Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs

Karol Desnos

► To cite this version:

Karol Desnos. Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs. Signal and Image processing. INSA de Rennes, 2014. English. NNT : 2014ISAR0004 . tel-01127297

HAL Id: tel-01127297
https://theses.hal.science/tel-01127297

Submitted on 7 Mar 2015

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

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

Thèse

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 des Images

présentée par
Karol Desnos
ECOLE DOCTORALE : MATISSE
LABORATOIRE : IETR

Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs

Thèse soutenue le 26.09.2014
devant le jury composé de :

Jarmo TAKALA
Professeur à l'Université Technologique de Tampere (Finlande) / Président
Alix MUNIER-KORDON
Professeur des Universités à l'Université Paris 6 / Rapporteur
Renaud SIRDEY
HDR, Commissariat à l'Energie Atomique de Saclay / Rapporteur
Shuvra S. BHATTACHARYYA
Professeur à l'Université du Maryland (USA) / Examinateur
Slaheddine ARIDHI
Docteur, Texas Instruments France / Encadrant
Maxime PELCAT
Maitre de Conférence à l'INSA de Rennes / Encadrant
Jean-François NEZAN
Professeur des Universités à l'INSA de Rennes / Directeur de thèse

Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs

Karol Desnos

Document protégé par les droits d'auteur

Contents

Acknowledgements **1**

1 Introduction **3**
1.1 General Context . . . 3
1.2 Scope of Thesis . . . 6
1.3 Outline . . . 7

I Background **9**

2 Dataflow Models of Computation **11**
2.1 Introduction . . . 11
2.2 What is a Dataflow Model of Computation? . . . 12
2.2.1 Models of Computation . . . 12
2.2.2 Dataflow Process Network . . . 13
2.2.3 Expression of Parallelisms . . . 14
2.2.4 Dataflow Models Properties . . . 16
2.3 Static Dataflow Models of Computation . . . 18
2.3.1 Synchronous Dataflow (SDF) . . . 18
2.3.2 Single-Rate SDF, Homogeneous SDF, and Directed Acyclic Graph . 21
2.3.3 Cyclo-Static Dataflow (CSDF) and Affine Dataflow (ADF) . . . 23
2.4 Hierarchical SDF Modeling . . . 24
2.4.1 Naive Hierarchy Mechanism for SDF . . . 24
2.4.2 Interface-Based SDF (IBSDF): a Compositional Dataflow MoC . . . 25
2.4.3 Deterministic SDF with Shared FIFOs . . . 27
2.5 Dynamic Dataflow Models of Computation . . . 28
2.5.1 Parameterized SDF (PSDF) . . . 28
2.5.2 Schedulable Parametric Dataflow (SPDF) . . . 31
2.5.3 Scenario Aware Dataflow (SADF) . . . 32
2.6 Summary of Presented Dataflow MoCs . . . 33

3 Rapid Prototyping Context **35**
3.1 Introduction . . . 35
3.2 What is Rapid Prototyping? . . . 35

3.2.1 Heterogeneous Architecture Modeling . . . 37
3.2.2 Parallel Application Modeling . . . 38
3.2.3 Application Deployment Constraints . . . 39
3.2.4 Mapping and Scheduling . . . 40
3.2.5 Simulation . . . 41
3.3 PREESM Rapid Prototyping Framework . . . 42
3.3.1 Related Work on Dataflow Programming . . . 42
3.3.2 PREESM Typical Rapid Prototyping Workflow . . . 43
3.3.3 PREESM Additional Features . . . 45
3.4 Memory Optimization for MPSoCs . . . 47
3.4.1 Modern Memory Architectures . . . 47
3.4.2 Memory Management for Multicore Architectures . . . 48
3.4.3 Usual Memory Allocation Techniques . . . 48
3.4.4 Literature on Memory Optimization and Dataflow Graphs . . . 50
3.5 Conclusion of the Background Part . . . 51

II Contributions **53**

4 Dataflow Memory Optimization: From Theoretical Bounds to Buffer Allocation **55**
4.1 Introduction . . . 55
4.2 Memory Exclusion Graph (MEG) . . . 55
4.2.1 IBSDF Pre-Processing for Memory Analysis . . . 55
4.2.2 MEG Definition . . . 57
4.2.3 MEG Construction . . . 61
4.3 Bounds for the Memory Allocation of IBSDF Graphs . . . 61
4.3.1 Least Upper Bound . . . 63
4.3.2 Lower Bounds . . . 63
4.3.3 Experiments . . . 67
4.4 Memory Allocation of a MEG . . . 70
4.4.1 MEG Updates with Scheduling and Time Information . . . 71
4.4.2 Memory Allocation Strategies . . . 72
4.4.3 Experiments . . . 74
4.5 Discussion . . . 79
4.5.1 Approach Limitations . . . 79
4.5.2 Comparison with a FIFO dimensioning technique . . . 79

5 Actor Memory Optimization: Studying Data Access to Minimize Memory Footprint **83**
5.1 Introduction . . . 83
5.2 Motivation: Memory Reuse Opportunities Offered by Actor Internal Behavior 83
5.2.1 Actors Inserted During Graph Transformations . . . 84
5.2.2 Actors with User-Defined Behavior . . . 86
5.3 Related Work . . . 88
5.3.1 Compiler optimization . . . 88
5.3.2 Dataflow Optimization . . . 89
5.4 Abstract Description of Actor Internal Behavior . . . 90
5.4.1 Matching Input and Output Buffers with Memory Scripts . . . 90
5.4.2 Port Annotations . . . 94

5.5 Match Tree Memory Minimization . . . 96
5.5.1 Applicability of Buffer Matches . . . 96
5.5.2 Match Trees Optimization Process . . . 101
5.5.3 MEG Update . . . 104

6 Memory Study of a Stereo Matching Application 107
6.1 Introduction . . . 107
6.2 System Presentation . . . 107
6.2.1 Stereo Matching Application . . . 107
6.2.2 Target Multicore Shared Memory Architectures . . . 111
6.3 Practical Issues . . . 112
6.3.1 Multicore Cache Coherence . . . 112
6.3.2 Data Alignment . . . 114
6.4 Experimental Results . . . 116
6.4.1 Static Memory Optimization . . . 116
6.4.2 Comparison with Dynamic Memory Allocation . . . 119

7 Towards Parameterized Results: The PiSDF Model 123
7.1 Introduction . . . 123
7.1.1 Motivation: Need for Reconfigurable Models of Computation . . . 123
7.2 Foundation Dataflow Models of Computation . . . 124
7.2.1 Synchronous Dataflow . . . 124
7.2.2 Interface-Based Synchronous Dataflow . . . 125
7.2.3 Parameterized Dataflow . . . 125
7.3 Parameterized and Interfaced Dataflow Meta-Modeling . . . 127
7.3.1 πSDF Semantics . . . 127
7.3.2 πSDF Reconfiguration . . . 129
7.4 Model Analysis and Behavior . . . 131
7.4.1 Compile Time Schedulability Analysis . . . 132
7.4.2 Runtime Operational Semantics . . . 132
7.5 PiMM Application Examples . . . 133
7.5.1 FIR Filter . . . 134
7.5.2 LTE PUSCH . . . 136
7.6 Comparison with Existing MoCs . . . 138
7.6.1 PiMM versus Parameterized Dataflow . . . 140
7.6.2 PiMM versus SADF . . . 141

8 Conclusion 143
8.1 Summary . . . 143
8.2 Future Work . . . 144
8.2.1 Support for Complex Memory Architectures . . . 144
8.2.2 Memory-Aware Scheduling Techniques . . . 144
8.2.3 Rapid Prototyping for πSDF graphs . . . 145

A French Summary 147
A.1 Introduction . . . 147
A.2 Modèles de calcul de type flux de données . . . 149
A.3 Prototypage rapide avec PREESM . . . 151
A.4 Optimisation mémoire des diagrammes de flux de données . . . 154

A.5 Optimisation mémoire des acteurs 155
A.6 Etude de cas : application d'appariement stéréoscopique 157
A.7 Vers plus de dynamisme : le modèle PiSDF 158
A.8 Conclusion 160

List of Figures **161**

List of Tables **165**

Acronyms **167**

Personal Publications **171**

Bibliography **173**

Acknowledgements

First, I would like to thank my advisors Dr. Maxime Pelcat and Dr. Slaheddine Aridhi, and my thesis director Pr. Jean-François Nezan for their guidance, their support, and their trust for the last three years. Thank you for giving me the opportunity to pursue a PhD on such an interesting topic and in such a motivating and friendly working environment. Maxime, thank you for your organizational and technical insights and thank you for your open-mindedness on research directions followed during this PhD. Slah, thank you for your warm welcome at Texas Instruments and for giving an industrial perspective to this PhD. Jeff, thank you for all your advice during these three years, and thank you for giving me the opportunity to work with international scientists of world renown.

I want to express my gratitude to Pr. Alix Munier-Kordon and Dr. Renaud Sirdey for being part of the PhD committee and for taking the time to review this thesis during their summer vacation. I also want to thank Pr. Jarmo Takala for presiding the PhD committee and Pr. Shuvra S. Bhattacharyya for being a member of the PhD committee. Shuvra, thanks also for welcoming me to the University of Maryland, for sharing your expertise on dataflow models, and for making this visit such an inspiring and fruitful collaboration.

I also would like to thank all members of the image group of the IETR for making me feel part of the team since the beginning. Special thanks to all my officemates during these three years: Jérome Gorin, Julien Heulot, Khaled Jerbi, Antoine Lorence, Erwan Nogues, and Fabien Racapé. Thanks also to Aurore Arlicot, Safwan El Assad, Clément Guy, Judicaël Menant, Yaset Oliva, Romina Racca and former members of the team Jonathan Piat and Matthieu Wipliez for their help and collaboration during this PhD. Also, thanks to Frédéric Garesché for his IT support and many thanks to Corinne Calo, Aurore Gouin and Jocelyne Trémier for their administrative support.

It has been a pleasure working with the High Performance Embedded Processing team from Texas Instruments France. Special thanks to Eric Biscondi, Raphael Defosseux, Renaud Keller, Lidwine Martinot, Filip Moerman, Alexandre Romana, Olivier Paviot, and Sébastien Tomas for helping me with the subtlety of multicore DSP programming.

Many thanks to all members of the DSPCAD group from the University of Maryland. Special thanks to Zheng "Joe" Zhou, it was a real pleasure to carpool and have these very interesting conversations with you everyday during my stay in Maryland. Thanks also to George Zaki for his helpful advice before and during my visit.

Thanks to Karina Aridhi for reading and correcting this documents and previous publications.

Thanks to the French taxpayers for, mostly reluctantly, funding the last 23 years of my education, including this PhD. Thanks also to all coffee producers around the world for making this work possible.

I am deeply grateful to my family and friends for their support and for helping me keep my sanity during these last three years. In particular, I would like to express my infinite gratitude to my parents, my sister, and my brother for their unconditional support and their love. I also want to thank the Martials for their encouragements and for making me feel part of the family. Thanks also to Nymeria and Burrito for their fluffiness.

Finally, and most importantly, I want to thank my dear love, Pauline, for sharing my life during the last 7 years. Pauline, thank you for your continuous support and for your indulgence to my crappy mood during the redaction of this document.

CHAPTER 1

Introduction

1.1 General Context

Over the last decade, **embedded systems** have been one of the most remarkable technological advances, driving the development of many industries, and entering the daily life of most human beings. An embedded system is an integrated electronic and computer system designed to serve a dedicated purpose. Car Anti-lock Braking Systems (ABS), Global Positioning Systems (GPS), e-readers, pacemakers, digital cameras, and autonomous vacuum cleaners are examples of modern appliances containing one or more embedded systems.

Embedded Systems Development Constraints

The development of an embedded system requires consideration of many constraints. These constraints can be classified in three categories, *application constraints*, *cost constraints* and *external constraints*.

- **Application constraints** refer to the requirements that an embedded system must satisfy to serve its intended purpose. For example, many embedded systems have performance requirements and must react to external events within a limited amount of time, or must produce results at a fixed rate. Another example of an application constraint is the reliability of an embedded system that restricts the probability of a system failure, primarily for safety reasons. Size limitation and power consumption are also major requirements for handheld or autonomous embedded systems.

- **Cost constraints** refer to all factors influencing the total cost of an embedded system. These factors cover the engineering development cost, the production cost, the maintenance cost, and also include the recycling cost of an embedded system.

- **External constraints** refer to the requirements that an embedded system must satisfy but that are nonessential to its purpose. Regulations and standards are examples of external constraints that dictate certain characteristics of an embedded system, but non-compliance would not prevent an embedded system from serving its purpose. The environment in which an embedded system is used can also have an impact on its design. Extreme temperatures, high humidity, rapidly changing pressure are examples of external constraints.

All these constraints are often contradictory, even within a single category. For example, reducing the power consumption of an embedded system can be achieved by lowering its clock frequency, which in turn will decrease the performance of this system. Hence, the development of an embedded system often consists of satisfying the most important constraints, and finding an acceptable trade-off between remaining ones.

Hardware and Software of Embedded Systems

Embedded systems, like most computer systems, are composed of two complementary parts: the *Hardware* and the *Software* part.

- **Hardware** refers to all the physical components that are assembled in order to create an embedded system. These components include processing elements, clock generators, sensors, actuators, analog-to-digital converters, memories, external communication interfaces, user interfaces, internal means of communication, and power management units among other elements. The set of hardware components contained in an embedded system and the connections between these components is defined as the **architecture** of this system. The most important hardware components are the processing elements that are responsible both for controlling the embedded system, and for performing its computation. Depending on the purpose of an embedded system, different kind of processing elements can be used. For example, simple low-power microcontrollers are often used in control systems that require little or no computation. For computation intensive systems, like audio and video Digital Signal Processing (DSP) systems, specialized processors providing a high computational power for a limited cost are used.

  In the early 1960s, following the creation of the first integrated circuit by Jack Kilby [Kil64], the first embedded systems were expensive, were composed of discrete hardware components, and were used only for military and space exploration projects. Progressively, the miniaturization of integrated circuits led to the integration of more and more hardware components within a single chip called a System-on-Chip (SoC). Nowadays, embedded systems are often based on complex integrated circuits called heterogeneous Multiprocessor Systems-on-Chips (MPSoCs). An heterogeneous MPSoC integrates all the elements of an embedded system, including different processing elements, on a single chip.

- **Software** refers to the computer programs executed by the processing elements of an embedded system. A program is a sequence of instructions stored as binary words in memory components of an embedded system. Processing elements of an embedded system iteratively read and execute the primitive operations encoded by the successive instructions of a program. Primitive operations include arithmetic and logic operations, jumps in the program sequence, configuration and control of other components of the embedded system, communication and synchronization with other processing elements, and read and write operations to the system memories.

  Programming an embedded system consists of writing a sequence of instructions that specify the behavior of its processing elements. In early embedded systems, program binary words were written directly by system developers. Progressively, this technique was replaced with higher-level programming methods that allow the specification of a program behavior with languages easier to understand for developers. Programs written in these high-level languages are then automatically translated into equivalent binary instructions by a computer program called a compiler.

In an effort to reduce development time of embedded systems, hardware and software parts are generally developed jointly as part of a hardware/software co-design process.

Embedded System Software Productivity Gap

In 1965, Moore predicted that the number of transistors in an integrated circuit would double every two years [Moo65]. Since then, this prediction has revealed itself to be true, driving an ever-increasing complexity of hardware components. In the meantime, the introduction of new programming languages with higher levels of abstraction has considerably improved the software productivity of developers. The software productivity measures the complexity of a program written by a developer within a given amount of time. As presented by Ecker et al. in [EMD09], the software productivity of developers is doubled every five years, or 2.5 times slower than hardware complexity. Hence, it is becoming more and more difficult for software developers to fully exploit the capabilities of hardware systems. This problem is often referred as the **software productivity gap**.

In recent years, the computational power of hardware was increased primarily by multiplying the number of processing elements running concurrently in MPSoCs. To fully exploit the computational power offered by these new architectures, programming languages must allow developers to specify applications where computations can be executed in parallel. A reason for the software productivity gap is that, as presented in [Bar09], the most popular language for programming embedded systems, namely the C language, is an imperative language whose basic syntax has a limited ability to express parallelism. This issue highlights the pressing need for new software programming techniques, such as dataflow programming, to bridge the software productivity gap.

Dataflow Programming

Dataflow Models of Computation (MoCs) have emerged as efficient programming paradigms to capture the parallelism of software. Applications modeled with dataflow MoCs consist of a directed graph where each vertex represents an independent computational module, and each edge represents an explicit communication channel between two vertices. The popularity of dataflow models and languages in the research, academic, and industrial communities is due to their natural expressivity of parallelism, their modularity, and their compatibility with legacy code. Indeed, dataflow graphs are used to specify networks of computational modules, but the specification of the internal behavior of these modules can be written in any programming language, including C code. The compatibility with legacy code has a positive impact on developers' productivity, since development time can be reduced by reusing previously developed and optimized programs.

Since the introduction of the first dataflow MoC by Kahn in [Kah74], many dataflow models have been proposed in the literature, each promoting a custom set of properties for application descriptions. For example, some dataflow models define construction rules for dataflow graphs that guarantee certain application properties at compile time, such as the performance or the reliability of applications. Such dataflow models are often used to capture the *application constraints* of embedded systems.

Rapid Prototyping Context

Rapid prototyping techniques and tools are developed as part of an effort to accelerate and ease the development of embedded systems. The classic embedded system design flow is a relatively straightforward process whose ultimate goal is to produce an embedded

system satisfying all design constraints. By contrast, the purpose of rapid prototyping techniques is to create an inexpensive prototype as early as possible in the development process. Generating new prototypes and analyzing their characteristics allow developers to identify critical issues of the prototype, and then to iteratively improve and refine the developed embedded system.

The work presented in this thesis is developed as part of a rapid prototyping framework called the Parallel and Real-time Embedded Executives Scheduling Method (PREESM). PREESM is a software rapid prototyping framework being developed at the Institute of Electronics and Telecommunications of Rennes (IETR) since 2007. The main objective of PREESM is to automate the deployment of applications modeled with dataflow graphs on heterogeneous MPSoCs. In addition to dataflow specification of the application, PREESM inputs include a System-Level Architecture Model (S-LAM) for specifying the targeted hardware components, and a *scenario* specifying design constraints of the embedded system. Using these inputs, PREESM can generate simulations of the embedded system behavior and can generate compilable code for the targeted MPSoC. PREESM has been successfully used for the rapid prototyping of real-time constrained Digital Signal Processing applications, computer vision, telecommunication, and multimedia applications on several heterogeneous MPSoCs [PAPN12, HDN+12, Zha13].

1.2 Scope of Thesis

After the processing elements, memories are the next most important components of an architecture. Indeed, from the hardware perspective, memory banks can cover up to 80% of the silicon area of an integrated circuit [DGCDM97]. Despite this large area overhead, and the associated power consumption, memory is still a scarce resource from the software perspective. For example, in the MPPA256 many-core chip from Kalray, each memory bank of 2 MBytes is shared between 16 processing elements, for a total of 32 MBytes of on-chip memory [Kal14]. Another example is the TMS320C6678 from Texas Instrument where 8.5 MBytes of memory are embedded on a chip with 8 DSP cores [Tex13]. These memory capacities remain relatively low compared to software requirements. For example, in a video decoding application, more than 3 MBytes are needed to store a single Full HD frame of 1920×1080 pixels. Although external memories can still be used as a complement to the memory embedded in MPSoCs, access to these external memory banks is much slower, and has a negative impact on the performance of an embedded system. Hence, the study and optimization of memory issues is a critical part in the development of an embedded system, as it can strongly impact the system performance and cost.

In this thesis, new techniques are introduced to study the memory issues encountered during the deployment of applications modeled with dataflow graphs onto heterogeneous multiprocessors architectures. The main contributions of this thesis are:

1. A method to **derive the memory bound requirements of an embedded system** in the early stages of its development, when there is a complete abstraction of the system architecture [DPNA12]. This method is based on an analysis of the system application, and allows the developer of a multi-core shared-memory system to adequately size the chip memory.

2. A flexible method to **minimize the amount of memory allocated** for applications implemented on a shared-memory MPSoC [DPNA13]. In this method, memory allocation can be performed at three distinct stages of the rapid prototyping process,

each offering a distinct trade-off between memory footprint and flexibility of the application execution.

3. A set of **annotations for dataflow graphs** that allows software developers to specify internal data dependencies of the computational modules. An optimization technique is also presented to automatically **exploit the memory reuse opportunities** offered by these new annotations and reduce the amount of memory allocated by the rapid prototyping process.

4. A case study of the deployment of **a state-of-the-art computer-vision application on a physical MPSoC**. In addition to showing the efficiency of the new memory optimization technique, this case study also presents technical solutions to support the execution of a dataflow graph on a cache-incoherent MPSoC [DPNA14].

5. **A new reconfigurable dataflow meta-model** that overcomes the limitations of the dataflow model currently used in Preesm while preserving its most advantageous characteristics [DPN+13].

All these contributions have been developed as part of a scientific collaboration between the IETR, Texas Instrument France (contributions 1 to 5), and the DSPCAD group from the University of Maryland (contribution 5).

1.3 Outline

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

In Part I, Chapter 2 formally defines the concept of dataflow Model of Computation (MoC) and presents the specific characteristics of all dataflow MoCs studied in this thesis. Then, Chapter 3 details the design challenges addressed in rapid prototyping frameworks and emphasizes the importance of memory-related issues.

In Part II, the memory bounding technique and the memory allocation techniques based on dataflow descriptions of applications are presented and evaluated in Chapter 4. Then, Chapter 5 presents the memory minimization technique based on a new set of annotations for the computational modules of a dataflow graph. In Chapter 6, the memory analysis and optimization techniques are evaluated for the rapid prototyping of a computer vision application on a real MPSoC. Chapter 7 defines the semantics of the new dataflow meta-model and compares its characteristics with existing MoCs. Finally, Chapter 8 concludes this work and proposes potential research directions for future research.

Part I

Background

CHAPTER 2

Dataflow Models of Computation

2.1 Introduction

Sketches are commonly used by engineers, scholars, and researchers as a way to conceptualize and illustrate systems and ideas in early stages of development. These informal graphical representations are used in many engineering domains to specify the purpose of the components of a system and the relations between these components.

Block diagrams are among the most popular informal models for the high-level specification of electronic and computer systems. A block diagram is a drawing composed of a set of boxes, usually representing the components of the system, and a set of lines and arrows representing the relations between components. In 1921, Gilbreth et al. [GG21] proposed a first formalization of block diagrams called "process chart" for the specification of production processes.

In the computer science domain, Kelly et al. [KLV61] introduced the BLOck DIagram compiler (BLODI) in 1961. BLODI is the first compiler for programs described with a formalized block diagram language. At the time of punched cards, the main objective of the BLODI language was to make computer programming accessible to persons with no knowledge in programming languages. The BLODI language was composed of a set of 30 primitive blocks, like adders, filters, and quantizers, that could be used to compose Digital Signal Processing (DSP) applications.

Nowadays, graphical languages are still popular for the specification and the development of software and hardware systems. For example, the MCSE (methodology for the design of electronic systems) [Cal93], is a hardware/software co-design methodology that is partly based on a formal block diagram description of the specified system. In computer science, the UML (Unified Modeling Language) [RJB04] is a collection of diagrams used for the high-level specification of software solutions. Following the BLODI language, several diagram-based alternatives to classical text-based programming languages have been proposed for different purposes. Labview® [Joh97] and Matlab Simulink® [Mat96] are among the most popular diagram-based programming languages used in commercial development environments.

This thesis focuses on the study of applications described with diagram-based Models of Computation (MoCs) called dataflow MoCs. Dataflow MoCs can be used to specify a wide

range of Digital Signal Processing (DSP) applications such as video decoding [TGB+06], telecommunication [PAPN12], and computer vision [NSD05] applications.

Dataflow MoCs and their properties are formally defined in Section 2.2. Sections 2.3, 2.4, and 2.5, respectively, present state-of-the-art static, hierarchical, and dynamic dataflow models proposed in the literature. Finally, Section 2.6 summarizes the properties of all dataflow MoCs presented in this chapter.

2.2 What is a Dataflow Model of Computation?

2.2.1 Models of Computation

A Model of Computation (MoC) is a set of operational elements that can be composed to describe the behavior of an application. The set of operational elements of a MoC and the set of relations that can be used to link these elements are called the semantics of a MoC.

As presented in [Sav98], MoCs can be seen as an interface between the computer science and the mathematical domain. A MoC specifies a set of rules that control how systems described with the MoC are executed. Each element of the semantics of a MoC can be associated to a set of properties such as timing properties or resource requirements. These rules and properties provide the theoretical framework that can be used to formally analyze the characteristics of applications described with a MoC. For example, using a mathematical analysis, it may be possible to prove that an application described with a MoC will never get stuck in an unwanted state or that it will always run in a bounded execution time.

There exists a wide variety of MoCs that each have their own specificities and objectives. The following MoCs are examples of popular MoCs among programmers and system designers.

- **Finite-State Machine (FSM):** The FSM MoC semantics consists of 2 elements: states and transitions between states. Each transition of the model is associated with a condition that guards the traversal of this transition. In an FSM, a unique state is active at any given time. A transition from a state to another can be traversed only when the associated condition is valid. FSMs are commonly used for the description of sequential control systems. For example, FSMs are used in VHDL to capture the control part of a hardware system [Gol94].

- **Lambda calculus:** The lambda calculus MoC semantics consists of a single element called lambda term [Chu32]. A lambda term is either a variable, a function with a unique lambda term argument, or the application of a function to a lambda term. Any lambda calculus is obtained by composing lambda terms. Lambda calculus was originally developed by mathematicians as a way to study and characterize functions [Ros84]. Today, MoCs derived from the lambda calculus MoC are the foundation of functional programming languages such as Haskell, Lisp, Scheme, and OCaml.

- **Process network:** The process network MoC semantics consists of 2 elements: processes and unbounded First-In First-Out queues (FIFOs). The processes of a process network are independent tasks that perform computation concurrently. Processes communicate with each other by sending quanta of data, called data tokens, through the FIFOs of the process network. The process network MoC was originally developed by Kahn as a semantics for parallel programming languages [Kah74]. The dataflow MoCs studied in this thesis are derivatives of the process network MoC.

- **Logic circuit:** The logic circuit MoC semantics consists of 2 elements: logic gates and boolean signals. In the logic circuit MoC, systems have no state and the boolean value (*true* or *false*) of an output signal of a logic gate only depends on the current values of the input signals of this logic gate [Sav98]. The logic circuit MoC is commonly used in VHDL to describe the basic blocks composing a hardware system.

Most programming languages implement the semantics of several MoCs. For example, the VHDL language supports many MoCs including *logic circuits*, *finite state machines*, and *process networks* [Ash90]. Composing several MoCs into a single programming language allows the developer of a system to choose the best suited MoC for the description of each part of the system.

2.2.2 Dataflow Process Network

In [LP95], Lee and Parks formalize the semantics of the Dataflow Process Network (DPN) MoC as a subset of the process network MoC. Although the Dataflow Process Network (DPN) MoC is not the first dataflow MoC published in the literature [Kah74, Den74], it constitutes a first attempt to provide a formal semantics for dataflow MoCs. In this thesis, a DPN is formally defined as follows:

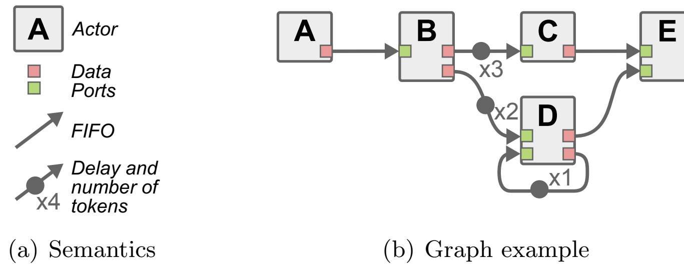
Definition 2.2.1. *A Dataflow Process Network (DPN) is a directed graph denoted by* $\mathbf{G} = \langle \mathbf{A}, \mathbf{F} \rangle$ *where:*

- ***A*** *is the set of vertices of* G*. Each vertex* $a \in A$ *is a computational entity named an* actor *of the DPN. Each actor* $a \in A$ *is defined as a tuple* $\mathbf{a} = \langle \mathbf{P}^{in}_{data}, \mathbf{P}^{out}_{data}, \mathbf{R}, \mathbf{rate} \rangle$ *where:*
  - $\mathbf{P}^{in}_{data}$ *and* $\mathbf{P}^{out}_{data}$ *respectively refer to the set of data input and output ports of the actor.*
  - $\mathbf{R} = \{R_1, R_2, ..., R_n\}$ *is the set of firing rules of the actor. A firing rule* $R_i \in R$ *is a condition that, when satisfied, can start an execution, called firing, of the associated actor.*
  - $\mathbf{rate} : (R, P^{in}_{data} \cup P^{out}_{data}) \to \mathbb{N}$ *associates a firing rule to the number of atomic data objects, called* data tokens*, consumed or produced on a given data port, for a firing of the actor resulting from the validation of this firing rule.*
- $\mathbf{F} \subseteq A \times A$ *is the set of edges of* G*. Each edge* $f \in F$ *is an unbounded First-In First-Out queue (FIFO) that transmits data tokens between actors. Each FIFO* $f \in F$ *is defined as a tuple* $\mathbf{f} = \langle \mathbf{prod}, \mathbf{cons}, \mathbf{src}, \mathbf{snk}, \mathbf{delay} \rangle$ *where:*
  - $\mathbf{prod} : F \to A$ *and* $\mathbf{cons} : F \to A$ *associate producer and consumer actors to a FIFO.*
  - $\mathbf{src} : F \to P^{out}_{data}$ *and* $\mathbf{snk} : F \to P^{in}_{data}$ *associate source and sink ports to a FIFO.*
  - $\mathbf{delay} : F \to \mathbb{N}$ *corresponds to a number of data tokens present in the FIFO when the described application is initialized.*

The semantics of the DPN MoC only specifies the *external* behavior of the actors: the firing rules specify when an actor should be executed and the port rates specify how many tokens are exchanged by an actor for each firing. The description of the *internal* behavior of an actor is not part of the DPN MoC. In order to specify the actual computation performed by the actor at each firing, a *host language* must be used. In most

dataflow programming frameworks, imperative languages, like C or Java, or hardware description languages, like VHDL, are used as *host languages* to describe the internal behavior of actors [GLS99]. Alternatively, dedicated languages, like the CAL Actor Language (CAL) [EJ03] or ΣC [Bod13, GSLD11, ABB+13], have been proposed to describe both the *external* and *internal* behaviors of actors.

Figure 2.1 illustrates the graphical elements associated to the semantics of the DPN MoC and gives an example of a DPN graph. The example graph presented in Figure 2.1(b) contains 5 actors interconnected by a network of 6 FIFOs. The FIFOs linking actors B to C, actors B to D, and actor D to itself contain 3, 2, and 1 initial tokens respectively.



(a) Semantics (b) Graph example

Figure 2.1: *Dataflow Process Network (DPN) MoC*

Dataflow MoCs

The semantics presented in Definition 2.2.1 serves as a basis for the semantics of many *dataflow MoCs*. Indeed, most dataflow MoCs are the result of a *specialization* or a *generalization* of the DPN semantics. *Specializing* the DPN semantics consists of adding new restrictions to the MoC. For example, the Synchronous Dataflow (SDF) MoC [LM87b] restricts the rates of data ports to a single scalar value [LM87b]. The DPN MoC itself is a *specialization* of the Kahn Process Network (KPN) MoC [Kah74]. *Generalizing* the DPN semantics consists of adding new elements to its semantics. For example, the Scenario-Aware Dataflow (SADF) MoC [TGB+06] introduces, among other new elements, a stochastic model of the execution time of actors. *Specialization* and *generalization* of the DPN semantics are often used jointly to derive new dataflow MoCs.

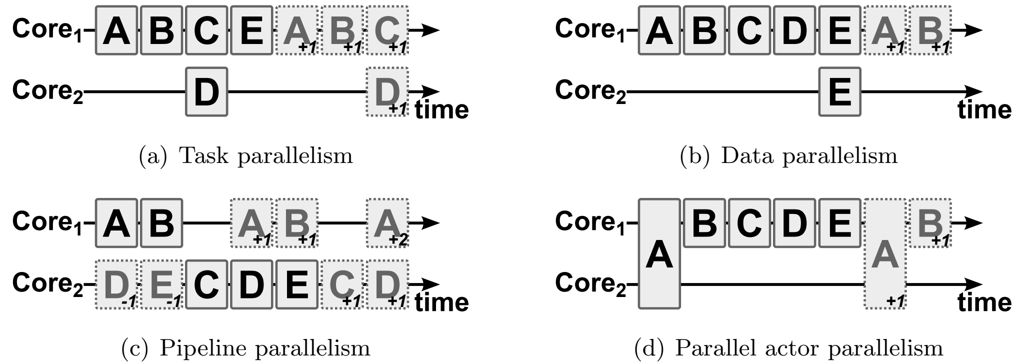
DPN and its derivatives (Sections 2.3 to 2.5) are popular MoCs for the description of Digital Signal Processing (DSP) applications and systems. Indeed, as shown in [LP95], data tokens provide an intuitive representation of the samples of a discrete signal, and networks of actors naturally capture the successive transformations applied to a digital signal. Finally, DPNs are often used to model repetitive DSP applications whose purpose is to process a continuous stream of data tokens.

2.2.3 Expression of Parallelisms

One of the major objectives of dataflow graphs is to ease the description of parallelism. In [ZDP+13], Zhou et al. identify 4 different sources of parallelism in a dataflow graph: *task parallelism*, *data parallelism*, *pipeline parallelism*, and *parallel actor parallelism*.

Figure 2.2 presents 4 Gantt diagrams that illustrate the different sources of parallelism. Each Gantt diagram represents a sequence of actor firings, called schedule, corresponding to an execution of the DPN presented in Figure 2.1(b) on two processing elements. In these schedules, it is assumed that the DPN is a repetitive process whose schedule can be repeated indefinitely. Each schedule contains a complete repetition, called **iteration**, of

the periodic schedule. Actors with a dotted border belong to prior or next iterations of the schedule, as indicated by the *+1*, *-1*, and *+2* notations on these actors.



(a) Task parallelism

(b) Data parallelism

(c) Pipeline parallelism

(d) Parallel actor parallelism

Figure 2.2: *Illustration of the 4 sources of parallelism in dataflow MoCs*

- **Task parallelism:** A FIFO of a dataflow graph induces a dependency constraint between its source and sink actors. A chain of actors linked by dependency constraints is called a **data-path**. Two actors belong to parallel data-paths if there exists no data-path between these actors. In such a case, there is no dependency constraint between these actors and their firings can be processed in parallel. For example, in Figure 2.1(b), actors C and D belong to parallel data-paths and can be executed concurrently, as can be seen in Figure 2.2(a).

- **Data parallelism:** In dataflow graphs, the computation of an actor only depends on the data tokens consumed during a firing. Hence, each firing of a single actor is independent from its other firings, and no state or context needs to be restored before firing an actor [LP95]. Since successive firings of an actor are independent, if enough data tokens are present in the input FIFOs, then several firings can be executed concurrently. For example, in Figure 2.2(b), it is assumed that actors C and D produce enough data tokens for actor E to be executed twice in parallel. If the behavior of an actor requires the definition of a state, this state must be explicitly specified in the dataflow graph with a self-loop FIFO conveying state data-tokens from a firing to the next, as illustrated by actor D in Figure 2.1(b).

- **Pipeline parallelism:** Pipelining an application consists of starting a new iteration of a dataflow graph before the end of the previous iteration. Pipelining is possible if there is no data dependency between successive iterations of a dataflow graph. In order to pipeline a dataflow graph, the sequence of actor firings is divided into subsequences, called stages, that can be executed concurrently. Initial data tokens, also called delays, can be used to separate two pipeline stages explicitly in the application graph. For example, in Figure 2.2(c), FIFOs BC and BD contain initial data tokens that can be used to fire actors C and D without waiting for the first data tokens produced by actor B. Hence, actors A and B and actors C, D, and E respectively compose the two pipeline stages that can be repeatedly executed in parallel.

- **Parallel actor parallelism:** A parallel actor is an actor that embeds inner (intra-firing) parallelism, and whose execution may be accelerated by the use of multiple processing elements [ZDP+13]. The inner parallelism of a parallel actor can have different sources:

– The host language describing the internal behavior of actors allows parallel computation. For example, thread-based programming languages can be used to describe parallel computations. The advantage of running several threads in parallel instead of firing several actors concurrently is that multiple threads may access shared variables whereas several actors can not.
– The parallel actor is a hierarchical actor whose internal behavior is itself described with a dataflow graph. To this purpose, several hierarchical dataflow MoCs have been proposed in the literature [PBR09, BB01, NL04].

In Figure 2.2(d), it is assumed that actor A is a parallel actor whose firings require the concurrent use of 2 processing elements of the architecture.

In Figure 2.2, each subfigure illustrates a single source of parallelism. In real schedules, these 4 sources of parallelism are combined in order to fully exploit the parallelism offered by dataflow graphs.

2.2.4 Dataflow Models Properties

Research on dataflow modeling leads to the continuing introduction of new dataflow models [JHM04]. New dataflow MoCs are often introduced to fulfill objectives that preexisting models failed to achieve. For example, some dataflow models are introduced with a mathematical formalism that enforces the analyzability of the model [SGTB11]. Other models simply extend the semantics of existing models in order to improve their expressivity and enable the description of a broader range of applications [BB01].

To help the developer of an application select the dataflow MoC that best suits his needs, a set of application and model properties can be used to compare the capabilities of existing dataflow MoCs.

Application properties

To fully understand the model properties used for comparing dataflow MoCs, some properties of applications when described with dataflow graphs must be in introduced first.

- **Schedulability:** A dataflow graph is *schedulable* if it is possible to find at least one sequence of actor firings, called a schedule, that satisfies all the firing rules defined in the graph. Depending on the dataflow MoC used to model an application, the schedulability of this application can be checked at compile time or during the execution of the system. A possible cause of non-schedulability for a dataflow graph is the possibility to reach a *deadlock* state where no Fifo contains enough data tokens to initiate a firing of an actor. The non-schedulability of an application can also be caused by external factors such as the lack of sufficient hardware resource to schedule a dataflow graph under a time constraint.

- **Consistency:** A dataflow graph is *consistent* if its execution does not cause an indefinite accumulation of data tokens in one or several Fifos of the graph. Although in Definition 2.2.1, the Fifos of a DPN are supposed to be unbounded, in practice, the amount of memory used to execute a DPN is always limited. Hence, if a dataflow graph is *inconsistent*, its execution will eventually generate more data tokens than the available storage capacity.

Dataflow MoCs properties

The following properties can be used to compare several dataflow MoCs. The presented list of properties is not meant to be an exhaustive list of all the properties used for comparing dataflow MoCs. Indeed, many other terms are used in the literature to compare dataflow MoCs, often with an overlapping meaning with one or several of the properties defined in this list.

- **Decidability:** A dataflow MoC is *decidable* if the schedulability and the consistency of applications described with this model can be proved statically (i.e. at compile-time) [BL06]. Hence, using a decidable dataflow MoC makes it possible to guarantee at compile-time that a dataflow graph will never reach a deadlock state and that its execution will require a finite amount of memory. As presented in [BL06], decidable dataflow MoCs are not Turing complete and may be unable to describe certain applications.

- **Determinism:** A dataflow MoC is *deterministic* if the output of an algorithm only depends on its inputs, but not on external factors such as time or randomness [LP95]. If determinism is a desired feature for most control and DSP applications, non-determinism may also be needed to describe applications reacting to unpredictable inputs [LP95].

- **Compositionality:** A dataflow MoC is *compositional* if the properties of a dataflow graph described with this MoC are independent from the internal specification of the actors that compose it [Ost95, TBG+13]. Compositionality is a desirable feature especially for hierarchical dataflow MoCs where the internal behavior of actors may be specified with a dataflow graph (cf. Section 2.4.2). In hierarchical dataflow MoCs, compositionality guarantees that modifications made to the subgraph of an actor will have no impact on the consistency or the schedulability of graphs in upper levels of hierarchy [TBG+13, PBR09, NL04].

- **Reconfigurability:** A dataflow MoC is *reconfigurable* if the firing rules and rates associated to the data ports of its actors can be changed dynamically depending on the application inputs [NL04]. DPN is a reconfigurable MoC since each data port can be associated to a set of firing rules and token rates that are dynamically selected during the application execution, depending on the number and value of available data tokens.

These four properties are predicates that can be used to characterize a dataflow MoC objectively. Indeed, a dataflow MoC is either *deterministic* or *non-deterministic* but cannot be somewhere in between. In addition to these objective properties, several informal properties can be used to compare dataflow MoCs. Although these properties cannot be objectively measured, they are commonly used as arguments for the relative comparison of new dataflow MoCs with existing MoCs.

- **Predictability:** The *predictability* property is related to the *reconfigurability* property of a dataflow MoC. This property evaluates the amount of time between the reconfiguration of the firing rules of an actor and the actual firing of this actor. The *predictability* of a dataflow MoC is inversely proportional to how often the firing rule of an application graph can be reconfigured [NL04].

- **Conciseness:** The *conciseness* (or *succinctness* [SGTB11]) of a dataflow MoC evaluates the ability of a MoC to model an application with a small number of actors. When a graphical interface is used to edit dataflow graphs, conciseness is an important property to limit the size of edited graphs.

- **Analyzability:** The *analyzability* of a dataflow MoC evaluates the availability of analysis and synthesis algorithms that can be used to characterize application graphs. For example, analysis algorithms can be applied at compile-time to compute the worst-case latency or the maximum memory requirements of a dataflow graph [SGTB11].

- **Expressivity:** The *expressivity*, or *expressive power* of a dataflow MoC evaluates the complexity of application behaviors that can be described with this MoC. For example, the *expressivity* of the DPN MoC has been proven to be equivalent to a Turing machine [BL93], and can thus be used to describe any application. Specializations of the DPN MoCs often restrict the expressivity of the MoC in order to increase its *analyzability* and *predictability*. *Expressivity* is often mistaken for *conciseness*. For example, the CSDF MoC is often said to be more *expressive* than the SDF MoC, but meaning instead that it has a better *conciseness* (Section 2.3).

The following sections present the semantics and the properties of the dataflow MoCs that will be studied in this thesis. These dataflow MoCs are sorted into two categories depending on their reconfigurability. *Static* MoCs, which are non-reconfigurable MoCs, are presented in Sections 2.3 and 2.4, and *dynamic* MoCs, which are reconfigurable MoCs, are presented in Section 2.5.

2.3 Static Dataflow Models of Computation

Static dataflow MoCs are *non-reconfigurable* and *deterministic* MoCs where the sequence of firing rules executed by all actors of a graph is known at compile time [LM87b]. Since the sequence of firing rules of an actor is known *a priori*, the production and consumption rates of an actor never depend on the value of the data tokens processed by the actor. This condition restricts the *expressivity* of static dataflow MoCs. For example, conditional behaviors such as `if-then-else` statements have no equivalent in static dataflow MoCs.

As a counterpart for their restricted *expressivity*, static dataflow MoCs are *decidable* MoCs for which all application graphs can be scheduled at compile-time [LP95].

2.3.1 Synchronous Dataflow (SDF)

Synchronous Dataflow (SDF) [LM87b] is the most commonly used static specialization of the DPN MoC. Production and consumption token rates set by firing rules are fixed scalars in an SDF graph. Formally, the SDF MoC is defined as follows:

Definition 2.3.1. *A Synchronous Dataflow (SDF) graph is a graph $G = \langle A, F \rangle$ respecting the Dataflow Process Network (DPN) MoC with the following restrictions:*

- *Each actor $a \in A$, with $a = \langle P^{in}_{data}, P^{out}_{data}, R, rate \rangle$, is associated to a unique firing rule: $R = \{R_1\}$*

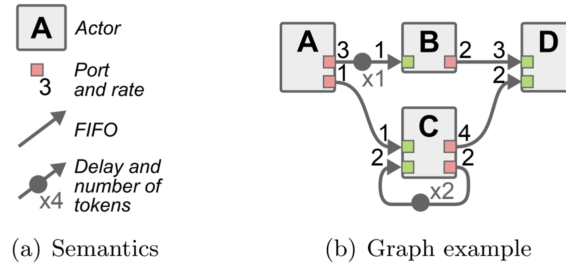
- *For each data input port $p \in P^{in}_{data}$ of an actor, the consumption rate associated to the unique firing rule R_1 of the actor is a static scalar that also gives the number of data tokens that must be available in the FIFO to start the execution of the actor.*

- *For each data output port $p \in P_{data}^{out}$ of an actor, the production rate associated to the unique firing rule R_1 of the actor is a static scalar.*

In addition to these restrictions, the following simplified notation is introduced.

- **rate** : $A \times F \to \mathbb{N}$ *is the production or consumption rate of actor $a \in A$ on* FIFO *$f \in F$. If a is both producer and consumer of f, then $rate(a, f)$ is the difference between the production and the consumption rates on f.*

Figure 2.3 illustrates the graphical elements associated to the semantics of the SDF MoC and gives an example of SDF graph.



(a) Semantics (b) Graph example

Figure 2.3: *Synchronous Dataflow (SDF) MoC*

The popularity of the SDF MoC comes from its great *analyzability*. Indeed, low complexity algorithms have been published to check the *consistency* and *schedulability* of SDF graphs [LM87b], to derive mono-core looped schedules of minimal length [BML96], but also to compute the maximum achievable throughput for the multicore execution of a graph [GGS+06].

Consistency and Schedulability

Checking the *schedulability* and the *consistency* of an SDF graph is a critical step in all dataflow programming frameworks, as it proves the absence of deadlock and the bounded memory requirements of an application.

The method introduced by Lee and Messerschmitt in [LM87b] to verify the *consistency* of an SDF graph is based on a **topology matrix** that characterizes the FIFOs and the *rates* of an SDF graph. The topology matrix is formally defined as follows:

Definition 2.3.2. *Considering an SDF graph $G = \langle A, F \rangle$, the associated topology matrix Γ is a matrix of size $|F| \times |A|$such that:*

- *Each column Γ_j of the matrix is associated to an actor $a_j \in A$ of G.*
- *Each row Γ_i of the matrix is associated to a* FIFO *$f_i \in F$ of G.*
- *Matrix coefficient* $\Gamma_{i,j} = \begin{cases} rate(a_j, f_i) & \text{if } a_j = prod(f_i) \\ -rate(a_j, f_i) & \text{if } a_j = cons(f_i) \\ 0 & \text{otherwise} \end{cases}$

The topology matrix associated to the SDF graph of Figure 2.3(b) is presented hereafter. The columns and rows of the matrix are labeled with the corresponding actors and

FIFOs respectively.

$$\Gamma = \begin{array}{c} \\ AB \\ AC \\ CC \\ BD \\ CD \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 3 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -3 \\ 0 & 0 & 4 & -2 \end{pmatrix} \end{array}$$

An interesting coefficient in this matrix is the coefficient corresponding to tokens produced and consumed by actor C on FIFO CC. Because actor C produces and consumes the same number of token on this FIFO, the positive and negative rates cancel each other, and the coefficient is set to 0. When an actor possesses a self-loop FIFO, its production and consumption rates on this FIFO should always be equal. Otherwise, tokens will either accumulate indefinitely on this FIFO, or this FIFO will eventually cause a deadlock.

The **state** of an SDF graph is characterized by the number of data tokens stored in each FIFO, and can be represented by a vector of size $|F|$. For example, the initial state of the SDF graph of Figure 2.3(b) is: $state(0) = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 \end{pmatrix}^T$. Given a state vector $state(n)$, the state resulting from the firing of the jth actor of the graph can be computed with the following equation: $state(n+1) = state(n) + \Gamma \cdot e^j$ where e^j is the jth canonical basis vector in Euclidean space (the vector with all coefficients equal to 0 but the jth equal to 1). For example, the state resulting from firing of actor B of the example SDF graph is $state(1) = state(0) + \Gamma \cdot e^1 = \begin{pmatrix} 0 & 0 & 2 & 2 & 0 \end{pmatrix}^T$.

Theorem 2.3.1. *A connected SDF graph $G = \langle A, F \rangle$ with a topology matrix Γ is consistent if and only if* $\text{rank}(\Gamma) = |A| - 1$

A non-connected SDF graph is a graph whose actors can be separated in two (or more) groups with no FIFO between actors belonging to different groups. Proving the consistency of a non-connected SDF graph consists of applying Theorem 2.3.1 separately to the connected SDF subgraphs formed by each group of actors.

A proof for Theorem 2.3.1 can be found in [LM87a]. Using Theorem 2.3.1, it is thus possible to prove that the repeated execution of an SDF graph will not result in an infinite accumulation of data tokens on a FIFO of this graph.

The *consistency* of an SDF graph implies the existence of a **Repetition Vector (RV)** of size $|A|$. The integer coefficients of the RV give the minimal number of firings of each actor to return the graph back to its original state. Executing an **iteration** of an SDF graph consists of firing each actor of this graph as many times as given by the RV.

Computing the RV q of a topology matrix Γ consists of finding a positive and integer vector that solves the following equation: $\Gamma \cdot q = \begin{pmatrix} 0 & \cdots & 0 \end{pmatrix}^T$. The RV for the SDF graph of Figure 2.3(b) is $q = \begin{pmatrix} 1 & 3 & 1 & 2 \end{pmatrix}^T$.

Theorem 2.3.2. *An SDF graph $G = \langle A, F \rangle$ is* schedulable *if and only if all following conditions are verified:*

- *G is consistent*
- *A sequence of actor firing can be constructed such that:*
  - *each actor is fired as many times as required in the Repetition Vector (RV).*
  - *the firing rule of each actor is respected: enough data tokens are available to start each firing of the sequence.*

The existence of the RV (i.e. the consistency) is not a sufficient condition to guarantee the schedulability of an application. For example in Figure 2.3(b), if there were no initial data token in the self-loop FIFO of actor C, the graph would still be consistent, since the topology matrix does not depend on delays, but it would never be possible to fire actor C.

In [BELP96], Bilsen et al. propose an algorithm to build a sequence of actor firings that verifies the second condition of Theorem 2.3.2. If the algorithm fails, the SDF cannot be executed without reaching a deadlock. An example of valid sequence of actor firings for the SDF graph of Figure 2.3(b) is: B, A, B, D, C, B, D.

It is important to note that *consistency* and *schedulability* of an SDF graph can be checked without any information on the actual hardware architecture executing the application.

2.3.2 Single-Rate SDF, Homogeneous SDF, and Directed Acyclic Graph

The **single-rate SDF** MoC is a specialization of the SDF MoC defined as follows:

Definition 2.3.3. *A single-rate SDF graph is an SDF graph where the production and consumption values on each FIFO are equal. Formally:*

$$\forall f \in F, rate(prod(f), f) = rate(cons(f), f)$$

The single-rate SDF MoC has the same *expressivity* as the SDF MoC. Consequently, for all consistent SDF graphs, there exists an equivalent single-rate SDF graph.

Unrolling an SDF graph into an equivalent single-rate SDF graph consists of duplicating the SDF actors by their number of firings in the RV [LM87b]. As a result of this transformation, data tokens produced by a single producer can be consumed by several consumers and vice-versa. Since the SDF MoC forbids the connection of multiple FIFOs to a single port of an actor, new *Fork* and *Join* actors are introduced [Pia10]. The purpose of *Fork* actors, also called *Split* [CDG+14] or *Explode* [FGG+13] actors, is to distribute data tokens produced by a single actor to several consumers. Conversely, *Join* actors, also called *Implode* [FGG+13] actors, are used to gather data tokens produced by several producers for a single consumer.

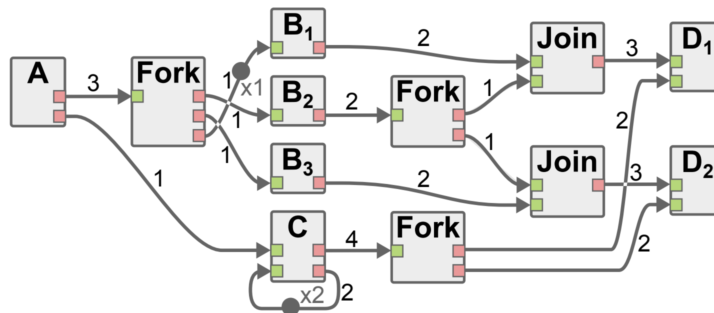


Figure 2.4: *Single-rate SDF graph derived from the SDF graph of Figure 2.3(b)*

Figure 2.4 illustrates the graph resulting from the conversion of the SDF graph of Figure 2.3(b) into its single-rate equivalent. In addition to the 7 actors resulting from duplication of the original 4 actors of the SDF graph, 3 *Fork* actors and 2 *Join* actors have been added to the graph. Each single-rate FIFO of the graph is labeled with its production/consumption rate. As illustrated in this example, FIFOs of a single-rate graph only contain a number of delays that is a multiple of their production/consumption rate.

In [SB09], Sriram and Bhattacharyya propose an algorithm to realize the transformation of an SDF graph into the equivalent single-rate SDF graph. As shown in [PBL95], the single-rate transformation may result in the addition of an exponential number of actors to the graph.

The single-rate transformation is commonly used in many dataflow programming frameworks as a way to reveal *data parallelism*, by transforming it into explicit *task parallelism* (Section 2.2.3). Another advantage of single-rate SDF graphs is that each actor needs to be fired only once per iteration of the graph. For this reason, single-rate SDF graphs are often used as an input to scheduling processes.

As presented in [PBL95], transforming an SDF graph into an equivalent single-rate SDF graph may require an exponential number of actor duplications. However, despite this limitation, the single-rate transformation is still used in many dataflow programming frameworks [PAPN12, SB09]. In practice, it is the developer responsibility to ensure that the specified production and consumption rates will not result in a prohibitively large single-rate graph. This constraint has been showed to be compatible with the description of real applications from the telecommunication [PAPN12], the image processing [HDN+12], and the computer vision domains [Zha13].

The **homogeneous SDF MoC** is a specialization of the single-rate SDF MoC where all production and consumption rates are equal to 1. The straightforward transformation of a single-rate SDF graph into an equivalent homogeneous SDF graph consists of setting the size of data tokens on each single-rate FIFO to the production/consumption rate of this FIFO.

The **Directed Acyclic Graph (DAG) MoC** is a specialization of the SDF MoC where cyclic data-paths are forbidden. Given a consistent and schedulable SDF graph, its transformation into an equivalent Directed Acyclic Graph (DAG) consists of replacing FIFOs that contain *delays* with a pair of special actors: *Save* and *Restore*. In order to be schedulable, all cyclic data-paths must contain at least one initial data token, consequently, replacing FIFOs with delays will naturally break all cycles. The purpose of the *Save* actor is to backup as many data tokens as the number of delays of the FIFO. The *Save* actor must be executed before the end of an iteration, saving data tokens that will be consumed during the next iteration. Then, the purpose of the *Restore* actor is to read the backed up data tokens and send them to the appropriate consumer. A *Save* actor must always be scheduled before its corresponding *Restore* actor. Indeed, the transformation of an SDF graph into a DAG hides a causality property.

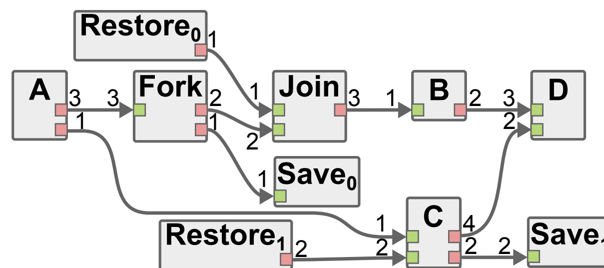


Figure 2.5: *Directed Acyclic Graph (DAG) derived from the SDF graph of Figure 2.3(b)*

Figure 2.5 illustrates the DAG derived from the SDF graph of Figure 2.3(b). In this DAG, the self-loop FIFO, also called feedback FIFO, of actor C has been replaced with a pair of *Save*/*Restore* actors that transmit the two data tokens from an iteration of the DAG to the next. A second pair of *Save*/*Restore* actors replaces the unique delay of FIFO

AB. Since only one data token out of the 3 produced by actor A is delayed, *Fork* and *Join* actors are added in order to isolate and regroup the delayed data token.

The single-rate and the DAG transformations are often combined to produce a single-rate DAG, also called an Acyclic Precedence Expansion Graph (APEG) [SL90].

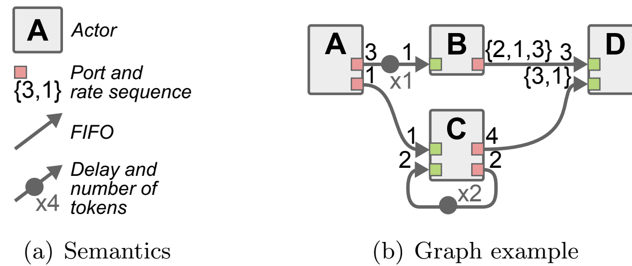
2.3.3 Cyclo-Static Dataflow (CSDF) and Affine Dataflow (ADF)

The Cyclo-Static Dataflow (CSDF) MoC is a generalization of the SDF MoC [BELP96], and a specialization of the DPN MoC, defined as follows:

Definition 2.3.4. *A Cyclo-Static Dataflow (CSDF) graph is a graph $G = \langle A, F \rangle$ respecting the Synchronous Dataflow (SDF) MoC with the following additions:*

- *Each port $p \in P^{in}_{data} \cup P^{out}_{data}$, is associated to a sequence of static integers of size n noted $seq(p) \in \mathbb{N}^n$.*
- *Considering an actor $a \in A$ and a port $p \in P^{in}_{data} \cup P^{out}_{data}$, the firing rule (i.e. the number of available tokens) and the production/consumption rates of p for the ith firing of actor a is given by the $(i \bmod n + 1)^{th}$ element of $seq(p)$.*

Figure 2.6 illustrates the graphical elements associated to the semantics of the CSDF MoC and gives an example of CSDF graph.



(a) Semantics (b) Graph example

Figure 2.6: *Cyclo-Static Dataflow (CSDF) MoC*

The expressivity of the CSDF MoC is equivalent to the expressivity of the SDF MoC. In [PPL95], Parks et al. propose a method to transform a CSDF graph into an equivalent SDF graph. Using this transformation enables the use of all optimization and analysis techniques of the SDF MoC for applications modeled with a CSDF graph.

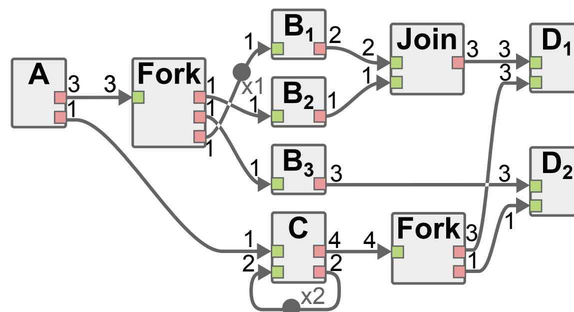


Figure 2.7: *SDF graph equivalent to the CSDF graph of Figure 2.6(b).*

The graph presented in Figure 2.7 results from the transformation of the CSDF graph of Figure 2.6(b) into its SDF equivalent. The SDF graph contains 6 additional actors and

8 additional FIFOs compared to the original CSDF graph. This example illustrates that, although the SDF and the CSDF MoCs have the same expressivity, the CSDF MoC has a better conciseness (Section 2.2.4).

The **Affine Dataflow (ADF)** MoC is a generalization of the CSDF MoC introduced by Bouakaz et al. in [BTV12]. In ADF, each port is associated to a finite initialization sequence in addition to the periodic sequence of rates. The sequence formed by the two sequences is called an ultimately periodic sequence. Hence, the firings of an actor can be separated in two successive phases: first, an initialization phase where the actor uses the rates defined in the initialization sequence, and then a periodic phase where the actor behaves as a CSDF actor. ADF has the same expressivity as SDF and CSDF, but offers a better conciseness than both.

2.4 Hierarchical SDF Modeling

Modularity in programming languages is an important feature that allows developers to specify the different elements that **compose** a system separately. Modular MoCs favor:

- **Composability:** Application and system descriptions consist of a set of independent components and a set of relationships between these components. Independent components of a composable system can, in general, be developed and compiled separately. In imperative programming languages, composability enables the incremental compiling of source code.

- **Dependability:** Modeling a system with several small components, instead of a unique large system, makes it easier to maintain. Indeed, each independent component can be tested separately and, if a problem arises, only the defective component will require fixing.

- **Reusability:** Similarly to classes in object-oriented programming, components of a modular MoC can be instantiated multiple times in a description. Components can also be reused in several applications, like DSP primitive building blocks (Fast Fourier Transform (FFT), Finite Impulse Response (FIR) filter, ...). Hence, several applications may benefit from the optimization of a single component.

Dataflow MoCs are inherently modular MoCs, since each actor is a clearly delineated module with its own behavior. In SDF, however, modularity is not part of the semantics as the internal behavior of actors is not part of the MoC. Several generalizations of the SDF MoC have been proposed [PBR09, TBG+13, LM87b] to make modularity an explicit part of the SDF MoC. These extensions of the SDF MoC rely on a **hierarchy mechanism** that enables the specification of the internal behavior of actors with a dataflow graph instead of host code.

2.4.1 Naive Hierarchy Mechanism for SDF

The first *hierarchy mechanism* for SDF graphs, introduced in [LM87b], simply consists of associating a hierarchical actor with an SDF subgraph in the development environment. When executing the application, the hierarchical actor is replaced with its content. Figure 2.8 shows an example of graph implementing this hierarchy mechanism. The *top-level* graph contains regular actors A and B and a hierarchical actor h. The subgraph describing the internal behavior of actor h contains 2 actors C and D.

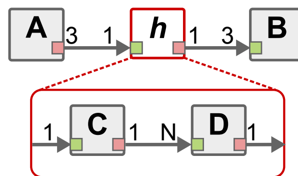
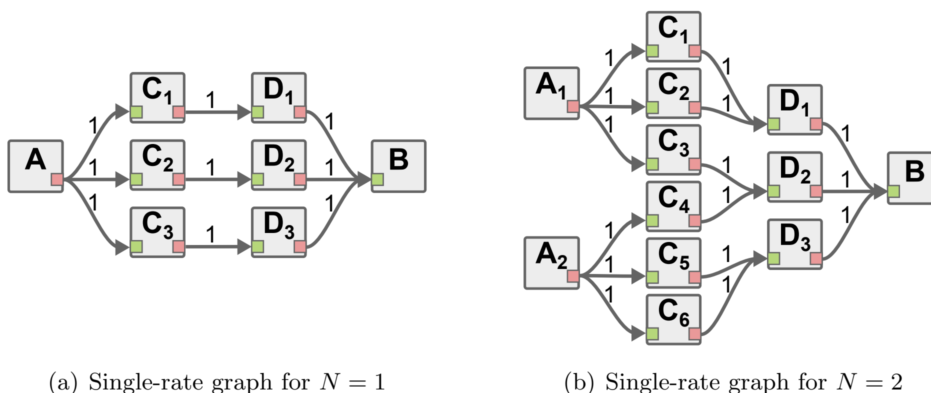


Figure 2.8: *Hierarchical SDF graph*

As presented in [Pia10, TBG+13], the main issue with this naive hierarchy mechanism is that the resulting MoC is not compositional as the properties of a graph depend on the internal specification of its components. To illustrate this issue, Figure 2.9(a) and 2.9(b) show the single-rate SDF graphs corresponding to the execution of the hierarchical SDF graph of Figure 2.8 with N, the consumption rate of actor D, set to 1 and 2 respectively. For clarity, $Fork$ and $Join$ actors are omitted in these single-rate graphs.



(a) Single-rate graph for $N = 1$ (b) Single-rate graph for $N = 2$

Figure 2.9: *Non-compositionality of naive hierarchical SDF MoC*

With $N = 1$, as expected with the production and consumption rates of actor h, actors A and B are each fired once per iteration, and actors C and D are fired three times each, once for each “firing” of actor h. With $N = 2$, the topology of the single-rate SDF graph is completely changed: in order to provide enough data tokens for the execution of actor B, actors D, C, and A must be fired 3, 6, and 2 times respectively. Hence, a modification in the graph associated to a hierarchical actor induces a modification of the RV of the graph containing this hierarchical actor, which is a non-compositional behavior.

Because of this non-compositional behavior, all hierarchical actors of a hierarchical SDF graph must be flattened (i.e. replaced with their content) before checking the consistency and schedulability of an application. Flattening the hierarchy may result in the creation of a graph with a large number of actors, which will require heavy computations to prove its schedulability.

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

The Interface-Based SDF (IBSDF) MoC [PBR09] is a compositional generalization of the SDF MoC defined as follows:

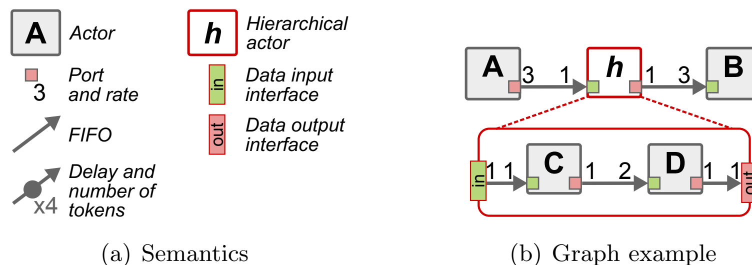
Definition 2.4.1. *An Interface-Based SDF (IBSDF) graph $G = \langle A, F, \mathbf{I} \rangle$ is a graph respecting the SDF MoC with the following additional elements of semantics:*

- *A **hierarchical actor** $a \in A$ is an SDF actor whose internal behavior is specified with an IBSDF graph, called the **subgraph** of actor a.*

- **I** $= (I_{data}^{in}, I_{data}^{out})$ *is a set of hierarchical interfaces. An interface* $i \in I$ *is a vertex of a subgraph. Interfaces enable the transmission of information between levels of hierarchy. Each interface* $i \in I$ *corresponds to a data port* $p \in P_{data}$ *of the enclosing hierarchical actor.*
  - *A* ***data input interface*** $i_{data}^{in} \in I_{data}^{in}$ *in a subgraph is a vertex transmitting to the subgraph the tokens received by its corresponding data input port. If more data tokens are consumed on a data input interface than the number of tokens received on the corresponding data input port, the data input interface behaves as a ring buffer, producing the same tokens several times. Formally, the* i^{th} *token produced by a data input interface is the* $(i \text{ mod } n + 1)^{th}$ *token consumed on the corresponding data input port, with* n *the consumption rate associated to this data input port*
  - *A* ***data output interface*** $i_{data}^{out} \in I_{data}^{out}$ *in a subgraph is a vertex transmitting tokens received from the subgraph to its corresponding data output port. If a data output interface receives too many tokens, it will transmit only the last received tokens to the upper level of hierarchy.*
  - *A subgraph of an actor* $a \in A$ *must be iterated until all data tokens produced by the data input interfaces have been read at least once, and all data tokens consumed by the data output interfaces have been written at least once.*

Contrary to the naive hierarchy mechanism presented in Section 2.4.1, the IBSDF MoC introduces new elements of semantics whose purposes are to insulate the different levels of hierarchy and guarantee the compositionality of the MoC. As proved in [PBR09], a necessary and sufficient condition to check the schedulability and consistency of an IBSDF graph is to check the schedulability of each subgraph separately. Hence, contrary to the naive hierarchy mechanism, a complete flattening of the hierarchy is not needed to prove the schedulability of an application.

An example of IBSDF graph and an illustration of the graphical elements associated to the semantics of IBSDF are given in Figure 2.10.



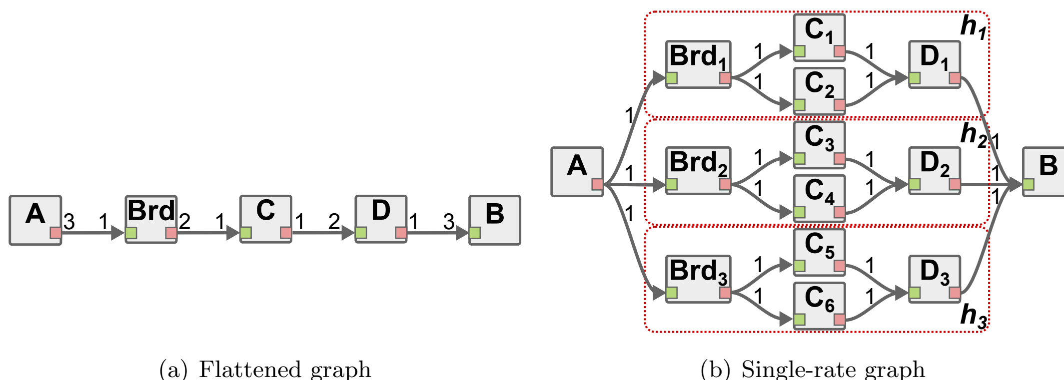
(a) Semantics (b) Graph example

Figure 2.10: *Interface-Based SDF (IBSDF) MoC*

As presented in Definition 2.4.1, the data input and output hierarchical interfaces of the IBSDF MoC have a special behavior if the number of data tokens exchanged in the subgraph is greater than the rate of the corresponding data port of the hierarchical actor. For example, in Figure 2.10(b), 1 firing of actor D and 2 firings of actor C are needed to produce one data token on the data output interface. Since only one data token is consumed on the data input port of actor h, this data token will be produced twice by the corresponding data input interface: once for each firing of actor C.

Hierarchy Flattening

Flattening the hierarchy of an IBSDF graph consists of replacing hierarchical actors with their subgraph and, when needed, replacing data interfaces with actors implementing their special behavior. As presented in [PBPR09], *Broadcast* (or *Brd*) actors implement the special behavior of data input interfaces that produce several times the same data tokens and *Roundbuffer* (or *Rnd*) actors implement the special behavior of data output interfaces that only transmit the last data tokens received to their output FIFO.



(a) Flattened graph (b) Single-rate graph

Figure 2.11: *Flattening and single-rate transformations of the IBSDF graph of Figure 2.10(b)*

Figure 2.11(a) and 2.11(b), respectively, present the flattened and the single-rate graphs derived from the IBSDF graph of Figure 2.10(b). In the flattened graph, the data input interface of the subgraph was replaced with a *Broadcast* actor that duplicates twice each data token produced by actor A. The number of data tokens produced by actor D in the subgraph is equal to the number of data tokens produced by the hierarchical actor h. Consequently, since the data output interface does not discard any data token, a *Roundbuffer* actor is not needed, and actor D can be directly connected to actor B.

In the single-rate graph of Figure 2.11(b), the firings of actors Brd, C, and D are separated in three identical groups, each corresponding to a distinct firing of the hierarchical actor h.

By flattening a selected set of hierarchical IBSDF actors, the developer of an application can easily change the *granularity* of an application without modifying its functional behavior. For example, if actor h had not been flattened in Figure 2.11(b), the single-rate graph would only contain 5 actors instead of 14. In such a case, the subgraph of non-flattened actors can be translated into *host language* using code generation techniques presented in [PBPR09].

The memory characteristics of applications modeled with the IBSDF MoC are studied in Chapters 4 to 6 of this thesis, and a reconfigurable generalization of the IBSDF MoC is introduced in Chapter 7.

2.4.3 Deterministic SDF with Shared FIFOs

In [TBG+13], Tripakis et al. propose another hierarchical and compositional generalization of the SDF MoC called Deterministic SDF with Shared FIFOs (DSSF). The main difference between the DSSF and the IBSDF MoCs is that in DSSF, a set of acceptable production and consumption rates on the data ports of a hierarchical actor is inherited from the topology of its subgraph (bottom-up approach) instead of being fixed by the interfaces (top-down approach), as in IBSDF. A second difference between the two models is that

in DSSF, firing a hierarchical actor does not necessarily imply a complete iteration of its subgraph. For example, considering actor h of Figure 2.10(b) as a DSSF actor with inherited consumption and production rates of 2 and 1, its single-rate equivalent is the single-rate graph of Figure 2.9(b). This single-rate graph corresponds to three firings of actor h, consisting of actors (C_1, C_2, D_1), actors (C_3, C_4, D_2), and actors (C_5, C_6, D_3) respectively.

2.5 Dynamic Dataflow Models of Computation

As presented in Section 2.3, static dataflow MoCs have a limited expressivity and cannot be used to model all applications. In particular, the sequence of firing rates of each actor is known at compile time in static MoCs, which lends static MoCs a total predictability and a great analyzability.

Many generalizations of the SDF MoC have been introduced over the years to enable the description of dynamically reconfigurable applications. While improving the expressivity of the static MoC, the purpose of these generalizations is to preserve as much predictability and analyzability as possible.

In [NL04], Neuendorffer an Lee propose a unified model to analyze the trade-off between reconfigurability and predictability of a dataflow MoC. In particular, they show that the predictability of a MoC can be characterized by allowing reconfiguration only at certain points, called quiescent points, during the execution of applications. Next sections present the PSDF, SPDF, and SADF dynamic dataflow MoCs.

2.5.1 Parameterized SDF (PSDF)

Parameterized dataflow is a meta-modeling framework introduced by Bhattacharya and Bhattacharyya in [BB01]. In the context of dataflow MoCs, a *meta-model* is a set of elements that can be added to the semantics of an existing MoC in order to bring new capabilities to this MoC.

The parameterized dataflow meta-model is applicable to all dataflow MoCs that present graph iterations. When this meta-model is applied, it extends the targeted MoC semantics by adding dynamically reconfigurable hierarchical actors. Examples of applications of the meta-model to the SDF and the CSDF MoCs can be found in [BB01] and [KSB+12] respectively. Parameterized dataflow is formally defined as follows:

Definition 2.5.1. *The **parameterized dataflow** meta-model extends the semantics of a targeted dataflow MoC with the following elements:*

- **param(a)** *is a set of parameters associated to an actor $a \in A$. A parameter $p \in param(a)$ is an integer value that can be used as a production or consumption rate for actor a, and that can influence the internal behavior of actor a. The value of parameters is not defined at compile time but instead is assigned at run time by another actor. Optionally, a parameter can be restricted to take values only in a finite domain noted $domain(p)$.*

- ***Hierarchy levels**, including subgraphs of **hierarchical actors**, are specified with 3 subgraphs, namely the init ϕ_i, the subinit ϕ_s, and the body ϕ_b subgraphs.*

  - *the $\boldsymbol{\phi_i}$ subgraph sets parameter values that can influence both the production and consumption rates on the ports of the hierarchical actor and the topology of the ϕ_s and ϕ_b subgraphs. The ϕ_i subgraph is executed only once per iteration*

of the graph to which its hierarchical actor belongs and can neither produce nor consume data tokens.

– *the* $\boldsymbol{\phi_s}$ *subgraph sets the remaining parameter values required to completely configure the topology of the* ϕ_b *subgraph. The* ϕ_s *subgraph is executed at the beginning of each firing of the hierarchical actor. It can consume data tokens on input ports of the hierarchical actor but can not produce data tokens.*

– *the* $\boldsymbol{\phi_b}$ *subgraph is executed when its configuration is complete, right after the completion of* ϕ_s*. The body subgraph behaves as any graph implemented with the MoC to which the* parameterized dataflow *meta-model was applied.*

In parameterized dataflow, a reconfiguration occurs when values are dynamically assigned to the parameters of an actor, causing changes in the actor computation and in the production and consumption rates of its data ports.

Parameterized SDF (PSDF) is the MoC obtained by applying the *parameterized dataflow* meta-model to the SDF MoC. Examples of real DSP applications described with the PSDF MoC can be found in [KSB+12, PAPN12].

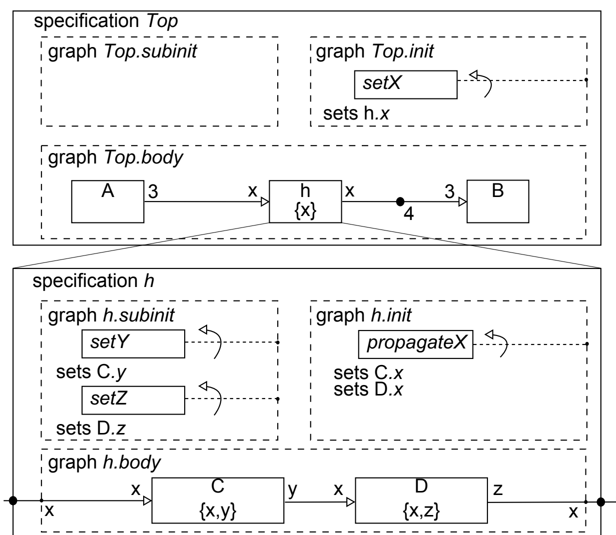


Figure 2.12: *Example of Parameterized SDF (PSDF) graph*

Figure 2.12 presents an example of PSDF graph using the graphical semantics proposed in [BB01]. In this example, the *top level* specification contains 4 actors. The *setX* actor, contained in the *Top.init* subgraph, assigns a value to parameter x, thus influencing the dataflow behavior of actor h in the *Top.body* subgraph. Actor h is a hierarchical actor whose subgraphs contain 5 actors. A parameter set in a level of hierarchy cannot influence directly parameters in body subgraphs of lower levels of hierarchy. For example, the value assigned to parameter x in the *Top.init* subgraph must be explicitly propagated by actor *propagateX* in subgraph *h.init* in order to be used in the body subgraph of actor h.

Runtime Operational Semantics

The runtime operational semantics of the PSDF MoC, as presented in [BB01], defines the successive steps followed during the execution of a hierarchical actor $a \in A$ whose internal behavior is specified with 3 subgraphs. The execution of actor a restarts to step 1 each time actor a is *instantiated* in its parent graph (cf. steps 2, 6, 9). Here, the *instantiate*

operation can be seen as a signal sent to an actor, triggering its initialization process and other operations such as reservation of memory for its execution, or retrieval of code to execute. The execution of actor a can be decomposed into the following steps:

1. Wait for actor a to be *instantiated* in its parent graph.
2. *Instantiate* all actors in the init subgraph $a.\phi_i$.
3. Fire all actors in the init subgraph $a.\phi_i$.
4. Compute the Repetition Vector (RV) of the subinit subgraph $a.\phi_s$ and pre-compute the RV of the body subgraph $a.\phi_b$. Parameter values used in this step are the values set in step 3 and default values for parameters whose values will be set in step 7. The computed RV is used to determine the production and consumption rates of actor a in its parent graph.
5. Wait for the next firing of actor a.
6. *Instantiate* all actors in the subinit subgraph $a.\phi_s$.
7. Fire all actors in the subinit subgraph $a.\phi_s$.
8. Compute the RV of the body subgraph $a.\phi_b$ with parameter values set in steps 3 and 7.
9. *Instantiate* all actors in the body subgraph $a.\phi_b$.
10. Fire all actors in the body subgraph $a.\phi_b$.
11. Go back to step 5.

Following this operational semantics, the actors of the PSDF graph of Figure 2.12 are fired in the following order, assuming that parameters x, y and z are always set to 1: $\{$*SetX, propagateX, A, 3×(SetY, SetZ, C, D), B*$\}$. Notation "3×(· · ·)" means that the content of the parenthesis is executed 3 times.

As presented in [SGTB11, NL04], this operational semantics makes the PSDF MoC less predictable than the SDF MoC, but more predictable than the DPN MoC. Indeed, contrary to the topology of SDF graphs, the topology of a PSDF graph is unknown at compile time, as it depends on dynamically set parameter values. However, this topology is fixed as soon as actors of the subinit subgraph are executed, and remains constant over several iterations of the body subgraph. Consequently, firing rates of actors are more stable than in the DPN MoC where the firing rates can non-deterministically change at each actor firing.

PSDF Analyzability

In *parameterized dataflow*, the schedulability of a graph can be guaranteed at compile time for certain applications by checking their *local synchrony* [BB01]. A PSDF (sub)graph is locally synchronous if it is schedulable for all reachable configurations and if all its hierarchical children are locally synchronous. As presented in [BB01], a PSDF hierarchical actor must satisfy the 5 following conditions in order to be locally synchronous:

1. ϕ_i, ϕ_s and ϕ_b must be locally synchronous, i.e. they must be schedulable for all reachable configurations.

2. Each invocation of ϕ_i must give a unique value to each parameter set by this subgraph.
3. Each invocation of ϕ_s must give a unique value to each parameter set by this subgraph.
4. Consumption rates of ϕ_s on interfaces of the hierarchical actor cannot depend on parameters set by ϕ_s.
5. Production/consumption rates of ϕ_b on interfaces of the hierarchical actor cannot depend on parameters set by ϕ_s.

The first condition cannot always be verified at compile time because it requires a formal analysis of all reachable configurations. Indeed, using numerous parameters with large domains of values will lead to an exponential number of reachable configurations, and testing all of them in reasonable time may not be possible. Nevertheless, failing to check the *local synchrony* of a PSDF graph at compile time does not mean that the graph is not schedulable. In such a case, the *consistency* and the *schedulability* of a PSDF graph will be checked in steps 4 and 8 of the operational semantics. In these steps, values of all parameters have been dynamically assigned, and the PSDF graph becomes an SDF graph whose *schedulability* can be checked using Theorem 2.3.2.

In Chapter 7, a novel integration of the PSDF and the IBSDF MoCs is introduced, with an enhanced conciseness and a new explicit parameterization mechanism.

2.5.2 Schedulable Parametric Dataflow (SPDF)

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

Definition 2.5.2. *A Schedulable Parametric Dataflow (SPDF) graph $G = \langle A, F, P \rangle$ is a graph respecting the SDF MoC with the following additional elements of semantics:*

- **P** *is a set of parameters. The value of a parameter $p \in P$ can influence the production and consumption rates of the actors of the graph. A parameter is defined as a tuple $p = \langle M, \alpha \rangle$ where:*
  - **M** $: P \to A$ *associates a* modifier *actor to a parameter. The* modifier *actor of a parameter is the only actor of the graph that can, when it fires, assign a new value to the parameter.*
  - $\boldsymbol{\alpha} : P \to \mathbb{N}$ *associates a* change period *to a parameter. The* change period *of a parameter is an integer that defines the number of firings of the modifier actor between two changes of the parameter value. The change period may itself depend on the value of a parameter.*

An example of SPDF graph from [FGP12], is shown in Figure 2.13. The $p[1]$ graphical notation within actor A means that this actor is the *modifier* actor for parameter p. The number between the square brackets depicts the *change period* associated to the parameter. As illustrated by actor B, the modifier actor for parameter q, the change period of a parameter can itself depend on the value of another parameter.

As illustrated in the SPDF graph of Figure 2.13, a modifier actor can, like actor A, change its firing rate at each firing. As shown in [BL93], adding this capability to the

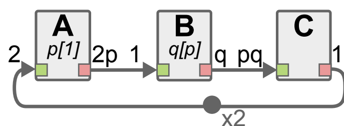


Figure 2.13: *Example of Schedulable Parametric Dataflow (SPDF) graph*

SDF MoC is a sufficient condition to make SPDF a Turing-complete model, and thus of equivalent expressivity with the DPN MoC.

To enforce the *analyzability* of the SPDF MoC, a *safety criterion* is also introduced in [FGP12]. In order to satisfy this safety criterion, the modifier actor must not change the value of a parameter during the iteration of a part of a graph where this parameter is used. For example, in Figure 2.13, parameter q is used in the part of the graph formed by actors B and C. According to the production and consumption rates of actors B and C, actor C is fired once every p firings of actor B. Consequently, the value of parameter q must remain constant over p firings of actor B, as is the case in this example. The consistency and the schedulability of SPDF graphs meeting this criterion can be checked at compile time.

The addition of parameters to the semantics of the SPDF MoC gives SPDF a better predictability than the DPN MoC. Indeed, the production and consumption rates of SPDF actors that are not modifier actors are fixed before the firing of the actor. For example, in Figure 2.13, the consumption rate of actor C is fixed as soon as actor B sets the value of parameter q. Since actor C is fired once every p firings of actor B, time for $p-1$ firings of actor B passes between the resolution of the consumption rate of actor C and its firing. This example illustrates the predictability of the SPDF MoC where production and consumption rates may be known some time before the actual firing of an actor. This predictability is in contrast with the DPN MoC where each actor sets it own consumption and production rates when it fires.

2.5.3 Scenario Aware Dataflow (SADF)

The Scenario-Aware Dataflow (SADF) MoC [TGB+06] is a reconfigurable generalization of the SDF MoC. Like the SPDF MoC, it has an equivalent expressivity with the DPN MoC. The semantics of the SADF MoC, defined hereafter, is designed to enforce the *analyzability* of applications.

Definition 2.5.3. *A Scenario-Aware Dataflow (SADF) graph $G = \langle A, F\rangle$ is a graph respecting the semantics of the SDF MoC with the following additions:*

- *Actors $a \in A$ are associated with a non-empty finite set of* ***scenarios*** *S_a. For each actor, a unique scenario $s \in S_a$ is active for each firing. The active scenario determines the production and consumption rates on the ports of the actor as well as the* ***execution time*** *t of the actor.*

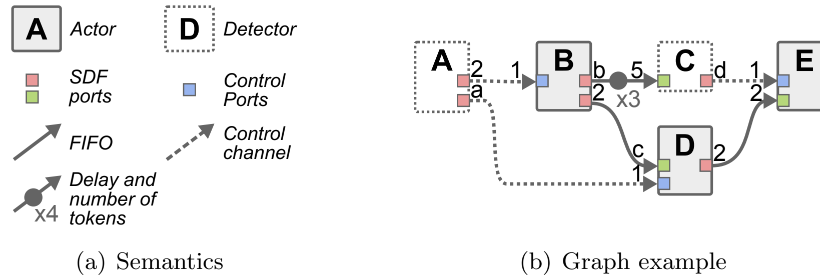
- *Actors $a \in A$ are associated with a possibly empty set of* ***control input ports****. Before firing an actor, a single* ***control token*** *is consumed from each control input port. The consumed control tokens are used to determine the scenario of the actor for the next firing.*

- **D** $\subset A$ *is a set of special actors called detectors. Each detector $d \in D$ is associated with a Markov chain. The* scenario *of a detector changes depending on the current state of the Markov chain. Detectors are the only actors that can write* control

tokens *on an output port. The value and the number of* control tokens *produced by a detector solely depend on the current state of its Markov chain.*

- **C** $\in F$ *is the set of special* FIFOs *called* ***control channels****. A control channel* $c \in C$ *is used to transmit* control tokens *from a* detector *to another actor or detector of the graph.*

Figure 2.14 presents the graphical elements associated to the semantics of the SADF MoC and an example of SADF graph.



(a) Semantics (b) Graph example

Figure 2.14: *Scenario-Aware Dataflow (SADF) MoC*

The SADF graph of Figure 2.14(b) contains 2 detectors (A and C), and 3 "regular" actors (B, D, and E). In this graph, production and consumption rates written with numbers are statically fixed rates whereas rates written with letters depend on the scenario of their actor. As required by Definition 2.5.3, the consumption rate of all control ports is statically set to 1.

The stochastic process used in SADF to determine the production and consumption rates and the execution time of actors has been shown to give a great *analyzability* to the MoC. Indeed, beside proving the consistency or the schedulability of an SADF graph [TGB+06], methods exist to derive useful metrics for real-time applications such as the worst-case latency, or the long-time average throughput of an application modeled with an SADF graph [SGTB11].

Although Markov chain of SADF lends a great analyzability to the MoC, this stochastic process is not practical for describing the functional behavior of applications. For this reason, an executable FSM-based SADF MoC is introduced in [SGTB11]. In the FSM-based SADF MoC, the Markov chains associated to the detectors of the MoC are replaced with deterministic FSMs.

2.6 Summary of Presented Dataflow MoCs

Table 2.1 summarizes the properties of the main dataflow MoCs presented in this chapter. A black dot indicates that the feature is implemented by a MoC, an absence of dot means that the feature is not implemented, and an empty dot indicates that the feature may be available for some applications described with this MoC. Vertical lines in this table, are used to separate the dataflow MoCs in groups of equivalent expressivity.

As presented in Section 2.2.4, the compositionality property is an important property when working with hierarchical dataflow MoCs. Table 2.1 reveals the lack of compositional dataflow MoC with a greater expressivity than the SDF MoC. To fill this gap, a new dataflow meta-model is introduced in Chapter 7, bringing reconfigurability, hierarchy, and compositionality properties to any dataflow MoC with a well defined concept of graph iteration.

Feature	SDF	CSDF	ADF	IBSDF	DSSF	PSDF	SADF	SPDF	DPN
Expressivity	low					med.	Turing		
Hierarchical				•	•	•			
Compositional				•	•				
Reconfigurable						•	•	•	•
Statically schedulable	•	•	•	•	•				
Decidability	•	•	•	•	•	○	•	○	
Variable rates		•	•			•	•	•	•
Non-determinism							•	•	•

Table 2.1: *Features comparison of presented dataflow MoCs*

Next chapter introduces the rapid prototyping context of this thesis and presents previous work on the dataflow-related issues studied in this thesis.

CHAPTER 3

Rapid Prototyping Context

3.1 Introduction

The memory analysis and optimization techniques for dataflow graphs presented in this thesis were developed as part of a rapid prototyping framework. Rapid prototyping consists of models and methodologies that give developers the possibility to quickly co-design and validate a hardware/software system. The purpose of rapid prototyping is to enable the creation of a simulated or a working prototype in early stages of development in order to assess the feasibility of a system with regards to design constraints.

The concepts and tasks involved in the rapid prototyping of a system are detailed in Section 3.2. Section 3.3 presents the rapid prototyping framework within which the contributions of this thesis were developed. Section 3.3 also compares the rapid prototyping framework used in this thesis with other State-of-the-Art dataflow programming environments. Then, Section 3.4 presents related work on the topic of memory optimization for applications modeled with dataflow graphs. Finally, Section 3.5 concludes the Background part of this thesis.

3.2 What is Rapid Prototyping?

As presented by Cooling and Hughes in [CH89], rapid prototyping in computer science relies on two pillars: models to describe the behavior and the requirements of systems, and automatic methods and tools to quickly generate system simulations or system prototypes from the system models.

Figure 3.1 presents an overview of a typical rapid prototyping design flow. This design flow can be separated in 3 parts:

- **Developer Inputs**: Developer inputs consist of high-level models that enable the specification of all important properties of a system. In the co-design context, where designed systems have both hardware and software parts, developer inputs often gather a model of the application (Section 3.2.2), a model of the targeted architecture (Section 3.2.1), and a set of constraints for the deployment of the application on the architecture (Section 3.2.3). As presented in the Algorithm-Architecture Adequation (AAA) methodology [GS03], the separation between the three inputs of the design

flow ensures the independence between them, which eases the deployment of an application on several architectures or the use of a single architecture to deploy several applications.

- **Rapid Prototyping**: The rapid prototyping part of the design flow regroups the tasks that are executed to automatically explore the design space and to generate a prototype of the system described in the developer inputs. An important characteristic of these tasks is the rapidity with which they can be executed, even for complex applications and architectures. Contrary to a classic design flow, the purpose of a rapid prototyping design flow is not to generate an optimal solution, but to rapidly assess the feasibility of a system by generating a functional prototype that respects the specified constraints. Ideally, the obtained prototype will be refined and optimized in later stages of development.

- **Legacy Development Toolchain**: Optionally, the prototype generated by the design flow may be executed on a real target. In such a case, the generation of the executable is supported by legacy development toolchains associated to the target. During the execution of the generated prototype, monitoring is generally used to record and characterize the system behavior in order to provide feedback to the rapid prototyping design flow and to the developer.

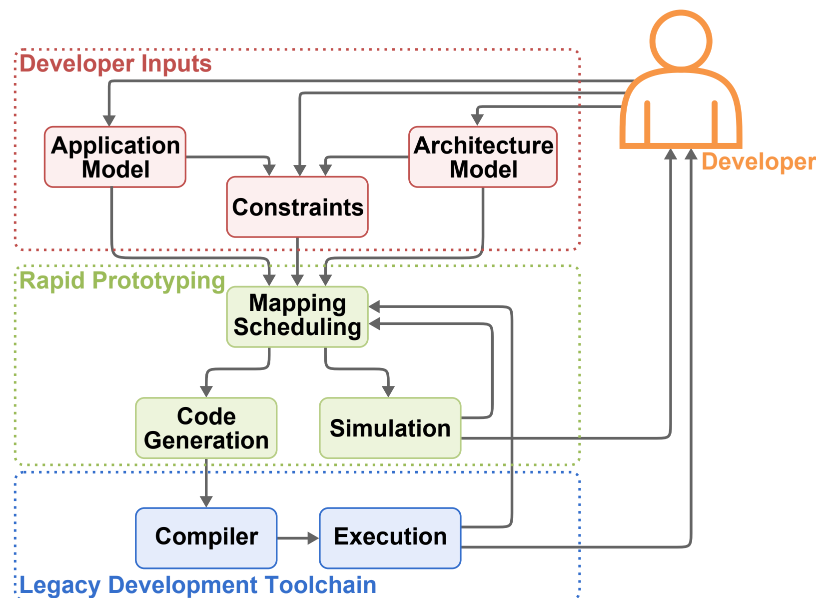


Figure 3.1: *Overview of a rapid prototyping design flow.*

As illustrated in Figure 3.1, the rapid prototyping design flow is an iterative process that can use feedback from the simulation and the execution of the generated prototype to improve its quality. More importantly, the simulation or the execution of the generated prototype gives valuable information to the developer to guide the evolution of the design flow inputs. For example, this feedback can reveal resource deficiency of the architecture model or the presence of contradictory constraints.

The following sections detail challenges behind the different elements of the rapid prototyping design flow and surveys existing solutions from the literature.

3.2.1 Heterogeneous Architecture Modeling

In recent years, following Moore's law [Moo65], the ever-increasing number of transistors in integrated circuits has lead to the introduction of more and more complex architectures. The heterogeneity of modern architectures, that assemble a wide variety of complex components into a single chip, makes their accurate modeling and simulation a tiresome task.

In rapid prototyping, the purpose of the architecture model is to provide a system-level description of the targeted architecture system [PNP+09]. A system-level model describes the components of an architecture and the relation between them with a limited accuracy compared to lower-level hardware descriptions. Hence, the components described in a system-level model mainly consist of *processing elements* and *communication channels*. *Processing elements* are the processors and accelerators performing the computation of the system, and *communication channels* are the buses, hardware queues, Direct Memory Accesses (DMAs), and shared memories that enable data transmissions from a processing element to another. A counterpart for the limited accuracy of system-level models is the possibility to quickly simulate the behavior of a described architecture.

As presented by Blake et al. in [BDM09], state-of-the-art multiprocessor architectures can be classified using 5 attributes: application domain, power/performance ratio, types of processing elements, memory hierarchy, and on-chip accelerators. Since these attributes can be captured by the components of system-level models, these models can successfully be used to describe state-of-the-art architectures, including:

- **Central Processing Units (CPUs)**. They are general purpose architectures that can be found in most desktop computers and servers. These architectures are characterized by their small number of homogeneous processing elements (1 to 8) organized around a shared memory. The cache-coherency mechanism, and the numerous accelerators and peripherals embedded on CPUs results in a poor power/performance ratio compared to other architectures. *Intel's i7-3610* [Int13] processor is an example of commercial CPU.

- **Graphics Processing Units (GPUs)**. They were originally dedicated to the rendering of 2D and 3D graphics, but have also been used for general purpose applications in recent years. These architectures are characterized by their large number of homogeneous processing elements (512 to 8192) organized in clusters with a global memory address space. Processing elements of a GPU have a Single Instruction, Multiple Data (SIMD) behavior [Fly72], which means that all processing elements of a cluster execute the same operation simultaneously with different inputs. Although the power/performance ratio of GPUs is better than the one of CPUs, their SIMD behavior restricts the range of suitable applications.

- **Many-core architectures**. They are massively parallel architectures characterized by their large number of homogeneous processing elements ($\geq$64). Contrary to CPUs and GPUs, the memory of many-core architectures is distributed and each processing element accesses a private address space. Hence, many-core architectures have a Multiple Instructions, Multiple Data (MIMD) behavior [Fly72], where all processing elements concurrently execute different operations on different inputs. Many-core architecture have a good power/performance ratio which makes them interesting architectures for embedded applications that involve heavy computations. *Kalray's MPPA256* [Kal14], *Tilera's Tile-Gx72* [Til14], and *Adapteva's Epipany-IV* [Ada14] are examples of commercial many-core processors.

- **Heterogeneous Multiprocessor Systems-on-Chips (MPSoCs).** They are embedded systems designed to support specific applications with performance, power, or size constraints. Heterogeneous MPSoCs combine different types of processing elements and accelerators on a single chip, with both shared and distributed memories. For example, *Texas Instrument's 66AK2H14* MPSoC [Tex14] combines a quad-core ARM processor, 8 high-performance DSP cores, and several hardware accelerators.

The work presented in this thesis focuses on the memory study of applications implemented on CPUs and heterogeneous MPSoCs with shared memory. Figure 3.2 presents a simple example of architecture model that will be used in the following sections to illustrate the different parts of the rapid prototyping process. This heterogeneous architecture consists of a shared memory connecting 3 processing elements, a CPU, a DSP processor, and a hardware accelerator.

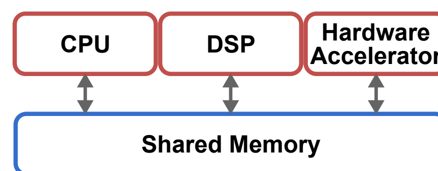


Figure 3.2: *Example of heterogeneous architecture model*

3.2.2 Parallel Application Modeling

The application model used as an input for a rapid prototyping design flow is a *coarse-grain* description of the application: a description where each atomic element represents an important amount of computation. A key property of application descriptions is their independence from any architectural consideration. Beside giving the possibility to deploy an application on several targets, the independence of the application model from architectural information is also the most important degree of freedom exploited by the rapid prototyping in its design space exploration tasks (Section 3.2.4).

Like the architecture model, a coarse-grain application model is preferable to ease the rapid prototyping process. Indeed, the fine granularity of low-level application models makes them more difficult to analyze and optimize, thus requiring substantially more time for the automated design space exploration of the rapid prototyping. Using coarse-grain models allows the application developer to specify high-level software components, such as actors, that are seen as indivisible elements from the rapid prototyping perspective. These coarse-grain software elements considerably ease the automated design space exploration compared to the fine-grain elements of low-level models. Indeed, analyzing data-dependency in imperative languages in order to identify the independent parts of an application will take much more time than exploiting explicitly independent actors of a dataflow graph.

As presented in Section 3.2.1, the trend in modern architectures is to increase the number of processing elements working concurrently. Hence, to fully exploit the computing power offered by these architectures, models must explicitly express the parallelism of applications. As presented in Section 2.2.3, dataflow MoCs are well suited for this purpose. There exist other parallel programming models that can be used to exploit the parallelism offered by multiprocessor architectures, including:

- **Threads**: An application modeled with threads consists of several sequences of instructions executed concurrently, each with a dedicated context. The main drawback

of threads is that they require the programmer to specify all synchronization points between threads explicitly, using mechanisms such as semaphores or mutexes. A number of threads close to the number of processing elements of the targeted architecture is usually preferable to avoid excessive context switching, i.e. to keep as low as possible the number of times the registers related to a thread execution need to be stored in memory.

- **Tasks**: An application modeled with tasks also consists of several sequences of instructions executed concurrently. Contrary to threads, tasks are non-preemptible, which means that once started, their execution cannot be interrupted. The granularity of tasks is much finer than threads and hundreds of light-weight tasks are used for the description of an application [Rei07].

- **Semi-automated parallelization**: This approach consists of adding `pragma` directives to the source code of an application to indicate loops or sections of code that can be parallelized. These directives can then be automatically analyzed in order to generate corresponding tasks or threads [CJVDP08]. Because it requires an analysis of the source code of an application, this parallel programming method is unsuitable for rapid prototyping.

The work presented in this thesis focuses on the rapid prototyping of applications modeled with dataflow MoCs. Figure 3.3 presents a simple example of application model that will be used in following sections to illustrate the different parts of the rapid prototyping process. This homogeneous SDF graph contains 6 actors linked by a network of single-rate FIFOs.

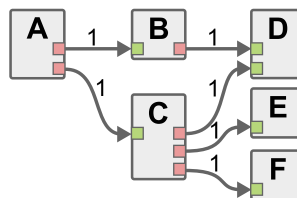


Figure 3.3: *Example of application model: a homogeneous SDF graph.*

3.2.3 Application Deployment Constraints

In addition to the independent application and architecture models, the user of a rapid prototyping design flow can also describe a set of specific constraints for the deployment of an application on a given architecture.

The first objective of these constraints is to model the cost of the design choices made by the rapid prototyping process. For example, the cost of mapping an actor of a dataflow application on a given processing element can be expressed in terms of execution time or power consumption. Evaluating the cost of design choices allows the rapid prototyping process to automatically optimize the overall cost of the generated prototype.

Tables 3.1(a) and 3.1(b) give, respectively, the execution time and energy consumption costs for executing actors of the application presented in Figure 3.3 on the processing elements of the architecture presented in Figure 3.2.

The second objective of the deployment constraints is to define requirements that the generated prototype must satisfy. Common requirements are:

Actor	CPU	DSP	Acc.
A	2	3	-
B	4	8	-
C	5	2	-
D	2	1	-
E	3	4	-
F	10	5	1

(a) Execution time (in seconds)

Actor	CPU	DSP	Acc.
A	2	1.5	-
B	4	4	-
C	5	1	-
D	2	0.5	-
E	3	2	-
F	10	2.5	0.1

(b) Energy consumption (in Joules)

Table 3.1: *Costs for the execution of actors from Figure 3.3 on processing elements from Figure 3.2.*

- **Mapping restrictions**: Some parts of the application can be executed only by a restricted set of processing elements of the architecture. For example, this may happen in heterogeneous architectures where IO modules can be accessed by a single processing element, or where floating point operations are not supported by all processing elements. For example, in Tables 3.1, no actor, except actor F, can be executed on the hardware accelerator, as denoted by the "-" in the accelerator column.

- **Real-time constraints**: The application must complete its processing within a limited amount of time. There are three types of real-time constraints: *latency* (total processing time), *throughput* (output rate), and *simultaneity* (synchronization between outputs or processes). For example, video decoding applications must process at least 25 Frames per second (fps) in order to be comfortably viewable by human beings.

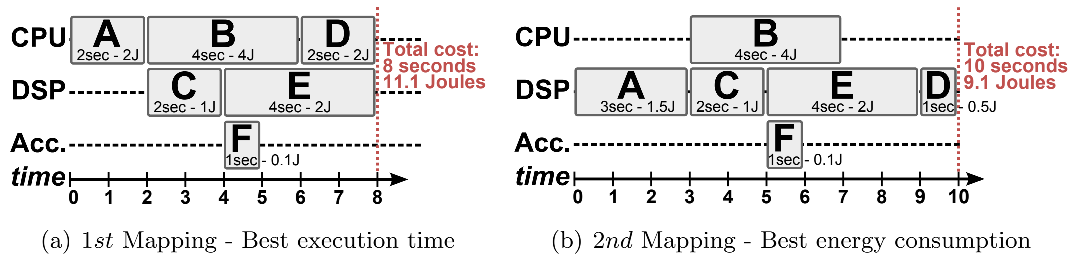
- **Power limitations**: The power consumption of the system is limited. For example, the power consumption of the application responsible for keeping a cellphone connected to the network must remain as low as possible in order to extend the battery life.

3.2.4 Mapping and Scheduling

Mapping an application on an architecture consists of assigning each part of the application to a specific processing element of the architecture. Hence, mapping a dataflow graph consists of selecting the processing elements that will execute each actor of the application. Scheduling (also called ordering) an application on an architecture consists of choosing the execution order of the different parts of the application mapped to a processing element.

As presented by Lee et al. in [LH89], the mapping and scheduling choices can be made at compile time or at runtime. Mapping and scheduling an application at runtime yields a great flexibility to the system, giving it the possibility to adapt its behavior to dynamically changing constraints, such as unpredictable actor execution time or external events. A counterpart of this dynamism is the need for a runtime management process that handles the mapping and scheduling choices dynamically. This runtime management process is executed concurrently with the application, which induces a performance overhead. Dynamic mapping and scheduling make the application performance hard to predict, because the external factors influencing the runtime management are unknown at compile time. For this reason, in the context of rapid prototyping, as much mapping and scheduling choices as possible should be shifted to compile time.

The objective of the mapping and scheduling processes is to find a solution that satisfies the constraints described by the developer (Section 3.2.3), while optimizing a tradeoff between other criteria such as application throughput, energy consumption of the system, or the balanced distribution of the computing load on the different processing elements [PAPN12]. Mapping and scheduling an application on a multiprocessor architecture under constraints has been showed to be an NP-complete problem [Bru07]. Nevertheless, many mapping and scheduling heuristic algorithms can be found in the literature [Kwo97, Bru07, ZDP+13].



(a) 1*st* Mapping - Best execution time (b) 2*nd* Mapping - Best energy consumption

Figure 3.4: *Gantt charts for 2 mappings of the application from Figure 3.3 on the architecture from Figure 3.2.*

Figure 3.4 presents two execution Gantt charts resulting from two different mappings of the application of Figure 3.3 on the architecture of Figure 3.2, using costs defined in Table 3.1. With a total execution time of 8 seconds, the first mapping presented in Figure 3.4(a) is a faster solution than the second mapping presented in Figure 3.4(b). However, from the energy consumption perspective, despite its longer execution time, the second mapping requires 2 joules less than the first. This simple example illustrates the tradeoff between the different system costs, and how user-defined constraints of the rapid prototyping process may influence the outcome of mapping and scheduling process.

3.2.5 Simulation

In the rapid prototyping context, the simulation process, or simulator, serves two purposes: predict the behavior of the generated prototype for the developer and evaluate the costs of the mapping and scheduling choices to improve them iteratively. The two most important characteristics of simulators are their accuracy and their speed of execution [PMAN09].

The accuracy of a simulator measures the difference between the system characteristics predicted by the simulation and the actual characteristics observed on the real system. The timing accuracy of a simulator is often qualified with one of the following terms [RK08]:

- **Functional**: A functional simulator only predicts the result produced by the simulated system by executing the same computation. Functional simulation is used by developers to quickly check the correct behavior of designed systems.

- **Approximately timed**: An approximately-timed simulator predicts the performance of a system based on simplified models of the architecture [PMAN09]. The advantage of approximately timed simulations is that they can be generated quickly, but the actual performance of the system may slightly differ from the performance predicted by the simulator.

- **Cycle accurate**: A cycle accurate simulator predicts the exact performance of the system. Cycle-accurate simulators are based on low-level models of the system

which make them significantly slower than other simulators. These simulators are often used when the hardware of the system is not available, and when the simulation results are used to prove that a safety critical system respects its real-time constraints.

These different degrees of accuracy also exist for other simulated characteristics such as the energy consumption [BASC+13] or the cache hit/miss ratio [RG13]. Approximately-timed simulators offer a good tradeoff between accuracy and speed of execution, which makes them good candidates for use in a rapid prototyping design flow. [PMAN09]

Next section presents the specificities of the rapid prototyping framework that served as a basis for the work presented in this thesis.

3.3 Preesm Rapid Prototyping Framework

The Parallel and Real-time Embedded Executives Scheduling Method (Preesm) is a rapid prototyping framework that provides methods to study the deployment of IBSDF applications onto heterogeneous multicore DSP systems [PDH+14]. Preesm is developed at the Institute of Electronics and Telecommunications of Rennes (IETR) as a set of open-source plugins for the Eclipse Integrated Development Environment (IDE) [IET14b].

Preesm is currently developed for research, development, and educational purposes. It has been successfully used for the rapid prototyping of real telecommunication, multimedia, DSP, and computer vision applications on several heterogeneous MPSoCs [PAPN12, HDN+12, Zha13].

After a presentation of related work on dataflow programming tools in Section 3.3.1, a typical workflow of Preesm is detailed in Section 3.3.2. Then, the graphical editors, the code generation, and the import/export features of Preesm are presented in Section 3.3.3

3.3.1 Related Work on Dataflow Programming

The creation of the Preesm rapid prototyping framework has been inspired by the Algorithm-Architecture Adequation (AAA) methodology [GS03]. AAA consists of simultaneously searching the best software and hardware configurations for respecting system constraints. The **SynDEx** tool [GLS99] is also based on the AAA methodology but it differs from Preesm in several ways: SynDEx is not open-source, it has a unique dataflow MoC that does not support schedulability analysis, and although the code generation task exists, it is not provided with the tool. Schedulability analysis is an important feature of Preesm because it ensures deadlock freeness in the generated code.

SDF For Free (SDF3) [Ele13] is an open-source dataflow analysis framework that supports the SDF, CSDF and SADF MoCs. SDF3 is developed by the Electronic Systems Group of the Eindhoven University of Technology. SDF3 is a command-line framework oriented towards transformation, analysis, and simulation of applications modeled with dataflow MoCs including the SDF, the CSDF and the SADF MoCs. In particular, SDF3 includes a tool to generate random SDF graphs that mimic the properties of DSP applications. This random graph generator is used in Chapter 4 to test the proposed memory allocation technique on a large number of SDF graphs with varying properties. SDF3 focuses on the theoretical study of the deployment of dataflow applications on MPSoCs, but cannot be used to generate an executable prototype.

Ptolemy (I and II) [BHLM94] is one of the first and most complete open-source frameworks for modeling and simulation of applications modeled with dataflow MoCs. Since 1990, Ptolemy is developed by the eponymous group led by Professor Edward A.

Lee at U.C. Berkeley. The specificity of the Ptolemy framework is the possibility to graphically edit hierarchical application graphs where each level of hierarchy respects a different MoC, including DPN, PSDF, and SDF. Ptolemy focuses on the theoretical study of real-time applications and their simulation on multicore CPUs. Like SDF3, Ptolemy cannot be used to generate code for MPSoCs.

The **DSPCAD Lightweight Dataflow Environment (LIDE)** [SWC+11] is a command-line tool supporting the modeling, simulation, and implementation of DSP systems modeled with dataflow graphs. LIDE is developed by the DSPCAD Research Group at the University of Maryland. LIDE supports applications modeled with the Enable-Invoke Dataflow (EIDF) MoC [PSK+08]: a specialization of the DPN MoC equivalent in expressivity. The **Open RVC-CAL Compiler (Orcc)** [YLJ+13] is an open-source tool that generates different types of hardware [SWNP12] and software codes from a unique dataflow-based language named RVC-CAL. An important difference between Orcc, LIDE and PREESM is the MoC used for describing applications. While PREESM uses the decidable [BL06] IBSDF MoC, the EIDF MoC and the DPN MoC implemented in RVC-CAL are not decidable. Consequently, in the general case, no guarantee can be given in LIDE and Orcc on the deadlock-freeness and memory boundedness of applications.

MPSoC Application Programming Studio (MAPS) [CCS+08] is a framework enabling the automated deployment of applications described with KPNs on heterogeneous MPSoCs. MAPS is a closed-source project developed at the RWTH Aachen University. A similar tool is the **AccessCore** IDE that supports the deployment of applications modeled with CSDF graphs on *Kalray*'s many-core processors [Kal14].

The features that differentiate PREESM from the related works and similar tools are:

- the tool is open-source and accessible online [IET14b];
- the algorithm description is based on a single well-known and predictable MoC;
- the scheduling is totally automatic;
- the functional code for heterogeneous MPSoC is generated automatically;
- rapid prototyping metrics are generated to help the system designer to take decisions;
- the IBSDF algorithm model provides a helpful hierarchical encapsulation;
- the System-Level Architecture Model (S-LAM) architecture model provides a high-level architecture description to study system bottlenecks [PNP+09].

3.3.2 PREESM Typical Rapid Prototyping Workflow

When using PREESM, a developer can customize the set of operations that are successively executed by the rapid prototyping framework. This feature is supported by a graphically edited directed acyclic graph called a **workflow**. Each vertex of the workflow represents an operation, called workflow task, that must be executed by the rapid prototyping framework. Examples of workflow tasks are the application of a graph transformation, the generation of compilable code, the simulation of the system, or the mapping and scheduling of the application. As in dataflow graphs, the directed edges of a workflow represent data dependencies between the workflow tasks. For example, edges can be used to transmit application and achitecture models to the mapping scheduling task, or mapping/scheduling choices to the code generation task.

PREESM provides a set of predefined tasks that can be used to compose a workflow. Each task is implemented in a different Eclipse plug-in, providing a high scalability to the tool. Workflow support is a feature that makes PREESM scalable and adaptable to designers' needs. A developer tutorial[1] provides all necessary information to create new workflow tasks and adapt PREESM (e.g. for exporting a graph in a custom syntax or for experimenting new scheduling methods).

Figure 3.5 presents a typical example of PREESM workflow for deploying an application modeled with an IBSDF graph on a heterogeneous architecture. As illustrated, PREESM workflow is consistent with the rapid prototyping design flow presented in Figure 3.1.

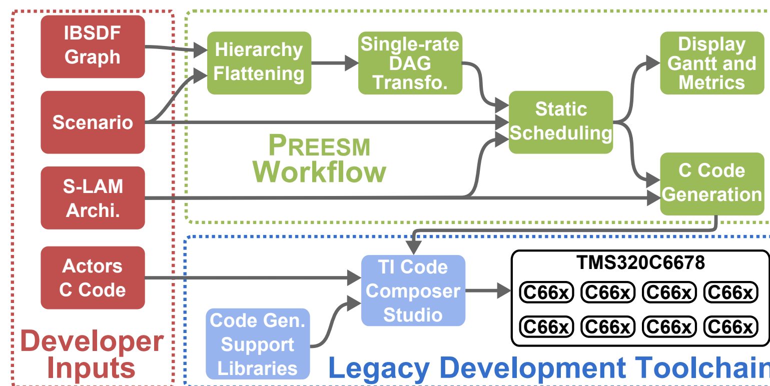


Figure 3.5: *Example of typical PREESM workflow*

Developer Inputs

In addition to the IBSDF graph and the S-LAM architecture description, the developer inputs of PREESM include a scenario that specifies the deployment constraints for a pair of application and architecture. In this example, the developer also provides C code describing the internal behavior of actors.

PREESM Workflow Tasks

In the workflow presented in Figure 3.5, the following workflow tasks are executed:

- **Hierarchy flattening.** By selecting the depth of the hierarchy flattening, the developer of an application can customize the granularity of the application before the mapping and scheduling process. Flattening an IBSDF graph into an equivalent SDF graph consists of instantiating subgraphs of hierarchical actors into their parent graph (cf. Section 2.4.2).

- **Single-rate DAG transformation.** The purpose of the single-rate transformation is to expose data parallelism of the flattened IBSDF graph. This parallelism will be exploited by the mapping/scheduling process when targeting a multiprocessor architecture. The transformation into a Directed Acyclic Graph (DAG) is applied to isolate an iteration of the original IBSDF graph where each vertex of the single-rate DAG corresponds to a single actor firing. This property will simplify the work of the mapping and scheduling task since the mapping of each actor of the DAG needs to be considered only once.

[1] http://preesm.sourceforge.net/website/new-workflow-task

- **Static scheduling.** Several static mapping and scheduling strategies are available in PREESM including LIST and FAST scheduling [Kwo97]. Schedulers are implemented using the Architecture Benchmark Computer (ABC) scalable scheduling framework introduced by Pelcat et al. in [PMAN09]. The ABC framework allows developers to customize the tradeoff between accuracy and speed of execution of the simulation process used within the mapping/scheduling task.

- **Display Gantt and Metrics.** The purpose of this task is to give a feedback to the developer on the characteristics of the generated prototype. The Gantt chart graphically represents the simulated execution of actors on the processing elements of the architecture as well as the data transiting on the communication channels. Beside the Gantt chart, this task also evaluates the throughput and load balancing of the generated solution, and display a *speedup assessment chart* that draws the expected application execution speedup depending on the number of cores [PAPN12].

- **C Code Generation.** In its current version, PREESM can generate C code for multi-C6x DSP architectures from Texas Instrument [Tex13], for multi-X86 CPUs running with Linux or Windows, and for OMAP4 heterogeneous platforms [HDN+12]. PREESM generates a specific C file for each processing element, containing actor C function calls, inter-core communication and synchronization.

Legacy Development Toolchain

PREESM provides runtime libraries to support the execution of generated code on the set of supported architectures. The purpose of these libraries is to abstract the communication and synchronization mechanisms of the targeted architecture and support calls to high-level primitives in the generated code.

Since all mapping and scheduling choices are statically made during the workflow execution, the execution of the application is *self-timed* [SB09]. In a self-timed execution, the execution time of an actor is not fixed and the precedence between actor firings is guaranteed by inter-core synchronizations and communications.

In the example of Figure 3.5, a compiler from Texas Instrument is used to compile the generated code for a multi-C6x DSP chip. Hence, the generated executable benefits from all optimization techniques implemented in the compiler.

3.3.3 PREESM Additional Features

Architecture, Dataflow, and Workflow Graphical Editors

One of the main objectives of PREESM is to ease the rapid prototyping of complex systems by enforcing the user friendliness of the framework. To this purpose, 3 graphical editors are implemented in PREESM for the edition of S-LAM architecture models, IBSDF application models, and PREESM workflows. These 3 editors provide a unified interface to the developer since they are implemented with the same open-source graph editor library named Graphiti [IET14a]. Figure 3.6 presents a screenshot of the IBSDF graph editor. The application graph illustrated in this example is a parallel version of a Sobel image filter.

Beside editing developer inputs, these graph editors also make it possible to display results produced by a workflow execution. For example, tasks can be added to a workflow to generate files containing the SDF graphs resulting from the hierarchy flattening and the single-rate transformation. These files can then be opened with the graph editor to check

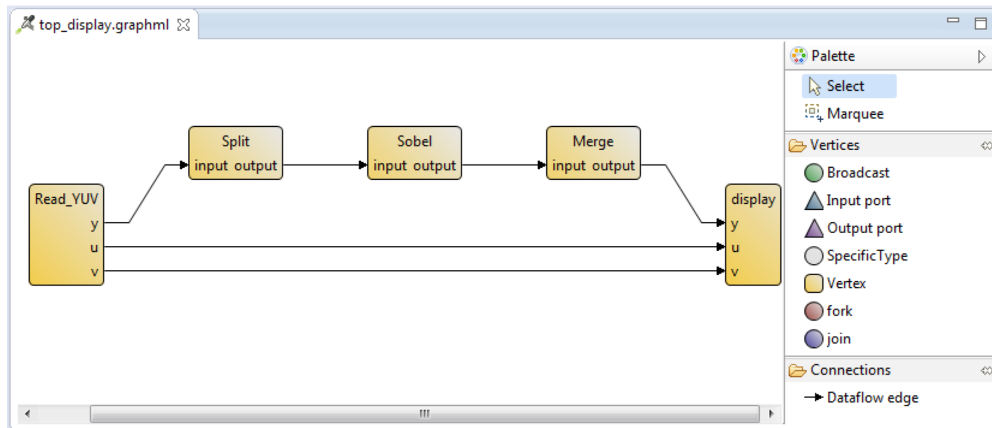


Figure 3.6: *Screenshot of the IBSDF graph editor of PREESM*

their properties, which can be helpful when debugging the original IBSDF graph. In the educational context, these intermediary graphs can also be used to illustrate the result of graph transformations.

Instrumented Code-Generation Optional Feature

The code generation workflow task of PREESM generates *self-timed* code for different multiprocessor architectures. Optionally, the generated code can be automatically instrumented in order to trace its execution and provide a feedback to the developer on the actual execution time of actors. As presented in an online tutorial [2], the instrumented code generates and fills a spreadsheet that automatically summarizes the results of an execution. This spreadsheet can then be imported into the *scenario* editor of PREESM in order to refine the execution time constraints used for subsequent executions of the mapping/scheduling process.

Compatibility with Other Tools

Workflow tasks have been implemented in PREESM to allow application developers to import and export dataflow graphs into formats supported by third-party dataflow programming tools. For example, an exporter of SDF graphs in the Dataflow Interchange Format (DIF) used in LIDE was used in [ZDP+13]. An tutorial explaining the import/export process of SDF graphs in the SDF3 format is available online[3]. This process is used in Chapter 4 to compare the efficiency of proposed memory allocation techniques with those implemented in SDF3.

A generator of PREESM projects was also developed in Orcc. Since the DPN MoC supported by Orcc has a greater expressivity than the IBSDF MoC used in PREESM, this generator can only be used for applications that are classified as *static*. This generator has been successfully used to export image processing [HDN+12], and computer vision [Zha13] applications.

[2]http://preesm.sf.net/website/index.php?id=automated-measurement-of-actor-execution-time
[3]http://preesm.sf.net/website/index.php?id=import-export-an-sdf3-graph

3.4 Memory Optimization for MPSoCs

Like most MPSoC programming environments, dataflow-based or not, the primary goal of PREESM is to generate solutions with optimal latency and throughput [PAPN12]. Hence, the traditional rapid prototyping design flow presented in Section 3.2 is naturally centered around the mapping and scheduling process that plays a critical role in the optimization of application performance.

Memory-related issues are often treated as secondary issues, subsequent to the scheduling of computations on the different processing elements. Nevertheless, the memory issues addressed during the development of an embedded system often yield a strong impact on system quality and performance. Indeed, the silicon area occupied by the memory can be as large as 80% of the chip and be responsible for a major part of the power consumption [DGCDM97]. A bad memory management can thus result in memory and power waste but also in poor system performances if, for example, memory accesses are a bottleneck of the system.

Moreover, memory issues often lead to modifications of the application description in order to make it fit in the available memory. These modifications are sometimes made at the expense of quality of the result produced by the application [MNMZ14].

3.4.1 Modern Memory Architectures

As presented in Section 3.2.1, memory hierarchy is one of the 5 attributes that can be used to characterize a modern multiprocessor architecture [BDM09]. In [Rai92], Raina proposes a comprehensive terminology to classify the organization of memory in a hardware architecture. This terminology focuses on two binary criteria: the *address space* accessed by the processing elements, and the *physical memory organization* of the architecture.

The *address space* of an architecture can either be *shared* or *disjoint*. In a *shared address space*, each address represents a unique place in memory that can be accessed by all processing elements of the architecture. In a *disjoint* address space, each processing element possesses a private address space that no other processing element can access directly.

The *physical memory organization* of an architecture can either be *shared* or *distributed*. A *shared memory* is a memory bank that is connected to several processing elements of an architecture. A *distributed memory* architecture is an architecture with several memory banks, each associated to a different processing elements.

Each combination of these two criteria characterizes a family of architecture. For example, in an SADM (Shared Address space, Distributed Memory) architecture, although each memory bank is associated to a processing element, it can be accessed by any processing element of the architecture using memory addresses that are globally associated to this memory bank.

The terminology proposed in [Rai92] also introduces terms to characterize the cache coherence protocols supported by multicore architectures.

A cache is a fast memory bank of small capacity that is accessible by a unique processing element, in addition to the slower main memory bank of the architecture. Cache replacement algorithm refers to the hardware mechanism responsible for selecting which data from the main memory will be duplicated in the cache in order to accelerate the computation performed by the associated processing element [Smi82]. A *scratchpad* is also a fast memory bank associated to a processing element, with the difference that the content of a scratchpad is controlled by the software running on this processing element.

In shared address space architectures where each processing element possesses its own cache (or scratchpad), consistency issues may arise if the same piece of data is stored and modified simultaneously in multiple caches. The purpose of coherence protocols is to ensure that accesses to a shared address range will produce the same content for all processing elements.

3.4.2 Memory Management for Multicore Architectures

Memory management designates the set of processes, mechanisms, and protocols used to administer the memory resources of an architecture. Memory management for multicore architectures primarily consists of allocation processes, cache replacement algorithms, and coherence protocols.

Allocating an application in memory consists of assigning a range of memory addresses to each piece of data and program that is read or written during the execution of this application.

The memory management supporting the execution of parallel applications on multicore architectures must address the following challenges:

- **Ensure data availability.** The memory allocation process must ensure that all **memory objects** (i.e. variables, constants, instructions, ...) used for the execution of an application are allocated in memory ranges that are accessible when and where they are needed. This constraint is particularly challenging in DADM (Disjoint Address space, Distributed Memory) architectures where each processing element possesses a private address space that no other processing element can access.

- **Minimize the memory footprint.** Memory resources are often limited, especially in embedded systems. Hence, an objective of memory allocation processes is to minimize the memory footprint of applications: the total range of memory addresses used at any moment during the execution of an application. A typical strategy to achieve this purpose is to reuse previously allocated memory spaces as soon as their content is no longer relevant to the running application.

- **Ensure data integrity.** Memory allocation must guarantee that data stored by an application will not be corrupted or lost until it is no longer needed by this application. This objective is not trivial in complex memory hierarchies where data corresponding to a given address range can be stored simultaneously in different levels of memory (caches, scratchpads, internal/external memories, ...). In such a case, the memory allocation process must work jointly with cache replacement algorithms and coherence protocols to ensure data integrity.

- **Optimize data locality.** The performance of an application can be optimized by allocating in adjacent address ranges the memory objects that are successively accessed by a processing element [WL91]. Since adjacent memory ranges are often duplicated in cache simultaneously, this strategy increases the chances of processing elements accessing data that is already duplicated in their cache.

3.4.3 Usual Memory Allocation Techniques

Like the mapping and scheduling process, the allocation of an application in memory can be realized statically or dynamically. The static allocation is performed during the compilation process and associates a fixed address to each memory object. In the dynamic

case, memory objects are given their address at runtime by a dedicated process that handles the available memory. Only the static approach will be covered in this thesis.

Static memory optimization for multicore systems has generally been studied in the literature as a post-scheduling memory allocation process. Using the scheduling information, the lifetimes of the different memory objects of an application are derived. The lifetime of a memory object is the period between the first and last scheduled accesses to this memory object. Minimization of the memory footprint of an application is achieved by allocating several memory objects whose lifetimes do not overlap in the same memory space.

As presented in [BAH09], the minimization of the memory footprint allocated for memory objects of variable sizes is a problem with an NP-Hard complexity. Therefore, it would be prohibitively long to find an optimal solution to this problem in the context of rapid prototyping. Instead, many heuristic algorithms have been proposed in the literature to perform the memory allocation of applications in reasonable computational time:

- **Online allocation (greedy) algorithms [Rau06].** Online allocators assign memory objects one by one in the order in which they are fed to the allocator. The most commonly used online allocators are the First-Fit (FF) and the Best-Fit (BF) algorithms [Joh73]. FF algorithm consists of allocating an object to the first available space in memory of sufficient size. The BF algorithm works similarly but allocates each object to the available space in memory whose size is the closest to that of the allocated object.

- **Offline allocation algorithm [DGCDM97, MB00].** In contrast to online allocators, offline allocators have a global knowledge of all memory objects requiring allocation, thus making further optimizations possible.

- **Coloring an exclusion graph [BAH09, Rau06].** An *exclusion graph* (or conflict graph) is a simple graph whose vertices are the memory objects and whose undirected edges represent exclusions (i.e. overlapping lifetimes) between objects. Coloring the exclusion graph consists of assigning a set of colors to each object such that two connected memory objects have no color in common. The purpose of graph coloring technique is to minimize the total number of colors used in the graph [BAH09, Rau06]. An equivalent approach is to use the complement graph, where memory objects are linked if they have non-overlapping lifetime, and perform a clique partitioning of its memory objects [KS02].

- **Using constraint programming [SK01]** where memory constraints can be specified together with resource usage and execution time constraints.

Although memory optimization has mostly been studied as a post-scheduling allocation process, pre-scheduling memory optimization techniques can also be found in the literature. These optimization techniques mostly consist of modifying the description of the application behavior to maximize the impact of later optimization processes. Variable renaming, instruction re-ordering, loop merging and splitting are examples of modifications for imperative languages that can reduce the memory needs of an application [Fab79]. Similar modifications can be applied to SDF graphs, as was done in [PBPR09].

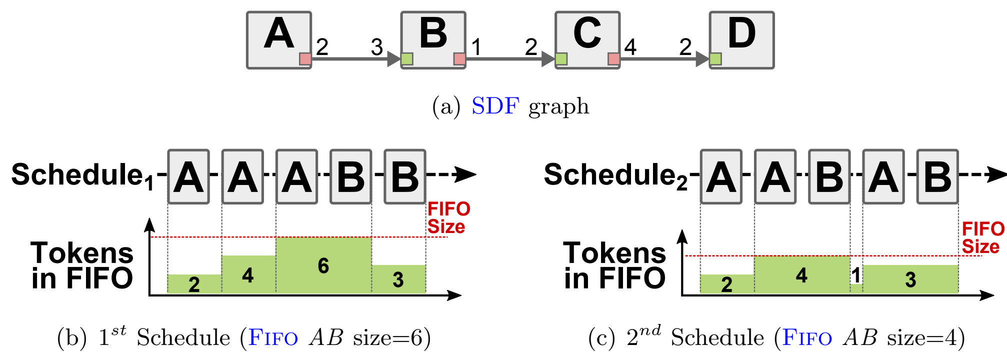
As explained in [BKV+08], these memory allocation and optimization techniques often require a partial system synthesis and the execution of time-consuming algorithms. Although these techniques provide an exact or highly optimized memory requirement, they may be too slow to be used in the rapid prototyping context. In [BKV+08], Balasa et al.

survey existing estimation techniques that provide a reliable memory size approximation in a reasonable computation time. Unfortunately, these techniques are based on the analysis of imperative code, and no such technique exists for higher level programming models such as dataflow graphs.

In this thesis, a new method is introduced to quickly evaluate the memory requirement of applications modeled with IBSDF graphs. As will be shown in Chapter 4, this method is independent from architectural consideration and can be used prior to any scheduling process.

3.4.4 Literature on Memory Optimization and Dataflow Graphs

Minimizing the memory footprint of dataflow applications is usually achieved by using FIFO dimensioning techniques [Par95, SGB06, MB10, BMMKU10]. FIFO dimensioning techniques, also called buffer sizing techniques, consist of finding a schedule of the application that minimizes the memory space allocated to each FIFO of the dataflow graph.



(a) SDF graph

(b) 1st Schedule (FIFO AB size=6)

(c) 2nd Schedule (FIFO AB size=4)

Figure 3.7: *Example of the impact of scheduling choice on FIFO size*

An illustration of the impact of scheduling choices on FIFO sizes is given in Figure 3.7. Figures 3.7(b) and 3.7(c) present two mono-core schedules of actors A and B from the SDF graph of Figure 3.7(a). The evolution of the number of data tokens stored in FIFO AB during an iteration of the graph is depicted below each schedule. It is assumed that an actor can access input tokens until the end of its firing, and that it can write its output data tokens from the start of its firing. In the first schedule, three firings of actor A are executed before the two firings of actor B. This schedule results in a maximum number of 6 data tokens stored in FIFO AB. In the second schedule, where a firing of actor B is inserted before the last firing of actor A, at most 4 data tokens are stored in the FIFO during an iteration.

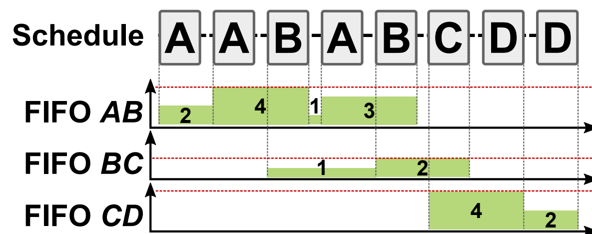


Figure 3.8: *Example of memory reuse opportunity*

The main drawback of FIFO dimensioning techniques is that they do not consider the reuse of memory since each FIFO is allocated in a dedicated memory space. For example, Figure 3.8 illustrates the content of the FIFOs throughout a complete iteration of the SDF graph from Figure 3.7(a). Even though FIFOs AB and CD never contain data tokens simultaneously, they will not be allocated in overlapping memory spaces. Hence, FIFO dimensioning techniques often result in wasted memory space [MB04].

The memory analysis and optimization techniques presented in this thesis focus on the revelation and the exploitation of memory reuse opportunities for minimizing the memory footprint of dataflow graphs.

3.5 Conclusion of the Background Part

In these background chapters, the challenges and existing solutions related to the deployment of parallel applications on modern heterogeneous MPSoCs have been surveyed. Chapter 2 introduced the semantics and the properties of the dataflow MoCs that will be used for describing applications in this thesis. In Chapter 3, the challenges addressed in the different steps of a rapid prototyping design flow were presented and the memory issues studied in this thesis were introduced.

Next chapters present a complete method to study and take advantage of the memory reuse opportunities offered by applications modeled with the IBSDF MoC. Chapters 4 and 5 focus on the exploitation of memory reuse opportunities at graph and actor levels respectively. In Chapter 6, these memory optimization techniques are applied to a state-of-the-art computer vision application. Then, Chapter 7 introduces a new dataflow meta-model, that combines advantages of the IBSDF and IBSDF MoCs.

Part II

Contributions

CHAPTER 4

Dataflow Memory Optimization: From Theoretical Bounds to Buffer Allocation

4.1 Introduction

Bounding the amount of memory needed to implement an application on a multicore architecture is a key step of a development process. Indeed, memory upper and lower bounds are crucial information in the co-design process. If these bounds can be computed during the early development of an embedded system, they might assist the developer in correct memory dimensioning. For example, memory bounds allow the developer to adjust the size of the architecture memory accordingly to the application requirements. Alternatively, memory bounds can drive the refactoring of an application description to comply with the amount of available memory on the targeted architecture [MNMZ14].

This chapter presents a complete method to study the memory characteristics of an application modeled with an IBSDF graph. Section 4.2 presents the intermediate representation used to model the memory characteristics of an IBSDF graph. The bounding and allocation techniques based on this intermediate representation are presented in Sections 4.3 and 4.4 respectively.

4.2 Memory Exclusion Graph (MEG)

Memory optimization techniques presented in this chapter require an accurate modeling of the memory characteristics of an input IBSDF graph. A preliminary step to these optimization techniques is the construction of a weighted graph, called Memory Exclusion Graph (MEG), that models the memory characteristics of an application. The Memory Exclusion Graph (MEG) then serves as a basis for the analysis and allocation techniques. Next sections detail how the original IBSDF is transformed to expose its memory characteristics, and how a MEG is built to capture these characteristics.

4.2.1 IBSDF Pre-Processing for Memory Analysis

The first step to derive the MEG of an application consists of successively flattening the hierarchy of its IBSDF graph, transforming the resulting SDF graph into a single-rate SDF graph, and then into a DAG. As presented in Section 3.3.2, these transformations

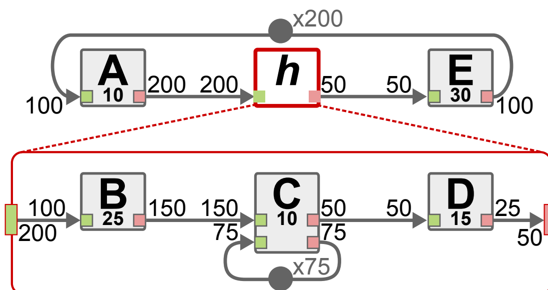


Figure 4.1: *Example of IBSDF graph*

are already applied by the rapid prototyping workflow of PREESM in order to reveal the embedded parallelism of applications before the mapping and scheduling process.

In the context of memory analysis and allocation, the single-rate and the DAG transformations are applied with the following objectives:

- **Break FIFOs into shared buffers:** In the IBSDF model, channels carrying data tokens between actors behave like FIFO queues. The memory needed to allocate each FIFO corresponds to the maximum number of tokens stored in the FIFO during an iteration of the graph [MB00]. As exposed in [SGB06, BMMKU10], this maximum number of tokens can be determined only by deriving a schedule of the SDF graph.

  In PREESM, memory analysis and allocation techniques can be independent from scheduling considerations. Since the exact size of FIFOs remains undefined until the scheduling process, FIFOs are replaced with synchronized buffers of fixed size during the transformation of the original application graph into a single-rate SDF graph.

- **Expose data parallelism:** Concurrent analysis of data parallelism and data precedence gives information on the lifetime of memory objects prior to any scheduling process. For example, if two FIFOs belong to parallel data-paths, a schedule can be derived so that the two FIFOs contain data tokens simultaneously. Consequently, FIFOs belonging to parallel data-paths cannot be allocated in overlapping address ranges because otherwise they would overwrite each others' content. Conversely, two single-rate FIFOs linked with a precedence constraint can be allocated in the same memory space since they will never store data tokens simultaneously. In Figure 4.3 for example, FIFO AB_1 is a predecessor to C_1D_1. Consequently, these two FIFOs may share a common address range in memory.

- **Derive an acyclic graph:** Cyclic data-paths in an SDF graph are an efficient way to model iterative or recursive calls to a subset of actors. In order to use efficient static scheduling algorithms [Kwo97], SDF models are often converted into a single-rate DAG before being scheduled. Besides revealing data-parallelism, this transformation makes it easier to schedule an application, as each actor is fired only once per execution of the resulting DAG. Similarly, in the absence of a schedule, deriving a single-rate DAG permits the use of single-rate FIFOs that will be written and read only once per iteration of DAG. Consequently, before a single-rate FIFO is written and after it is read, its memory space will be reusable to store other objects.

Figures 4.1 to 4.3 illustrate the successive transformations applied to an IBSDF graph to reveal its memory characteristics. The DAG resulting from these transformations will be used to build the MEG that serves as a basis to the memory optimization techniques of PREESM.

The first transformation consists of flattening the hierarchy of the graph by replacing all hierarchical actors with their content. The IBSDF graph from Figure 4.1 is thus transformed into an SDF graph presented in Figure 4.2. More information on the rules for flattening the IBSDF hierarchy is given in Section 2.4.2.

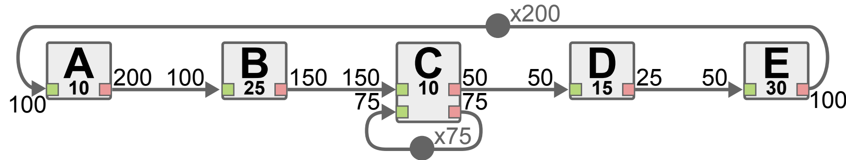


Figure 4.2: *SDF graph resulting from the flattening of Figure 4.1*

The second transformation applied to the input graph is a conversion into a single-rate SDF graph. In Figure 4.3, actors B, C, and D are each split in two instances and new FIFOs are added to ensure the equivalence with the SDF graph of Figure 4.2. For clarity, the *Fork* and *Join* actors added during the single-rate transformation are not depicted in Figure 4.3 nor in subsequent illustrations. However, the reader should keep in mind that a strict application of the SDF MoC forbids the connection of multiple FIFOs to a single data port of an actor.

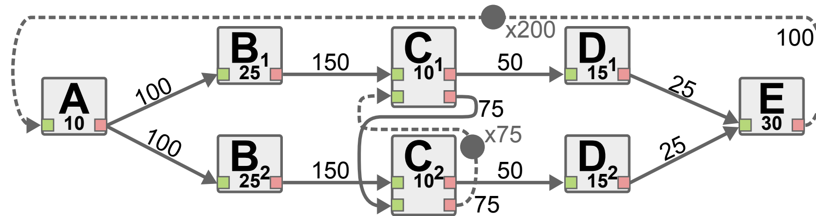


Figure 4.3: *Single-rate SDF graph derived from IBSDF graph of Figure 4.1 (Directed Acyclic Graph (DAG) if dotted FIFOs are ignored)*

The last conversion consists of generating a Directed Acyclic Graph (DAG) by isolating one iteration of the algorithm. This conversion is achieved by ignoring FIFOs with initial tokens, or delays, in the single-rate SDF graph. In the example, this approach means that the feedback FIFO C_2C_1, which stores 75 initial tokens, and FIFO EA, which stores 100 initial tokens, are ignored.

4.2.2 MEG Definition

Once an IBSDF graph has been pre-processed into a DAG to expose its memory characteristics, a MEG is built to capture these characteristics.

Definition 4.2.1. *A Memory Exclusion Graph (MEG) is an undirected weighted graph denoted by $G = \langle M, E, w \rangle$ where:*

- **M** *is the set of vertices of the MEG. Each vertex represents an indivisible* memory object *of the application.*
- **E** $\subseteq M \times M$ *is the set of edges representing the memory exclusions, i.e. the impossibility to share memory.*
- **w** $: M \to \mathbb{N}$ *is a function with $w(m)$ the weight of a memory object m. The weight of a memory object corresponds to the amount of memory required for its allocation.*

In addition to this formal definition, the following notations are also introduced:

- $\mathbf{N(m)}$ *the neighborhood of* $m \in M$*, i.e. the set of memory objects linked to* m *by an exclusion* $e \in E$*. Memory objects of this set are said to be adjacent to* m*.*
- $|\mathbf{S}|$ *the cardinality of a set* S*.* $|M|$ *and* $|E|$ *are the number of memory objects and exclusions respectively of a graph.*
- $\boldsymbol{\delta}(\mathbf{G}) = \frac{2 \cdot |E|}{|M| \cdot (|M|-1)}$ *the edge density of the graph corresponding to the ratio of existing exclusions to all possible exclusions.*

Memory Objects

A *memory object* $m \in M$ is an indivisible quantum of memory that must be allocated (i.e. be mapped in a system memory) to enable the execution of an application. The DAG resulting from the transformations of an IBSDF graph contains three types of memory objects:

- **Single-rate FIFOs/buffers:** The first type of memory object is the buffers used to transfer data tokens between consecutive actors. These buffers correspond to the single-rate FIFOs of the DAG.
  Formally, the memory object associated to a single-rate FIFO AB is noted m_{AB}.
- **Working memory of actors:** The second type of memory object corresponds to the maximum amount of memory allocated by an actor during its execution. This working memory represents the memory needed to store the data used during the computations of the actor but does not include the input nor the output buffers storage. In PREESM as in most dataflow programming frameworks [MB04], it is assumed that an actor keeps exclusive access to its working memory during its execution. This memory is equivalent to a task stack space in an operating system. In Figures 4.1 to 4.3, the size of the working memory associated with each actor is given by the number below the actor name.
  Formally, the memory object associated to the working memory of an actor A is noted m_A.
- **Feedback/pipeline FIFOs:** The last type of memory object corresponds to the memory needed to store feedback FIFOs ignored by the transformation of a single-rate SDF into a DAG. In Figure 4.3, there are two feedback FIFOs: $C_2 C_1$ and EA. Each feedback FIFO is composed of 2 memory objects: the *head* and the (optional) *body*. The *head* of the feedback FIFO corresponds to the data tokens consumed during an iteration of the single-rate SDF graph. The *body* of the feedback FIFO corresponds to data tokens that remain in the feedback FIFO for several iterations of the graph before being consumed. A *body* memory object is needed only if the amount of delay on the feedback single-rate FIFO is greater than its consumption rate, as is the case with feedback FIFO EA in Figure 4.3. On the contrary, the feedback FIFO $C_2 C_1$ does not need a *body* memory object.
  Formally, the memory objects associated to the head and body of feedback FIFO AB are noted $m_{head(AB)}$ and $m_{body(AB)}$ respectively.

Exclusions

Two memory objects of any type exclude each other in the MEG if they can not be allocated in overlapping address ranges.

Definition 4.2.2. *Considering a DAG* $= \langle A, F \rangle$ *and its derived MEG* $= \langle M, E, w \rangle$*, two memory objects* $m_a, m_b \in M$ *are linked by an exclusion* $e \in E$ *if they may store valid data simultaneously during the execution of the DAG.*

Sources of exclusion are given by the following rules. Two memory objects exclude each other in the MEG if:

1. The DAG can be scheduled in such a way that both these memory objects store data tokens simultaneously. Memory objects belonging to parallel data-paths fall within this category. For example, in Figure 4.3, there is no precedence relationship between m_{AB_2} and m_{D_1}. Hence, actor D_1 may be scheduled after actor A and before actor B_2, thus using its working memory m_{D_1} while data tokens are stored in m_{AB_2}.

2. The memory objects are working memory, input, or output buffers of the same actor. In PREESM, the memory allocated to these buffers is reserved from the execution start of the producer actor until the completion of the consumer actor. This choice is made to enable custom token accesses throughout actor firing time. As a consequence, the memory used to store an input buffer of an actor should not be reused to store an output buffer of the same actor. In Figure 4.3, the memory used to carry the 100 data tokens of memory object m_{AB_1} can not be reused, even partially, to transfer data tokens from actor *B1* to actor *C1*.

3. The first memory object of a feedback FIFO f is a *head* memory object $m_{head(f)}$, and the second does not belong to the consumer-producer data-path of f. In Figure 4.3, actors B, C, and D and the FIFOs between them are both successors to the consumer A and predecessors to the producer E of the feedback FIFO EA. Consequently, the working memory of these actors (m_{B_1}, m_{B_2}, m_{C_1}, m_{C_2}, m_{D_1}, m_{D_2}) and the memory allocated to the single-rate FIFOs ($m_{B_1C_1}$, $m_{B_2C_2}$, $m_{C_1C_2}$, $m_{C_1D_1}$, $m_{C_2D_2}$) can be allocated in an overlapping memory space with $m_{head(EA)}$.

4. One of the memory objects is a *body* memory object. For example, the *body* memory object $m_{body(EA)}$ associated to the feedback FIFO EA will have exclusions with all other memory objects of the MEG.

These four rules are formally expressed as follows, with **pred** $: A \cup F \to \{A \cup F\}^{n \in \mathbb{N}}$ a function that associates an actor $a \in A$ (or a FIFO $f \in F$) to its list of *predecessors*, i.e. the list of actors and FIFOs that precede a (or f) in the data dependency order of the graph.

Theorem 4.2.1. *Considering a DAG* $= \langle A, F \rangle$ *and its derived MEG* $= \langle M, E, w \rangle$
$\forall a, b \in \{A, F\} \mid a \neq b$, ***if*** $a \notin pred(b) \wedge b \notin pred(a)$ ***then*** $\exists e \in E$ *linking* m_a *to* m_b.

Proof. Since there is no precedence relationship between a and b, the scheduling order of a (or its producer if $a \in F$) is independent from the scheduling order of b. Consequently, nothing prevents a and b from being scheduled in parallel on two distinct processing elements. In such a case, the application will access data stored in m_a and m_b simultaneously, which requires the presence of an exclusion $e \in E$ between m_a and m_b according to Definition 4.2.2. □

Theorem 4.2.2. *Considering a DAG* $= \langle A, F \rangle$ *and its derived MEG* $= \langle M, E, w \rangle$,
$\forall f \in F$, $\forall b \in \{A, F\} \mid b \neq f$,

$$\textbf{if} \begin{cases} b \in A \wedge [prod(f) = b \vee cons(f) = b] \\ b \in F \wedge [prod(f) = prod(b) \vee prod(f) = cons(b)] \end{cases} \quad \textbf{then } \exists e \in E \text{ linking } m_f \text{ to } m_b$$

Proof. In the case where $b \in A \wedge [prod(f) = b \vee cons(f) = b]$, f is an input or an output buffer of actor b. In the case where $b \in A \wedge [prod(f) = b \vee cons(f) = b]$, f and b are output buffers, or input and output buffer, of the same actor $prod(f)$.

Since an actor $a \in A$ keeps access to its input and output buffers (and its working memory) during its whole firing time, these buffers can simultaneously store valid data. Consequently, following Definition 4.2.2, the memory objects corresponding to these buffers must be linked by an exclusion in the corresponding MEG. □

Theorem 4.2.3. *Considering a DAG* $= \langle A, F \rangle$ *and its derived MEG* $= \langle M, E, w \rangle$*, let* f *be a feedback* FIFO *of the corresponding single-rate SDF graph.*
$\forall b \in \{A, F\}$, ***if*** $b \notin pred(prod(f)) \vee cons(f) \notin pred(b)$ ***then*** $\exists e \in E$ *linking* $m_{head(f)}$ *to* m_b

Proof. During a graph iteration, the *head* memory object $m_{head(f)}$ of a feedback FIFO f stores data tokens before the FIFO consumer $cons(f)$ is fired, and after the FIFO producer $prod(f)$ is fired. If b is a FIFO or an actor of the DAG that is both a predecessor to $prod(f)$ and a successor to $cons(f)$, its associated memory objects will never store valid data simultaneously with the feedback FIFO *head*. If both these conditions are not met, a schedule of the DAG can be derived where $cons(f)$ is scheduled after b, or $prod(f)$ is scheduled before b. In such cases, $m_{head(f)}$ and m_b store valid data simultaneously, so they must exclude each other in the MEG according to Definition 4.2.2. □

Theorem 4.2.4. *Considering a DAG* $= \langle A, F \rangle$ *and its derived MEG* $= \langle M, E, w \rangle$*, let* f *be a feedback* FIFO *of the corresponding single-rate SDF graph.*
$\forall m \in M \setminus m_{body(f)}$, $\exists e \in E$ *linking* $m_{body(f)}$ *to* m

Proof. Body memory objects store data tokens for periods spanning over several iterations of the graph. Since all other memory objects have a lifespan bounded by a graph iteration, they will all store data tokens simultaneously with *body* memory objects. Consequently, following Definition 4.2.2, *body* memory objects must always have exclusions with all other memory objects in their MEG. □

A MEG is valid only if the previous rules are respected for all pairs of memory objects.

MEG Example

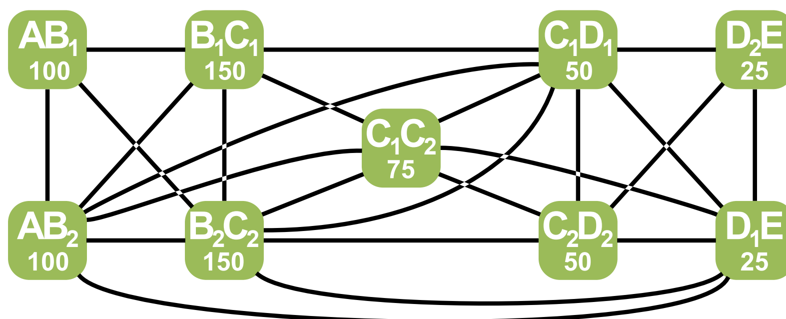


Figure 4.4: *Memory Exclusion Graph (MEG) derived from the IBSDF graph of Figure 4.1*

The MEG presented in Figure 4.4 is derived from the IBSDF graph of Figure 4.1. The complete MEG contains 20 memory objects and 95 exclusions but, for clarity, only the memory objects corresponding to the single-rate FIFOs (1st type memory objects) are

presented. The values printed below the vertices names represent the weight w of the memory objects.

4.2.3 MEG Construction

Building a MEG based on a DAG consists of scanning its actors and single-rate FIFOs in order of precedence, so as to identify its parallel branches. As part of this scan, the memory objects and the exclusions caused by a precedence relationship are added to the MEG. Then, exclusions are added to the MEG between all memory objects belonging to parallel data-paths.

The pseudo-code of an algorithm to build the complete MEG of an application is given in Algorithm 4.1.

Algorithm 4.1 can be divided in three main parts each responsible for the creation of one type of memory objects. Creation of working memory of actors is handled between lines 5 and 10. Single-rate FIFOs are processed between lines 12 and 21, and the memory objects associated to feedback FIFOs are generated between lines 24 and 39.

The MEG obtained at this point of the method models all possible exclusions for all possible schedules. Hence, from the memory reuse perspective, this MEG corresponds to the worst-case scenario. As will be shown in Section 4.4, it is possible to update a MEG with scheduling information in order to reduce the number of exclusions, thus favoring memory reuse.

An alternative way of building a MEG is to first build its complement graph, within which two memory objects are linked if they can share a memory space. Then, the exclusion graph is simply obtained by considering that two of its memory objects exclude each other if they are not connected by an edge in the complement graph. Using a complement graph instead of a MEG may be advantageous when allocating MEGs with exclusion density greater than 0.5. Indeed, in such a case, the complement graph will have fewer edges than the MEG and hence, it will be faster to check if two memory objects exclude each other by checking the absence of link between them in the complement graph than the presence of an exclusion in the MEG.

The next two sections present memory analysis and optimization techniques based on MEGs derived from IBSDF graphs.

4.3 Bounds for the Memory Allocation of IBSDF Graphs

A MEG is an intermediary representation that captures the memory characteristics of an IBSDF graph. Hence, a MEG can serve as a basis for analyzing and optimizing the allocation of memory objects for the execution of an application on an embedded MPSoC. In this section, an analysis technique is presented for deriving the memory allocation bounds (Figure 4.5) of an application modeled with an IBSDF graph. This bounding technique has been the subject of a publication in an international conference [DPNA12].

Figure 4.5: *Memory Bounds*

The upper and lower bounds of the static memory allocation of an application are a maximum and a minimum limit respectively to the amount of memory needed to run this

Algorithm 4.1: Building the Memory Exclusion Graph (MEG)

```
   Input: a single-rate SDF srSDF =< A, F > with:
          A the set of actors
          F the set of FIFOs
   Output: a Memory Exclusion Graph (MEG) MEG =< V, E, w >
 1 Define Pred[], I[], O[] : A → V* ⊂ V;
 2 Sort A in the DAG precedence order;
 3 for each a ∈ A do
 4 |  /* Process working memory of a */
 5 |  workingMem ← new v ∈ V;
 6 |  w(workingMem) ← workingMemorySize(a);
 7 |  for each v ∈ V \ {Pred[a], workingMem} do
 8 |  |  Add e ∈ E between workingMem and v;
 9 |  endfor
10 |  I[a] ← I[a] ∪ {workingMem};
11 |  /* Process output buffers of a */
12 |  for each f ∈ (F \ feedbackFIFOs) ∩ outputs(a) do
13 |  |  bufMem ← new v ∈ V;
14 |  |  w(bufMem) ← size(f);
15 |  |  for each v ∈ V \ {Pred[a], bufMem} do
16 |  |  |  Add e ∈ E between bufMem and v;
17 |  |  endfor
18 |  |  Pred[consumer(f)] ← Pred[a] ∪ I[a];
19 |  |  I[consumer(f)] ← I[consumer(f)] ∪ {bufMem};
20 |  |  O[a] ← O[a] ∪ {bufMem};
21 |  endfor
22 endfor
23 /* Process Feedback FIFOs */
24 for each ff ∈ F ∩ feedbackFIFOs(F) do
25 |  headMem ← new v ∈ V;
26 |  w(headMem) ← rate(ff);
27 |  set ← (V ∩ P[producer(ff)]) \ P[consumer(ff)];
28 |  set ← set \ I[consumer(ff)] ∪ O[consumer(ff)];
29 |  for each v ∈ V \ set do
30 |  |  Add e ∈ E between headMem and v;
31 |  endfor
32 |  if rate(ff) < delays(ff) then
33 |  |  bodyMem ← new v ∈ V;
34 |  |  w(bodyMem) ← delays(ff) − rate(ff);
35 |  |  for each v inV do
36 |  |  |  Add e ∈ E between bodyMem and v;
37 |  |  endfor
38 |  end
39 endfor
```

application, as presented in Figure 4.5. The following four sections explain how the upper bound can be computed and give three techniques to compute the memory allocation lower bound. These three techniques offer a trade-off between accuracy of the result (Figure 4.8) and complexity of the computation.

4.3.1 Least Upper Bound

The least upper memory allocation bound of an application corresponds to the size of the memory needed to allocate each memory object in a dedicated memory space. This allocation scheme is the least compact allocation possible as a memory space storing a memory object is never reused to store another.

Definition 4.3.1 (Least Upper Bound)**.** *Given a MEG* $= \langle M, E, w\rangle$*, its upper memory allocation bound is the sum of the weights of its memory objects:*

$$Bound_{Max}(G) = \sum_{m \in M} w(m)$$

The upper bound for the MEG of Figure 4.4 is 725 memory units without considering the working memory of actors. As presented in Figure 4.8, using more memory than the upper bound means that part of the memory resources is wasted. Indeed, if a memory allocation uses an address range larger than this upper bound, some addresses within this range will never be read nor written. The upper bound is 1140 memory units for the complete MEG when the working memory of actors are considered.

Although the least upper bound gives a pessimistic evaluation of the memory needed for the allocation of an application, its $O(|N|)$ complexity, where $|N|$ is the number of memory objects, makes it an interesting metric to quickly evaluate the memory requirements of an application.

4.3.2 Lower Bounds

The greatest lower memory allocation bound of an application is the least amount of memory required to execute it. A lower bound to the memory allocation is a value inferior or equal to the greatest lower bound, and thus inferior to the smallest achievable memory allocation. This section presents two state-of-the-art methods and a new heuristic to compute lower bounds of an application. These methods offer a tradeoff between accuracy of the result and speed of the computation.

Method 1 to Compute the Greatest Lower Bound: Interval Coloring Problem

Finding the optimal allocation, and thus the greatest lower bound, using a MEG can be achieved by solving the equivalent *Interval Coloring Problem* [BAH09, Fab79].

A *k-coloring* of a *MEG* $= \langle M, E, w\rangle$ is the association of each memory object $m_i \in M$ with an interval $I_i = \{a, a+1, \cdots, b-1\}$ of consecutive integers called colors, such that $b - a = w(m_i)$. Two memory objects m_i and m_j linked by an exclusion $e \in E$ must be associated to non-overlapping intervals. Assigning an interval of integers is equivalent to allocating a range of memory addresses to a memory object. Consequently, a k-coloring of a MEG corresponds to an allocation of its memory objects.

Solving the *Interval Coloring Problem* consists of finding a *k-coloring* of the MEG with the fewest integers used in the I_i intervals. This objective is equivalent to finding the allocation of memory objects that uses the least memory possible, thus giving the greatest lower bound of the memory allocation.

As presented in [BAH09], the *Interval Coloring Problem* has an NP-Hard complexity. Therefore, it would be prohibitively long to solve this problem for applications with hundreds or thousands of memory objects.

A sub-optimal solution to the *Interval Coloring Problem* [Rau06] corresponds to an allocation that uses more memory than the minimum possible: more memory than the greatest lower bound. Consequently, a sub-optimal solution to the *Interval Coloring Problem* fails to achieve the bounding objective which is to find a lower bound to the size of the memory allocated for a given application.

Method 2 to Compute a Lower Bound: Exact Solution to the Maximum-Weight Clique Problem

Since the greatest lower bound can not be found in reasonable time, an alternative is to find a lower bound close to the size of the optimal allocation. In [Fab79], Fabri introduces another lower bound derived from a MEG: the weight of the Maximum-Weight Clique (MWC).

A clique is a subset of vertices that forms a subgraph within which each pair of vertices is linked with an edge. Formally:

Definition 4.3.2. *Considering a MEG* $= \langle M, E, w \rangle$*, a clique* $C \subseteq M$ *is such that:* $\forall m_i, m_j \in C, m_i \neq m_j => \exists e \in E | m_i \in N(m_j)$

Since memory objects belonging to a clique can not share memory space, their allocation requires a memory as large as the sum of the weights of the clique elements, also called the clique weight.

Theorem 4.3.1. *Considering a clique* $C \subseteq M$ *of a MEG* $= \langle M, E, w \rangle$*, the memory space needed to allocate* C *in memory is:*

$$weight(C) = \sum_{m \in C} w(m)$$

Proof. Let m_0 and m_1 be memory objects of a clique $C \subseteq M$. The memory required for their allocation is $w(m_0)$ and $w(m_1)$ respectively.

Following Definition 4.3.2, since there is an exclusion $e \in E$ linking m_0 to m_1, the two memory objects form a clique C_2 of the MEG, and they must be allocated in non-overlapping address ranges. The amount of memory required for the allocation of C_2 is thus

$$weight(C_2) = w(m_0) + w(m_1) = \sum_{m \in C_2} w(m) \tag{4.1}$$

Now, let's consider a clique C_n whose weight is $weight(C_n) = \sum_{m \in C_n} w(m)$. Following Definition 4.2.2, a new clique C_{n+1} can be formed by adding a new memory object m_n to C_n if $\forall m_i \in C_n, \exists e \in E$ linking m_i to m_n. Consequently, m_n can not be allocated in an address range overlapping with any memory object of C_n. Hence,

$$weight(C_n \cup \{m_n\}) = weight(C_n) + w(m_n) \tag{4.2}$$

Which implies:

$$weight(C_{n+1}) = \sum_{m \in C_n} w(m) + w(m_n) = \sum_{m \in C_{n+1}} w(m) \tag{4.3}$$

Recursively, using equations 4.1 and 4.3, the theorem equation is obtained for a clique C of size $|C| \in \mathbb{N}$. $\square$

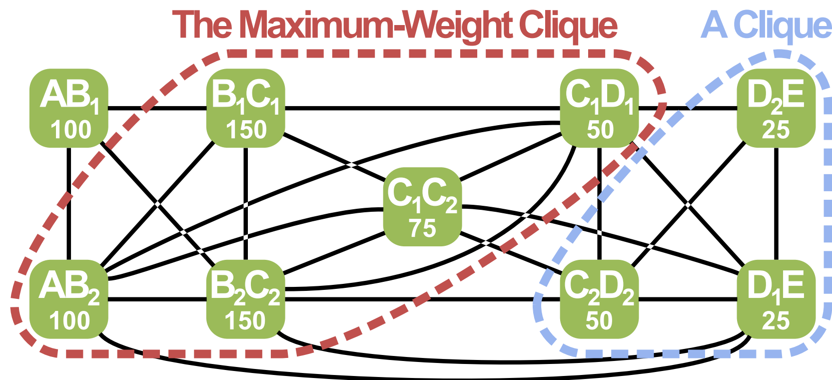


Figure 4.6: *Cliques examples*

Subsets C_a :={D$_2$E, C$_2$D$_2$, D$_1$E} and C_b :={AB$_2$,B$_1$C$_1$,B$_2$C$_2$,C$_1$C$_2$,C$_1$D$_1$} are examples of cliques in the MEG of Figure 4.6. Their respective weights are 100 and 525. Following Definition 4.3.2, a single memory object can be considered as a clique. A clique is called maximal if no memory object from its MEG can be added to it to form a larger clique. In Figure 4.6, clique C_b is maximal, but clique C_a is not as C$_1$D$_1$ is linked to all the clique memory objects and can therefore be added to the clique.

The Maximum-Weight Clique (MWC) of a graph is the clique whose weight is the largest of all cliques in the graph. Solving the MWC problem consists of finding the MWC in a MEG. Like the *Interval Coloring Problem*, the MWC problem is known to be NP-Hard. Several branch-and-bound algorithms can be found in the literature to solve the MWC problem efficiently. In [Ö01], Östergård proposes an exact algorithm which is, to our knowledge, the fastest algorithm for MEGs with an edge density $\delta(G) \leq 0.80$. For graphs with an edge density $\delta(G) \geq 0.80$, a more efficient algorithm was proposed by Yamaguchi et al in [YM08]. Both algorithms are recursive and use a similar branch-and-bound approach. Beginning with a subgraph composed of a single memory object, they search for the MWC C_i in this subgraph. Then, a memory object is added to the considered subgraph, and the weight of C_i is used to bound the search for a larger clique C_{i+1} in the new subgraph. In Section 4.3.3, the two algorithms were implemented to compare their performances on exclusion graphs derived from different applications.

The weight of the MWC corresponds to the amount of memory needed to allocate the memory objects belonging to this subset of the graph. By extension, the allocation of the whole graph will never use less memory than the weight of its MWC. Therefore, this weight is a lower bound to the memory allocation and is less than or equal to the greatest lower bound, as illustrated in Figure 4.8. In the MEG of Figure 4.4, the MWC is {AB$_2$,B$_1$C$_1$,B$_2$C$_2$,C$_1$C$_2$,C$_1$D$_1$} with a weight of 525 memory units.

Method 3 to Compute a Lower Bound: Heuristic for the Maximum-Weight Clique Problem

Östergård's and Yamaguchi's algorithms are exact algorithms and not heuristics. Since the MWC problem is an NP-Hard problem, finding an exact solution in polynomial time can not be guaranteed. For this reason, a new heuristic algorithm for the MWC problem was developed in PREESM.

The proposed heuristic approach, presented in Algorithm 4.2, is an iterative algorithm whose basic principle is to remove a judiciously selected memory object from the MEG

at each iteration. This operation is repeated until the remaining memory objects form a clique.

Algorithm 4.2: Maximum-Weight Clique Heuristic Algorithm

```
   Input: a Memory Exclusion Graph G =< M, E, w >
   Output: a maximal clique C
 1 C ← M;
 2 nb_edges ← |E|;
 3 for each m ∈ C do
 4 |  cost(m) ← w(m) + Σ_{m'∈N(m)} w(m')
 5 endfor
 6 while |C| > 1 and (2·nb_edges)/(|C|·(|C|−1)) < 1.0 do
 7 |  Select m* from C that minimizes cost(·);
 8 |  C ← C \ {m*};
 9 |  nb_edges ← nb_edges − |N(m*) ∩ C|;
10 |  for each m ∈ N(m*) ∩ C do
11 |  |  cost(v) ← cost(m) − w(m*);
12 |  endfor
13 end
14 Select a vertex m_random ∈ C;
15 for each m ∈ N(m_random) \ C do
16 |  if C ⊂ N(m) then
17 |  |  C ← C ∪ {m};
18 |  end
19 endfor
```

Our algorithm can be divided into 3 parts:

- *Initializations (lines 1-5):* For each memory object of the MEG, the *cost* function is initialized with the weight of the memory object summed with the weights of its neighbors. In order to keep the input MEG unaltered through the algorithm execution, its set of memory objects M and its number of edges $|E|$ are copied in local variables C and nb_{edges}.

- *Algorithm core loop (lines 6-13):* During each iteration of this loop, the memory object with the minimum cost m^* is removed from C (line 8). In the few cases where several memory objects have the same cost, the lowest number of neighbor $|N(m)|$ is used to determine the memory object to remove. If the number of neighbors is equal, then selection is performed based on the smallest weight $w(m)$. By doing so, the number of edges removed from the graph is minimized and the edge density of the remaining memory objects will be higher, which is desirable when looking for a clique. If there still are multiple vertices with equal properties, a random memory object is selected among them. For clarity, these criteria were not included in Algorithm 4.2.

  This loop is iterated until the memory objects in subset C become a clique. This condition is checked line 6, by comparing 1.0 (the edge density of a clique) with the edge density of the subgraph of G formed by the remaining memory objects in C. To this purpose nb_{edge}, the number of edges of this subgraph, is decremented line 9 by the number of edges in E linking the removed memory object m^* to vertices in C.

Lines 10 to 12, the costs of the remaining memory objects are updated for the next iteration.

- *Clique maximization (lines 14-19):* This last part of the algorithm ensures that the clique C is maximal by adding neighbor memory objects to it. To become a member of the clique, following Definition 4.3.2, a memory object must be adjacent to all its members. Consequently, the candidates to join the clique are the neighbors of a memory object randomly selected in C. If a memory object among these candidates is linked to all memory objects in C, it is added to the clique.

The complexity of this heuristic algorithm is of the order of $O(|N|^2)$, where $|N|$ is the number of memory objects of the MEG.

In Table 4.1, the heuristic algorithm is applied to the MEG of Figure 4.4. For each iteration, the costs associated to the remaining memory objects are given. The memory object removed during an iteration is crossed out. Column $\delta(C)$ corresponds to the edge density of the subgraph formed by the remaining memory objects. For example, in the first iteration, the memory object D_2E has the lowest cost and is thus removed from the MEG. Before beginning the second iteration, the costs of memory objects C_1D_1, C_2D_2, and D_1E are decremented by 25: the weight of the removed memory object.

		Costs								
Iter	$\delta(C)$	AB_1	AB_2	B_1C_1	B_2C_2	C_1C_2	C_1D_1	C_2D_2	D_2E	D_1E
1	0.67	500	650	625	700	600	625	375	~~150~~	475
2	0.75	500	650	625	700	600	600	~~350~~		450
3	0.81	500	650	625	650	550	550			~~400~~
4	0.87	~~500~~	625	625	625	525	525			
5	1.00		525	525	525	525	525			

Table 4.1: *Algorithm proceeding for the MEG of Figure 4.4*

In this example, the clique found by the heuristic algorithm and the exact algorithm is the same, and its weight also correspond to the size of the optimal allocation (Figure 4.7). This example proves that, as shown in Figure 4.8, the result of the heuristic can be equal to the exact solution of the MWC problem, whose size can also equal that of the optimal allocation.

Figure 4.7: *Example of optimal memory allocation for the MEG of Figure 4.4.*

Figure 4.8 summarizes the positions of the 4 bounding techniques presented in this section on the memory allocation axis. Next section presents a comparison of the performance of these 4 techniques on a set of real and randomly generated MEGs.

4.3.3 Experiments

To compare the performance and accuracy of the four bounding techniques introduced in previous sections (Figure 4.8), each of them has been implemented in PREESM. All results presented in this section are obtained by running the algorithms on an Intel Xeon E3-1225 quad-core CPU clocked at 3.1GHz.

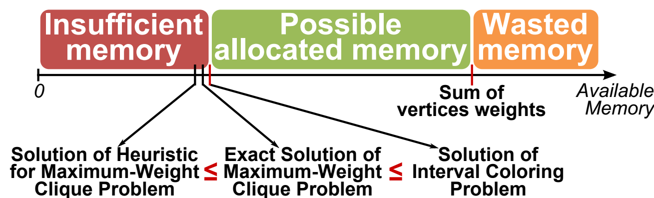


Figure 4.8: *Four techniques for the computation of memory bounds.*

MEGs properties		Exact algorithms		Heuristic	
$\|M\|$	$\delta(G)$	Östergård's [Ö01]	Yamaguchi's [YM08]	Time	Efficiency
60	0.80	0.05 s	0.25 s	0.004 s	91%
80	0.80	0.43 s	2.04 s	0.009 s	89%
100	0.80	3.4 s	11.73 s	0.014 s	87%
120	0.80	17.93 s	55.23 s	0.024 s	86%
60	0.90	0.35 s	0.56 s	0.004 s	94%
80	0.90	9.34 s	7.83 s	0.009 s	93%
100	0.90	188.00 s	90.90 s	0.016 s	91%

Efficiency: Ratio of the size of the clique found by the heuristic algorithm over the size of the MWC

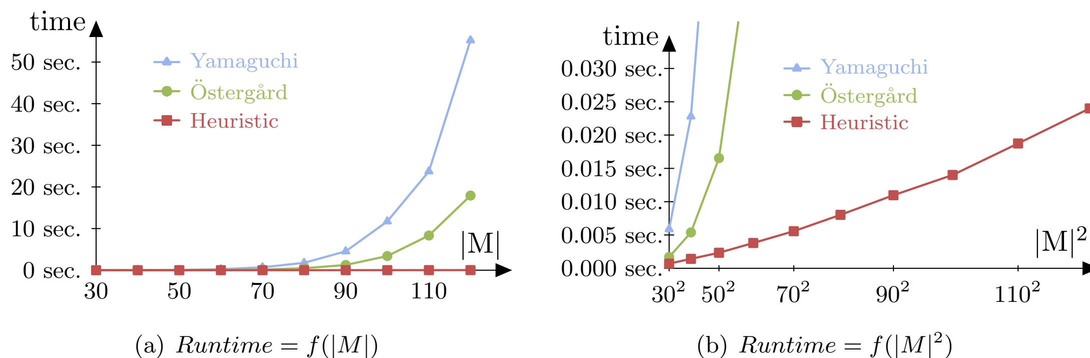
Table 4.2: *Performance of Maximum-Weight Clique algorithms on random MEGs*

Table 4.2 shows the performance of three algorithms for the MWC problem. Each entry presents the mean performance obtained from 400 randomly generated MEGs with a fixed number of memory objects ($|M|$) and a fixed density of edges ($\delta(G)$). For each MEG, the weights of its memory objects are uniformly distributed in a predefined range. The 400 graphs are generated with ranges varying from $[1000; 1010]$ to $[1000; 11000]$. Columns *Östergård's*, *Yamaguchi's* and *Time* respectively give the mean runtime of each of the three algorithms. The *Efficiency* column gives the average ratio of the clique size found by the heuristic algorithm over the size of the MWC. Results for MEG density are plotted in Figure 4.9.

It should be noted that the clique maximization part of the heuristic Algorithm 4.2 was deactivated in all tests of this section. Indeed, several tests showed that this maximization improved the mean efficiency of the heuristic algorithm by only 2%, while multiplying the runtime of the heuristic by a factor 1.6.

Table 4.2 and Figure 4.9(a) show that the runtime of exact algorithms grows exponentially with the number of memory objects of the MEGs. The runtime of exact algorithms is also highly dependent on the edge density of the graphs. On average, a change of density from 0.80 to 0.90 slows down exact algorithms by a factor 15. Conversely, as illustrated in Figure 4.9(b), the runtime of the heuristic algorithm is roughly proportional to $|M|^2$ and is not strongly influenced by the edge density of the MEGs. The results in table 4.2 also reveal that the mean efficiency of the heuristic algorithm for random MEGs is of the order of 90%, and decreases slightly as the number of vertices increases. Finally, as showed in [YM08], these results confirm that Yamaguchi's algorithm has better performance than Östergård's algorithm for graphs with more than 80 memory objects and an edge density higher than 0.80.

The performance of the three algorithms was also tested using MEGs derived from IBSDF graphs of functional applications. Table 4.3 shows the characteristics of the tested graphs. The first three entries of this table, namely *RACH*, LTE_1 and LTE_2, correspond to application graphs describing parts of the Long Term Evolution (LTE) wireless com-



(a) *Runtime* $= f(|M|)$ (b) *Runtime* $= f(|M|^2)$

Figure 4.9: *Runtime of the three MWC algorithms for random MEGs of density 0.80.*

munication standard [PAPN12]. The last two entries, *MP4P2* and *Diff*, respectively, are a description of the MPEG-4 (Moving Picture Experts Group) Part2 video encoder, and a dummy application that computes the difference between successive video frames. The values given for *Actors* and *FIFOs* are those of the flattened IBSDF graph, before its conversion into a DAG. The lower memory bounds stated in Table 4.3 correspond to the size of the MWC. Because the application models did not specify the working memory of actors, only the memory objects corresponding to the FIFOs were considered to generate the MEGs.

SDF graph			MEG		Memory Bounds	
Graph	Actors	FIFOs	$\|M\|$	$\delta(G)$	Lower	Upper
RACH	233	468	457	0.83	317 kB	752 kB
LTE_1	667	907	4240	0.72	≤3492 kB	4899 kB
LTE_2	56	84	606	0.82	451 kB	714 kB
MP4P2	143	146	143	0.80	963 kB	2534 kB
Diff	19	27	165	0.93	779 kB	1378 kB

RACH: LTE Preamble detection
LTE_1: Coarse Grain Physical+MAC Layer
LTE_2: Coarser Grain Physical+MAC Layer
MP4P2: MPEG-4 Part2 Encoder
Diff: Difference of 2 CIF pictures

Table 4.3: *Properties of the test graphs*

To take advantage of a multi-core architecture, an application modeled with an SDF graph must present a high degree of parallelism. MEGs derived from such applications will therefore have a high edge density, as is the case with the graphs of Table 4.3. The performance of each of the three algorithms on these graphs are related in Table 4.4.

	Exact algorithms		Heuristic	
Graph	Östergård's	Yamaguchi's	Time	Efficiency
RACH	∞	207.00 s	1.200 s	99.9%
LTE_1	∞	∞	869.320 s	-
LTE_2	996.70 s	219.60 s	3.300 s	100.0%
MP4P2	1.12 s	0.50 s	0.052 s	99.9%
Diff	0.42 s	0.49 s	0.120 s	100.0%

Table 4.4: *Performance of MWC algorithms on MEGs derived from the test graphs of Table 4.3*

As shown in Table 4.4, the efficiency of the heuristic algorithm for MEGs derived from real applications is much higher than for randomly generated MEGs. Indeed, the heuristic algorithm always finds a clique with a weight almost equal to the weight of the MWC. For these real application graphs, the runtime of the heuristic Algorithm 4.2 is at least 4 times faster than its exact counterparts. Moreover, contrary to the exact algorithms which sometimes fail to find a solution within 12 hours (as denoted by ∞), the runtime of the heuristic algorithm is highly predictable as it is solely dependent on the number of memory objects $|M|$ of the MEG. In the case of LTE_1, because of the large number of memory objects in the MEG, exact algorithms never ran to completion. Consequently, neither the MWC exact size nor the efficiency of the heuristic algorithm can be computed for this graph. This example shows that the heuristic algorithm may succeed in finding a lower bound to memory requirements in cases where exact approaches fail. Additionally, it can also be noted that Yamaguchi's algorithm presents a slightly better performance than Östergård's algorithm for MEGs derived from SDF graphs.

Finally, the algorithms were tested against 120 MEGs derived from randomly generated SDF graphs. The resulting exclusion graphs presented edge densities from 0.49 to 0.93 and possessed between 50 and 500 memory objects. These tests confirmed that Yamaguchi's algorithm is faster than Östergård's for exclusion graphs derived from SDF graphs. These tests also showed that the heuristic approach finds the optimal solution 81% of the time. When the optimal solution is not found, the average efficiency of the heuristic algorithm is 96.5%.

The next section describes and evaluates several allocation strategies that are based on MEGs.

4.4 Memory Allocation of a MEG

Computing memory bounds of an application provides key information to the developer of an embedded system. A rapid computation of the memory bounds, such as the one enabled by heuristic Algorithm 4.2, can also be used to rapidly assess the quality of an allocation algorithm or the result of a memory optimization process.

A MEG constructed from a non-scheduled DAG can be used as an input for a bounding process (Section 4.3), and can also serve as a basis for the allocation of the application in memory. As illustrated in Figure 4.10, a MEG can be allocated in shared memory at four different implementation stages: prior to any scheduling process, after an untimed multicore scheduling of actors, after a timed multicore scheduling of the application, or dynamically during the application execution. The scheduling flexibility resulting from the four alternatives is detailed in the following sections. The memory allocation techniques presented in this section have been published in [DPNA13].

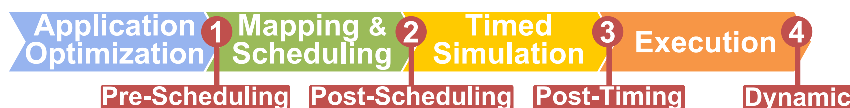


Figure 4.10: *Simplified development flow of* PREESM *with possible stages for memory allocation.*

4.4.1 MEG Updates with Scheduling and Time Information

As presented in Section 4.2, the MEG built from the non-scheduled DAG is a worst-case scenario that models all possible exclusions for all possible schedules of the application on any number of cores. If a multicore schedule of the application is known, this schedule can be used to update the MEG and lower its density of exclusions.

Post-Scheduling MEG Update

Scheduling a DAG on a multicore architecture introduces an order of execution between the actors of the graph, which is equivalent to adding new precedence relationships between actors. Adding new precedence edges to a DAG results in a decreased inherent parallelism of the application. For example, Figure 4.11 illustrates the new precedence edges that result from the dual-core schedule of the DAG presented in Figure 4.13. In this example, $Core_1$ executes actors B_1, C_1, D_1 and D_2; and $Core_2$ executes actors A, B_2, C_2 and E. A new precedence relationship between D_1 and D_2 results from the schedule of $Core_2$.

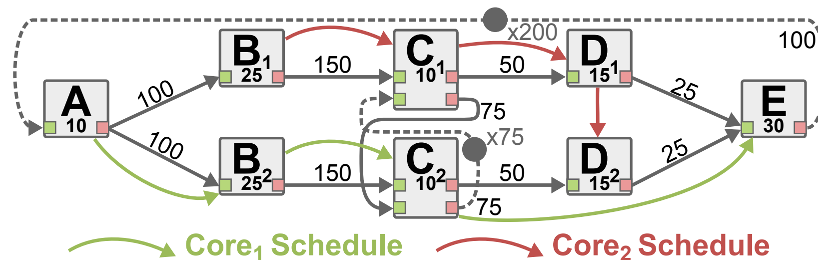


Figure 4.11: *Update of the DAG from Figure 4.3 with scheduling information.*

As exposed in Theorem 4.2.1, memory objects belonging to parallel data-paths may have overlapping lifetimes. Reducing the parallelism of an application results in creating new precedence paths between memory objects, thus preventing them from having overlapping lifetimes and removing exclusions between them. Since all the parallelism embedded in a DAG is explicit, the scheduling process cannot augment the parallelism of an application and cannot create new exclusions between memory objects. In Figure 4.12, updating the MEG with the multicore schedule of Figure 4.11 results in removing the exclusion between C_1D_1 and D_2E.

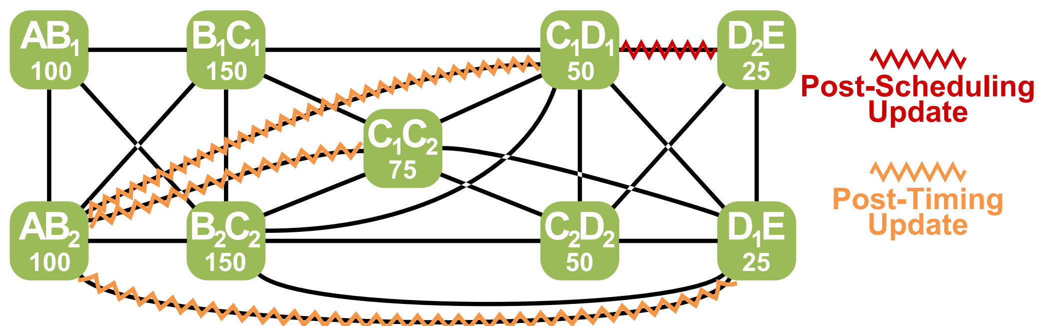


Figure 4.12: *MEG updated with scheduling information from Figure 4.11 and 4.13.*

Post-Timing MEG Update

A second update of the MEG is possible if a timed schedule of the application is available. A timed schedule is a schedule where not only the execution order of the actors is fixed, but also their absolute starting and ending times. Such a schedule can be derived if, for example, the exact or the worst-case execution times of all actors are known at compile time [PAPN12]. Updating a DAG with a timed schedule consists of adding new precedence edges between all actors with non-overlapping lifetimes.

Following Theorem 4.2.2, the lifetime of a memory object begins with the execution start of its producer, and ends with the execution end of its consumer. In the case of working memory, the lifetime of the memory object is equal to the lifetime of its associated actor because, contrary to the stack of a thread, working memory can be discarded between two actor firings. Using a timed schedule, it is thus possible to update a MEG so that exclusions remain only between memory objects with overlapping lifetimes. For example, the timed schedule of Figure 4.13 introduces a precedence relationship between actors B_2 and C_1 which translates into removing from the MEG exclusions between memory objects AB_2 and: C_1D_1, C_1C_2, and D_1E.

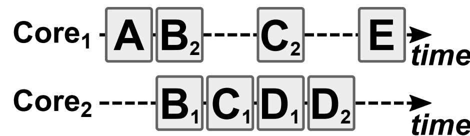


Figure 4.13: *Example of timed schedule for the DAG of Figure 4.3.*

Each version of the MEG obtained during the successive updates with scheduling and timing information can be used as a basis for the allocation process. The next section details the advantages and drawbacks offered by the different versions of the MEG for its memory allocation.

4.4.2 Memory Allocation Strategies

During the development of an application modeled by an IBSDF graph, three versions of its corresponding MEG are obtained at three stages of the implementation process: prior to the scheduling process, after an untimed multicore schedule is decided, and after a timed multicore schedule is decided. Each of these three alternatives offers a distinct trade-off between the amount of allocated memory and the flexibility of the application multicore execution.

Pre-Scheduling Allocation

A compile-time memory allocation of a pre-scheduling MEG will never violate any exclusion for any multicore schedule of its original IBSDF graph on any shared-memory architecture. Indeed, before scheduling the application, the MEG models all possible exclusions that may prevent memory objects from being allocated in overlapping memory spaces (Definition 4.2.2). Hence, a pre-scheduling MEG models all possible exclusions for all possible multicore schedules of its corresponding application IBSDF graph.

Since a compile-time memory allocation based on a pre-scheduling MEG is compatible with any multicore schedule, it is compatible with any runtime schedule. The great flexibility of this first allocation approach is that it supports any runtime scheduling policy for the DAG and can accommodate any number of cores that can access a shared memory.

A typical scenario where this pre-scheduling compile-time allocation is useful is the concurrent execution of multiple applications on a multicore architecture. In such a scenario, the number of cores used for an application may change at runtime to accommodate applications with high priority or those with high processing needs. In this scenario using a compile-time allocation relieves runtime management from the weight of a dynamic allocator while guaranteeing a fixed memory footprint for the application.

The downside of this first approach is that, as will be shown in the results of Section 4.4.3, this allocation technique requires substantially more memory than post-scheduling allocators.

Post-Scheduling Allocation

Post-scheduling memory allocation offers a trade-off between amount of allocated memory and multicore scheduling flexibility. The advantage of post-scheduling allocation over pre-scheduling allocation is that updating the MEG greatly decreases its density which results in using less allocated memory.

Like pre-scheduling memory allocation, the flexibility of post-scheduling memory allocation comes from its compatibility with any schedule obtained by adding new precedence relationships to the scheduled DAG. Indeed, adding new precedence edges will make some exclusions useless but it will never create new exclusions. Any memory allocation based on the updated MEG of Figure 4.11 is compatible with a new schedule of the DAG that introduces new precedence edges. For example, the post-scheduling MEG of Figure 4.12 can be updated with a single core schedule combining schedules of $Core_1$ and $Core_2$ as follows A, B_2, B_1, C_1, C_2, D_1, D_2 and E. Updating the MEG with this schedule would result in removing the exclusions between memory objects AB_2 and: B_1C_1, C_1C_2, C_1D_1, and D_1E.

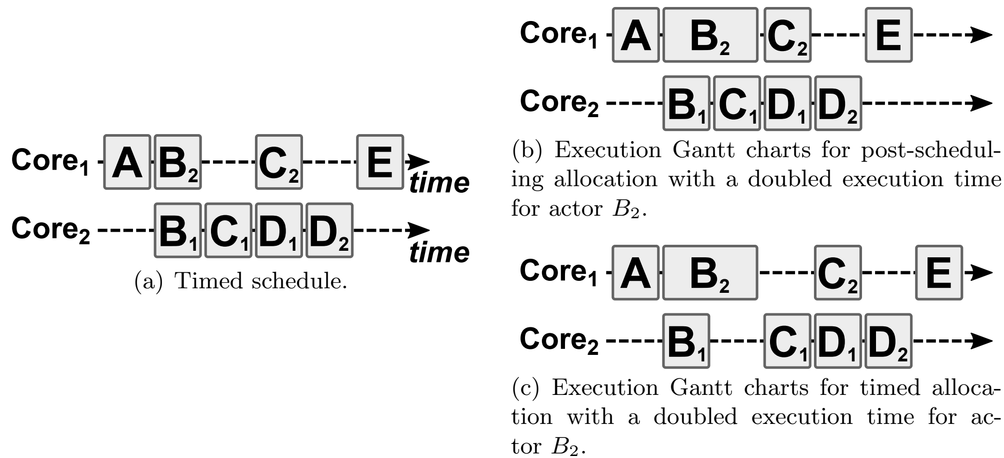
The scheduling flexibility for post-scheduling allocation is inferior to the flexibility offered by pre-scheduling allocation. Indeed, the number of cores allocated to an application may be only decreased at runtime for post-scheduling allocation while pre-scheduling allocation allows the number of cores to be both increased and decreased at runtime.

Post-Timing Allocation

A MEG updated with a timed schedule has the lowest density of the three alternatives, which leads to the best results in terms of allocated memory size. However, its reduced parallelism makes it the least flexible scenario in terms of multicore scheduling and runtime execution.

Figure 4.14 illustrates the possible loss of flexibility resulting from the usage of post-timing allocation. In the timed schedule of Figure 4.14(a), the same memory space can be used to store memory objects AB_2 and C_1D_1 since actor B_2 execution ends before actor C_1 execution starts. In Figures 4.14(b) and 4.14(c), the execution time of actor B_2 is double that of the timed schedule. With timed allocation (Figure 4.14(c)), the execution of C_1 must be delayed until B_2 completion; otherwise actor C_1 might overwrite and corrupt data of the AB_2 memory object. With post-scheduling allocation (Figure 4.14(b)), only the actor order on each core must be guaranteed. Actor C_1 can thus start its execution before actor B_2 completion since memory objects AB_2 and C_1D_1 exclude each other in the corresponding MEG (Figure 4.12).

This example illustrates why timed allocation is a bad option for implementation of an application on a non-deterministic time MPSoC. Indeed, using post-timing allocation on such an architecture requires the addition of synchronization points after each actor



(a) Timed schedule.

(b) Execution Gantt charts for post-scheduling allocation with a doubled execution time for actor B_2.

(c) Execution Gantt charts for timed allocation with a doubled execution time for actor B_2.

Figure 4.14: *Loss of runtime flexibility with timed allocation for DAG of Figure 4.11.*

completion to ensure that the exact same order of execution is always respected. These synchronization points would cause a large overhead for the execution of the application that would greatly decrease the performance of the system.

Although timed allocation provides the smallest memory footprints (Section 4.4.3), its lack of runtime flexibility makes it a bad option for implementation on MPSoCs with non-deterministic behavior because of caches, pipelined ALUs, or non-deterministic buses. Nevertheless, computing the memory bounds for a MEG updated with a timed schedule is a convenient way to approximate the memory footprint that would be allocated by a dynamic allocator. Indeed, like timed allocation, dynamic allocation allows memory reuse as soon as the lifetime of a memory object is over.

Next section illustrates the memory allocation efficiency offered by the three implementation stages.

4.4.3 Experiments

The allocation of MEGs at the four implementation stages presented in Figure 4.10 was tested within the PREESM rapid prototyping framework.

Memory Allocators

For each stage, the following allocation algorithms were tested to allocate the MEGs in memory:

- **First-Fit (FF) algorithm:** the FF algorithm consists of allocating each memory object to the first available space in memory that is sufficiently large to store it [Joh73].

- **Best-Fit (BF) algorithm:** The BF algorithm works similarly to the FF algorithm but allocates each memory object to the available space in memory whose size is the closest to the size of the allocated object [Joh73].

- **Placement algorithm:** The placement algorithm was introduced by DeGreef et al. in [DGCDM97]. In the placement algorithm, each memory object is associated to an offset that is set to 0 at the beginning of the allocation process. The memory objects to allocate are stored in a list that is sorted according to the offsets associated to

the memory objects. Iteratively, the placement algorithm checks if the head memory object of the list can be allocated in memory at its associated offset. If no exclusion forbids this allocation, the memory object is allocated and removed from the sorted list. Otherwise, the offset associated to the memory object is updated to the first valid offset possible, and the memory object is put back in the sorted list.

The FF and the BF algorithms are *online* allocators that allocate memory objects in order in which they are received. Three ordering techniques were tested to feed these online allocators:

- **Largest-first order:** In the largest-first order, memory objects are allocated in decreasing order of size.
- **Stable-sets order:** The stable-sets order is inspired by interval coloring techniques [BAH09]. The memory objects are first grouped into stables: sets of memory objects within which no pair of objects exclude each other. Then, one by one, the stables are allocated in the largest-first order by the online allocator. To compose the stables, heuristic Algorithm 4.2 is used to find cliques in the complement graph of the MEG. Indeed, as presented in [KS02], if a set of memory objects forms a clique in the complement graph of a MEG, this same set of memory objects forms a stable in this MEG.
- **Scheduling order:** Possible only for post-scheduling and timed allocations, the allocation of the memory objects in scheduling order consists of feeding the online allocators with memory objects sorted in order in which they are created in the schedule. Using the scheduling order is equivalent to using an online allocator at runtime.

Test Graphs

The different allocation algorithms and implementation stages were tested on a set of MEGs derived from IBSDF and SDF graphs of real applications. Table 4.5 shows the characteristics of the tested graphs.

	Single-rate SDF graph		Memory Exclusion Graph (MEG)					
Nr	Graph	Actors	FIFOs	$\|M\|$	δ_{pre}	δ_{sch}	δ_{tim}	B_{max}
1	MPEG4 Enc.	74	143	143	0.80	0.60	0.50	2475 kB
2	H263 Enc.*	207	402	603	0.98	0.76	0.50	5115 kB
3	MP3 Dec.*	33	44	71	0.64	0.55	0.31	354 kB
4	PRACH	308	897	897	0.94	0.67	0.56	4413 kB
5	Sample Rate	624	1556	1289	0.50	0.22	0.03	1.601 kB

*: Actors of this graph have working memory

Table 4.5: *Properties of the test graphs*

The first three entries of this table, namely *MPEG4 Enc.*, *H263 Enc.*, and *MP3 Dec.*, model standard multimedia encoding and decoding applications. The *Sample Rate* graph models an audio sample rate converter. The *MPEG4 Enc.*, *H263 Enc.* and *Sample Rate* graphs were taken from the SDF3 example database [Ele13]. The *PRACH* graph models the preamble detection part of the LTE telecommunication standard [PAPN12]. Information on the working memory of actors was only available for the *MPEG4 Enc.* and the *MP3 Dec.* graphs. In Table 4.5, columns δ_{pre}, δ_{sch}, and δ_{tim} respectively correspond

to the density of the MEG in the pre-scheduling, post-scheduling, and timed stages. B_{max} gives the upper bound for the memory allocation of the applications.

To complete these results, the allocation algorithms were also tested on 45 random SDF graphs. The random SDF graphs were randomly generated with SDF3 tool [Ele13]. The corresponding 45 MEGs cover a wide range of complexities with a number of memory objects $|M|$ ranging from 47 to 2208, and exclusion densities δ ranging from 0.03 to 0.98.

Experimental Results

For each implementation stage, a table presents the performance of the allocators for each application. Performance is expressed as a percentage corresponding to the amount of memory allocated by the algorithm compared to the smallest amount of memory allocated by all algorithms. So, 0% means that the algorithm determined the best allocation. A positive percentage value indicates the degree of excess memory allocated by an allocator compared to the value of the *Best Found* column. The B_{min} column gives the lower memory bound found using the heuristic presented in Section 4.3.

For each implementation stage, a box-plot diagram presents performance statistics obtained with all 50 graphs. For each allocator, the following information is given: the leftmost and rightmost marks are the best and worst performance achieved by the allocator, left and right sides of the rectangle respectively are inferior and superior to 75% of the 50 measured performances, and the middle mark of the rectangle is the median value of the 50 measured performances.

Pre-Scheduling Allocation

			First-Fit (FF)		Best-Fit (BF)		
Nr	Best	B_{min}	LF	SS	LF	SS	PA
1	988470	-3%	+3%	+15%	0%	+15%	+13%
2	2979776	0%	+1%	+0%	+1%	+5%	0%
3	144384	-3%	+2%	+5%	+2%	+5%	0%
4	2097152	0%	0%	+5%	+1%	+5%	+29%
5	347	-17%	0%	+8%	+4%	+18%	+18%

LF: Algorithm fed in largest-first order SS: Algorithm fed in stable-sets order

PA: Placement Algorithm from [DGCDM97]

Table 4.6: *Pre-scheduling memory allocation*

Performances obtained for pre-scheduling allocation are displayed in Table 4.6 and Figure 4.15. These results clearly show that the FF algorithm fed in the largest-first order tends to generate a smaller footprint than the other algorithms. Indeed, the *FF-LF* algorithm finds the best allocation for 29 of the 50 tested graphs. When it fails to find the best solution, it assigns only 1.35% more memory, on average, than the *Best Found* allocation. Moreover, the solution of the *FF-LF* is 4% superior, on average, to the lower bound B_{min}.

Since the pre-scheduling allocation is performed at compile-time, it is possible to execute all allocation algorithms and keep only the best results. Indeed, in the 50 tests, the BF allocator fed in largest-first order found the best solution for 13 graphs and the placement algorithm for 12 graphs.

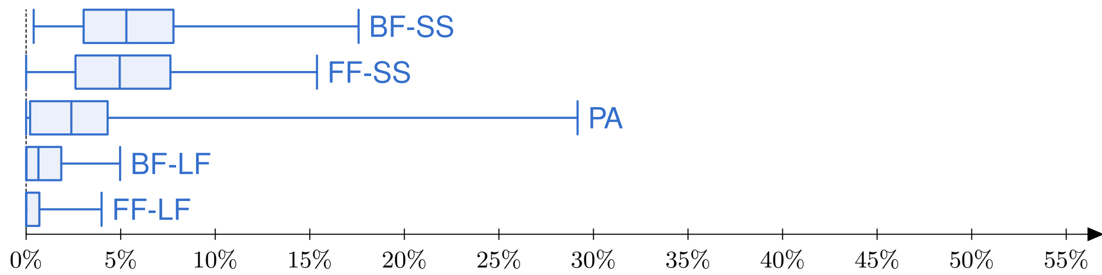


Figure 4.15: *Performance of pre-scheduling allocation algorithms for 50 graphs.*

Post-scheduling Allocation

			First-Fit (FF)			Best-Fit (BF)			
Nr	Best	B_{min}	LF	SS	Sch.	LF	SS	Sch.	PA
1	861726	-3%	0%	+18%	+6%	+3%	+18%	+6%	+9%
2	1570240	-37%	+0%	0%	+5%	+2%	0%	+17%	0%
3	117184	-1%	+8%	+8%	+34%	+8%	+8%	+55%	0%
4	1365906	-16%	0%	+19%	+51%	+11%	+26%	+51%	+13%
5	185	-2%	+1%	+2%	0%	0%	+2%	+4%	+5%

LF: Algorithm fed in largest-first Order SS: Algorithm fed in stable-sets order
PA: Placement Algorithm from [DGCDM97] Sch.: Algorithm fed in Scheduling Order

Table 4.7: *Post-scheduling memory allocation*

Table 4.7 and Figure 4.16 present the performance obtained for post-scheduling allocation on a multicore architecture with 3 cores. Because the 50 graphs have very different degrees of parallelism, mapping them on the same number of cores decreases their parallelism differently and results in a wide variety of test cases.

On average, updating MEGs with scheduling information reduces their exclusion density by 39% which in turns leads to a diminution of the amount of allocated memory by 32%.

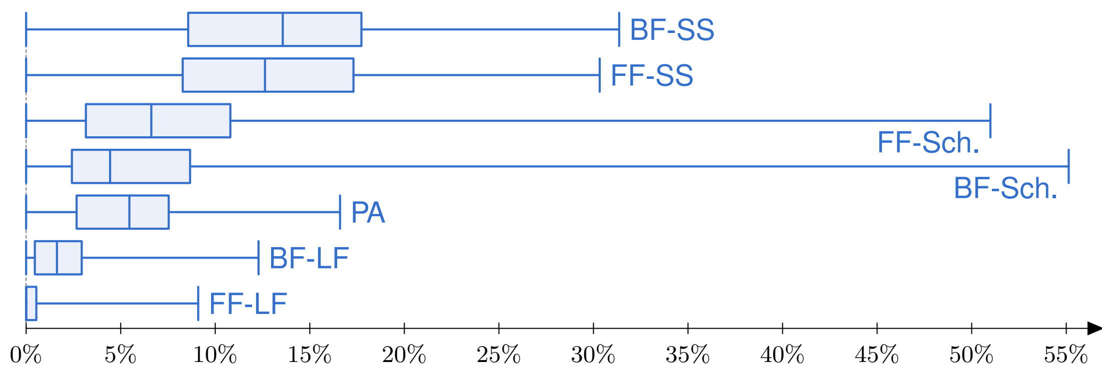


Figure 4.16: *Performance of post-scheduling allocation algorithms for 50 graphs.*

As for pre-scheduling allocation, the *FF-LF* is the most efficient algorithm as it finds the best allocation for 29 of the 50 graphs. For the 21 remaining graphs, it allocates on average 1.74% more memory than the *Best Found* solution. Although the average performance of algorithms fed in scheduling order is better than those of the placement

algorithm or the stable-sets order, these algorithms generate allocations up to 56% larger than the best solution. Contrary to other allocators, these online algorithms do not exploit global knowledge of all memory objects and simply allocate them in the raw scheduling order. This allocation order sometimes results in a greater fragmentation of the memory, which in turn leads to larger memory footprints.

Timed allocation

Performances obtained for timed allocation are presented in Table 4.8 and Figure 4.17. The *FF*-*LF* allocator is once again the most efficient algorithm as it finds the best allocation for 31 of the 50 graphs, including all 5 real applications.

			First-Fit (FF)			Best-Fit (BF)			
Nr	Best	B_{min}	LF	SS	Sch.	LF	SS	Sch.	PA
1	760374	-0%	0%	+13%	+13%	+0%	+13%	+13%	+13%
2	1243072	-0%	0%	+2%	+28%	+10%	+3%	+48%	+3%
3	111008	-3%	0%	+8%	+3%	0%	+8%	+3%	+1%
4	1231968	-8%	0%	+9%	+17%	+14%	+13%	+22%	+26%
5	41	-10%	0%	+7%	+5%	+5%	+7%	+2%	+17%

LF: Algorithm fed in largest-first Order SS: Algorithm fed in stable-sets order
PA: Placement Algorithm from [DGCDM97] Sch.: Algorithm fed in Scheduling Order

Table 4.8: *Timed memory allocation*

The online allocators fed in scheduling order assign more memory than the *FF*-*LF* algorithm for 38 graphs with up to 48% more memory being assigned. In the timed stage, allocators fed in scheduling order assign only 7% less memory, on average, than the *FF*-*LF* algorithm in the post-scheduling stage. In 5 cases, including the *PRACH* application, online allocators assign even more memory in the timed stage than which was allocated by the *FF*-*LF* algorithm in the post-scheduling stage. Considering the $O(nlogn)$ complexity of these online allocators [Joh73], where n is the number of allocated memory objects, using the post-scheduling allocation is an interesting alternative. Indeed, using post-scheduling allocation removes the computation overhead of dynamic online allocation while guaranteeing a fixed memory footprint slightly superior to that which could be achieved dynamically. This problem is illustrated in the case study presented in Chapter 6.

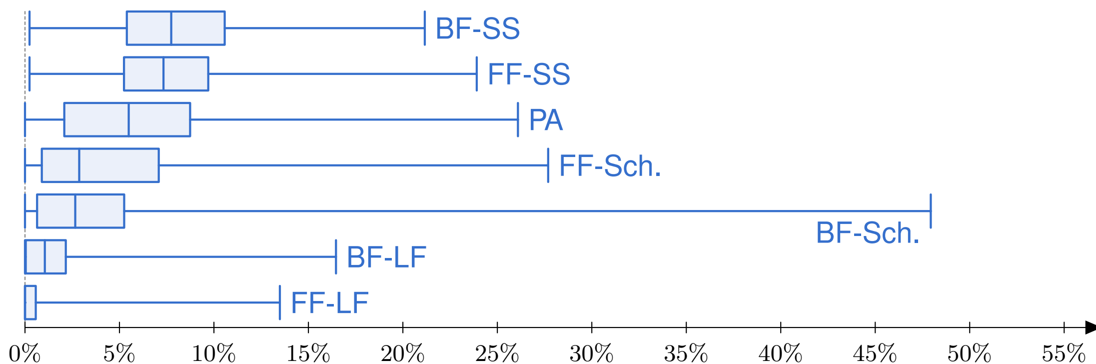


Figure 4.17: *Performance of timed allocation algorithms for 50 graphs.*

On average, the *Best Found* timed allocation uses only 11% less memory than the post-scheduling allocations and only 2.7% more memory than the minimum bound B_{min}.

Considering the small gain in footprint and the loss of flexibility induced by this implementation stage (Section 4.4.2), timed allocation appears to be a good choice only for systems with restricted memory resources where flexibility is not important. However, for systems where the memory footprint is important, but scheduling flexibility is also desired, the post-scheduling allocation offers the best trade-off. Finally, for systems where a strong flexibility is essential, the pre-scheduling allocation offers all the required parallelism while ensuring a fixed memory footprint.

4.5 Discussion

4.5.1 Approach Limitations

Using a MEG produced by Algorithm 4.1 as a basis for the memory analysis and optimization techniques presents some limitations:

- **MEG-based optimization techniques do not allow the concurrent execution of successive graph iterations.** When building a MEG, the lifetime of memory objects is bounded by the span of a graph iteration. Indeed, one of the objectives of the transformation of the original IBSDF graph into a DAG is to isolate one iteration of the algorithm. Since MEGs are built from DAGs, the precedence relationships causing exclusions between memory objects are thus bounded by the span of a DAG iteration. To make-up for this limitation, as presented in [LM87b], delays can be used in a dataflow graph as a way to pipeline an application. By doing so, the developer can divide a graph into several unconnected graphs whose iterations can be executed in parallel, thus improving the application throughput. From the memory perspective, pipelining a graph will increase the graph parallelism and consequently the number of exclusions in its MEG and the amount of memory required for its allocation.

- **Transformation of an SDF graph into its equivalent single-rate SDF can be exponential in terms of number of actors [PBL95].** As a consequence, the proposed analysis and allocation technique should only be applied to SDF graphs with a relatively coarse grained description: graphs whose single-rate equivalent have at most hundreds of actors and thousands of single-rate Fifos. Despite this limitation, the single-rate transformation has proven to be efficient for many real applications, notably in the telecommunication [PAPN12] and the multimedia [DPNA13] domains.

4.5.2 Comparison with a Fifo dimensioning technique

As presented in Chapter 3, Fifo dimensioning is currently the most widely used technique to minimize the memory footprint of applications modeled with a dataflow graph [Par95, SGB06, MB10]. Fifo dimensioning techniques consist of finding a schedule of the application that minimizes the memory space allocated to each Fifo of the SDF graph. The main drawback of Fifo dimensioning techniques is that, contrary to the MEG allocation technique, they do not consider memory reuse since each Fifo is allocated in a dedicated memory space.

Table 4.9 compares allocation results of a Fifo dimensioning algorithm with the reuse technique based on a MEG allocation for 4 application graphs. The Fifo dimensioning technique tested is presented in [SGB06] and its implementation is part of the SDF For

Free (SDF3) framework [Ele13]. The *stereo* graph is the application studied in Chapter 6. The *H263 Enc.* graph is a video encoder taken from the SDF3 database [Ele13]. The *sobel* and *chaotic* graphs are a sobel video filtering application and a generator of chaotic sequences inspired by [EAN13].

Graph	Upper Bound[1]	Pre-sched.[1]	Post-sched.[1]	FIFO dim.
stereo	+109%	+20%	-15%	0%
h263 enc.	+116%	-1%	-17%	0%
sobel	+46%	-43%	-43%	0%
chaotic	+222%	+77%	+33%	0%

1: Percentages are relative to the FIFO dimensioning result.

Table 4.9: *Comparison of MEG allocation results with FIFO dimensioning technique from SDF3 [Ele13].*

Results presented in Table 4.9 consist of: the upper bound for the allocation of the MEG in memory (Section 4.3), the MEG allocation result in the pre-scheduling and post-scheduling implementation stages, and the result of the FIFO dimensioning technique. For each application, the results are expressed as percentages relative to the FIFO dimensioning case which is marked with 0%.

For the first 3 graphs, the post-scheduling scenario of MEG allocation offers the best results, with memory footprints up to 43% lower than the FIFO dimensioning technique. The FIFO dimensioning technique itself offers memory footprints on average 51% lower than the computed upper bound.

In Table 4.9, the FIFO dimensioning technique provides the best result for the *chaotic* graph, with 25% less memory than the post-scheduling memory allocation of the MEG. This result reveals two current limitations of the MEG allocation.

- **Bad handling of fork/join operations.** During the single-rate transformation presented in Section 4.2, special *fork* and *join* actors are introduced to replace FIFOs with unequal production and consumption rates. These actors are responsible for dividing a memory object produced (or consumed) by an actor into subparts consumed (or produced) by other actors. Since the divided memory object and its subparts are input and output buffers of a single special actor, they exclude each other in the MEG and their allocation requires twice the size of the divided memory object in memory. This issue is not present in the FIFO dimensioning technique since buffer division is naturally implemented by successive data token reads in FIFOs. The numerous fork and join operations of the single-rate *chaotic* graph are thus responsible for its higher memory footprint. A solution to this issue is presented in Chapter 5.

- **Absence of memory-aware scheduling process.** As presented in Chapter 3, FIFO dimensioning techniques consist of finding the schedule of the application that minimizes the memory space allocated to each FIFO of the graph. In PREESM, the aim of the scheduling process is to minimize the latency of the schedule, independent of memory allocation concerns. This policy often results in bad choices from the memory perspective, as is the case for the *chaotic* application where several actors producing large memory objects are executed before any of the large memory objects are consumed. With FIFO dimensioning techniques, the consuming actor of the large memory object would be scheduled immediately after its producer.

In this chapter, a new method was introduced to study and optimize the memory allocation of applications modeled with IBSDF graphs. In this method, minimization of

the memory footprint is achieved by exploiting the memory reuse opportunities derived from dataflow graphs. In the next chapter, an abstract model is introduced to specify memory reuse opportunities between ports of an actor, and enable a further reduction of the memory footprint of applications.

CHAPTER 5

Actor Memory Optimization: Studying Data Access to Minimize Memory Footprint

5.1 Introduction

As presented in Chapter 2, dataflow MoCs are high-level Models of Computation (MoCs) that do not specify the internal behavior of actors. Hence, memory optimization techniques presented in Chapter 4 only rely on an analysis of the high-level data dependencies specified in IBSDF graphs. The memory reuse opportunities exploited by these optimization techniques is limited by the high abstraction level considered and by the lack of information about the internal data dependencies of actors.

In this chapter, a new set of annotations for dataflow graphs is introduced to allow the developer of an application to specify memory reuse opportunities between input and output buffers of an actor. Section 5.2 details the numerous memory reuse opportunities that can be revealed by considering internal data dependencies of dataflow actors. Then Section 5.3 presents related work on this topic. The new graph annotations enabling the description of actor-level memory reuse opportunities are introduced in Section 5.4. Finally, Section 5.5 shows how these new memory reuse opportunities can be exploited to reduce the memory footprint allocated for IBSDF graphs.

5.2 Motivation: Memory Reuse Opportunities Offered by Actor Internal Behavior

A major objective of the PREESM rapid prototyping framework is to generate executable code for shared memory MPSoCs from an IBSDF model of applications. To achieve this goal, PREESM automatically performs the mapping and scheduling of the IBSDF actor firings and the memory allocation of the IBSDF FIFOS (Section 3.3). Then, the code generation task of PREESM translates mapping and scheduling choices and memory allocation into compilable code for the targeted architecture.

In the generated code, the firing of an actor on a processing element of the target architecture is translated into a call to the function implementing the actor behavior for this processing element. To enable the correct execution of actors, each function call must be parameterized with references to the memory allocated for its inputs and outputs.

Listing 5.1 gives an example of a function prototype that can be called in the generated C code. As shown in this example, in C code, references to the memory allocated for input and output buffers are given by pointers.

```
void sobel(int height, int width, char *in, char *out);
```

Listing 5.1: *Sobel actor prototype in C code*

To simplify the description of the internal behavior of actors, and for performance reasons, it is convenient to assume that the memory allocated to each input or output buffer of an actor constitutes a contiguous memory space. By using contiguous memory spaces, the developer of an application avoids the multiple jumps in memory that would have a negative impact on the system performance. By doing so, the developer also avoids the writing of complex pointer operations that would decrease the source code readability [CDG+14].

From the PREESM framework perspective, actors firings are thus seen as function calls accessing indivisible buffers. As presented in Chapter 4, since PREESM has no knowledge of the internal behavior of actors, the worst-case scenario must be assumed. The memory allocated for input and output buffers of actors is thus reserved for the complete execution time of actors and each input and output buffer must be allocated in a separate memory space.

To allow memory reuse between input and output buffers of actors, new information must be provided to PREESM about the internal behavior of actors and how they access the data contained in their input and output buffers. Next section presents the memory reuse opportunities offered by IBSDF graphs through the example of an image processing application presented in Figure 5.1

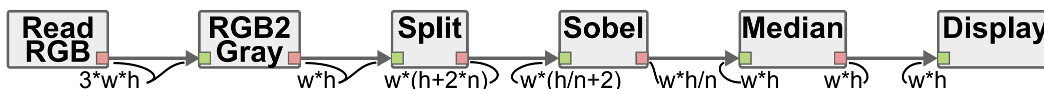


Figure 5.1: *Image processing SDF graph*

5.2.1 Actors Inserted During Graph Transformations

As presented in [PAPN12] and in Section 4.2.1, a preliminary step to the mapping/scheduling process and the allocation process is the application of a set of transformations to the IBSDF graph of an application. As presented in Section 3.3.2, the hierarchy flattening and the single-rate transformations are successively applied to expose the application characteristics implicitly embedded in its IBSDF model.

During these graph transformations, new actors are introduced to ensure the equivalence with the original IBSDF graph. Figure 5.2 illustrates the new *Fork* and *Join* actors introduced during the single-rate transformation of the SDF graph of Figure 5.1.

Figure 5.3 illustrates the Memory Exclusion Graph (MEG) derived from the input and output buffers of the actors introduced during graph transformations.

Fork/Join Actors

The MEG associated to a *Fork* actor is illustrated in Figure 5.3(d). It is composed of $n+1$ memory objects, where n is the number of FIFOS connected to the output ports of the actor. Since these memory objects form a clique (Definition 4.3.2), they must be

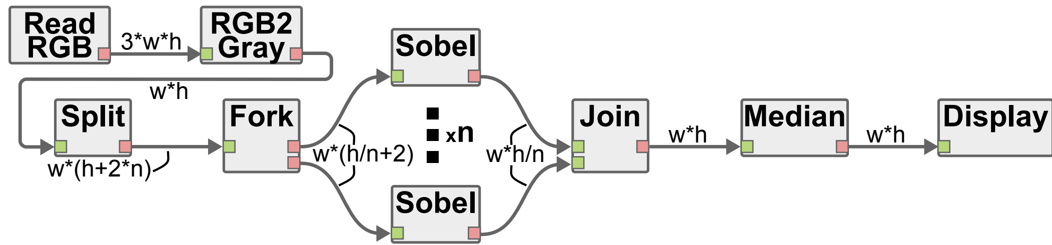


Figure 5.2: *Single-rate SDF graph derived from the SDF graph of Figure 5.1.*

allocated in non-overlapping memory spaces, which require twice as much memory as the *size* of the input buffer.

As illustrated in Figure 5.3(a), the purpose of a *Fork* actor is to distribute the tokens received on its input port on the FIFOs connected to its output ports. Hence, the internal behavior of the *Fork* actor simply consists of copying a part of the data from its input buffer to each of its output buffers.

An obvious way to take advantage of this internal behavior is to force the allocation of output buffers directly in the corresponding subranges of the input buffer memory. Applying this optimization will result in a reduction by 50% of the memory footprint of *Fork* actors.

A similar optimization is possible with *Join* actors, by allocating input buffers directly in corresponding subranges of the output buffer memory.

Broadcast Actors

The MEG associated to a *Broadcast* actor is illustrated in Figure 5.3(e). It is composed of $n + 1$ memory objects, where n is the number of FIFOs connected to the output ports of the actor. Since these memory objects form a clique, they must be allocated in non-overlapping memory spaces, which require $n+1$ times as much memory as the *size* of the input buffer.

As illustrated in Figure 5.3(b), the purpose of a *Broadcast* actor is to produce a copy of the tokens received on its input port on each FIFO connected to its output ports. Hence, the internal behavior of the *Broadcast* actor consists of copying all the data from its input buffer into each output buffer.

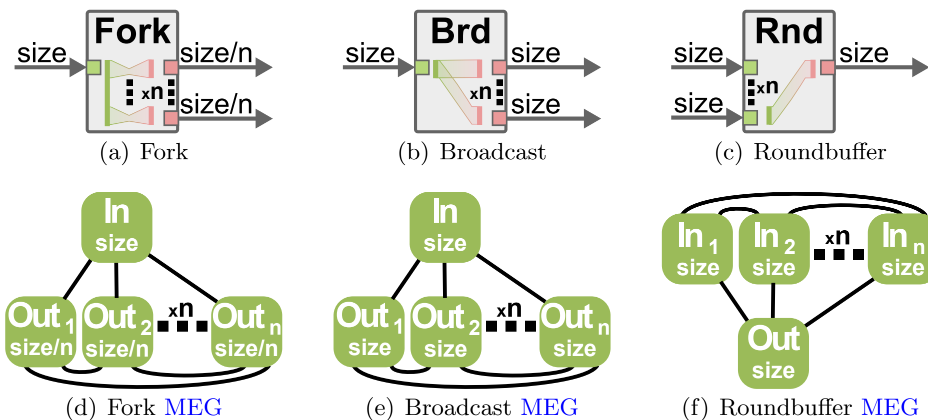


Figure 5.3: *Automatically inserted actors and corresponding MEG. Diagram within actors represent the copy operations realized by the actor.*

To take advantage of this internal behavior the allocation of output buffers can be matched directly in the input buffer memory. By doing so, the memory footprint of *Broadcast* actors is divided by a factor $n + 1$.

The allocation of the output buffers of a *Broadcast* actor directly within the input buffer requires additional precautions. Indeed, if this allocation is used, all actors receiving a copy of the broadcasted buffer will access the same memory space. To ensure a correct functioning of the application, the developer must make sure that actors accessing the broadcasted data do not write anything in the shared memory space. If such an actor exists, it must be given a private copy of the broadcasted data, allocated in a distinct memory space.

Roundbuffer Actors

The MEG associated to a *Roundbuffer* actor is illustrated in Figure 5.3(f). Like the MEG of the *Broadcast* actor, its allocation requires $n + 1$ times as much memory as the *size* of the input buffer, where n is the number of input ports of the actor.

As illustrated in Figure 5.3(c), the purpose of a *Roundbuffer* actor is to forward on its output port only the last tokens received on its input ports. Hence, the internal behavior of the *Roundbuffer* actor consists of copying the last tokens consumed on its input buffers to its output buffer.

To take advantage of this internal behavior the allocation of the last input buffer can be matched directly in the output buffer memory. Moreover, since only the last data tokens consumed by a *Roundbuffer* actor are forwarded to the output port, all other data tokens consumed by this actor are useless. Since the content of these buffers is not used, they can be allocated in overlapping memory spaces. By doing so, the memory footprint of a *Roundbuffer* actor can be reduced to a footprint only twice as large as the *size* of the output buffer.

Besides reducing the memory footprint of the *Fork*, *Join*, *Broadcast*, and *Roundbuffer* actors, these optimizations will also improve the performance of applications since copying data from input to output buffers is no longer required. The memory reuse opportunities presented in this section can be applied systematically since the concerned actors are "special" actors whose behaviors are not defined by the application developer. The next section presents memory reuse opportunities resulting from the internal behavior of user-defined actors.

5.2.2 Actors with User-Defined Behavior

The description of the internal behavior of a dataflow actor is not part of the SDF MoC. Consequently, many SDF programming frameworks have no knowledge of the internal behavior of actors [PAPN12, Ele13].

Last section showed that advanced memory optimization is possible for input and output buffers of "special" actors whose behaviors are not user-defined. Although their behavior is defined by the application developer, user-defined actors also present internal behaviors that favor the use of advanced memory reuse techniques.

In-place Algorithms

In-place algorithms is a first example of data access pattern that a developer might want to use to optimize the memory footprint of his application [HKB05].

An in-place algorithm is an algorithm that can write its results directly in its input buffer during its execution. Beside the input buffer, the execution of an in-place algorithm requires a small constant working memory whose size is independent of the size of the processed data. Examples of in-place signal processing algorithms can be found in the literature such as FFT algorithms [BE81], and sorting algorithms [GG11].

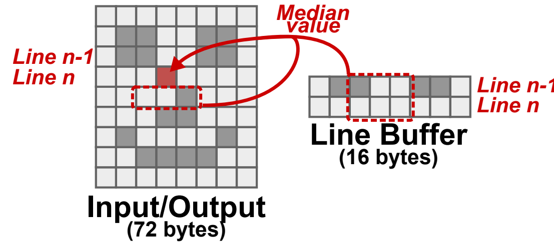


Figure 5.4: Median *actor internal behavior with in-place results.*

Figure 5.4 illustrates the in-place behavior of the *Median* actor from the SDF graph of Figure 5.2. The *Median* actor purpose is to replace each pixel with the median value of its 3x3 pixels neighborhood. Its implementation is based on a sliding-window of 3x3 pixels that successively scans the lines of the filtered image [Tex08].

To compute the filtered value of a pixel, the median filter must keep in memory the original values of the 8 pixels around it. If the result of the filter is directly written in the original image, the algorithm behavior will be corrupted since the new value assigned to a pixel will be used for the computation of its neighbor pixels.

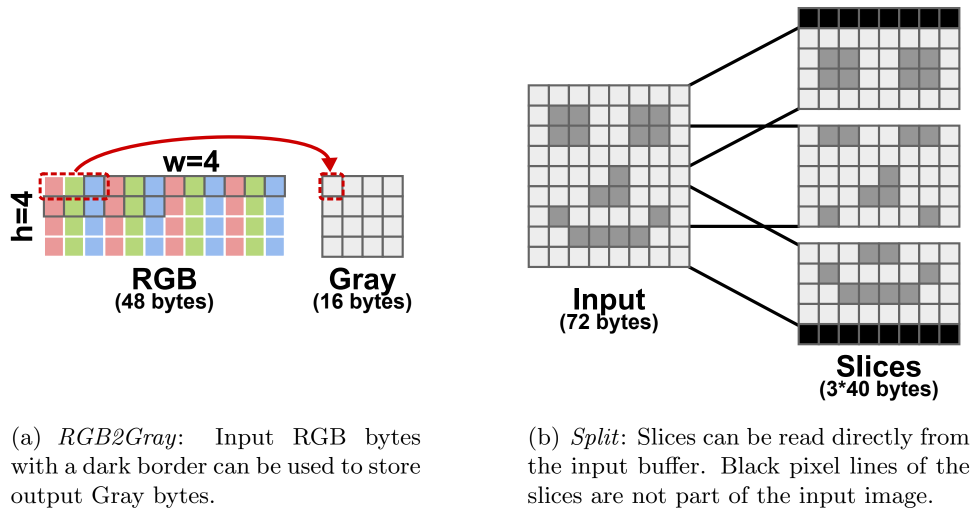
By buffering two pixel lines of the input image, the *Median* actor can be implemented in such a way that its results are written directly in its input buffer. As illustrated in Figure 5.4, to compute the value of a pixel, 3 pixels are read from the original image and 6 pixels are read from the line buffer. The line buffer stores the original value of pixels that have been overwritten in the input image by the application of the median filter on previous lines of pixels. It is important to note that the *Median* actor is not an in-place algorithm in the strict sense as it requires a working memory (i.e. the line buffer) proportional to the width of the processed input frame.

Mergeable Input and Output Buffers

Information on the data dependencies between the input and output buffers of an actor can be used to reveal the disjoint lifetimes of **subranges** of the buffers [DGCDM97].

Data dependencies can be used to identify when a subrange of an input buffer is last read, and when an output buffer is first written. If an input subrange is last read before an output buffer is first written, this two subranges might be allocated in the same memory space. Indeed, in such a case, corrupting the input buffer by writing the result into it is not an issue since this input will no longer be used. An allocation technique taking advantage of the data dependencies exposed by an automatic analysis of application data access is presented in [DGCDM97].

Figure 5.5(a) illustrates the data dependencies of the *RGB2Gray* actor for a 4x4 pixels image. In this example, 3 input bytes are read to produce each output byte. Assuming that a raster scan of the input pixels is used, the input bytes will never be read again and the result can be stored directly in the input. Hence, as illustrated in Figure 5.5(a), the 16 output bytes can be stored in a contiguous subrange of the input buffer. The chosen subrange must ensure that no output byte overwrites an input byte that has not yet been read. In the example, the 16 bytes of the *Gray* output buffer are stored between the $[2; 17]$



(a) *RGB2Gray*: Input RGB bytes with a dark border can be used to store output Gray bytes.

(b) *Split*: Slices can be read directly from the input buffer. Black pixel lines of the slices are not part of the input image.

Figure 5.5: *Memory reuse opportunities for custom actors.*

subrange of the *RGB* input buffer. Another valid solution would be to store these 16 bytes in the $[0; 15]$ subrange.

A special case of mergeable buffers is the *Split* actor presented in Figure 5.5(b). The purpose of the *Split* actor is to divide an input image into several overlapping slices. In this example, each slice has a 1-pixel line overlap with the previous and the next slices. Like special actors (*Fork*, *Join*, *Broadcast*, *Roundbuffer*), output values of the *Split* actors are simple copies of the inputs. Provided that the consumer of the slices do not write within the slices memory, slices can be allocated directly within the input memory. As illustrated in Figure 5.5(b), border slices of the image may require the allocation of extra lines of pixel. If each slice is allocated in its corresponding contiguous memory space of the input buffer, then these extra lines will be allocated before and after the memory allocated for the input buffer.

Next section presents related work on memory optimization techniques taking advantage of application data access and actor internal behavior. Then, Sections 5.4 and 5.5 present a new generic technique to enable input and output buffers allocation optimization for any "special" and user-defined actors.

5.3 Related Work

The analysis of data dependencies to optimize the memory allocation of an application has been an important research subject for many years and has been studied at many levels of abstraction.

5.3.1 Compiler optimization

Data dependencies analysis is an important optimization process of modern compilers. This optimization consists of automatically parsing the source code of an application to identify data accesses and expose variables with disjoint lifetimes. Graph coloring [QC93] and clique partitioning [KS02] techniques are then used to perform the memory allocation of these variables with a minimal memory footprint. Such approaches have been widely

used to optimize the memory allocation for procedural programming languages such as C [DGCDM97] and Fortran [Fab79].

For example, in [DGCDM97], De Greef et al. expose the order in which array elements are last read and first written in a C program. Using this information, they propose an allocation algorithm that partially merges the memory allocated for arrays with overlapping lifetimes.

Keywords have been introduced in programming languages to specify explicitly the type of access for a given variable or array. For example, the `const` keyword in C or the `final` keyword in Java both specify that the associated primitive object will be read but never written. There exists other keywords such as the `restrict` or the `volatile` keywords in C language [Mag05]. Using these keywords, the developer of an application gives information to the compiler that can not be deduced from code analysis because of complex nested function calls or call to external libraries. This information is used by the compiler to minimize the allocated memory footprint of applications.

5.3.2 Dataflow Optimization

In most SDF programming frameworks [Ele13, SWC+11, Rau06], memory optimization only consists of graph-level optimization of the memory allocated to the FIFOs. Indeed, the internal behavior of actors is often unknown to the SDF frameworks and cannot be exploited for optimization purposes. However, separate solutions to the issues presented in Section 5.2 can be found in the literature.

Several solutions to the *Broadcast* actor issue can be found in the literature. Non-destructive reads, or FIFO peeking, is a well-known way to read data tokens without popping them from FIFOs, hence avoiding the need for *Broadcast* actors [FWM07]. Unfortunately, this technique cannot be applied without considerably modifying the underlying SDF MoC. Indeed, using FIFO peeking means that an actor does not have the same behavior for all firings. Otherwise, tokens of peeked FIFOs would never be consumed and would accumulate indefinitely. Another solution to the *Broadcast* issue is to use a *Broadcast FIFO* [MFK+11]. *Broadcast FIFOs* are single-writer, multiple-readers FIFOs that discard data tokens only when all readers have consumed them. The drawbacks of this solution is that it requires a modification of the SDF MoC semantics but only solves the *Broadcast* issue.

In [CDG+14], Cudennec et al. propose a compile-time optimization technique for CSDF graphs that enables access to a shared buffer for the producers and consumers of *special actors* (*Fork, Join, Broadcast, ...*) called *system agents* in this publication. Contrary to the method presented in this chapter, this technique does no allow buffer merging for input and output buffers of actors with a used-defined behavior.

The allocation of input and output buffers of an actor in overlapping memory spaces has been studied in [MB04, HKB05]. In [MB04], Murthy and Bhattacharyya introduce an annotation system for SDF actors to specify a relation between the number of data tokens produced and consumed for a pair of input and output buffers of an actor. This information is then used jointly with scheduling information to enable the merging of the annotated buffers. In [HKB05], another annotation system is introduced to specify buffers that may be used for the in-place execution of actors. The SynDEx framework also provides supports for in-place computations through the use of *input/output ports* [FGG+13].

The advantage of these annotation-based techniques is that they do not require any modification of the underlying SDF MoC. Even though to fully benefit from these annotations, the SDF FIFOs must be replaced with buffers, a regular SDF graph can still

be obtained by ignoring these annotations. The main drawback of these two annotation systems is that they only allow pairwise merging of input and output buffers. Hence, these annotation systems are unable to model the behavior of *Fork*, *Broadcast*, or *Split* actors that require merging several output buffers into a single input. Moreover, the optimization technique presented in [MB04] relies on a single-core scheduling of the application graph. The application of this optimization technique to multicore architectures and schedules is not straightforward.

Like the existing annotation systems, the buffer merging technique presented in following sections does not require any modification of the underlying MoC. Contrary to existing techniques, the buffer merging technique can be used for any number of input and output buffers.

5.4 Abstract Description of Actor Internal Behavior

To enable the merging of input and output buffers of an SDF actor, development tools must be given information on their internal behavior. Besides the application SDF graph, the memory optimization technique presented in this chapter relies on two additional inputs abstracting the internal behavior of actors: a script-based specification of mergeable buffers, and a set of annotations of the ports of the SDF actors. Advantages of this technique are:

- **No impact on SDF graphs:** Annotating an application with these new inputs does not require any modification of the underlying SDF graph. These new optional inputs are only used by the development framework as part of a compile time optimization process. If an annotated application were to be implemented on a target that does not support these optimizations, these annotations could simply be ignored without any impact on the application behavior. Conversely, the proposed annotations are not indispensable for the description of a functional application. Annotations can be added to an application description only in late stages of development, when optimization of the application memory footprint is needed.

- **Independence from the *host* language:** The *host* language is the language used to specify the internal behavior of actors. The proposed optimization technique is not based on an automated analysis of the host code of actors. Instead, a script-based description of the mergeable buffers is provided by the application developer. This abstract description can be suitable for several implementations of a given actor in different *host* languages.

- **Semi-automated graph annotation:** Since the behavior of some "special" actors is predefined (cf. Section 5.2.1), annotations associated to these actors can be added automatically during the graph transformations.

5.4.1 Matching Input and Output Buffers with Memory Scripts

The first input used to abstract the internal behavior of actors is a script-based specification of the mergeable input and output buffers of an actor.

As presented in Section 5.2, many actors offer opportunities for reusing memory allocated to input buffers for the allocation of output buffers. The objective of memory scripts is to allow the application developer to specify directly which input buffer can be merged with which output buffer, and what the relative position of the merged buffers is. To this purpose, each actor of the SDF graph can be associated with a memory script.

The memory scripts are executed by the development framework at compile time. The script inputs are similar to those provided for the execution of actors: a list of the input buffers of the actor, a list of the output buffers of the actor, and a list of parameters influencing the internal behavior of actors. The script execution produces a list of matches between the input and output buffers of the actor. Each match corresponds to the association of a subrange of memory from an input buffer with a subrange of memory from an output buffer. Applying a match consists of merging the memory allocated to the two subranges in a unique address range.

The following definitions give the formal notations associated to the buffers and matches concepts used in this chapter.

Definition 5.4.1 (Buffers)**.** *Considering an actor* $a \in A$ *of an SDF graph* $G = \langle A, F \rangle$*:* $\mathbf{B_a}$ *is the set of input and output buffers of actor* a*.* $\mathbf{B_a^{in}} \subseteq \mathbf{B_a}$ *and* $\mathbf{B_a^{out}} \subseteq \mathbf{B_a}$ *are the sets of input and output buffers respectively associated to ports* P_{data}^{in} *and* P_{data}^{out} *of actor* a*. Each buffer* $b \in B_a$ *is defined as a tuple* $\mathbf{b} = \langle \mathbf{r_{bytes}}, \mathbf{size} \rangle$ *where:*

- $\mathbf{r_{bytes}}$ *is the range of bytes associated with buffer* $b \in B_a$*. By definition* $r_{bytes} = [start, end[$ *with* $start, end \in \mathbb{Z}$ *and* $start < end$*. The default value of* r_{bytes} *for buffer* $b \in B_a$ *associated to port* $p \in P_{data}^{in} \cup P_{data}^{out}$ *is* $b.r_{bytes} = [0, rate(p)[$*.*

- ***size****:* $B \rightarrow \mathbb{N}^*$ *is the amount of memory needed to allocate a given buffer in memory. The size of a buffer* $b \in B_a$ *is deduced from the range of bytes* $b.r_{bytes}$ *associated to this buffer:* $size(b) := b.r_{bytes}.end - b.r_{bytes}.start$*.*

Definition 5.4.2 (Matches)**.** *Considering an actor* $a \in A$ *of an SDF graph* $G = \langle A, F \rangle$*:* $\mathbf{M_a}$ *is the set of matches associated to buffers* B_a *of actor* a*. Each match* $m \in M_a$ *is defined as a tuple* $\mathbf{m} := \langle \mathbf{b_{src}}, \mathbf{r_{src}}, \mathbf{b_{dst}}, \mathbf{r_{dst}} \rangle$ *where:*

- $\mathbf{b_{src}}, \mathbf{b_{dst}} \in \mathbf{B_a}$ *are the source and the destination buffers of match* $m \in M_a$*.*

- $\mathbf{r_{src}}$ *and* $\mathbf{r_{dst}}$ *are the matched subranges of bytes from buffers* b_{src} *and* b_{dst}*.*

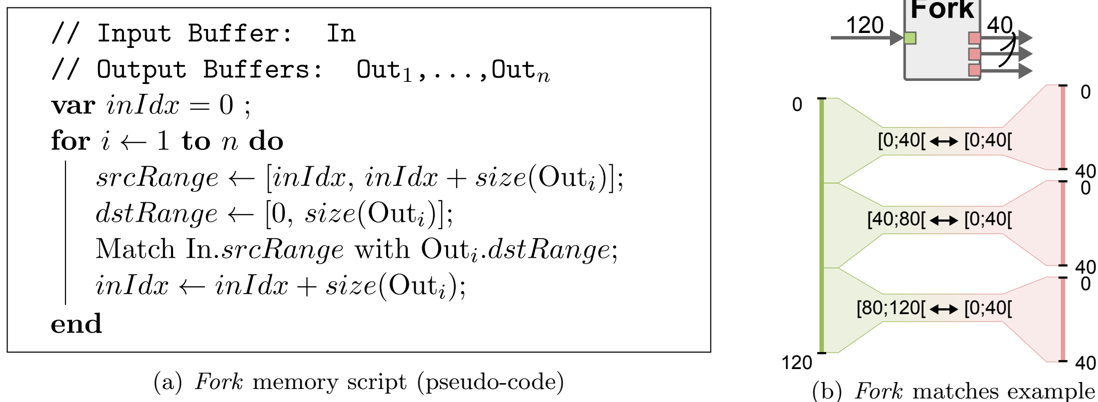
It is important to note that a match $m \in M$ *can be applied in both directions: the destination subrange* $m.r_{dst}$ *can be merged into the source buffer* $m.b_{src}$ *or the source subrange* $m.r_{src}$ *can be merged into the destination buffer* $m.b_{dst}$*.*

Examples

Figures 5.6 to 5.8 present examples of memory scripts and illustrate the matches resulting from an execution of these scripts.

The memory script of Figure 5.6(a) is a generic script supporting *Fork* actors with any number of output buffers. The only condition for this script to run correctly is that the sum of the sizes of output buffers ($\sum_{i=1..n} size(Out_i)$) equals the size of the input buffer $size(In)$. As illustrated in Figure 5.6(b), the execution of this script results in matching the output buffers of the *Fork* actor one after the other within the input buffer.

For clarity, memory scripts presented in Figures 5.6(a), 5.7(a), and 5.8(a) are written in pseudo-code. In PREESM, memory scripts are written with a dynamically interpretable derivative of the Java language called BeanShell [Nie14]. Listing 5.2 presents the BeanShell memory script for the *Fork* actor.



(a) *Fork* memory script (pseudo-code)

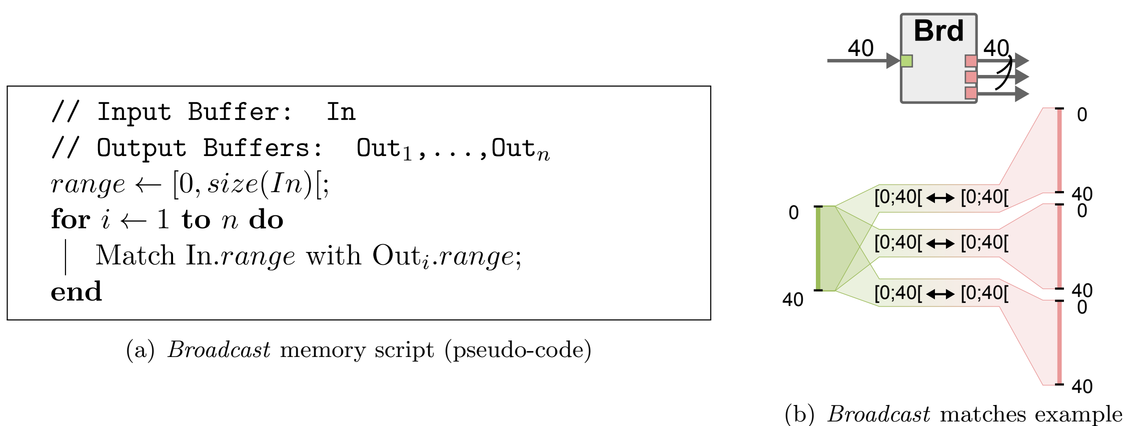
(b) *Fork* matches example

Figure 5.6: *Memory script for* Fork *actors*

```
inIdx = 0;
for(output : outputs){
  outSize = output.getNbTokens();
  inputs.get(0).matchWith(inIdx,output,0,outSize);
  inIdx += outSize;
}
```

Listing 5.2: *Fork actor memory script in BeanShell*

Figure 5.7(a) presents the memory script that can be used for *Broadcast* actors with any number of output buffers. For a correct execution of this memory script, all output buffers must have the same size as the input buffer. As illustrated in Figure 5.7(b), the execution of this memory script results in matching all output buffers at the beginning of the input buffer. The source ranges r_{src} of all the matches produced by the script execution are thus identical.



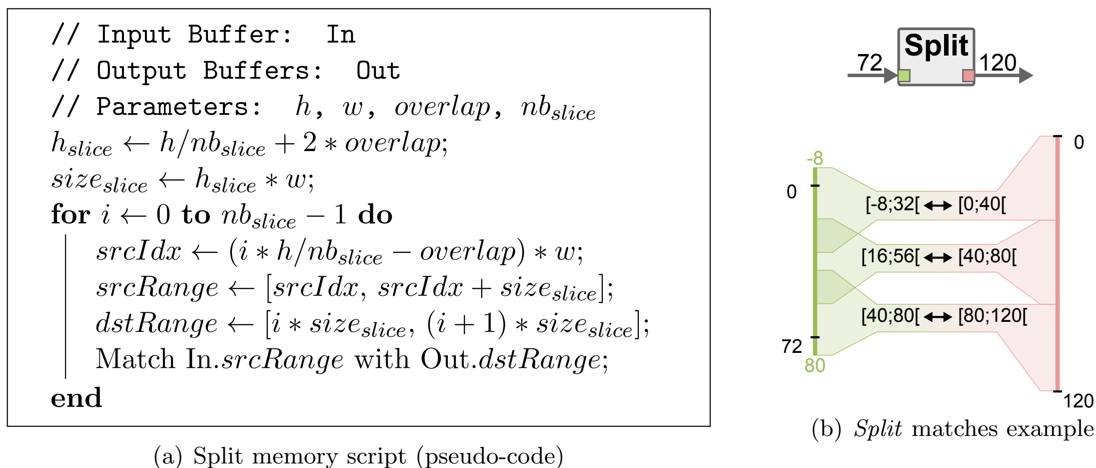
(a) *Broadcast* memory script (pseudo-code)

(b) *Broadcast* matches example

Figure 5.7: *Memory script for* Broadcast *actors*

The memory script associated to user-defined *Split* actors is presented in Figure 5.8. This memory script is more complex than the previous ones as its execution requires a set of parameters in addition to the list of input and output buffers. The h, w, $overlap$, and nb_{slice} parameters correspond respectively to the height and width of the sliced image, to the number of pixel lines overlapping between slices, and to the number of slices produced.

Figure 5.8(b) illustrates the matches obtained for $h = 9$, $w = 8$, $overlap = 1$, and $nb_{slice} = 3$. The output buffer of the *Split* actor is divided into three subranges of equal size $(h/nb_{slice} + 2 * overlap) * w$. Each output subrange is matched in the corresponding subrange of the input buffer, thus creating overlaps between the matched input subranges. It is interesting to note that the first and the last subranges of the output buffer are partially matched outside the boundaries of the input buffer byte range $r_{bytes} = [0, 72[$.



(a) Split memory script (pseudo-code)

(b) *Split* matches example

Figure 5.8: *Memory script for* Split *actors*

These 3 examples illustrate the great expressiveness of memory scripts that allows the specification of complex matching patterns with only a few script lines. Indeed, as illustrated in these examples, it is possible to match a single buffer with several other buffers, to match several buffers in overlapping subranges, to match contiguous subranges into non-contiguous ranges, and to match subranges partially outside the range of bytes of another buffer.

Matching Rules

The matches resulting from the memory script execution serve as an input to the optimization process presented in Section 5.5. Although memory scripts offer a great liberty for defining custom matching patterns, a set of rules must be respected to ensure the correct behavior of an application.

R1. Both subranges r_{src} and r_{dst} of a match $m \in M_a$ must cover the same number of bytes: $r_{dst}.end - r_{dst}.start = r_{src}.end - r_{src}.start$.

R2. A match $m \in M_a$ can only be created between an input buffer $b_{src} \in B_a^{in}$ and an output buffer $b_{dst} \in B_a^{out}$.

R3. A subrange of bytes of an output buffer $b_{dst} \in B_a^{out}$ can not be matched several times by overlapping matches. Formally, if $\exists\, m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle \in M_a$ then $\nexists\, m' = \langle b'_{src}, r'_{src}, b'_{dst}, r'_{dst} \rangle \in M_a | b'_{dst} = b_{dst}$ and $r'_{dst} \cap r_{dst} \neq \emptyset$

R4. All matches $m \in M_a$ must involve at least one byte from the default byte range $b_{src}.r_{bytes}$ and one byte from the default byte range $b_{dst}.r_{bytes}$. Formally, the following condition must always be true: $\forall m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle \in M_a$, $r_{src} \cap b_{src}.r_{bytes} \neq \emptyset$ and $r_{dst} \cap b_{dst}.r_{bytes} \neq \emptyset$.

R5. Only bytes within the default byte range $b.r_{bytes}$ of their buffer $b \in B_a$ can be matched with bytes falling outside the default byte range of the matched buffer. Formally, considering a match $m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle \in M_a$, for each byte $n \in r_{src}$ and its matched byte $n' \in r_{dst}$, if $n \notin b_{src}.r_{bytes}$ then $n' \in b_{dst}.r_{bytes}$.

The first rule R1 enforces the validity of the matches. It is impossible to allocate a contiguous subrange of n bytes within a memory range of m bytes if $m < n$. Indeed, a subrange of a buffer can not be matched or merged within a smaller subrange of another buffer. Since matches are always bidirectional, it is also impossible to match a subrange of a buffer within a larger subrange of another buffer and thus $m = n$ is needed.

Rules R2 and R3 forbid the creation of useless matching patterns. Merging several input buffers of an actor together would result in allocating the corresponding single-rate FIFOs in overlapping memory ranges. Consequently, all actors producing data tokens on these FIFOs would write these data tokens in the same memory, thus overwriting each other results. For this reason, merging several inputs together is forbidden both directly, by matching them together, or indirectly, by matching several inputs in overlapping ranges of output buffers. However, the indirect merge of several output buffers through overlapping matches with an input buffer is allowed. This matching pattern means that several actors consuming data tokens from the merged outputs will read these data tokens from the same memory. This matching pattern is thus valid as long as the consuming actors do not write in the shared memory (cf. Figure 5.7 and 5.8).

Rules R4 and R5 forbid the creation of matches that, when applied, would result in merging "virtual" bytes from the input and output buffers. A byte is called "virtual" if it does not belong to the default range of bytes of a buffer, otherwise the byte is called "real". All bytes of a buffer, "virtual" and "real", are mapped in memory when a buffer is allocated. As illustrated by the *Split* actor (Figure 5.8), memory scripts can be used to match a subrange of a buffer partially outside the range of "real" bytes, or default byte range, of the remote buffer. Misused, this feature could be used to merge a buffer completely out of the "real" byte range of the remote buffer, thus resulting in no memory reuse between the two buffers. Such matches are made impossible by forcing matches to have at least one "real" byte on both sides of the match, and by forbidding matches between "virtual" bytes.

5.4.2 Port Annotations

As illustrated by the *Broadcast* and the *Split* actors, memory scripts allow the creation of overlapping matches. Applying overlapping matches results in merging several subranges of output buffers in the same input buffer. Hence, actors reading data from the merged output buffers are accessing the same memory. To ensure the correct behavior of the application, the memory optimization process must check that actors accessing the merged buffer do not write in this shared memory. If one of the consumer actor does not respect this condition, its corresponding output buffer should not be merged and it should be given a private copy of the data. Indeed, writing in the shared memory would modify the input data of other actors, which might change their behavior.

By default, the most permissive actor behavior is assumed. All actors are supposed to be both writing to and reading from all their input and output buffers. Since this assumption forbids the application of overlapping matches, a set of graph annotations has been introduced. Each data port $p \in P^{in}_{data} \cup P^{out}_{data}$ can be annotated with the following keywords:

- **Read-Only:** The actor possessing a *read-only* input port can only read data from this port. Like a `const` variable in C or a `final` variable in Java, the content of a buffer associated to a *read-only* port can not be modified during the computation of the actor to which it belongs.

- **Write-Only:** The actor possessing a *write-only* output port can only write data on this port. During its execution, an actor with a *write-only* buffer is not allowed to read data from this buffer, even if data in the buffer was written by the actor itself.

- **Unused:** The actor possessing an *unused* input port will never write nor read data from this port. Like the `/dev/null` device file in Unix operating systems, an *unused* input port can be used as a sink to consume and immediately discard data tokens produced by another actor.

It is important to note that if the application of a match created by a memory script results in merging an input buffer into an output buffer written by the actor, then the input buffer should not be marked as *read-only*. This condition holds even if all write operations are performed in ranges of the output buffer that fall outside the ranges merged with the input buffer. For example, the input port of the *Split* actor presented in Figure 5.8 cannot be marked as *read-only* because it is matched in an output buffer whose first and last lines of pixel will be written by the *Split* actor. Input ports of *Fork* (Figure 5.6) and *Broadcast* (Figure 5.7) actors can be marked as *read-only* ports since, if the matches are applied, no write operation will be issued to the output buffers of these actors.

An example of use-case for the *write-only* and *unused* port annotations are the *Roundbuffer* actors. As illustrated in Figure 5.3(c), the only purpose of *Roundbuffer* actors is to forward on their output port the last data tokens consumed on their input ports. All input ports of a *Roundbuffer* actor except the last one can thus be marked as *unused*. If a FIFO connects an *unused* input port to a *write-only* output port, the memory allocated for this FIFO can be reused to store other memory objects as soon as the FIFO producer completes its execution.

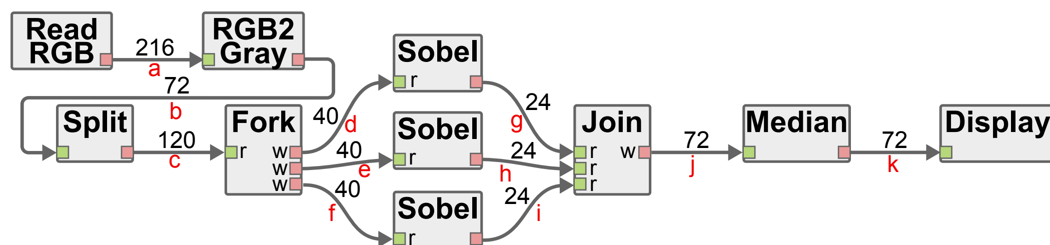


Figure 5.9: *Single-rate SDF graph from Figure 5.2 for $h = 9$, $w = 8$, and $n = 3$.* r *and* w *mark* read-only *and* write-only *ports respectively. Red letters uniquely identify the* FIFO*s.*

In PREESM, annotations are automated for the data ports of the *Fork*, *Join*, *Broadcast*, and *Roundbuffer* actors that are inserted during graph transformations. All output ports of these special actors are thus automatically marked as *write-only* ports, and all input ports, except the first ports of the *Roundbuffer* actors, are marked as *read-only* ports. For user-defined actors, it is the developer's responsibility to make sure that the actors internal behavior is consistent with the port annotations. In the single-rate graph presented in Figure 5.9, input ports of the *Sobel* actors must be marked as *read-only* ports to allow these actors to read their input slices directly from the input buffer of the *Split* actor if all matches created by the scripts of the *Fork* and *Split* actors are applied.

The memory scripts and port annotations presented in this section allow the developer of an application to specify how the input and output buffers can be merged, and how actors access these buffers. In the next section, an optimization process is presented to make use of these inputs to reduce the memory footprints of SDF graphs.

5.5 Match Tree Memory Minimization

The execution of a memory script associated to an actor produces a list of matches that represent merging opportunities for the input and output buffers of this actor. Once all the memory scripts associated to the actors of an SDF graph have been executed, the memory minimization process builds trees of buffers and matches by chaining the lists of matches produced by the memory scripts.

Definition 5.5.1. *A match tree is a directed tree denoted by* $T = \langle B, M \rangle$ *where:*

- **B** *is the set of vertices of the tree. Each vertex is a buffer* $b \in B_a$ *of an actor* a *from the SDF graph (cf. Definition 5.4.1). In the match tree, a single buffer* $b \in B$ *is used to represent two buffers* $b_o \in B^{out}_{prod}$ *and* $b_i \in B^{in}_{cons}$ *linked by a single-rate* FIFO.
- $\mathbf{M} \subseteq B \times B$ *is the set of directed edges of the tree. Each edge* $m \in M$ *is a match produced by the memory script of an actor (cf. Definition 5.4.2).*

The combination of all the buffers and matches of an application results in the creation of a forest (i.e. the creation of several unconnected match trees).

An example of match tree is given in Figure 5.10 below the corresponding single-rate SDF graph. In this figure, the single-rate FIFOs between actors *RGB2Gray* and *Split*, and actors *Split* and *Fork* each are represented by a single buffer.

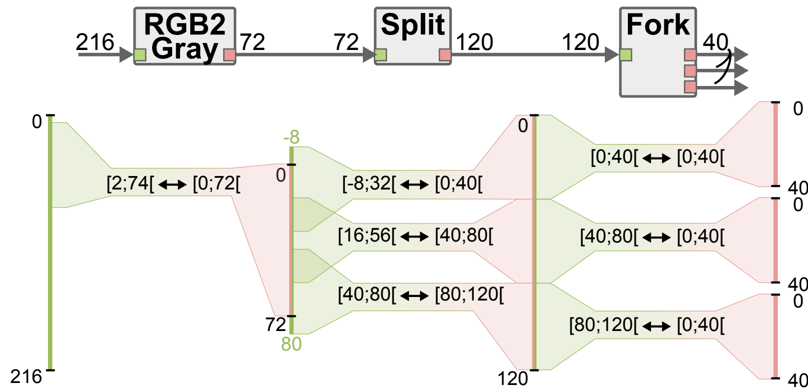


Figure 5.10: *Match tree associated to buffers of actors* RGB2Gray, Split, *and* Fork.

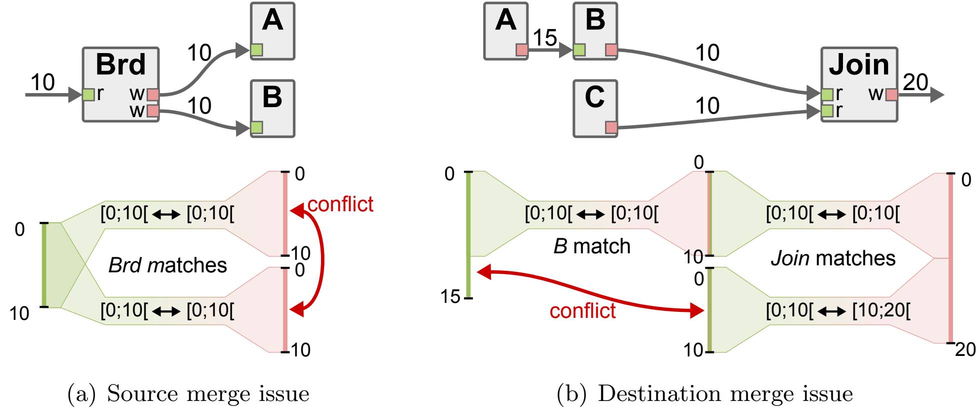
The match trees derived from an application are used by the optimization process to identify the matches that can be applied without corrupting the behavior of the application. All applicable matches are applied by merging their corresponding memory objects in the MEG derived from the SDF graph.

5.5.1 Applicability of Buffer Matches

A match $m \in M_a$ is said to be applicable if it can be applied without changing the behavior of the application. The 5 matching rules presented in Section 5.4.1 are necessary conditions to ensure the applicability of a created match. However, following these rules is not sufficient to guarantee the applicability of the created matches.

Buffer Merging Issues

Figure 5.11 gives examples of matches that respect the matching rules presented in Section 5.4.1 but that can not be applied without corrupting the application behavior.



(a) Source merge issue (b) Destination merge issue

Figure 5.11: *Matching patterns with inapplicable matches.*

- **Source merge issue:** Matches with overlapping source subranges can be applied only if their destination buffers are *read-only* or *unused*. In Figure 5.11(a), only one of the *Broadcast* matches can be applied because neither actor A nor actor B have a *read-only* input port. If both matches were applied, actors A and B would write in each other input buffer and corrupt the application behavior.
- **Destination merge issue:** A chain of matches cannot be applied if it results in merging several source subranges in overlapping destination subranges. In Figure 5.11(b), if all matches were applied, the output buffers of actors A and C would be merged in ranges $[0, 15[$ and $[10, 20[$ respectively of the output buffer of the *Join* actor. Hence, if all matches were applied and actor A was scheduled after actor C, then actor A would partially overwrite the data tokens produced by actor C, thus corrupting the application behavior.

In order to detect and avoid the application of matches involved in a merging issue, a set of new properties is introduced by the memory optimization process.

Definition 5.5.2. *The tuple associated to a buffer $b \in B$ is extended as follows:* $b = \langle r_{bytes}, size, \mathbf{r_{merge}} \rangle$ *where r_{merge} is the set of ranges of mergeable bytes of the buffer. Buffers corresponding to* FIFOs *connected to a* read-only *or an* unused *input buffer are initialized with a mergeable range $r_{merge} = \{r_{bytes}\}$, otherwise $r_{merge} = \{\emptyset\}$.*

r_{merge} is defined as a set of ranges because it can contain several non-contiguous ranges as a result of an update of the buffer properties (cf. Section 5.5.2).

A range of bytes of a buffer is *mergeable* only if the consumer actor associated to the corresponding FIFO does not write in this buffer (i.e. sink port of the FIFO is either *read-only* or *unused*). If all ranges of bytes involved in a *source merge issue* are *mergeable*, then the corresponding matches can be applied without corrupting the application behavior.

Property 5.5.1 (Source merge issue)**.** *Two matches $m_1, m_2 \in M$ with overlapping source subranges $m_1.r_{src}$ and $m_2.r_{src}$ can both be applied if and only if $(m_1.r_{src} \cap m_2.r_{src}) \subseteq (m_1.r_{merge} \cap m_2.r_{merge})$. If this condition is not satisfied, the two matches are said to be* ***in conflict****, and at most one of them can be applied.*

Property 5.5.2 (Destination merge issue)**.** *Two matches $m_1, m_2 \in M$ with overlapping destination subranges $m_1.r_{dst}$ and $m_2.r_{dst}$ are **in conflict** and can never be both applied. Formally, if $m_1.r_{dst} \cap m_2.r_{dst} \neq \emptyset$, then m_1 and m_2 mutually exclude each other.*

Properties 5.5.1 and 5.5.2 present the applicability rules for the source merge issue and the destination merge issue respectively.

Buffer Division Issue

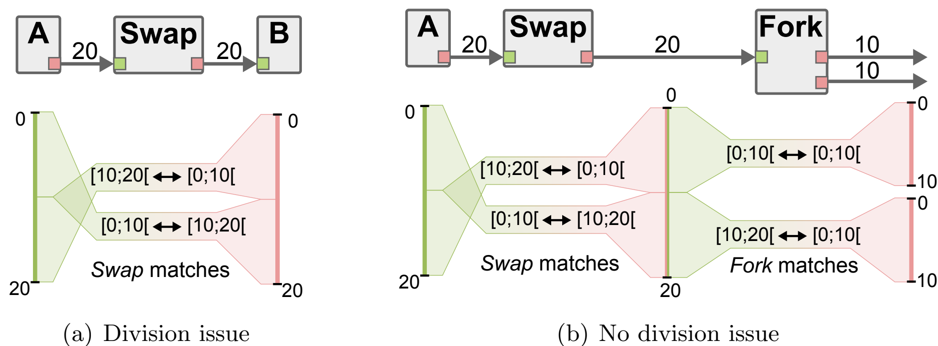
To ensure that the behavior of an application is not modified when buffer merging is applied, a necessary condition is that all actors must always have access to all their input and output buffers. For example, as illustrated in Listing 5.3, a *null* output pointer can be given to an *in-place* actor like *RGB2Gray*, if the memory allocated for its output buffer is merged into its input buffer memory.

```
// Call to RGB2Gray actor
// 16 output bytes are merged in byte range [2, 17] of the input buffer.
rgb2gray(4 /*height*/, 4 /*width*/,
         ptr /*pointer to 48 bytes*/, NULL /*null pointer*/);
```

Listing 5.3: *Possible call to RGB2Gray actor if buffers are merged as in Figure 5.5(a).*

As illustrated by the memory script of actor *Split* in Figure 5.8, contiguous subranges of bytes can be matched in non-contiguous ranges of bytes. The application of this matching pattern requires the division of the output buffer of actor *Split* into several subranges, each matched in a distinct location.

A buffer can be divided into non-contiguous subranges only if all actors accessing this buffer can still access all its subranges. Hence, a divided buffer remains accessible to an actor only if the memory script of this actor matches all the subranges of this buffer into other buffers accessible by this actor. To apply the matching pattern illustrated in Figure 5.12(a), either the output buffer of actor A or the input buffer of actor B must be divided in two non-contiguous subranges in memory. If actors A and B are not associated with memory scripts, neither one of them expects the buffer division. Since applying the *Swap* matches can only be achieved by dividing the output buffer of actor A or the input buffer of actor B, these matches cannot be applied.



(a) Division issue (b) No division issue

Figure 5.12: *Divisibility of buffers*

In Figure 5.12(b), the *Swap* actor is followed by a *Fork* actor. In this case, all subranges of the input buffer of actor *Fork* are matched within its output buffers. Consequently, actors *Swap* and *Fork* both expect a division of the buffer corresponding to the FIFO

between them. In such a case, the buffer can be divided into two non-contiguous ranges of bytes merged respectively into the input buffer of actor *Swap* and into the output buffers of actor *Fork*.

A set of new properties is introduced by the memory optimization process to detect and check the applicability of matching patterns requiring the division of a buffer.

Definition 5.5.3. *The tuple associated to a buffer $b \in B$ is extended as follows:* $b = \langle r_{bytes}, size, r_{merge}, \mathbf{r_{div}} \rangle$ *where r_{div} is a set of* indivisible *ranges of bytes of the buffer. If a range of bytes is* indivisible *it will compulsorily be allocated in a contiguous memory space.*

Considering the set of matches $S \subset M$ involving a buffer $b \in B$, the set of indivisible ranges r_{div} of this buffer is initialized as the *lazy union* of the matched subranges.

Definition 5.5.4. *Considering two ranges of bytes $r_1 = [a, b[$ and $r_2 = [c, d[$, the* lazy union $\overset{lazy}{\cup}$ *of these ranges is computed as follows:*

$$r_1 \overset{lazy}{\cup} r_2 = \begin{cases} \textit{if } r_1 \cap r_2 \neq \emptyset & \textit{then } [min(a, c); max(b, d)[\\ \textit{if } r_1 \cap r_2 = \emptyset & \textit{then } \{[a, b[; [c, d[\} \end{cases}$$

Considering two consecutive ranges of bytes $r_1 = [a, b[$ and $r_2 = [b, c[$, contrary to the standard union operator whose result is $r_1 \cup r_2 = [a, c[$, the result of the *lazy union* is $r_1 \overset{lazy}{\cup} r_2 = \{[a, b[; [b, c[\}$.

Property 5.5.3. *Considering a buffer $b \in B$ and the set of associated matches*

$$S = \{m \in M | m.b_{dst} = b \lor m.b_{src} = b\}$$

Let $m.r_b$ be the matched subrange (source or destination) of $m \in S$ associated to buffer b. The indivisible range r_{div} associated to buffer b is initialized as follows:

$$b.r_{div} = \begin{cases} \textit{if } \bigcup_{m \in S} m.r_b = b.r_{bytes} & \textit{then } \overset{lazy}{\bigcup_{m \in S}} m.r_b \\ & \textit{else } b.r_{bytes} \end{cases}$$

In Figure 5.12(b), the input buffer of actor *Swap* and the output buffer of actor *Swap* (and input buffer of actor *Fork*) both have several indivisible ranges $r_{div} = \{[0, 10[; [10, 20[\}$. The two output buffers of actor *Fork* each have a single indivisible range that covers their complete range of bytes: $r_{div} = r_{bytes} = [0; 10[$.

If only part of the range of bytes of a buffer $b \in B$ is matched, then this buffer is indivisible, and its indivisible range is $b.r_{div} = b.r_{bytes}$. For example, in Figure 5.10, the indivisible range of bytes associated to the input buffer of actor *RGB2Gray* is $[0; 216[$.

Property 5.5.3 specifies how the indivisible ranges of a buffer can be computed. The following property gives the conditions that must be satisfied for a buffer to be divided into these indivisible subranges.

Property 5.5.4. *Considering a buffer $b \in B$ and the set of associated matches $S \subset M$ (as defined in Property 5.5.3), buffer b can be divided into subranges $b.r_{div}$ and matched into non-contiguous ranges of bytes if and only if all the following conditions are met:*

1. *Matches in S with $m.b_{src} = b$ completely cover the range of bytes of buffer b and have no overlap. Formally, with $S^{src} = \{m \in S | m.b_{src} = b\}$, the condition is $(\bigcup_{m \in S^{src}} m.r_{src} = b.r_{bytes})_1 \land (\bigcap_{m \in S^{src}} m.r_{src} = \emptyset)_2$.*

2. *Matches in S with $m.b_{dst} = b$ completely cover the range of bytes of buffer b and have no overlap. Formally, with $S^{dst} = \{m \in S | m.b_{dst} = b\}$, the condition is $(\bigcup_{m \in S^{dst}} m.r_{dst} = b.r_{bytes})_1 \wedge (\bigcap_{m \in S^{dst}} m.r_{dst} = \emptyset)_2$.*

3. *All matches in S are applicable under Properties 5.5.1 and 5.5.2*

4. *All matches in S must match buffer b only with indivisible buffers. Formally, with $m.b_{remote}$ the second buffer ($\neq b$) matched by $m \in S$, the condition is $\forall m \in S, m.b_{remote}.r_{div} = m.b_{remote}.r_{bytes}$.*

When a buffer $b \in B$ is divided, it can no longer be accessed as a contiguous memory space. Consequently, a unique reference can not be sufficient to access all the memory associated to this buffer. It is thus assumed that when a buffer is divided, no reference at all can be given for this buffer.

In order to preserve the behavior of the application, actors must find another access to the divided buffer. Consequently, all actors accessing a divided buffer must have matched the divided buffer into other buffers accessed by the actor. Hence, a buffer b can be divided only if both its producer and consumer actors completely match b into other buffers. This condition is expressed by the first parenthesis $()_1$ in the formal expression of conditions 1 and 2 of Property 5.5.4. The programmer is responsible for accessing the divided buffer through the subranges matched in other buffers.

To avoid ambiguities, all subranges of the divided buffer must be matched exactly once in other buffers accessed by the actors. This condition is expressed by the second parenthesis $()_2$ in the formal expression of conditions 1 and 2 of Property 5.5.4.

Following conditions 1 and 2, each subrange of a divided buffer is matched exactly once in buffers of its producer actor, and once in buffers of its consumer actor. If one of this match is not applied, the corresponding actor will not be able to access the unmatched subrange of the divided buffer. Consequently, all matches must be applicable for the buffer to be divided (condition 3).

Finally, subranges of a divided buffer can only be accessed in the remote buffers in which they were merged. Consequently, these remote buffers cannot be divisible themselves since their content must remain accessible to the actors through a simple reference (condition 4).

In Figure 5.12(b), the buffer corresponding to the output of the *Swap* actor satisfies these conditions. Hence, if this buffer is divided, a *null* pointer will be given in its place to actors *Swap* and *Fork*. Listing 5.4 presents the function call of these two actors if the buffer between them was divided.

```
// Call to actor Swap
// Output range [0;10[ is accessible in range [10;20[ of the input
// Output range [10;20[ is accessible in range [0;10[ of the input
swap(ptrIn /*Input*/, NULL /*Output: null pointer*/);
// Call to actor Fork
// Input range [0;10[ is accessible in range [0;10[ of the first output
// Input range [10;20[ is accessible in range [0;10[ of the second output
fork(NULL /*Input: null pointer*/, ptrOut1 /*Output*/, ptrOut2 /*Output*/);
```

Listing 5.4: *Call to actors* Swap *and* Fork *from Figure 5.12(b) if the buffer between them is divided.*

In Figure 5.10, the buffer corresponding to the output of the *Split* actor satisfies these conditions.

Properties 5.5.1, 5.5.2, and 5.5.4 give the necessary conditions that a match must satisfy to be applicable. Using these properties, the purpose of the minimization process is to select and apply as many matches as possible in order to minimize the memory footprint of an application.

5.5.2 Match Trees Optimization Process

The minimization process responsible for selecting the matches to apply can be divided in two steps. First, the forest of match trees obtained by combining results of the memory scripts for all actors are separated into independent match trees. Then, a heuristic algorithm is used to process each match tree and select the matches to apply.

Algorithm 5.1 gives the pseudo-code of the optimization process. The purposes of the different parts of this optimization process are detailed in following sections.

Algorithm 5.1: Optimization process for the application of buffer matches.

Input: B a set of buffers
M a set of matches
Output: $M_{applied}$: a set of applied matches

```
1  Separate T = ⟨B, M⟩ into independent match trees TreeSet = {T_i = ⟨B_i, M_i⟩};
2  // Fold match tree T_i
3  for each T_i ∈ TreeSet do
4  |  repeat
5  |  |  M_sel ← ∅;
6  |  |  // A match m = ⟨b_src, r_src, b_dst, r_dst⟩ can be selected only if:
7  |  |  // ∄m' ∈ M_sel such that m'.b_dst = m.b_src or m'.b_src = m.b_dst
8  |  |  M_sel ← Select applicable match(es) in M_i ;
9  |  |  Apply selected matches M_sel in T_i;
10 |  |  M_applied ← M_applied ∪ M_sel;
11 |  until M_i = ∅ or M_sel = ∅;
12 endfor
```

Independent Match Trees

The first line of the optimization process (Algorithm 5.1) consists of separating the buffers and matches into a set of independent match trees.

Considering all actors of an SDF graph, the number of buffers that can be matched by the memory scripts is equal to the number of single-rate FIFOs of the graph. As shown in [PBL95], the number of FIFOs of an SDF graph might grow exponentially during the transformation of this graph into a single-rate equivalent. To keep the performance overhead of the optimization process as low as possible, the forest of match trees produced by the execution of the memory scripts is clustered into independent match trees. Clustering is achieved by ignoring buffers that are not involved in any match, and identifying the subsets of connected buffers. Because the *Sobel* actor is not associated with a memory script, there is no match between its input and output buffers. Consequently, the single-rate SDF graph of Figure 5.9 contains two unconnected match trees: a tree formed by FIFOs a to f presented in Figure 5.10, and a tree formed by FIFOs g to k. Processing the match trees separately presents two main advantages:

- Some match trees might be much faster to optimize than others. The minimization heuristic algorithm presented in Algorithm 5.1 is an iterative algorithm that repetitively tests the applicability of matches until none is applicable. If the whole forest of match trees was processed at once, the applicability of some matches might be tested unnecessarily several times while these same matches only require a single test as part of a smaller independent match tree.

- Each match tree can be processed in parallel by the optimization process. Since match trees are independent from each other, there is no common buffer between match trees nor matches linking a match tree to another. Consequently, there can be no merge issues between the matches of independent match trees, and the processing can be done in parallel.

Applying Matches

The heart of the optimization process is the application of selected matches at line 9 of Algorithm 5.1. Applying a match consists of merging a buffer, or a subrange of a buffer in case of a division, into a target buffer. The merged subrange can either be part of the source or the destination buffer, respectively merged in a target subrange at the other end of the match (cf. Definition 5.4.2).

Applying a set of matches is a complex operation that requires many transformations in the corresponding match tree. The following list describes the transformations resulting from the application of a match $m = \langle b_{src}, r_{src}, b_{dst}, r_{dst} \rangle \in M$ where b_{src} is the target buffer and b_{dst} is the merged buffer. All notations can thus be reversed ($src \leftrightarrows dst$) for the application of a match where the target is the destination range.

Applying a match consists of:

- **Updating the target buffer properties:** The mergeable ranges r_{merge}, the indivisible ranges r_{div}, and the range of bytes r_{bytes} of the target buffer must be updated with the properties of the merged subrange. Formally,

$$b_{src}.r_{bytes} \leftarrow [min(b_{src}.r_{bytes}.start, r_{src}.start), max(b_{src}.r_{bytes}.end, r_{src}.end)[$$

$$(b_{src}.r_{merge} \cap r_{src}) \leftarrow [b_{src}.r_{merge} \cap (b_{dst}.r_{merge})^{dst \xrightarrow{m} src}] \cap r_{src}$$

$$(b_{src}.r_{div} \cap r_{src}) \leftarrow [b_{src}.r_{div} \overset{lazy}{\cup} (b_{dst}.r_{div})^{dst \xrightarrow{m} src}] \cap r_{src}$$

  Where $()^{dst \xrightarrow{m} src}$ denotes the translation of a range of bytes of the destination buffer into the corresponding range of the source buffer according to match m. Notation $b_{src}.r_{merge} \cap r_{src}$ on the left-hand part of an assignation is used to specify that only the subrange r_{src} of the mergeable range $b_{src}.r_{merge}$ is updated (i.e. r_{src} can be seen as mask for the assignation).

- **Reconnecting matches of the merged subrange:** The merged buffer is not necessarily a leaf of the match tree. In other words, the merged subrange of the destination buffer b_{dst} may itself be the source of other matches $m' \in M$. In such case, these matches should be reconnected to the target buffer as follows:

$$m'.b_{src} \leftarrow b_{src} \qquad \text{and} \qquad m'.r_{src} \leftarrow (m'.r_{src})^{dst \xrightarrow{m} src}$$

- **Removing conflicting matches:** If the applied match $m \in M$ is in conflict with other matches (cf. properties 5.5.1 and 5.5.2), these other matches will no longer be

applicable once m is applied. The inapplicable matches are thus removed from the match tree.

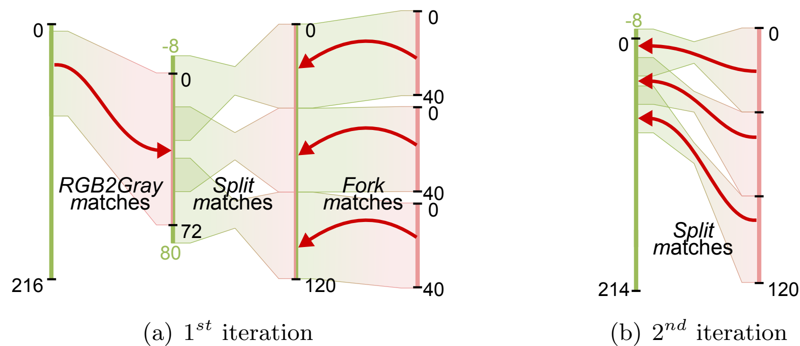
- **Removing applied matches and merged buffers:** Once a match is applied, the merged buffer and the applied match no longer exist in the match tree, and they should be removed from it.

Match Tree Folding

The match tree folding algorithm is the iterative optimization process responsible for selecting the matches to apply (lines 2 to 11 of Algorithm 5.1). This process iterates until no more applicable match can be found in the match tree.

The order in which the matches are applied is important to maximize the number of applied matches. In particular, matches that are not involved in any conflict should be applied first. Since the application of such matches does not require the removal of any conflicting match from the match tree, applying them first is a good way to maximize the number of applied matches. For example, if all but one output buffers of a *Broadcast* actor are mergeable, applying the match with the non-mergeable output first would be a bad choice since it would forbid the application of all other matches.

It is important to note that a match can be applied during an iteration only if neither its source nor its destination buffer is itself merged during the same iteration. Indeed, when a first match is applied, the properties of the target buffer are updated in such a way that may prevent the application of a second match. For example, in the match tree of Figure 5.11(b), all matches are applicable during the first iteration of the folding algorithm, but only one of the matches implying the output buffer of actor B can be applied at a time to avoid destination merge issue.

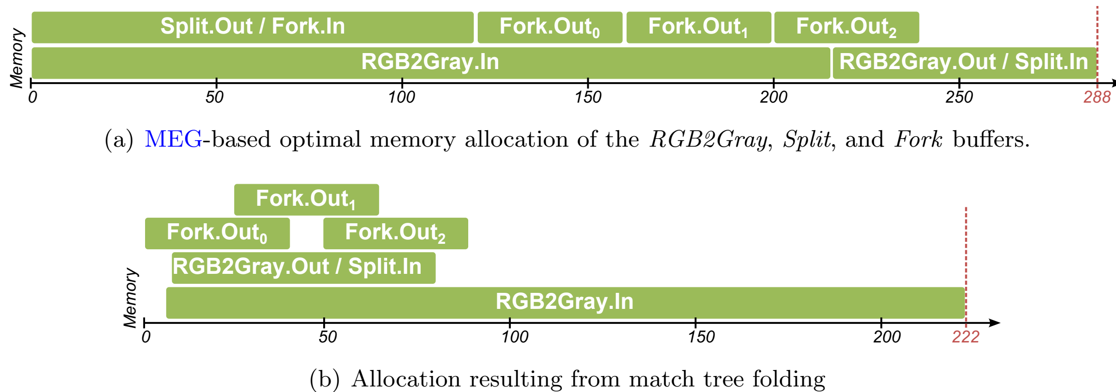


(a) 1st iteration (b) 2nd iteration

Figure 5.13: *Folding the match tree formed by* RGB2Gray-Split-Fork

Figure 5.13 illustrates the processing of the folding algorithm on the match tree from Figure 5.10. In the first iteration (Figure 5.13(a)), 4 matches can be applied simultaneously: the three output buffers of the *Fork* actor are merged into the output buffer of the *Split* actor, and the input buffer of the *RGB2Gray* actor is merged into the input buffer of the *Split* actor. As a result of this last merge, the input buffer of the *Split* actor is enlarged to cover a range of bytes $r_{bytes} = [0, 214[$. In the second iteration (Figure 5.13(b)), the output of the *Split* actor is divided into three subranges, each corresponding to a previously merged output buffer of the *Fork* actor. These three subranges are merged into overlapping subranges of the input buffer of the *Split* actor. Since all matches of the match tree were applied, the execution of the match tree folding algorithm is terminated and a single buffer of 222 bytes remains in the folded match tree. The memory allocation resulting

from this optimization is presented in Figure 5.14(b). A comparison of this result with the memory allocation of the same buffers without buffer merging technique (Figure 5.14(a)) reveals a reduction of the memory footprint by 23% for these buffers.



(a) MEG-based optimal memory allocation of the *RGB2Gray*, *Split*, and *Fork* buffers.

(b) Allocation resulting from match tree folding

Figure 5.14: *Allocation resulting from match tree folding vs. MEG-based allocation*

As presented in Algorithm 5.1, the application of the folding algorithm on all the match trees of an application produces a list of all the matches that can be applied without corrupting the behavior of the application. Next section shows how this information can be integrated in the Memory Exclusion Graph (MEG) of an application to combine the results of the buffer merging technique with the memory reuse techniques presented in Chapter 4.

5.5.3 MEG Update

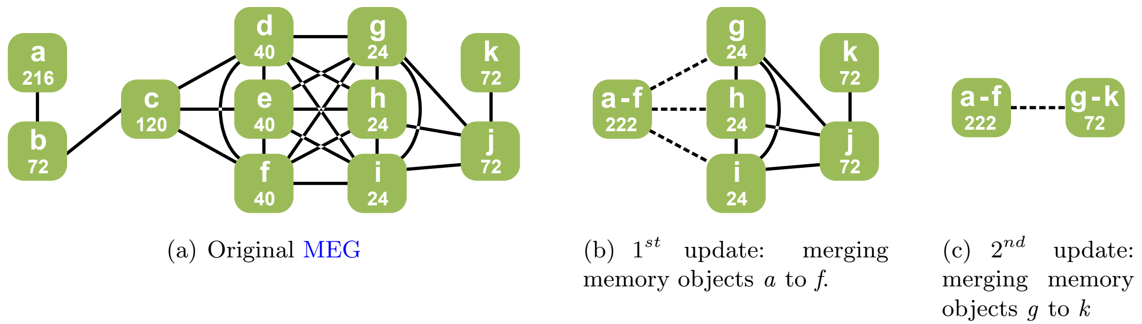
The memory optimization technique presented in Chapter 4 enables the allocation of single-rate FIFOs with disjoint lifetimes in overlapping address ranges of a shared memory MPSoC. Contrary to **match trees that express the merging opportunities** between buffers, this memory reuse technique is based on a **Memory Exclusion Graph (MEG)** that **only models the exclusions between buffers**. This section presents how the MEG of an application can be updated to benefit from both optimization techniques.

Two updates of the MEG can be successively applied. The first update consists of integrating the results produced by the match tree minimization process (Section 5.5). The second update consists of using port annotations provided by the application developer (Section 5.4.2) to remove exclusions from the MEG.

Merging Memory Objects

Each buffer of the match trees corresponds to a single vertex, called memory object, of the MEG derived from the same single-rate SDF graph. For example, each memory object in Figure 5.15(a) corresponds to a single-rate FIFO of the SDF graph of Figure 5.9.

Updating the MEG with the results of the match tree optimization process consists of replacing the memory objects corresponding to a set of merged buffers with a single memory object. The weight of this new memory object corresponds to the size of the merged buffer obtained at the end of the match tree folding process. For example, the MEG presented in Figure 5.15(b) results from the merge of buffers a to f into a single buffer of size 222, following buffer merging results of Figure 5.14(a). This MEG can be



(a) Original MEG

(b) 1st update: merging memory objects a to f.

(c) 2nd update: merging memory objects g to k

Figure 5.15: *Update of the MEG derived from the single-rate SDF graph of Figure 5.9.*

updated a second time by applying the matches of the match tree formed by buffers g to k, as presented in Figure 5.15(c).

As depicted by the dotted exclusions in Figure 5.15(b) and 5.15(c), memory objects resulting from the merge of several buffers still have exclusions with other memory objects. An exclusion is added between a memory object m and a merged memory object m_{merged} if any memory object of m_{merged} previously had an exclusion with m. For example, in Figure 5.15(b), exclusions are added between the merged memory object a-f and memory objects g, h, and i because there were exclusions between these memory objects and memory objects d, e, and f in the original MEG.

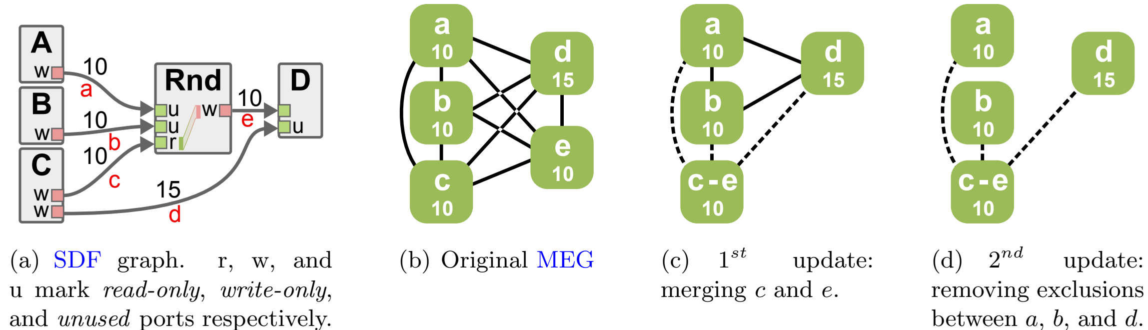
Exclusions linking a merged memory object to other memory objects of a MEG are special exclusions that may express a partial exclusion between memory objects. For example, in Figure 5.15(b), memory object g had exclusions with three memory objects d, e, and f of the merged memory object a-f. Since these three memory objects are packed in the first 80 bytes of the merged memory object (cf. Figure 5.14(a)), memory object g can safely be allocated between the 80th byte and the end of memory object a-f. The memory reuse opportunities offered by partial exclusions can be exploited during the memory allocation process by checking the validity of overlapping allocations for memory objects linked by a partial exclusion.

Removing Exclusions

Port annotations *unused* and *write-only* can be respectively used by the developer of an application to specify that the data tokens of an input port of an actor will never be used, and to specify that the data tokens written on an output port will never be read again by their producer. Hence, if a single-rate FIFO is connected both to a *write-only* output port and an *unused* input port, the data tokens written on this FIFO will never be used by the application and can be discarded as soon as they are produced. Even though single-rate FIFOs connected to a *write-only* and an *unused* port contain only useless data tokens, their allocation in memory is still required. Indeed, the actor producing the data tokens on this FIFO has no knowledge that these tokens will never be used, and an output buffer must still be provided.

Since the allocation of *write-only* and *unused* FIFOs is mandatory, but their content is useless, all corresponding buffers can be allocated in overlapping memory spaces, regardless of their respective lifetimes. Indeed, if two *write-only* and *unused* buffers are allocated in overlapping memory ranges, the content of one buffer might be corrupted when the other content is written. However, this data corruption will not change the application behavior since the content of these buffers will never be used. This mechanism is especially useful

for IBSDF graphs whose hierarchy flattening results in the insertion of many *Roundbuffer* actors with *unused* input ports (cf. Chapter 2).



(a) SDF graph. r, w, and u mark *read-only*, *write-only*, and *unused* ports respectively.

(b) Original MEG

(c) 1st update: merging c and e.

(d) 2nd update: removing exclusions between a, b, and d.

Figure 5.16: *Removing exclusion between* write-only *and* unused *memory objects of a MEG.*

To enable the allocation of memory objects corresponding to *write-only* and *unused* FIFOs in overlapping memory ranges, the exclusion between them should simply be removed from the MEG. Figure 5.16(a) gives an example of SDF graph with *unused* and *write-only* FIFOs. The MEG derived from this SDF graph is presented in Figure 5.16(b). A first update of this MEG (Figure 5.16(c)) consists of merging memory objects c and e according to the matches created by the memory script of actor *Roundbuffer*. Then, a second update consists of removing exclusions between memory objects a, b and d that correspond to single-rate FIFOs connecting *write-only* ports to *unused* ports (Figure 5.16(d)).

Since all the memory objects of the original MEG exclude each other, 55 bytes of memory are needed for their allocation. After the first update, the remaining memory objects still form a clique and 45 bytes are still needed for their allocation. Finally, after the last update, memory objects a, b, and d are no longer linked by an exclusion and can be allocated in overlapping memory ranges. Hence, only 25 bytes of memory are needed to allocate the MEG of Figure 5.16(d). On this application, the application of the buffer merging technique leads to a reduction of the memory footprint by 55%.

An evaluation of the memory optimization and buffer merging technique on a real application is presented in the next chapter.

CHAPTER 6

Memory Study of a Stereo Matching Application

6.1 Introduction

The goal of this chapter is to show, through the example of a computer vision application, how memory optimization techniques presented in Chapters 4 and 5 can be used to efficiently address the memory challenges encountered during the development of an application on an embedded multicore processor.

The computer vision application which serves as a memory case study and the target MPSoC architectures are described in Section 6.2. The practical challenges addressed in this chapter are presented in Section 6.3. Finally, experimental results on the computer vision application are presented in Section 6.4.

6.2 System Presentation

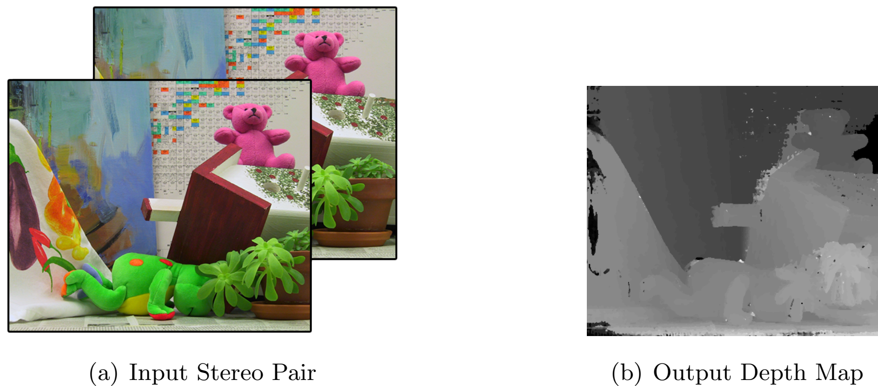
6.2.1 Stereo Matching Application

Over the last decade, the popularity of data-intensive computer vision applications has rapidly grown. Research in computer vision traditionally aims at accelerating execution of vision algorithms with Desktop GPUs or hardware implementations. The recent advances in computing power of embedded processors have made embedded systems promising targets for computer vision applications. Nowadays, computer vision is used in a wide variety of applications, ranging from driver assistance [ABBB13], to industrial control systems [MPZ+03], and handheld augmented reality [Wag07]. When developing data-intensive computer vision applications for embedded systems, addressing the memory challenges is an essential task as it can dramatically impact the performance of a system.

Despite the large silicon area allocated to memory banks, the amount of internal memory available on most embedded MPSoCs is still limited. Consequently, supporting the development of computer vision applications on high-resolution images remains a challenging objective.

The computer vision application studied in this chapter is a stereo matching algorithm. Stereo matching algorithms are used in many computer vision applications such as [ABBB13, Emb13]. As illustrated in Figure 6.1, the purpose of stereo matching algorithms is to process a pair of images (Figure 6.1(a)) taken by two rectified cameras

separated by a small distance in order to produce a disparity map (Figure 6.1(b)). A disparity map is an image the same size as the input image that corresponds to the 3rd dimension (the depth) of the captured scene. The disparity map is obtained by matching each pixel of the first image of the stereo pair with a corresponding pixel in the second image of the pair. The disparity produced in the output map corresponds to the horizontal distance between the positions of two matched pixels in the stereo pair. The large memory requirements of stereo matching algorithms make them interesting case studies to validate the memory analysis and optimization techniques presented in Chapters 4 and 5.



(a) Input Stereo Pair (b) Output Depth Map

Figure 6.1: *Stereo Matching Example from [SS02] database*

Stereo matching algorithms can be sorted in two classes [SZ00]:

- **Global algorithms**, such as graph cuts [Roy99], are minimization algorithms that produce a depth map while minimizing a cost function on one or multiple lines of the input stereo pair. Despite the good quality of the results obtained with *global* algorithms, their high complexity make them unsuitable for real-time or embedded applications.

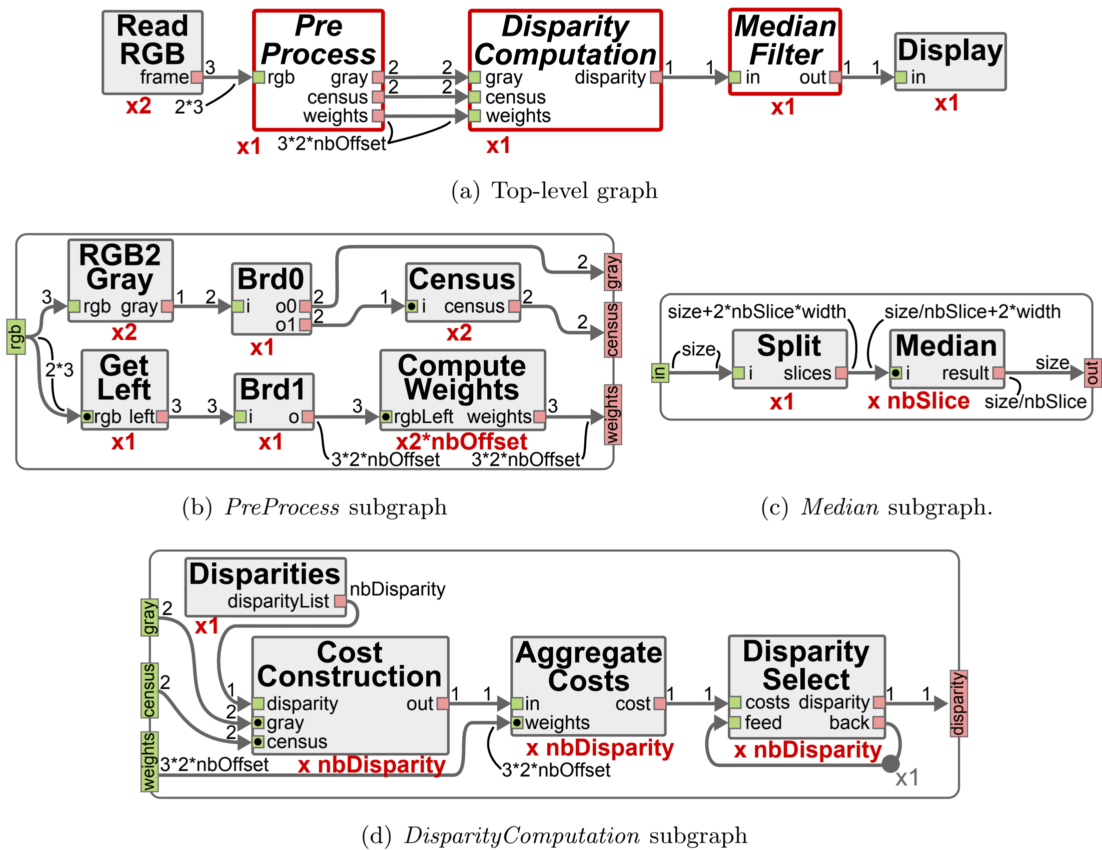
- **Local algorithms** independently match each pixel of the first image with a pixel selected in a restricted area of the second image [ZNPC13]. The selection of the best match for each pixel of the image is usually based on a correlation calculus.

The stereo matching algorithm studied in this chapter is the algorithm proposed by Zhang et al. in [ZNPC13]. The low complexity, the high degree of parallelism, and the good accuracy of the result make this algorithm an appropriate candidate for implementation on an embedded MPSoC.

IBSDF Graph

The IBSDF graph of the stereo matching algorithm is presented in Figure 6.2. For the sake of readability, except in Figure 6.2(c), all token production and consumption rates displayed in the IBSDF graph are simplified and should be multiplied by the number of input image pixels (*size*) to obtain the real exchange rates.

Below each actor, in bold, is a repetition factor which indicates the number of executions of this actor during each iteration of the graph. This number of executions is deduced from the data productions and consumptions of actors. Three parameters are used in these graphs: namely *nbDisparity*, *nbOffset*, and *nbSlice*. *nbDisparity* represents the number of distinct values that can be found in the output disparity map. *nbOffset* is



(a) Top-level graph

(b) *PreProcess* subgraph

(c) *Median* subgraph.

(d) *DisparityComputation* subgraph

Figure 6.2: *Stereo-matching IBSDF graphs.*[1]

a parameter influencing the size of the pixel area considered for the correlation calculus of the algorithm [ZNPC13]. *nbSlice* parameterizes the number of image slices processed in parallel in the *MedianFilter* subgraph.

Overall, the *Top-level* IBSDF graph (Figure 6.2(a)) and its 3 hierarchical subgraphs contain 12 distinct actors.

- **ReadRGB** produces the 3 color components of an input image by reading a stream or a file. This actor is called twice: once for each image of the stereo pair.
- ***PreProcess* subgraph** (Figure 6.2(b))
  - **GetLeft** gets the RGB left view of the stereo pair.
  - **RGB2Gray** converts an RGB image into its grayscale equivalent.
  - **BrdX** is a Broadcast actor. Its only purpose is to duplicate on its output ports the data tokens consumed on its input port.
  - **Census** produces an 8-bit signature for each pixel of an input image. This signature is obtained by comparing each pixel to its 8 neighbors: if the value of the neighbor is greater than the value of the pixel, one signature bit is set to 1; otherwise, it is set to 0.
  - **ComputeWeights** produces 3 weights for each pixel using characteristics of neighboring pixels. *ComputeWeights* is executed twice for each offset: once

[1] All rates of the graphs are implicitly multiplied by the picture size, except in graph 6.2(c)

considering a vertical neighborhood of pixels, and once with a horizontal neighborhood.

- ***DisparityComputation* subgraph** (Figure 6.2(d))
  - **Disparities** produces the list of disparities whose matching cost must be computed.
  - **CostConstruction** is executed once per possible disparity level. By combining the two images and their census signatures, it produces a value for each pixel that corresponds to the cost of matching this pixel from the first image with the corresponding pixel in the second image shifted by a disparity level.
  - **AggregateCosts** computes the matching cost of each pixel for a given disparity. Computations are based on an iterative method that is executed *nbOffset* times.
  - **DisparitySelect** produces a disparity map by selecting, for each pixel, the disparity of the input cost map with the lowest matching cost. The first input is a cost map computed for a specific disparity level by the *AggregateCosts* actor, and the second input is a cost map resulting from a previous firing of the actor. Hence, the *nbDisparity* successive firings of this actor iteratively select the disparity with the lowest matching cost among all tested disparities.
- ***MedianFilter* subgraph** (Figure 6.2(c))
  - **Split** divide the input disparity map into *nbSlice* equal parts. To ensure the correct behavior of the median filter, two extra lines of pixels are added to each slice: the last line of the previous slice, and the first line of the next slice.
  - **Median** applies a 3×3 pixels median filter to the input slice of the disparity map to smooth the results.
- **Display** displays the result of the algorithm or writes it in a file.

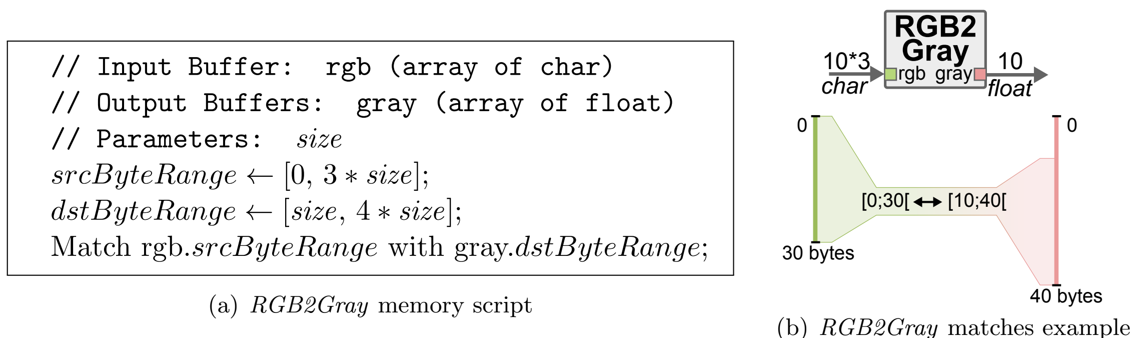
This IBSDF description of the algorithm provides a high degree of parallelism since it is possible to execute in parallel the repetitions of the three most computationally intensive actors, namely *CostConstruction*, *AggregateCosts*, and *ComputeWeights*. A detailed description of the original stereo matching algorithm can be found in [ZNPC13] and the IBSDF implementation studied in this chapter is available online [DZ13].

The *DisparityComputation* subgraph presented in Figure 6.2(d) illustrates the special behavior of data interfaces in the IBSDF MoC. Each firing of the *AggregateCosts* actor consumes $3*2*nbOffset$ data tokens from the *weights* data input interface of the subgraph. Because only $3*2*nbOffset$ tokens are available on the *weights* data input interface, the data input interface will behave as a ring buffer and the same data tokens will be "produced" on the interface for each firing of the *AggregateCosts* actor. An equivalent behavior can be obtained by adding a *Broadcast* actor between the *weights* interface and the *AggregateCosts* actor. *Broadcast* actors are also implicitly used after the *gray* and *census* data input interfaces of the subgraph, and after the *rgb* data input interface of the *PreProcess* subgraph (Figure 6.2(b)). Similarly, the *disparity* data output interface behaves as a *Roundbuffer* actor, only forwarding the last consumed data tokens to the corresponding data output port of the hierarchical actor.

Memory Scripts

An important challenge to face when implementing the stereo matching application is the explosion of the memory space requirements caused by the *Broadcast* actors. For example in Figure 6.2(d), with $nbOffset = 5$, $nbDisparity = 60$ and a resolution of $size = 450 * 375$ pixels, the implicit *Broadcast* actor for the *weights* IBSDF interface produces $3*2*nbOffset*nbDisparity*size$ float values, or 1.13 GBytes of data. Beside the fact that this footprint alone largely exceeds the 512 MBytes capacity of the targeted multicore DSP architecture (cf. Section 6.2.2), this amount of memory is a waste as it consists only of 60 duplications of the 19.3 MBytes of data produced by the firings of the *ComputeWeights* actor.

Memory scripts and port annotations (Section 5.4) were added to the stereo matching IBSDF graph to prevent the allocation of excessively large memory footprints. In Figure 6.2, *read-only* input ports are marked with a black dot within the port anchor. Figure 6.3 presents the memory script associated to the *RGB2Gray* actor. As illustrated in the figure, the *RGB2Gray* actor transforms each pixel coded on 3 `char` values (r, g, and b) into a single *float* value (gray). Because the input and output buffers have different data types, ranges of single precision bytes are used in the memory script instead of ranges of data tokens (Figure 6.3(a)). To maximize memory reuse, three quarters of the output buffer can be matched directly within the input buffer, as illustrated in Figure 6.3(b).



(a) *RGB2Gray* memory script

(b) *RGB2Gray* matches example

Figure 6.3: *Memory script for* RGB2Gray *actors*

Memory scripts associated to *Broadcast* and *Split* actors are presented in Chapter 5, in Figures 5.7(a) and 5.8(a) respectively.

6.2.2 Target Multicore Shared Memory Architectures

This section presents the two multicore architectures that were considered for the implementation of the stereo matching algorithm.

Intel i7-3610QM Multicore CPU

The *i7-3610QM* is a multicore Central Processing Unit (CPU) manufactured by Intel [Int13]. For simplicity, this processor will be called *i7* in the remainder of this chapter. This 64-bit processor contains 4 physical hyper-threaded cores that are seen as 8 virtual cores from the application side. This CPU has a clock speed varying between 2.3 GHz and 3.3 GHz. Using virtual memory management technique, this CPU provides virtually unlimited memory resources to the applications it executes. A detailed description of this architecture can be found in [Int13].

Texas Instrument TMS320C6678 Multicore DSP Architecture

The *TMS320C6678* is a multicore Digital Signal Processing (DSP) architecture manufactured by Texas Instruments [Tex13]. For simplicity, this processor will be called *C6678* in the remainder of this chapter. This MPSoC contains 8 C66x DSP cores, each running at 1.0 GHz on the experimental evaluation module. Each C66x DSP core possesses 32 KBytes of L1 data cache, 32 KBytes of L1 program cache, and 512 KBytes of L2 unified cache. In addition, the 8 DSP cores also share 4 MBytes of L2 memory and an addressable memory space of 8 Gbytes. Although the size of the addressable memory space is 8 GBytes, the targeted evaluation module contains only 512 MBytes of shared memory.

Contrary to the Intel's CPU, the *C6678* does not have a hardware cache coherence mechanism to manage the private caches of each of its 8 cores. Consequently, it is the developer's responsibility to use *writeback* and *invalidate* functions to make sure that data stored in the two levels of private cache of each core is coherent.

The diverse memory characteristics and constraints of the two architectures must be taken into account when implementing an application. Section 6.3 presents the memory challenges encountered when implementing the stereo matching application on these two architectures.

6.3 Practical Issues

6.3.1 Multicore Cache Coherence

Cache management is a key challenge when implementing an application on a multicore target without automatic cache coherence. Indeed, as shown in [URND06], the use of cache dramatically improves the performance of an application on multicore DSP architectures, with execution times up to 24 times shorter than without cache.

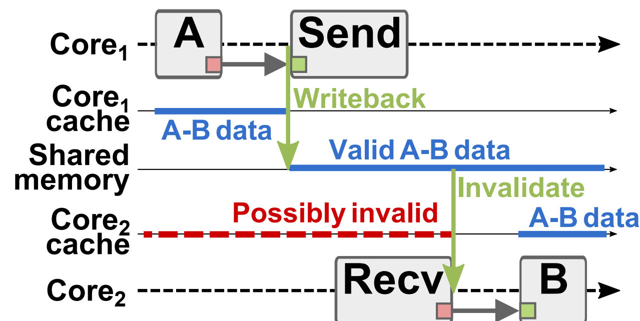


Figure 6.4: *Cache coherence solution without memory reuse*

An automatic method to insert calls to *writeback* and *invalidate* functions in code generated from an SDF graph is presented in [URND06]. As depicted in Figure 6.4, this method is applicable for shared memory communications between two processing elements. Actors A and B both have access to the shared memory addresses where data tokens of the AB FIFO are stored. The synchronization between cores is ensured by the *Send* and *Recv* actors which can be seen as *post* and *pend* semaphore operations respectively. A *writeback* call is inserted before the *Send* operation to make sure that all AB data tokens from Core$_1$ cache are written back in the shared memory. Similarly, an *invalidate* call

is inserted after the *Recv* operation to make sure that cache lines corresponding to the address range of buffer AB are removed from Core_2 cache.

Problem

As depicted in Figure 6.5, a problem arises if the method presented in [URND06] is used jointly with the memory reuse techniques introduced in Chapters 4 and 5.

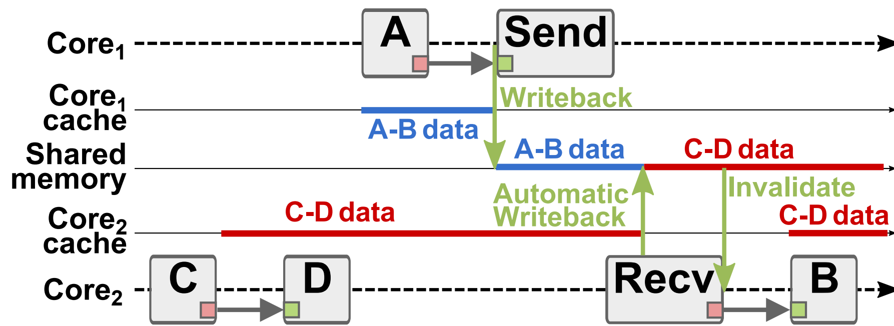


Figure 6.5: *Cache coherence issue with memory reuse*

In the example of Figure 6.5, overlapping memory spaces in shared-memory are used to store data tokens of two FIFOs: AB and CD. After the firings of actors C and D, the cache memory of Core_2 is "dirty", containing data tokens of FIFO CD that were not written back in the shared memory. Because these data tokens are "dirty", the local cache manager, which runs the cache replacement algorithm (Section 3.4.1), might generate an automatic *writeback* to put new data in the cache. If, as in the example, this automatic *writeback* occurs after the *writeback* from Core_1, then the data tokens of FIFO CD will overwrite tokens of FIFO AB in the shared memory, thus corrupting the data accessed by actor B.

Solution

Figure 6.6 presents a solution to the issue illustrated in Figure 6.5.

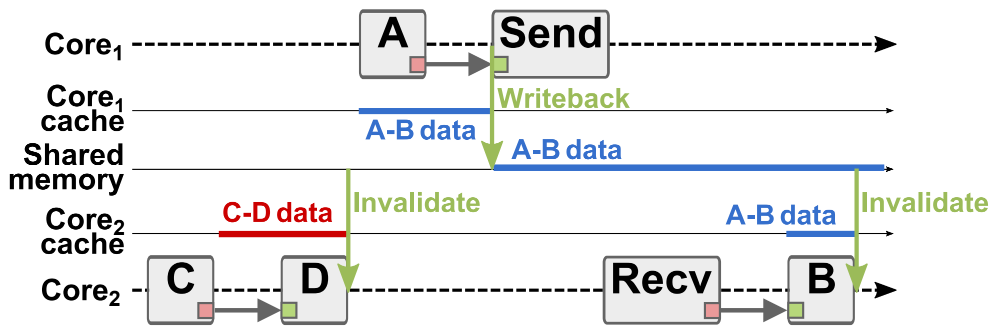


Figure 6.6: *Multicore cache coherence solution*

The proposed solution to prevent unwanted *writebacks* is to make sure that no dirty lines of cache remains once the data tokens of a FIFO have been consumed. To this purpose, a call to the *invalidate* function is inserted for each input buffer, after the completion of the actor consuming this buffer. As illustrated in Figure 6.6, new calls to the *invalidate*

function replace those inserted after the *Recv* synchronization actor. Insertion of calls to the *writeback* function before *Send* actors is not affected by this solution.

Special care must be taken when inserting cache coherence function calls around the firing of an actor associated to a memory script. For example, when firing the *RGB2Gray* actor from the stereo matching application (Figure 6.3), the memory accessed on the *rgb* input port should not be invalidated. Indeed, if the match created by the associated memory script is applied, the result of the actor firing is stored in a memory range overlapping the memory allocated for the input buffer. Invalidating the input buffer would result in discarding the data tokens produced by the actor. To avoid this issue, a call to the *invalidate* function is inserted only for ranges of input buffers that are not matched within output buffers of the actor.

6.3.2 Data Alignment

Aligning a data structure in memory consists of allocating this data structure in a memory range whose starting address is a multiple of n bytes, where n is generally a power of 2. A data structure respecting this condition is said to be *n-aligned*. An aligned memory access is an access to a data structure of n bytes stored at an *n-aligned* address. For example, if a 32-bit float variable is stored at address `0xA8`, accesses to this variable will be aligned since `0xA8` is a multiple of 4 bytes. However, if the same float variable is stored at address `0xA3`, accesses to this variable will not be aligned.

Aligning data in memory has a positive impact on application performance. Many architectures, including the two target architectures [Tex13, Int13], have assembly instructions that are optimized for aligned data accesses. For example, as exposed in [KLC10], SSE instructions that are available on most commercial CPU architectures are optimized for memory accesses aligned on 16 bytes (i.e. 128 bits).

The memory allocators implemented in PREESM and presented in Section 4.4.3 can be configured to allocate memory objects only on aligned addresses [Des13].

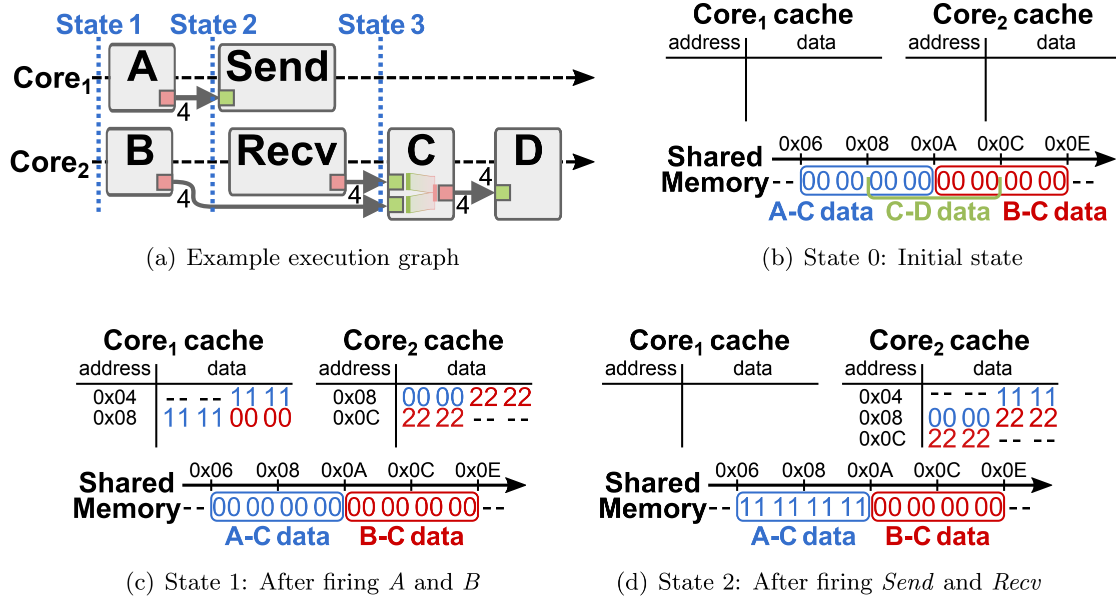
Problem with memory scripts and cache line alignment

Figure 6.7 illustrates an issue that arises when incoherent caches are used on a multicore architecture.

Figure 6.7(a) presents an SDF graph mapped on 2 cores. All production and consumption rates of this graph are equal to 4 bytes. To ensure the synchronization between the 2 cores, *Send* and *Recv* actors have been added to the SDF graph. Actor C is associated with a memory script that matches both input buffers into the output buffer. Figure 6.7(b) shows the memory allocation resulting from the application of the matches created by the memory script, as well as the initial content of the memory and private caches of the architecture.

Figure 6.7(c) presents the state of the system after the firing of actors A and B. At this point, each cache of the architecture has been filled with two lines of 4 bytes. Caches of $Core_1$ and $Core_2$ contain the result produced by actors A and B respectively. At this point, the content of the shared memory is unchanged since no *writeback* call was issued by the cores.

Figure 6.7(d) presents the state of the system during the firing of actor C. As presented in Section 6.3.1, lines of $Core_1$ cache corresponding to the A-C data sent by the *Send* actor were *written back* to the shared memory. B-C data however is not *written back* to the shared memory since it is not involved in an inter-core communication. Because the last two bytes of A-C data and the first two bytes of B-C data are allocated in consecutive



(a) Example execution graph (b) State 0: Initial state

(c) State 1: After firing A and B (d) State 2: After firing *Send* and *Recv*

Figure 6.7: *Cache Alignment Issue*

memory spaces, these bytes are cached as a single 4 bytes line of cache beginning at address `0x08`. After firing actor B on $Core_2$, line `0x08` of the cache contains valid data produced by this actor. When actor C is executed, line `0x08` of the cache will be used as it is, despite the presence of invalid A-C bytes at the beginning of the cache line. Hence, in the absence of cache coherence mechanism, actor C will read corrupted A-C data from $Core_2$ cache.

Solution

In the absence of memory scripts, preventing several buffers from being cached in the same line of cache is achieved by allocating all buffers at addresses that are multiples of the cache line size. When memory scripts are used, aligning buffers on cache line size is not sufficient. For example, in Figure 6.7, output buffer of actor C is allocated at address `0x08` which is aligned with the cache line size of 4 bytes. However, because input buffers of actor C are partially merged in the output buffer, their allocation is not aligned on the cache line size.

The solution to this issue is to prevent the application of matches that would result in caching several buffers in the same line of cache. To achieve this purpose, Algorithm 6.1 is executed before applying the matches created by the memory scripts associated to the actors of the application graph.

The principle of Algorithm 6.1 is to find all output buffers that are involved in a match (lines 1 to 3), and enlarge the ranges of bytes of these buffers. Enlarging the range of bytes of all selected buffers is achieved by adding $size_{cacheLine} - 1$ bytes at the beginning and at the end of the buffers (lines 6 and 7). If, as in *Fork* or *Broadcast* actors, no write operation is executed on an output buffer, the corresponding lines of cache will not contain dirty data for these buffers. Consequently, several buffers respecting this condition may be cached in the same line of cache without corrupting the application behavior. Such buffers are identified in lines 4 and 5 of the algorithm and their ranges of bytes are not enlarged.

Algorithm 6.1: Prevent caching of multiple buffers in a same cache line

```
   Input: G = (A, F) an IBSDF graph
          T = (B, M) the match tree associated to G
1  for each actor a ∈ A do
2    for each output buffer b ∈ B_a^out do
3      if ∃ match m ∈ M such that m.b_src = b or m.b_dst = b then
4        if b.r_merge = ∅
5          or (∃ match m ∈ M such that m.b_dst = b and m.b_src.r_merge = ∅) then
6            b.r_bytes.start ← 0 − (size_cacheLine − 1);
7            b.r_bytes.end ← b.r_bytes.end + (size_cacheLine − 1);
8        end
9      end
10   endfor
11 endfor
```

Adding new ranges of bytes to buffers results in the creation of new conflicts between matches of the match tree. For example, in Figure 6.7(a), the ranges of bytes of the output buffers of actors A and B will both be equal to $[-3, 7[$, instead of $[0, 4[$, after the application of Algorithm 6.1. As a consequence, an overlap appears between the destination ranges of the two matches created by actor C. As presented in Section 5.5.1, an overlap between destination ranges is a source of conflict that prevents the concurrent application of corresponding matches.

The $size_{cacheLine} - 1$ bytes added to the range of bytes of the buffers can be seen as a "safety distance" between a buffer and the next. For example, if the last byte of a buffer is aligned on the $size_{cacheLine}$, no buffer will be merged in the following $size_{cacheLine} - 1$ bytes, which corresponds to the remaining bytes of the line of cache. Hence, this "safety distance" guarantees that as soon as a single byte of a buffer is cached in a line of cache, this whole line of cache will be dedicated to this buffer in the buffer matching process.

A negative counterpart of solutions to data alignment issues is an augmentation of the memory footprint allocated for applications. Indeed, forcing the allocation of buffers on aligned addresses and preventing the application of some matches for cache coherence purposes diminish the efficiency of the memory optimization techniques. However, experimental results presented in the next section show that this augmentation of the memory footprint is a negligible side effect compared to the performance improvement caused by the activation of caches.

6.4 Experimental Results

The stereo matching algorithm presented in Figure 6.2 was implemented within the PREESM rapid prototyping framework. Beside modeling the stereo matching IBSDF graph, PREESM was used for mapping and scheduling the application, optimizing the memory allocation, and generating compilable code for each of the two targeted architectures.

6.4.1 Static Memory Optimization

Table 6.1 shows the memory characteristics resulting from the application of memory optimization techniques presented in this thesis to the IBSDF graph of the stereo matching

algorithm. The memory characteristics of the application are presented for 4 scenarios, each corresponding to a different implementation stage of the algorithm. The $|M|$ and $\delta(G)$ columns respectively give the number of memory objects and the density of exclusion of the MEG derived from the application graph. The next two columns present the *upper* and *lower* allocation bounds for each scenario. Finally, the last two columns present the actual amount of memory allocated for each target architecture. The allocation results are expressed as the supplementary amount of memory allocated compared to the *lower* bound. These results were obtain with $nbOffset = 5$, $nbDisparity = 60$ and a resolution of $size = 450 * 375$ pixels.

	MEG		Bounds (MBytes))		Allocations[2] (MBytes)	
Scenarios	$\|M\|$	$\delta(G)$	Upper	Lower	i7	C6678
Pre-schedule[1]	1000	0.68	1453.4	1314.2	+0	+0.051
Pre-schedule	437	0.57	170.2	99.6	+0.164	+0.679
Post-schedule	437	0.47	170.2	79.7	+0	+0.014
Post-timing	437	0.39	170.2	68.4	+0	+0.342

1: Memory scripts not applied in this scenario.
2: Relatively to the lower bound.

Table 6.1: *MEGs characteristics and allocation results*

Application of Buffer Merging

A comparison between the two pre-schedule scenarios of Table 6.1 reveals the impact of the merging of buffers presented in Chapter 5. The first pre-schedule scenario presented in the table corresponds to the memory characteristics of the stereo matching application when buffer merging is not applied. With a memory footprint of 1314.2 MBytes, this scenario forbids the allocation of the application in the 512 MBytes shared memory of the multicore DSP architecture. The application of the buffer merging technique in the second scenario leads to a reduction of the memory footprint by 92%, from 1314.2 MBytes to 99.6 MBytes.

Another positive aspect of the buffer merging technique is the simplification of the MEG. Indeed, 563 vertices are removed from the MEG as a result of the buffer merging technique. The computation of the memory bounds of the MEG and the allocation of the MEG in memory are both accelerated by a factor of 6 with the simplified MEG.

In addition to the large reduction of the memory footprint, buffer merging also has a positive impact on the application performance. On the *i7* multicore CPU, the stereo matching algorithm reaches a throughput of 3.50 fps when buffer merging is applied, and a throughput of 1.84 fps otherwise. Hence, the suppression of the `memcpy` calls associated to *Broadcast*, *Fork*, and *Join* actors results in a speedup ratio of 90%. On the *C6678* DSP architecture, the suppression of the `memcpy` results in a speedup ratio of 40%, rising from 0.24 fps to 0.34 fps.

Memory Footprints

Results presented in Table 6.1 reveal the memory savings resulting from the application of the memory reuse techniques presented in Chapter 4. 170.2 MBytes of memory are required for the allocation of the last three scenarios if, as in existing dataflow frameworks [BMKDdD12, Par95], memory reuse techniques are not used. In the pre-scheduling scenario, memory reuse techniques lead to a reduction of the memory footprint by 41%.

This reduction of the memory footprint does not have any counterpart since the MEG is compatible with any schedule of the application (cf. Section 4.4). In the post-scheduling and in the post-timing scenarios, the memory footprints are respectively reduced by 53% and 59% compared to the memory footprint obtained without memory reuse. The memory footprints allocated on the *i7* CPU for these scenarios are optimal since the lower bounds for the MEGs and the allocation results are equal.

The memory footprints presented in Table 6.1 result from the allocation of the MEG with a Best-Fit (BF) allocator fed with memory objects sorted in the largest-first order. This allocator was selected because it was the most efficient allocation algorithm for the MEG derived from the stereo matching IBSDF graph.

Since all production and consumption rates of the stereo matching SDF graph are multiples of the image resolution, the memory footprints allocated with our method are proportional to the input image resolution. Using memory reuse techniques with $nbOffset = 5$ and $nbDisparity = 60$, the 512 MBytes of the *C6678* DSP architecture allow the processing of stereo images with a resolution up to 720p (1280*720 pixels). Without memory reuse, the maximum resolution that can fit within the *C6678* memory is 576p (704*576 pixels), which is 2.27 times less than when memory reuse is in effect.

Cache Activation

Because of cache alignment constraints, the memory allocation results presented in Table 6.1 for the *C6678* multicore DSP architecture are slightly larger than the results for the *i7* CPU. On average, the alignment of buffers on L2 cache line of 128 bytes results in an allocation increase of only 0.3% compared with the unaligned allocation of the *i7* CPU.

As presented in Section 6.3.1, the insertion of *writeback* and *invalidate* calls in the code generated by PREESM allows the activation of the caches of the *C6678* multicore DSP architecture. Without caches, the stereo-vision application reaches a throughput of 0.06 fps. When the caches of the *C6678* architecture are activated, the application performance is increased by a factor of 5.7 and reaches 0.34 fps.

Evolution of the Performance and Memory Footprint Depending on the Number of Cores

Figure 6.8 shows the performance obtained by deploying the stereo matching algorithm on a variable number of cores of the *C6678* multicore DSP chip. On eight cores, a throughput of 0.34 fps is reached. This throughput corresponds to a speed-up by a factor 3.4 compared to the execution of the application on one DSP core.

Figure 6.8 also plots the theoretical greedy scheduling throughputs [PAPN12] computed by PREESM for the stereo matching application. In the current version, the computation of this theoretical throughput does not take into account inter-core communications nor cache operations. Consequently, the actual throughput of the stereo matching algorithm appears to be inferior to the theoretical throughput.

Figure 6.9 shows the memory footprint allocated for the execution of the stereo matching algorithm on a variable number of cores of the *C6678* multicore DSP chip. The smallest memory footprint of 68.4 MBytes is obtained when the application is executed on a single core of the architecture. When the number of cores executing the application is increased, more parallelism of the application is preserved, and the allocated memory footprint is increased. As illustrated in Figure 6.9, the memory optimization techniques presented in previous chapters help limit this increase of the memory footprint, and only 79.7 MBytes of memory are needed to execute the application on 3 to 8 cores.

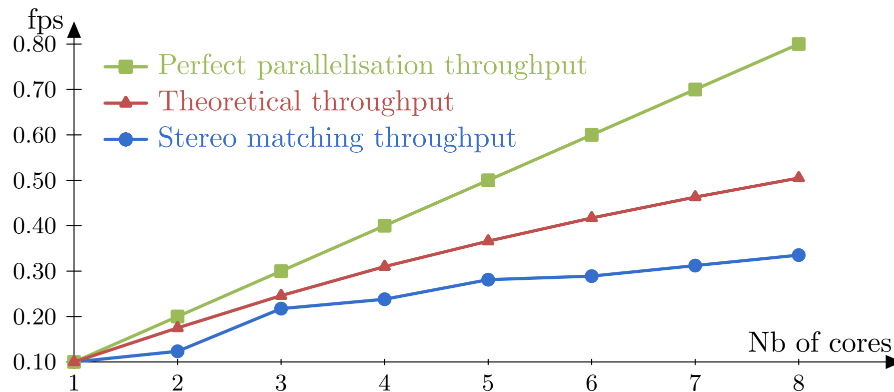


Figure 6.8: *Throughput of the stereo matching application depending on the number of cores.*

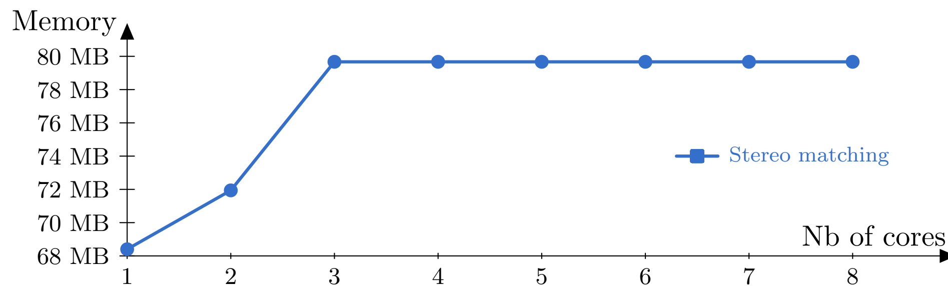


Figure 6.9: *Memory footprint of the stereo matching application depending on the number of targeted C6x cores.*

6.4.2 Comparison with Dynamic Memory Allocation

As presented in Section 4.4.2, similar footprints are obtained with dynamic allocation and static allocation in the post-timing scenario. In both cases, the memory allocated to a memory object can be reused as soon as the lifetime of this memory object ends. However, although dynamic allocators provide low memory footprints, their runtime overhead and their unpredictability make them bad choices when compared to static allocation.

Runtime Overhead

	Throughput		
Target	Static Allocation	Dynamic Allocation	Overhead
i7 CPU	3.57 fps	2.79 fps	22%
C6678 DSP	0.39 fps	0.26 fps	32%

Table 6.2: *Comparison of the stereo matching performance with static and dynamic allocations*

Table 6.2 presents the performance of the stereo matching algorithm on the *i7* CPU and on the *C6678* multicore DSP architectures. Two versions of the code were generated with PREESM: the first with post-scheduling allocation, and the second with dynamic memory allocation (`malloc` and `free`). For a fair comparison, the same schedule was used for both allocation strategies. To increase the application throughput in these tests, a software pipeline stage was added between the *AggregateCost* and the *DisparitySelect* actors.

Dynamic allocation has a negative impact on the performance of the application. On the *C6678* DSP architecture, the throughput reduction of 32% has four main sources:

- **The overhead of the dynamic allocator.** Each time a memory object is dynamically allocated, the online allocation algorithm searches for a free space of sufficient size in the heap to store this memory object.
- **The creation of critical sections.** Since a unique heap is shared between all cores of the architecture, access to this heap must be protected with a mutex. Each time a core executes a dynamic allocation primitive, a few clock cycles may be lost waiting for the mutex to be released by another core.
- **The extra synchronization added to the generated code to dynamically support the merging of buffers.** A semaphore is associated with each merged buffer and initialized with the number of actors accessing this buffer. Each actor accessing the merged buffer decrements the value of the semaphore after its firing. When the semaphore value reaches zero, a *free* operation is issued for the merged buffer.
- **The insertion of cache operations for the memory object pointers.** Each time a buffer is allocated on one core, a *writeback* call is issued to ensure that the pointer value is written back in the shared memory. Similarly, a call to *invalidate* is required when a core uses the pointer of a buffer allocated on another core.

On the *i7* CPU, the dynamic allocator overhead and the dynamic support for merged buffers also cause a throughput reduction of 22%.

Unpredictable Footprint

Although dynamic allocation provides memory footprints similar to post-timing allocation, the dynamic memory footprint cannot be bounded at compile time. To illustrate this issue, the dynamic memory footprint of the stereo matching algorithm was measured during 200 iterations of the graph execution, i.e. the processing of 200 stereo image pairs. This experiment was conducted on the *C6678* by measuring, after each iteration, the maximum size of the heap on which the memory objects were dynamically allocated. The experiment was repeated 12 times with the same code but with different cache configurations (activation of level 1 and level 2 caches, location of the code, debug or release). These different configurations modify actor execution times and thus the order of memory allocation primitive calls. Each blue curve in Figure 6.10 represents the footprints measured during one of the 12 experiments.

This experiment shows that the dynamic memory footprint of an application increases during the first iterations. This increase of the memory footprint is caused by the fragmentation of the memory. Memory fragmentation happens when a free space in the heap is too small to allocate new memory objects. Because the multicore DSP architecture has no defragmenting process, the memory fragmentation tends to accumulate during the first iterations of the application, which results in an increase of the heap size.

The memory footprints measured in Figure 6.10 range between 118.5 MBytes and 125.7 MBytes. The 6% difference between these two values illustrates the unpredictability of the dynamic memory footprint of applications. Finally, post-scheduling allocation for this schedule results in a memory footprint of 125.4 MBytes. Consequently, these experiments show that despite a slight reduction in the memory footprint with dynamic

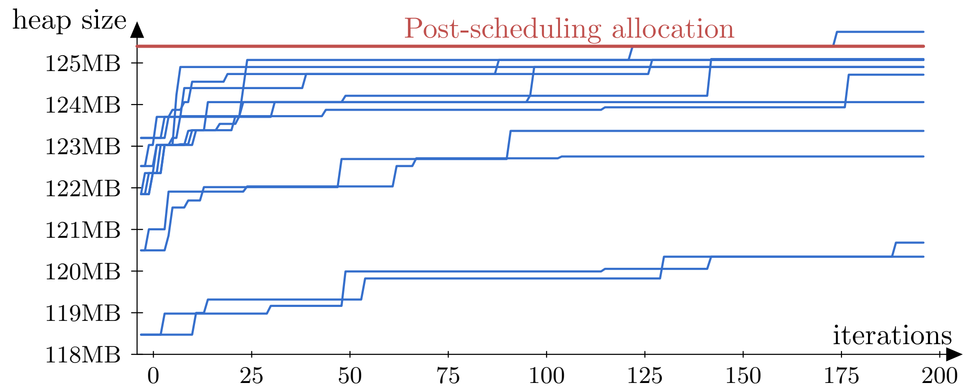


Figure 6.10: *Dynamic allocation: Heap size after N iterations. Each blue line represents the heap size for an execution of the stereo matching application.*

allocation, the exact memory footprint cannot be predicted with dynamic allocation and this dynamic footprint might exceed its static equivalent.

CHAPTER 7

Towards Parameterized Results: The PiSDF Model

7.1 Introduction

In this chapter, a new meta-model called Parameterized and Interfaced dataflow Meta-Model (PiMM) is introduced. PiMM extends the semantics of a targeted dataflow MoC to enable the specification of dynamically reconfigurable DSP applications. PiMM builds on a novel integration of two previously developed dataflow modeling techniques called parameterized dataflow [BB01] and interface-based dataflow [PBR09].

PiMM extends the semantics of a targeted dataflow MoC by introducing explicit parameters and parameter dependencies. Parameters can influence, both statically and dynamically, different properties of a dataflow MoC such as the firing rules of actors. PiMM also adds to the targeted MoC an interface-based hierarchy mechanism that enforces the compositionality of the extended model. This hierarchy mechanism also improves the targeted MoC predictability by restricting the scope of its parameters and by enabling a top-down parameterization. In this chapter, the capabilities of the PiMM meta-model are demonstrated by applying it to the SDF MoC.

The semantics of the dataflow MoCs used as a basis for building PiMM are reminded in Section 7.2. The hierarchy and parameterization semantics of PiMM is introduced in Section 7.3. Then, the compile time and runtime analyzability of PiMM is discussed in Section 7.4. Finally, Section 7.5 presents examples of applications modeled with PiMM and Section 7.6 compares PiMM with state-of-the-art dataflow MoCs.

7.1.1 Motivation: Need for Reconfigurable Models of Computation

The IBSDF MoC studied in previous chapters has a limited expressivity. Indeed, in IBSDF like in the underlying SDF MoC, the production and consumption rates of actors are fixed at compile time and cannot be dynamically changed depending on the application inputs or any other external factor (time, randomness, ...). Hence, the IBSDF MoC cannot be used to specify applications whose computation strongly depends on external factors.

An example of such an application is the PUSCH part of the LTE mobile telecommunication standard presented in [PAPN12]. The purpose of this application, executed by a base station of the network, is to decode data sent by the mobile phones connected to an antenna. The number of mobile phones connected to an antenna, and the bandwidth

allocated to each mobile phone are two parameters that can not be known and fixed at compile time. The variation domains of these two parameters results in more than 190 millions possible configurations for the PUSCH application. This great variability makes it impossible to store a statically computed schedule for each configuration. Hence, the MoC used to specify this application must support dynamic reconfiguration.

7.2 Foundation Dataflow Models of Computation

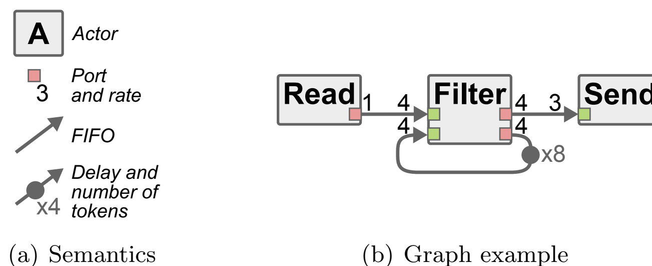
The Parameterized and Interfaced dataflow Meta-Model (PiMM) builds on a novel integration of two previously developed dataflow modeling techniques, called interface-based dataflow and parameterized dataflow. As presented in Chapter 2, the objective of these two MoCs is to enable the description of hierarchical and parameterizable graphs respectively. The semantics and important properties of these two models, as well as these of the SDF MoC, are reminded and detailed in the following sections. These MoCs will then serve as a basis for the formal definition of PiMM in Section 7.3.

7.2.1 Synchronous Dataflow

Synchronous Dataflow (SDF) [LM87b] is the most commonly used dataflow MoC. Production and consumption token rates set by firing rules are fixed scalars in an SDF graph. A static analysis of an SDF graph can be used to ensure consistency and schedulability properties of an application. The consistency and schedulability properties are crucial information for an application developer as they imply the deadlock-free execution and the bounded FIFO memory needs of the modeled application.

SDF Brief Definition

An SDF graph $G = (A, F)$ (Figure 7.1) contains a set of actors A that are interconnected by a set of FIFOs F. An actor $a \in A$ comprises a set of data ports $(P^{in}_{data}, P^{out}_{data})$ where P^{in}_{data} and P^{out}_{data} respectively refer to a set of data input and output ports. Data ports are used as anchors for FIFO connections. Functions $src : F \rightarrow P^{out}_{data}$ and $snk : F \rightarrow P^{in}_{data}$ associate source and sink data ports to a given FIFO. A data rate is specified for each port by the function $rate : A \times F \rightarrow \mathbb{N}$ corresponding to the fixed firing rules of an SDF actor. A delay $d : F \rightarrow \mathbb{N}$ is set for each FIFO. A delay corresponds to a number of tokens initially present in the FIFO.



(a) Semantics (b) Graph example

Figure 7.1: *Image processing example: SDF graph*

Properties

If an SDF graph is consistent and schedulable, a fixed sequence of actor firings can be repeated indefinitely to execute the graph, and there is a well defined concept of a minimal sequence for achieving an indefinite execution with bounded memory. Such a minimal sequence is called *graph iteration* and the number of firings of each actor in this sequence is given by the graph Repetition Vector (RV).

Graph consistency means that no FIFO accumulates tokens indefinitely when the graph is executed (preventing FIFO overflow). Consistency can be proved by verifying that the graph topology matrix has a non-zero vector in its null space [LM87b]. When such a vector exists, it gives the RV for the graph. The topology of an SDF graph characterizes actor interconnections as well as token production and consumption rates on each FIFO. A graph is schedulable if and only if it is consistent and has enough initial tokens to execute the first graph iteration (preventing deadlocks by FIFO underflow).

7.2.2 Interface-Based Synchronous Dataflow

Interface-Based SDF (IBSDF) [PBR09] is a hierarchical extension of the SDF model interpreting hierarchy levels as code closures. IBSDF fosters subgraph composition, making subgraph executions equivalent to imperative language function calls. Hence, IBSDF is a compositional MoC: the properties (schedulability, deadlock freeness, ...) of an IBSDF graph composed of several sub-graphs are independent from the internal specifications of these sub-graphs [Ost95]. IBSDF has proved to be an efficient way to model dataflow applications [PAPN12]. IBSDF interfaces are inherited by the PiMM meta-model proposed in Section 7.3.

IBSDF Brief Definition

In addition to the SDF semantics, IBSDF adds interface elements to insulate levels of hierarchy in terms of schedulability analysis. An IBSDF graph $G = (A, F, I)$ contains a set of interfaces $I = (I^{in}_{data}, I^{out}_{data})$ (Figure 7.2).

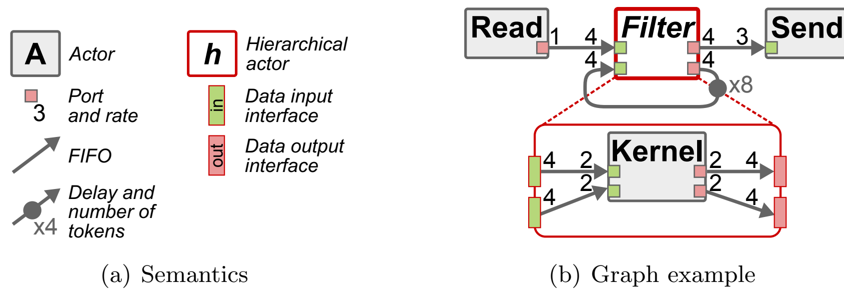
A *data input interface* $i^{in}_{data} \in I^{in}_{data}$ in a subgraph is a vertex transmitting to the subgraph the tokens received by its corresponding data input port. If more tokens are consumed on a data input interface than the number of tokens received on the corresponding data input port, the data input interface behaves as a ring buffer, producing the same tokens several times.

A *data output interface* $i^{out}_{data} \in I^{out}_{data}$ in a subgraph is a vertex transmitting tokens received from the subgraph to its corresponding data output port. If a data output interface receives too many tokens, it will behave like a ring buffer and output only the last pushed tokens.

[PBR09] details the behavior of IBSDF data input and output interfaces as well as the IBSDF properties in terms of compositionality and schedulability checking. Through PiMM, interface-based hierarchy can be applied to other dataflow models than SDF with less restrictive firing rules.

7.2.3 Parameterized Dataflow

Parameterized dataflow is a meta-modeling framework introduced in [BB01] that is applicable to all dataflow MoCs that present graph iterations. When this meta-model is applied, it extends the targeted MoC semantics by adding dynamically reconfigurable hierarchical actors. A reconfiguration occurs when values are dynamically assigned to the



(a) Semantics (b) Graph example

Figure 7.2: *Image processing example: IBSDF graph*

parameters of a reconfigurable actor, causing changes in the actor computation and in the production and consumption rates of its data ports.

PSDF Brief Definition

In *parameterized dataflow*, each hierarchical actor is composed of 3 subgraphs, namely the init ϕ_i, the subinit ϕ_s, and the body ϕ_b subgraphs.

The ϕ_i subgraph sets parameter values that can influence both the production and consumption rates on the ports of the hierarchical actor and the topology of the ϕ_s and ϕ_b subgraphs. The ϕ_i subgraph is executed only once per iteration of the graph to which its hierarchical actor belongs and can neither produce nor consume data tokens.

The ϕ_s subgraph sets the remaining parameter values required to completely configure the topology of the ϕ_b subgraph. The ϕ_s subgraph is executed at the beginning of each firing of the hierarchical actor. It can consume data tokens on input ports of the hierarchical actor but can not produce data tokens.

The ϕ_b subgraph is executed when its configuration is complete, right after the completion of ϕ_s. The body subgraph behaves as any graph implemented with the MoC to which the *parameterized dataflow* meta-model was applied.

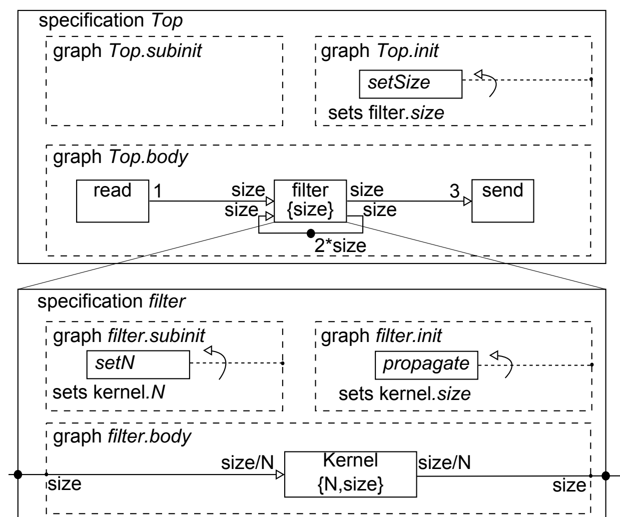


Figure 7.3: *Image processing example: PSDF graph*

PSDF is the MoC obtained by applying the *parameterized dataflow* meta-model to the SDF MoC. It has been shown to be an efficient way to prototype streaming applications [KSB+12]. The objective of PiMM is to further improve parameterization compared to

parameterized dataflow by introducing explicit parameter dependencies and by enhancing graph compositionality. Indeed, in a PSDF graph, ports are simple connectors between data FIFOs that do not insulate levels of hierarchy. For example, in the PSDF graph presented in Figure 7.3 the consumption rate on the data input port of the *filter* hierarchical actor depends on the RV of the actor subgraphs.

7.3 Parameterized and Interfaced Dataflow Meta-Modeling

The Parameterized and Interfaced dataflow Meta-Model (PiMM) can be used similarly to the *parameterized dataflow* to extend the semantics of all dataflow MoCs implementing the concept of graph iteration. PiMM adds both interface-based hierarchy and explicit parameter dependencies to the semantics of the extended MoC.

In this section PiMM is formally presented through its application to the SDF MoC. The model resulting from this application is called Parameterized and Interfaced SDF (πSDF) or (PiSDF).

7.3.1 πSDF Semantics

The pictograms associated to the different elements of the Parameterized and Interfaced SDF (πSDF) semantics are presented in Figure 7.4.

The πSDF semantics is formally defined as follows:

Definition 7.3.1. *A πSDF graph $G = (A, F, I, \Pi, \Delta)$ contains, in addition to the SDF actor set A and FIFO set F:*

- **I** *is a set of hierarchical interfaces. An interface $i \in I$ is a vertex of the graph. Interfaces enable the transmission of information (data tokens or configurations) between levels of hierarchy.*
- **Π** *is a set of parameters. A parameter $\pi \in \Pi$ is a vertex of the graph. Parameters are the only elements of the graph that can be used to configure the application and modify its behavior.*
- **Δ** *is a set of parameter dependencies. A parameter dependency $\delta \in \Delta$ is a directed edge of the graph. Parameter dependencies are responsible for propagating configuration information from parameters to other elements of the graph.*

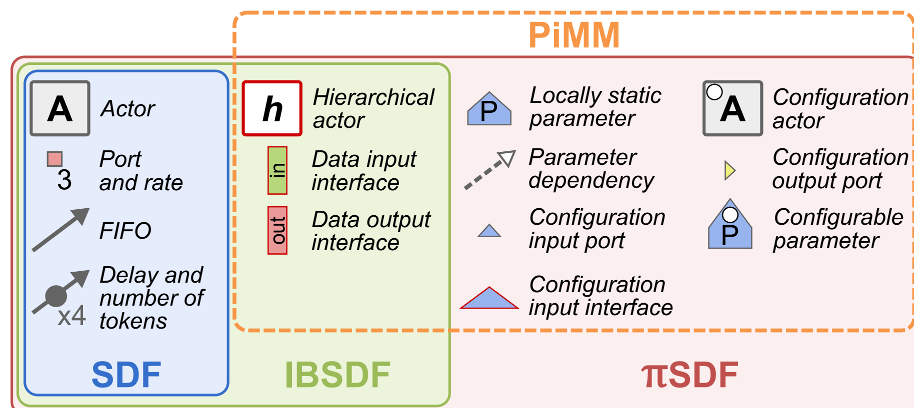


Figure 7.4: *PiMM semantics*

Parameterization Semantics

A parameter $\pi \in \Pi$ is a vertex of the graph associated to a parameter value $v \in \mathbb{N}$ that is used to configure elements of the graph. For a better analyzability of the model, a parameter can be restricted to take only values of a finite subset of $\mathbb{N}$. A *configuration* of a graph is the assignation of parameter values to all parameters in Π.

In a πSDF graph, an actor $a \in A$ is associated to a set of ports (P^{in}_{data}, P^{out}_{data}, P^{in}_{cfg}, P^{out}_{cfg}) where P^{in}_{cfg} and P^{out}_{cfg} are a set of configuration input and output ports respectively. A configuration input port $p^{in}_{cfg} \in P^{in}_{cfg}$ of an actor $a \in A$ is an input port that depends on a parameter $\pi \in \Pi$ and can influence the computation of a and the production/consumption rates on the dataflow ports of a. A configuration output port $p^{out}_{cfg} \in P^{out}_{cfg}$ of an actor $a \in A$ is an output port that can dynamically set the value of a parameter $\pi \in \Pi$ of the graph (Section 7.3.2).

A parameter dependency $\delta \in \Delta$ is a directed edge of the graph that links a parameter $\pi \in \Pi$ to a graph element influenced by this parameter. Formally a parameter dependency δ is associated to the two functions $setter : \Delta \rightarrow \Pi \cup P^{out}_{cfg}$ and $getter : \Delta \rightarrow \Pi \cup P^{in}_{cfg} \cup F$ which respectively give the source and the target of δ. A parameter dependency set by a configuration output port $p^{out}_{cfg} \in P^{out}_{cfg}$ of an actor $a \in A$ can only be received by a parameter vertex of the graph that will dispatch the parameter value to other graph elements, building a parameter dependency tree. Dynamism in PiMM relies on parameters whose values can be used to influence one or several 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. In PiMM, if an actor has all its production/consumption rates set to 0, it will not be executed.

A Parameter dependency Directed Acyclic Graph (PDAG) $T = (\Pi, \Delta)$ is formed by the set of parameters Π and the set of parameter dependencies Δ of a πSDF graph. A PDAG T is similar to a set of combinational relations where the value of each parameter is resolved virtually instantly as a function of the parameters it depends on. This PDAG is in contrast to the precedence graph formed by (A, F) where the firing of actors is enabled by the data tokens flowing on the Fifos.

πSDF Hierarchy Semantics

The hierarchy semantics used in πSDF inherits from the IBSDF model introduced in [PBR09]. In πSDF, a hierarchical actor is associated to a unique πSDF subgraph. The set of interfaces I of a subgraph is extended as follows: $I = (I^{in}_{data}, I^{out}_{data}, I^{in}_{cfg}, I^{out}_{cfg})$ where I^{in}_{cfg} is a set of *configuration input interfaces* and I^{out}_{cfg} a set of *configuration output interfaces*.

Configuration input and output interfaces of a hierarchical actor are respectively seen as a configuration input port $p^{in}_{cfg} \in P^{in}_{cfg}$ and a configuration output port $p^{out}_{cfg} \in P^{out}_{cfg}$ from the upper level of hierarchy.

From the subgraph perspective, a *configuration input interface* is equivalent to a locally static parameter whose value v is left undefined.

A *configuration output interface* enables the transmission of a parameter value from the subgraph of a hierarchical actor to upper levels of hierarchy. In the subgraph, this parameter value is provided by a Fifo linked to a data output port p^{out}_{data} of an actor that produces data tokens with values $v \in \mathbb{N}$.

In cases where several values are produced during an iteration of the subgraph, the configuration output interface behaves like a data output interface of size 1 and only the last value written will be produced on the corresponding configuration output port of the enclosing hierarchical actor.

It is important to note that *configuration output interfaces* are the only elements of the πSDF semantics that can be used to automatically convert a data token from a data output port into a parameter value. In the absence of *configuration output interface*, an actor firing is needed to convert a data token read on a data input port into a parameter value written on a configuration output port.

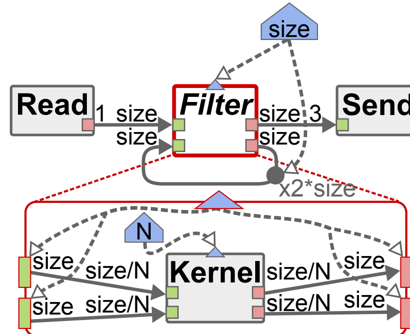


Figure 7.5: *Image processing example: Static πSDF graph*

Figure 7.5 presents an example of a static πSDF description. Compared to Figure 7.2, it introduces parameters and parameter dependencies that compose a PiMM PDAG. The modeled example illustrates the modeling of a test bench for an image processing algorithm. In the example, one token corresponds to a single pixel in an image. Images are read, pixel by pixel, by actor *Read* and sent, pixel by pixel, by actor *Send*. A whole image is processed by one firing of hierarchical actor $Filter$. A feedback edge with a delay stores previous images for comparison with the current one. Actor $Filter$ is refined by an actor $Kernel$ processing one Nth of the image. In Figure 7.5, the size of the image $size$ and the parameter N are locally static.

7.3.2 πSDF Reconfiguration

As introduced in [NL04], the frequency with which the value of a parameter is changed influences the predictability of the application. A constant value will result in a high predictability while a value which changes at each iteration of a graph will cause many reconfigurations, thus lowering the predictability.

There are two types of parameters $\pi \in \Pi$ in πSDF: configurable parameters and locally static parameters. Both restrict how often the value of the parameter can change. Regardless of the type, a parameter must have a constant value during an iteration of the graph to which it belongs.

Configurable Parameters

A configurable parameter $\pi_{cfg} \in \Pi$ is a parameter whose value is dynamically set once at the beginning of each iteration of the graph to which it belongs. Configurable parameters can influence all elements of their subgraph except the production/consumption rates on the data interfaces I_{data}^{in} and I_{data}^{out}. As explained in [BB01, NL04], this restriction is essential to ensure that, as in IBSDF, a parent graph has a consistent view of its actors throughout an iteration, even if the topology may change between iterations.

The value of a configurable parameter can either be set through a parameter dependency coming from another configurable parameter or through a parameter dependency coming from a configuration output port p_{cfg}^{out} of an actor. In Figure 7.6, N is a configurable parameter.

Locally Static Parameters

A locally static parameter $\pi_{stat} \in \Pi$ of a graph has a value that is set before the beginning of the graph execution. Contrary to configurable parameters whose values change at each graph iteration, the value of a locally static parameter remains constant over one or several iterations of this graph. In addition to the parameters properties listed in Section 7.3.1, a locally static parameter belonging to a subgraph can also be used to influence the production and consumption rates on the I_{data}^{in} and I_{data}^{out} interfaces of its hierarchical actor.

The value of a locally static parameter can be statically set at compile time, or it can be dynamically set by configurable parameters of upper levels of hierarchy via parameter dependencies. From a subgraph perspective, a configuration input interface is equivalent to a locally static parameter. A configuration input interface can take different values at runtime if its corresponding configuration input port is connected to a configurable parameter. In Figure 7.6, *size* is seen as a locally static parameter both in main graph and in subgraph *Filter*.

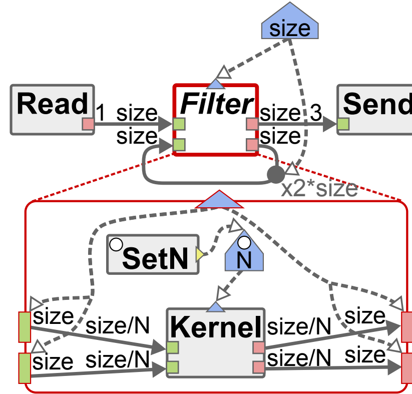


Figure 7.6: *Image processing example: πSDF graph*

A **partial configuration state** of a graph is reached when the parameter values of all its locally static parameters are set. **Hierarchy traversal** is the execution instant when a hierarchical actor is replaced by its subgraph in order to be executed. Hierarchy traversal of a hierarchical actor is possible only when the corresponding subgraph has reached a partial configuration state. The partial configuration state of a πSDF graph is equivalent to the state reached by a *parameterized dataflow* actor on completion of its init subgraph ϕ_i. In both cases the production/consumption rates on the interfaces of the hierarchical actor are fixed when this state is reached.

A **complete configuration state** of a graph is reached when the values of all its parameters (locally static and configurable) are set. If a graph does not contain any configurable parameter, its partial and complete configurations are equivalent. Only when a graph is completely configured is it possible to check its consistency, compute a schedule, and execute it. The complete configuration state of a πSDF graph is equivalent to the state reached by a *parameterized dataflow* actor on completion of its subinit subgraph ϕ_s.

Configuration Actors

A firing of an actor a with a configuration output port p_{cfg}^{out} produces a parameter value. Using a parameter dependency δ, a parameter value produced on a configuration output port can be used to dynamically set the value of a configurable parameter π, triggering

a reconfiguration of the graph elements depending on π. In PiMM, actors triggering reconfigurations of their graph are called *configuration actors*.

Since the execution of a *configuration actor* is the cause of a reconfiguration, it may happen only at quiescent points during the graph execution, as explained in [NL04]. To ensure the correct behavior of πSDF graphs, a *configuration actor* $a_{cfg} \in A$ of a subgraph G is subject to the following restrictions:

R1. a_{cfg} must be fired exactly once per iteration of G. This unique firing must happen before the firing of any non-configuration actor.

R2. a_{cfg} must consume data tokens only from hierarchical interfaces of G and must consume all available tokens during its unique firing.

R3. The production/consumption rates of a_{cfg} can only depend on locally static parameters of G.

R4. Data output ports of a_{cfg} are seen as a data input interface by other actors of G. (i.e. tokens produced on these ports are made available using a ring-buffer and can be consumed more than once).

These restrictions naturally enforce the local synchrony conditions of *parameterized dataflow* defined in [BB01] and reminded in Section 7.4.1.

The firing of all configuration actors of a graph is needed to obtain a complete configuration of this graph. Consequently, configuration actors must be executed before all other (non-configuration) actors of the graph to which they belong. Configuration actors are the only actors whose firing is not data-driven but driven by hierarchy traversal.

The sets of configuration and non-configuration actors of a graph are respectively equivalent to the subinit ϕ_s and the body ϕ_b subgraphs of *parameterized dataflow* [BB01]. Nevertheless, configuration actors provide more flexibility than subinit graphs as they can produce data tokens that will be consumed by non-configuration actors of their graph. The init subgraph ϕ_i has no equivalent in PiMM as its responsibility, namely the configuration of the production/consumption rates on the actor interfaces, is performed by configuration input interfaces and parameter dependencies.

Figure 7.6 presents an example of a πSDF description with reconfiguration. It is a modified version of the example in Figure 7.5. In Figure 7.6, parameter N is a configurable parameter of subgraph *Filter*, while parameter *Size* remains a locally static parameter. The number of firings of actor *Kernel* for each firing of actor *Filter* is dynamically configured by the configuration actor *setN*. In this example, the dynamic reconfiguration dynamically adapts the number N of firings of *Kernel* to the number of cores available to perform the computation of *Filter*. Indeed, since *Kernel* has no self-loop FIFO, the N firings of *Kernel* can be executed concurrently due to data parallelism.

7.4 Model Analysis and Behavior

The πSDF MoC presented in Section 7.3 is dedicated to the specification of applications with both dynamic and static parameterizations. This dual degree of dynamism implies a two-step analysis of the behavior of applications described in πSDF: a compile time analysis and a runtime analysis. In each step a set of properties of the application can be checked, such as the consistency, the deadlock freeness, and the boundedness. Other operations can be performed during one or both steps of the analysis such as the computation of a schedule or the application of graph transformation to enhance the performance of the application.

7.4.1 Compile Time Schedulability Analysis

πSDF inherits its schedulability properties both from the interface-based dataflow modeling and the *parameterized dataflow* modeling.

In interface-based dataflow modeling, as proved in [PBR09], a (sub)graph is schedulable if its precedence SDF graph (A, F) (excluding interfaces) is consistent and deadlock-free. When a πSDF graph reaches a complete configuration, all its production and consumption rates are decided and the graph becomes equivalent to an IBSDF graph. Hence, given a complete configuration, the schedulability of a πSDF graph can be checked using the same conditions as in interface-based dataflow.

In *parameterized dataflow*, the schedulability of a graph can be guaranteed at compile time for certain applications by checking their *local synchrony* [BB01]. A PSDF (sub)graph is locally synchronous if it is schedulable for all reachable configurations and if all its hierarchical children are locally synchronous. As presented in Section 2.5.1, a PSDF hierarchical actor composed of three subgraphs ϕ_i, ϕ_s and ϕ_b must satisfy the 5 following conditions in order to be locally synchronous:

1. ϕ_i, ϕ_s and ϕ_b must be locally synchronous, i.e. they must be schedulable for all reachable configurations.
2. Each invocation of ϕ_i must give a unique value to parameters set by this subgraph.
3. Each invocation of ϕ_s must give a unique value to parameters set by this subgraph.
4. Consumption rates of ϕ_s on interfaces of the hierarchical actor cannot depend on parameters set by ϕ_s.
5. Production/consumption rates of ϕ_b on interfaces of the hierarchical actor cannot depend on parameters set by ϕ_s.

The last four of these conditions are naturally enforced by the πSDF semantics presented in Section 7.3. However, the schedulability condition number 1, which states that all subgraphs must be schedulable for all reachable configurations, cannot always be checked at compile time. Indeed, since values of the parameters are chosen in $\mathbb{N}$, an infinite number of configurations can theoretically be reached. Consequently, only a formal verification can be used to check if the schedulability condition is satisfied. In graphs where all parameters take their values in a restricted and finite subset of $\mathbb{N}$, only a finite set of configurations are possible and the schedulability condition may be checked. In most cases, it is the responsibility of the developer to make sure that an application will always satisfy the schedulability condition; this responsibility is similar to that of writing non-infinite loops in imperative languages.

πSDF inherits from PSDF the possibility to derive quasi-static schedules at compile time for some applications. A quasi-static schedule is a schedule that statically defines part of the scheduling decisions but also contains parameterized parts that will be resolved at runtime. An example of quasi-static schedule is given for the application case in Section 7.5.2.

7.4.2 Runtime Operational Semantics

Based on the πSDF semantics presented in Section 7.3, the execution of a subgraph $G = (A, F, I, \Pi, \Delta)$ associated to a hierarchical actor a_G follows the following steps. The execution of G restarts from step 1 each time the parent graph of a_G begins a new iteration.

1. Wait for a partial configuration of G, i.e. wait for all configuration input interfaces in I_{cfg}^{in} to receive a parameter value.

2. Compute the production and consumption rates on the data interfaces I_{data}^{in} and I_{data}^{out} using the partial configuration.

3. Wait until a_G is fired by its parent graph. (enough data tokens must be available on the FIFOs connected to the data input ports P_{data}^{in} of a_G.)

4. Fire the *configuration actors* of A that will set the configurable parameters of G: a complete configuration is reached.

5. Check the local synchrony of G with the rates and delays resulting from the complete configuration and compute a schedule (if possible).

6. Fire the non-configuration actors of A following the computed schedule: this corresponds to an iteration of G.

7. Produce on the output ports P_{data}^{out} and P_{cfg}^{out} of a_G the data tokens and parameter values written by the actors of G on the output interfaces I_{data}^{out} and I_{cfg}^{out}.

8. Go back to step 3 to start a new iteration of G, i.e. a new firing of a_G.

Steps 1 and 2 correspond to a configuration phase of G. These first two steps are not clocked by data but are the result of the virtually instantaneous propagation of parameter values in the PDAG T. Steps 3 to 8 correspond to a firing of the enclosing actor a_G. These last six steps are executed each time actor a_G is scheduled during the execution of its parent graph.

If the schedulability of the graph can not be verified at compile time (Section 7.4.1), it will be checked at runtime in the 5th execution step. If a non-locally synchronous behavior is detected, i.e. if the graph is not consistent or has deadlocks, the execution of the application is terminated.

The runtime verification of the schedulability in step 5 can be used as a debug feature that can be deactivated to improve the performance of the application, thus assuming that a valid schedule can always be found in this step.

As introduced in Sections 7.3 and 7.3.2, the operational semantics of the πSDF MoC is equivalent to the one of the PSDF presented in [BB01]. Steps 1 and 2 are equivalent to the execution of the init subgraph ϕ_i, steps 3 to 5 are equivalent to the execution of the subinit subgraph ϕ_s, and steps 6 to 8 are equivalent to the execution of the body subgraph ϕ_b.

7.5 PiMM Application Examples

This section presents two examples of real applications that can be modeled with parameterized and interfaced dataflow models. These two applications, an FIR filter and a part of the LTE telecommunication standard, demonstrate how PiMM fosters the conciseness, the parallelism, and the composability of application description.

7.5.1 FIR Filter

The Finite Impulse Response (FIR) filter is an essential basic block of Digital Signal Processing (DSP) applications. Applying an FIR filter consists of computing the convolution of the filter coefficients with an input digital signal. Formally,

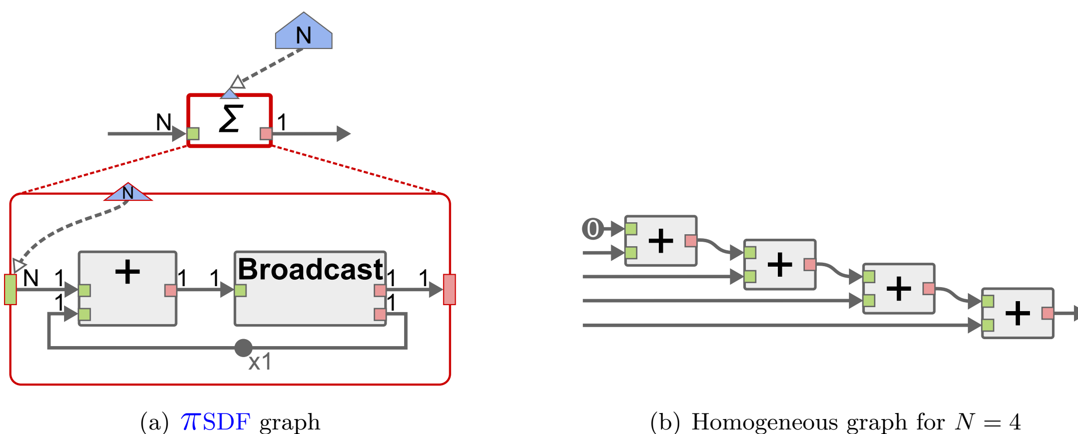
$$y[n] = \sum_{k=0}^{N-1} c_k \cdot x[n-k]$$

Where x and y are the input and output signals respectively, and c_0 to c_{N-1} are the filter coefficients.

The frequency response of an FIR filter depends on two parameters: the number and the value of the filter coefficients. In some DSP applications, such as adaptive audio filtering [GC04], the number and the value of these coefficients need to be changed dynamically. To fulfill these requirements, a flexible description of an FIR filter is proposed in this section. This description allows the developer of an application to set both statically and dynamically the number and the values of the coefficient while offering the highest possible degree of parallelism.

Adder PiMM Models

The computation of an FIR filter can be decomposed in two operations: the multiplication of the input samples with the filter coefficients, and the summation of the multiplication results. A πSDF description of the Σ adder subgraph responsible for the N additions of the FIR filter is presented in Figure 7.7.



(a) πSDF graph

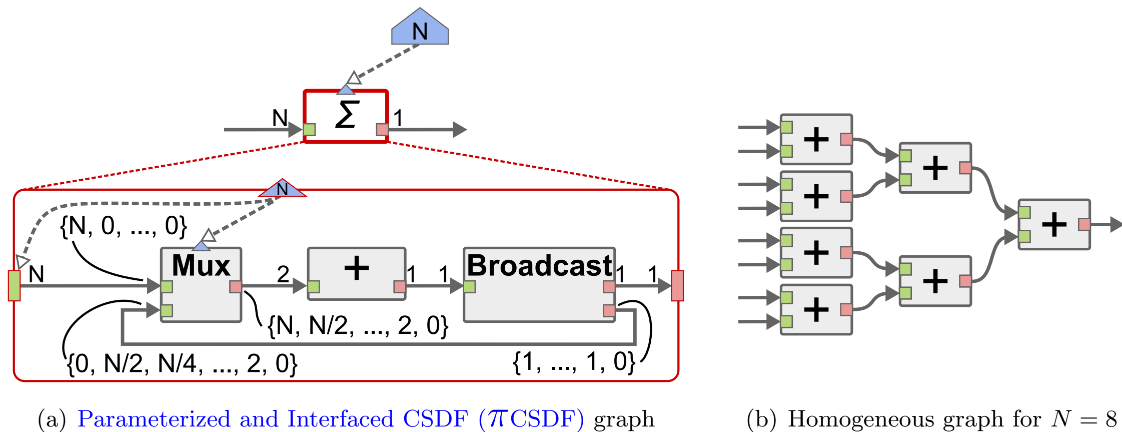
(b) Homogeneous graph for $N = 4$

Figure 7.7: *πSDF description of a configurable adder*

The πSDF description of the adder (Figure 7.7(a)) is composed of two actors: the $+$ actor, that computes the sum of the tokens consumed on its two input ports, and the *Broadcast* actor that duplicates data tokens from its input port to its output ports. Configuration input port N is used to parameterize the number of additions realized by the adder. In this πSDF graph, each data token represents a sample. To compute the addition between the N data tokens, this πSDF description recursively sums each data token with the intermediate result obtained by summing previous data tokens. The feedback loop, which is responsible for feeding the intermediate result to the $+$ actor, initially contains a single data tokens whose value is 0.

Figure 7.7(b) shows the homogeneous graph obtained by unrolling the adder πSDF graph for $N = 4$. For the sake of readability, only + actors are represented in this homogeneous graph. As revealed by this homogeneous graph, the drawback of this πSDF description of the adder is that all actor firings must be executed sequentially one after the other. Hence, this πSDF description should be used only in cases where parallelism is not needed.

A πCSDF description of the Σ adder subgraph is given in Figure 7.8. Parameterized and Interfaced CSDF (πCSDF) is the MoC obtained by applying the PiMM meta-model to the CSDF MoC. CSDF is a generalization of the SDF MoC where the scalar production/consumption rates of actors are replaced with cyclically changing rates [BELP96].



(a) Parameterized and Interfaced CSDF (πCSDF) graph (b) Homogeneous graph for $N = 8$

Figure 7.8: *πCSDF description of a configurable adder*

The πCSDF graph of the adder (Figure 7.8(a)) is composed of 3 actors: the two actors used in the πSDF graph of Figure 7.7(a) and a *Mux* actor that is responsible for selecting the data tokens fed to the + actor. The lists of consumption and production rates of the *Mux* actor each contain $log_2(N)+1$ values. The first firing of the *Mux* actor forwards the N data tokens from the data input interface of the subgraph to the + actor. The following firings of actor *Mux* forward the intermediate result from the *Broadcast* actor to the + actor.

Figure 7.8(b) shows the homogeneous graph obtained by unrolling the adder πCSDF graph for $N = 8$. For the sake of readability, only + actors are represented in this homogeneous graph. Contrary to the πSDF description, the πCSDF description exposes the parallelism of the addition process. Indeed, as shown in Figure 7.8(b), the πCSDF MoC enables the specification of a cascade summation [Hig93]. In the first stage of the cascade, $N/2$ additions are computed between pairs of successive data tokens. The results from the first stage are then used as inputs for the $N/4$ additions of a second stage. The process is repeated until a single value is produced. All additions executed in a stage of this cascade can be executed in parallel.

The πCSDF model presented in Figure 7.8 is only valid if N is a power of 2. A first solution to support any value of N is to artificially insert new data tokens with value equals to 0. The drawback of this solution is that it unnecessarily increases the number of additions. A better solution is to use the production and consumption pattern of the *Mux* actor to forward at each firing the greatest possible number of pair of data tokens, combining tokens from the data input interface and tokens from the *Broadcast* actor. Formally, considering u_i a recursive sequence such that $u_i = u_{i-1} - \lfloor u_{i-1}/2 \rfloor$ and $u_0 = N$, then the production pattern of the *Mux* actor is $\{2 \cdot \lfloor u_i/2 \rfloor |$ for $i = 0$ to $\lfloor log_2(N) \rfloor\}$.

For example, for $N = 7$, the *Mux* actor successively feeds 6, 4 and 2 data tokens to the $+$ actor. Besides revealing the parallelism of the addition process, cascade summation also improves the accuracy of the computed result [Hig93].

FIR filter πSDF model

A generic πSDF description of an FIR filter is presented in Figure 7.9. The *FIR* actor processes a single sample at each firing. The *FIR* actor possesses 5 data ports: *coefs* that receives the coefficients of the filter, *sample* that receives the sample to process, *d.in* that receives the delayed samples, *out* that produces the result sample, and *d.out* that produces the delayed samples for the next firing of the *FIR* actor. The *FIR* hierarchical actor possesses a unique configuration input port *nbCoef* that represents the number of coefficients of the filter. The self-loop between ports *d.out* and *d.in* is used to transmit the delayed data tokens from a firing of the FIR actor to the next.

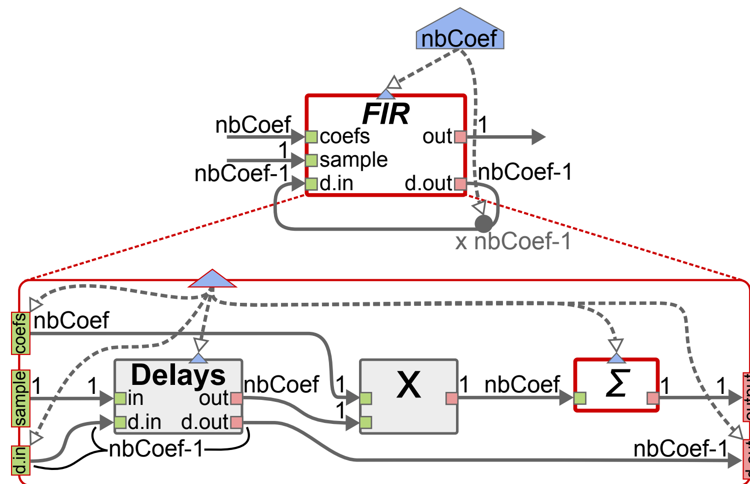


Figure 7.9: *πSDF model of an FIR filter*

The subgraph of the *FIR* actor contains 3 actors: the Σ hierarchical actor described in previous paragraph, actor X which performs multiplications, and actor *Delays* that is responsible for feeding the last *nbCoef* samples to the X actor and forwarding the last $nbCoef - 1$ samples to the $d.out$ output interface.

The $nbCoef$ firings of the X actor can be executed in parallel for each sample processed by the FIR filter. Hence, if the πCSDF description of the adder (Figure 7.7(a)) is used, most of the computation can be performed in parallel. This example shows that compact πSDF model can be used to describe configurable applications without sacrificing their parallelism. Through the example of the Σ actor, the FIR example also illustrates how the hierarchy mechanism of PiMM can be used to divide applications into smaller, generic, and reusable subgraphs. Next example focuses on the dynamic configuration semantics offered by PiMM.

7.5.2 LTE PUSCH

The LTE Physical Uplink Shared Channel (PUSCH) decoding is executed in the physical layer of an LTE base station (eNodeB). It consists of receiving multiplexed data from several User Equipments (UEs), decoding it and transmitting it to upper layers of the LTE standard.

πSDF Application Model

Figure 7.10 presents a πSDF specification of the bit processing algorithm of the PUSCH decoding which is part of the LTE telecommunication standard.

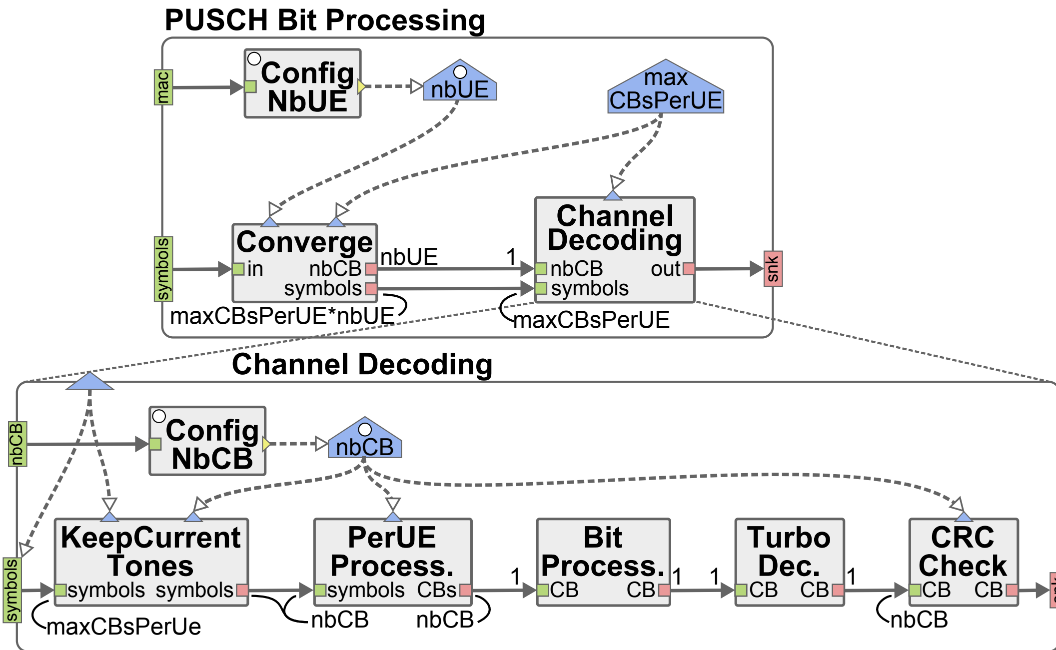


Figure 7.10: *πSDF model of the bit processing part of the LTE PUSCH decoding.*

As presented in Section 7.1.1, because the number of UEs connected to an eNodeB and the rate for each UE can change every millisecond, the bit processing of PUSCH decoding is inherently dynamic and cannot be modeled with static MoCs such as SDF [PAPN12].

The bit processing specification is composed of two hierarchical actors: the *PUSCH Bit Processing* actor and the *Channel Decoding* actor. For clarity, Figure 7.10 shows a simplified specification of the LTE PUSCH decoding process where some actors and parameters are not depicted.

The *PUSCH Bit Processing* actor is executed once per invocation of the PUSCH decoding process and has a static parameter, *maxCBsPerUE*, that represents the maximum number of data blocks (named Code Block (CB)) per UE. *maxCBsPerUE* statically sets the configuration input interface of the lower level of the hierarchy, according to the eNodeB limitation of bitrate for a single UE. The *ConfigNbUE* configuration actor consumes data tokens coming from the Medium Access Control (MAC) layer and sets the configurable parameter *NbUE* with the number of UEs whose data must be decoded. The *converge* actor consumes the multiplexed CBs received from several antennas on the *symbols* data input interface of the graph, produces *NbUE* tokens, each containing the number of CBs for one UE, and produces *NbUE* packets of *maxCBsPerUE* CBs, each containing the CBs of an UE.

The *Channel Decoding* hierarchical actor fires *NbUE* times, once for each UE, because each UE has specific channel conditions. This actor has a configuration input interface *maxCBsPerUE* that receives the eponymous locally static parameter from the upper hierarchy level. The *ConfigNbCB* configuration actor sets the *NbCB* parameter with the number of CBs allocated for the current UE. A detailed explanation of the role of the remaining actors is beyond the scope of this thesis and can be found in [PAPN12].

The LTE PUSCH bit processing algorithm is a good illustration of the conciseness of the πSDF model (Figure 7.10) compared to the PSDF model (Figure 7.11). Indeed, only 2 πSDF graphs are needed to specify the application whereas 2 sets of 3 subgraphs (ϕ_i, ϕ_s and ϕ_b) are needed to specify it with the PSDF MoC.

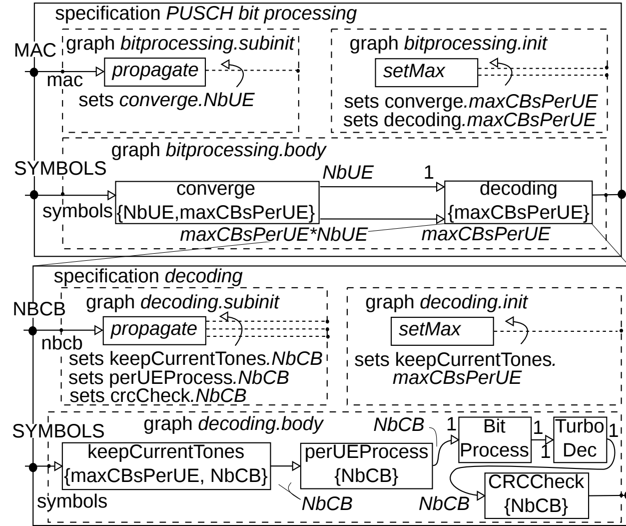


Figure 7.11: *PSDF model of the bit processing part of the LTE PUSCH decoding.*

Quasi-Static Schedule

The use of quasi-static schedules is highly desirable in many contexts compared to dynamic schedules. In particular, quasi-static schedules for *parameterized dataflow* graphs provide significant reductions in runtime overhead, and improvements in predictability (e.g. see [BB01, Bou09]). By facilitating systematic construction of parameterized schedules — either manually as part of the design process or automatically as part of a graph transformation flow — the PiMM framework enhances the efficiency and confidence with which dynamically structured signal processing systems can be implemented.

The dynamic topology of a πSDF graph usually prevents the computation of a static schedule since the production/consumption rates are unknown at compile time. In the example of Figure 7.10, despite the dynamic rates, the production rate on all FIFOs always is a multiple of the consumption rate, or vice versa. Consequently, the dynamic RV is an affine function of the graph parameters and a quasi-static schedule can be computed. Based on an affine formulation, the following quasi-static schedule (Figure 7.12) can be derived for the graph of Figure 7.10.

7.6 Comparison with Existing MoCs

Table 7.1 presents a comparison of the properties of the πSDF MoC with the properties of the dataflow MoCs presented in Chapter 2. The compared MoCs are the SDF [LM87b], the ADF [BTV12], the IBSDF [PBR09], the PSDF [BB01], the SADF [TGB+06], the DPN [LP95], the DSSF [TBG+13], the SPDF [FGP12], and the πSDF. In Table 7.1, a black dot indicates that the feature is implemented by a MoC, an absence of dot means that the feature is not implemented, and an empty dot indicates that the feature may be available for some applications described with this MoC. It is important to note that the full

```
while true do
    /* Execute Top PUSCH */
    fire ConfigNbUE ;              // Sets nbUE
    fire Converge;
    repeat nbUE times
        /* Execute Channel Decoding */
        fire ConfigNbCB ;          // Sets nbCB
        fire KeepCurrentTones;
        fire PerUEProcess;
        repeat nbCB times
            fire BitProcess;
            fire TurboDec;
        end
        fire CrcCheck;
    end
end
```

Figure 7.12: *Quasi-static schedule for graph in Figure 7.10*

semantics of the compared MoCs is considered here. Indeed, some features can be obtained by using only a restricted semantics of the compared MoCs. For example, all MoCs can be restricted to describe a SDF graph, thus benefiting from the static schedulability and the decidability but losing all reconfigurability.

Feature	SDF	ADF	IBSDF	DSSF	PSDF	πSDF	SADF	SPDF	DPN
Hierarchical			•	•	•	•			
Compositional			•	•		•			
Reconfigurable					•	•	•	•	•
Configuration dependency						•	•		
Statically schedulable	•	•	•	•					
Decidability	•	•	•	•	○	○	•	○	
Variable rates		•			•	•	•	•	•
Non-determinism							•	•	•

Table 7.1: *Features comparison of different dataflow MoCs*

The following features are compared in Table 7.1:

- *Hierarchical*: composability can be achieved by associating a subgraph to an actor.
- *Compositional*: graph properties are independent from the internal specifications of the subgraphs that compose it [Ost95].
- *Reconfigurable*: actors firing rules can be reconfigured dynamically.
- *Configuration dependency*: the MoC semantics includes an element dedicated to the transmission of configuration parameters.
- *Statically schedulable*: a fully static schedule can be derived at compile time [LM87b].

- *Decidability*: the schedulability is provable at compile time.
- *Variable rates*: production/consumption rates are not a fixed scalar.
- *Non-determinism*: output of an algorithm does not solely depend on inputs, but also on external factors (e.g. time, randomness).

7.6.1 PiMM versus Parameterized Dataflow

PiMM and the *parameterized dataflow* meta-model are equivalent in many points. However, PiMM introduces new elements of semantics, such as the PDAG, that enhance the predictability of the model and increase the performance of applications described with the new meta-model.

Faster parameter propagation

In PiMM, the explicit parameter dependencies enable the instant propagation of parameter values to lower levels of hierarchy through configuration input ports P_{cfg}^{in} and corresponding configuration input interfaces I_{cfg}^{in}. With this instant propagation, setting a parameter in a hierarchical graph may instantly influence parameters deep in the hierarchy, causing some subgraphs to reach a partial or a complete configuration.

The instant parameter propagation of PiMM is in contrast with the more complex configuration mechanism of the *parameterized dataflow*. The body subgraph ϕ_b of an actor a_G can be configured only by parameters set by the init subgraph ϕ_i or the subinit subgraph ϕ_s, but cannot directly depend on parameters defined in the parent graph of a_G [BB01]. This semantics implies that a complete configuration of ϕ_b cannot be reached before the execution of ϕ_i, even if actors in ϕ_i simply propagate parameters from upper levels of hierarchy. Consequently, a complete configuration of a subgraph may be reached earlier for an application modeled with PiMM, providing valuable information to the runtime management system and leading to better scheduling or resource allocation choices, and therefore better performance.

Lighter runtime overhead

In *parameterized dataflow*, the production and consumption rates on the data interfaces of a hierarchical actor are obtained by computing the Repetition Vector (RV) [LM87b] of its subgraph. For dynamically scheduled applications, two computations of the RV are performed at runtime. The first computation is done using a partial configuration completed with default values for undefined parameters. The second computation is done when a complete configuration is reached. The default parameter values used in the first computation must be carefully chosen to ensure that the two RVs present the same production/consumption rates on the interfaces of the actor, or otherwise the application will be terminated.

In PiMM, the production and consumption rates on the interfaces of a hierarchical actor only depend on the value of locally static parameters. Since neither the first computation of a RV nor the use of default parameter values are needed, the runtime overhead is lighter and the development of the application simpler as the developer does not need to specify default values for configurable parameters.

Improved User-friendliness

In *parameterized dataflow*, the specification of a parameterized hierarchical actor is composed of three subgraphs, which may lead to a rapid increase in the number of graphs to maintain when developing an application. For example, the development of an application with only a dozen parameterized hierarchical actors requires the specification of almost forty graphs.

In PiMM, a single subgraph is needed to specify the behavior of all hierarchical actors, parameterized or not. The employment of a single subgraph is enabled by the introduced parameters and parameter dependencies that replace the init subgraph ϕ_i and by the configuration actors that replace the subinit subgraphs ϕ_s. Using explicit parameter dependencies also makes it possible to lower the number of actors of a graph, by eliminating the actors whose only purpose was to propagate parameters from the upper levels of the hierarchy. Moreover, using parameter dependencies instead of referencing parameters by their names makes it easier to identify what is influenced by a parameter. All these features enhance the readability of πSDF graphs, and hence make the model more user-friendly.

7.6.2 PiMM versus SADF

In [TGB+06], Theleen et al. introduce the Scenario-Aware Dataflow (SADF): a generalization of the SDF model where dynamism is handled by special actors, called detectors. Detector actors can reconfigure other actors of their graph by sending them control tokens sequentially through specific FIFOs called control channels. When consumed by an actor, these control tokens change the scenario in which the actor is running, possibly modifying the nature of its computation, its run time, and its production and consumption rates.

A first difference between SADF and πSDF is that in SADF, each actor has a unique status that denotes the current scenario of the actor. Because of this status, an actor cannot be fired multiple times in parallel. In πSDF as in SDF, actors have no state unless explicitly specified with a self-loop FIFO [LM87b]. Consequently, the parallelism embedded in a πSDF description is implicitly greater than the one of an SADF graph.

A second difference between SADF and πSDF lies in the motivations behind the two models. SADF is an analysis-oriented model that has proved to be an efficient model to quickly derive useful metrics such as the worst-case throughput or the long-run average performance [SGTB11]. To provide such metrics, SADF relies on a timed description of the actors behavior which corresponds to the execution time of the actor on a specific type of processing elements. Conversely, like PSDF, πSDF is an implementation-oriented, untimed, and architecture-independent model which favors the development of portable applications for heterogeneous MPSoCs. Moreover, it was shown in [SGTB11] that implementation of applications described in SADF are less efficient than PSDF applications. Finally, the hierarchical compositionality mechanism of πSDF has no equivalent in SADF.

Next chapter concludes this thesis and proposes possible axes for future work.

CHAPTER 8

Conclusion

8.1 Summary

In recent years, it has become critical to find new development techniques to manage the increasing complexity of embedded systems, both in terms of hardware and software. In particular, new programming models and languages must be found to **exploit the parallel processing capabilities** of modern MPSoCs. For this purpose, **dataflow MoCs** have emerged as a popular programming paradigm to capture intuitively the parallelism of applications.

The contributions presented in this thesis address key memory challenges encountered throughout the deployment of an application modeled with an IBSDF graph on an shared-memory MPSoC. These contributions were implemented as part of the PREESM software rapid prototyping framework.

In Chapter 4, new techniques were presented to analyze and optimize the memory characteristics of applications modeled with IBSDF graphs, from the estimation of the application memory footprint in early stages of development, to the static reduction of this memory footprint during the application implementation on a shared memory MPSoC. Experimental results showed a reduction of the static memory footprint by up to 43% compared to a state-of-the-art memory minimization technique.

In Chapter 5, a new set of annotations was introduced, allowing application developers to specify new memory reuse opportunities for the static allocation of input and output buffers of each dataflow actor. The minimization technique taking advantage of these reuse opportunities to reduce the memory footprint allocated to applications was also presented in this chapter.

In Chapter 6, an in-depth study of the memory characteristics of a state-of-the-art computer vision application was presented. Besides showing the efficiency of the proposed static memory optimization techniques compared to dynamic allocation, this study also presented technical solutions for supporting cache-incoherent multiprocessor architectures.

In Chapter 7, a new dataflow meta-model called PiMM was introduced to overcome the limitations of the IBSDF MoC. PiMM can be applied to a dataflow MoC to increase its expressivity, to enable the specification of reconfigurable applications, and to promote derivation of quasi-static schedules. While bringing dynamism and compositionality, the explicit parameter dependency tree and the interface-based hierarchy mechanism intro-

duced by PiMM maintain strong predictability for the extended model and enforce the conciseness and readability of application descriptions.

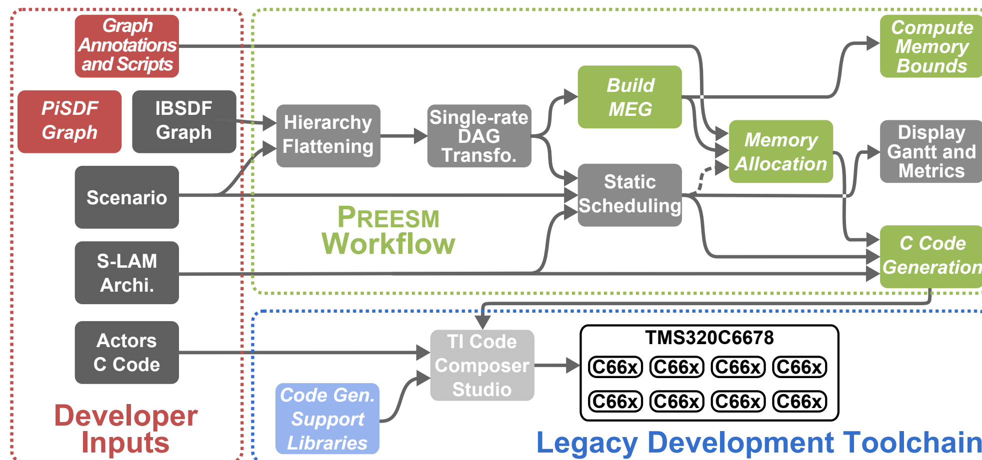


Figure 8.1: *PREESM typical workflow including new contributions from this thesis.*

Figure 8.1 summarizes these new contributions and presents their role in the typical workflow of PREESM. New contributions and improvements to the rapid prototyping framework during this thesis are depicted by colored boxes with italic text.

It is important to note that all contributions presented in this thesis were developed as part of the PREESM open-source project, and can therefore be freely and legally accessed online for any purposes [IET14b].

8.2 Future Work

The work presented in this thesis opens many opportunities for future research on dataflow MoCs and rapid prototyping.

8.2.1 Support for Complex Memory Architectures

The memory analysis and allocation techniques presented in this thesis are applicable only for MPSoCs where all processing elements can access a unique shared-memory. Although shared-memory MPSoCs are still the most commonly used architectures for embedded systems, more and more companies propose products that break with this memory organization. For example, many-core processors from Tilera [Til14], Kalray [Kal14], and Adapteva [Ada14] are all based on a distributed memory architecture where each processor, or each group of processors, is associated with a local memory.

Hence, a potential direction for future research would be to extend the memory study and allocation techniques presented in this thesis to support distributed memory architectures. This work could also be further extended to encompass memory allocation for complex memory architectures with a hierarchical organization of memory banks with variable speeds.

8.2.2 Memory-Aware Scheduling Techniques

The current objective of the mapping and scheduling process implemented in PREESM is to optimize the throughput, the latency, and the load balancing of applications on heterogeneous MPSoCs. As presented in Chapter 4, this scheduling policy sometimes

results in bad choices from the memory perspective. Indeed, actors consuming and freeing the memory allocated to large buffers are not prioritized by the scheduling process, even when the throughput is not impacted.

An interesting perspective for future work would be to introduce an evaluation of the memory footprint in the mapping and scheduling process. As in FIFO dimensioning techniques [SGB06], a memory-aware scheduling process would allow developers to customize the tradeoff between memory footprint and throughput of applications.

8.2.3 Rapid Prototyping for πSDF graphs

In the current version of PREESM, a graphical editor has been implemented for πSDF graphs. A converter of static πSDF graphs into IBSDF graphs was also implemented to enable use of all previously developed workflow tasks on static πSDF graph. Finally, a runtime manager for applications modeled with πSDF graphs has been developed for the *TMS320C6678* multiprocessor DSP chip from Texas Instrument [Tex13] and is being ported on the *MPPA256* many-core chip from Kalray [Kal14].

Possibilities for future work on the PiMM meta-model include the development of analysis techniques to guarantee the local synchrony of πSDF graphs at compile time, the creation of an algorithm to automate the static scheduling of applications on multi-core architecture, or the extension of the memory optimization techniques presented in this thesis for reconfigurable πSDF graphs.

ANNEXE A

French Summary

A.1 Introduction

Les systèmes embarqués constituent l'une des avancées technologiques les plus remarquables des dix dernières années. Ces systèmes sont devenus un vecteur de développement majeur pour de nombreuses industries, et ont progressivement envahi de nombreux aspects de la vie quotidienne. Un système embarqué est un système électronique et informatique intégré, conçu pour réaliser une tâche précise. Les systèmes d'aide au freinage d'urgence, les terminaux de géolocalisation par satellite, les liseuses, les stimulateurs cardiaques, les appareils photo numériques (APN), et les aspirateurs autonomes sont autant d'exemples d'appareils fonctionnant grâce à un ou plusieurs systèmes embarqués.

Matériel et logiciel d'un système embarqué

Comme la plupart des systèmes informatiques, les systèmes embarqués sont composés de deux parties complémentaires : la partie matérielle et la partie logicielle.

- **Le matériel** est l'ensemble des composants physiques qui constituent un système embarqué, comme des unités de traitement, des générateurs d'horloges, des capteurs, des convertisseurs analogique-numérique, des mémoires, des moyens de communication interne et externe, ou des unités de gestion de l'alimentation. L'architecture d'un système embarqué spécifie le type des composants matériels contenus dans ce système ainsi que les connexions entre ces composants. Les unités de traitement, processeurs ou accélérateurs matériels, sont les composants matériels les plus importants d'un système embarqué. En effet, ces unités réalisent les calculs nécessaires au fonctionnement d'un système embarqué et sont responsables du contrôle des autres éléments matériels de ce système. Selon le domaine d'application visé par un système embarqué, différents types d'unités de traitement peuvent être utilisées. Par exemple, des processeurs spécialisés offrant une forte puissance de calcul pour un coût limité sont souvent utilisés pour les systèmes nécessitant des calculs intensifs, comme les systèmes de traitement du signal et de l'image.

  De nos jours, les systèmes embarqués sont souvent basés sur des circuits intégrés complexes appelés “systèmes multiprocesseur sur puce” (MPSoC). Un MPSoC hétérogène

regroupe tous les composants matériels d'un système embarqué, dont plusieurs unités de traitement de différents types, sur un seul circuit intégré.

- **Le logiciel** désigne les programmes informatiques exécutés par les unités de traitement d'un système embarqué. Un programme est une séquence d'instructions exécutées de manière itérative par les unités de traitement d'un système embarqué. Chaque instruction correspond à une opération primitive, comme une opération arithmétique et logique, un saut dans le déroulement du programme, la configuration et le contrôle d'autres composants du système embarqué, la communication et la synchronisation avec d'autres unités de traitement, ou une opération de lecture ou d'écriture dans la mémoire du système.

  Programmer un système embarqué consiste à écrire une séquence d'instructions qui spécifie le comportement de ses unités de traitement. Les méthodes de programmation, dites de haut-niveau, permettent l'écriture du programme dans des langages facilement manipulables par les développeurs. Lors d'une étape de compilation, les programmes écrits dans ces langages de haut-niveau sont automatiquement convertis en instructions binaires exécutables par les unités de traitement.

Afin de réduire le temps de développement d'un système embarqué, ses parties matérielles et logicielles sont généralement développées conjointement.

En quelques années, la multiplication du nombre d'unités de traitement intégrées dans les MPSoCs a permis une augmentation rapide de leur puissance de calcul. Afin d'exploiter la puissance de calcul offerte par ces nouvelles architectures, les langages de programmation doivent permettre la spécification d'applications où les calculs peuvent être exécutés en parallèle. Actuellement, le langage C est le langage de programmation le plus couramment utilisé pour la programmation de systèmes embarqués. La syntaxe du langage C ne permettant d'exprimer qu'un faible degré de parallélisme, de nouveaux langages et modèles de calcul sont nécessaire pour exploiter pleinement la puissance de calcul des MPSoCs.

Programmation flux de données

Les modèles de calcul de type flux de données [Kah74] sont des paradigmes de programmation efficaces pour exprimer le parallélisme d'applications. Une application modélisée avec un diagramme de flux de données se présente sous la forme d'un graphe orienté dans lequel chaque sommet représente un module de calcul indépendant, appelé un acteur, et chaque arête représente un canal de communication entre deux sommets. La popularité des modèles de flux de données dans les milieux scientifiques, universitaires, et industriels est due à leur expressivité naturelle du parallélisme, à leur modularité, et à leur compatibilité avec le code existant. En effet, le comportement interne des acteurs d'un diagramme de flux de données peut être spécifié dans n'importe quel langage de programmation, y compris en langage C. Le temps de développement d'un système peut donc être considérablement réduit en réutilisant des programmes précédemment développés et optimisés.

Prototypage rapide

L'objectif des outils de prototypage rapide est d'accélérer et de faciliter le développement des systèmes embarqués. Le flot de conception classique pour systèmes embarqués est un processus simple dont le l'objectif final est de produire un système embarqué répondant à toutes les contraintes de conception. Le principe des techniques de prototypage rapide est de créer un prototype peu coûteux, aussi tôt que possible dans le processus de

développement. L'analyse des caractéristiques de ce prototype permet aux développeurs d'identifier les points critiques du système, et d'en améliorer les caractéristiques de manière itérative en générant de nouveaux prototypes.

Problématique de la thèse

Les composants de mémoire sont, après les unités de traitement, les composants les plus importants d'une architecture matérielle embarquée. En effet, les composants mémoires peuvent couvrir jusqu'à 80% de la surface de silicium d'un circuit intégré [DGCDM97]. Malgré ce fort coût en surface de silicium et la forte consommation énergétique qui en découle, la mémoire est une ressource encore rare du point de vue logiciel. Par exemple, dans le processeur *many-core* MPPA256 de Kalray [Kal14], chaque banc mémoire de 2 Mo est partagé par 16 unités de traitement, pour un total de 32 Mo de mémoire embarquée sur la puce. Un autre exemple est la TMS320C6678 de Texas Instrument dans lequel 8,5 Mo de mémoire sont partagés par les 8 processeurs de traitement du signal [Tex13]. Ces capacités mémoires restent relativement faibles par rapport aux besoins logiciels. Par exemple, pour une application de décodage vidéo, plus de 3 Mo sont nécessaires pour stocker une seule image en résolution Full HD de 1920×1080 pixels. Même si des mémoires externes peuvent être utilisées en complément de la mémoire intégrée dans un MPSoC, l'accès à ces bancs mémoires externes est plus lent, et leur utilisation a un impact négatif sur les performances d'un système embarqué. Par conséquent, l'étude et l'optimisation des aspects mémoires constituent une étape de développement essentiel qui peut fortement influer sur les performances et le coût d'un système embarqué.

Plan

Ce résumé reprend l'organisation des chapitres du corps de la thèse. La Section A.2 présente quelques modèles de flux de données utilisés dans ce résumé. La Section A.3 décrit l'environnement de prototypage rapide au sein duquel les travaux de cette thèse ont été développés. Ensuite, les Sections A.4 et A.5 présentent deux techniques pour optimiser la quantité de mémoire allouée pour l'exécution d'une application modélisée par un diagramme flux de données sur un système multiprocesseur sur puce. Ces méthodes d'optimisation mémoire sont appliquées à un algorithme d'appariement stéréoscopique en Section A.6. La Section A.7 introduit un nouveau modèle de flux de données permettant la modélisation d'applications reconfigurables. Enfin, la Section A.8 conclut ce résumé.

A.2 Modèles de calcul de type flux de données

Un modèle de calcul (MoC) est un ensemble d'éléments opérationnels pouvant être assemblés afin de décrire le comportement d'une application. La sémantique d'un MoC désigne l'ensemble des éléments opérationnels de ce MoC ainsi que l'ensemble des relations pouvant être utilisées pour assembler ces éléments opérationnels. Par exemple, la sémantique du MoC des automates finis est composé de deux éléments opérationnels : des états et des transitions. À un instant t, un unique état d'un automate fini est actif. Chaque transition d'un automate fini est associée à une condition. Le passage d'un état actif à un autre état ne peut se faire que si la transition reliant ces deux états est validée. Le MoC des automates finis est généralement utilisé pour la modélisation d'applications de contrôle séquentiel [Gol94].

Il existe de nombreux autres MoCs, tels que le lambda calculus [Chu32] ou les circuits logiques [Sav98], chacun adapté à la modélisation d'un certain type d'applications. Le reste

de cette section présente la sémantique des modèles de calculs de type flux de données qui sont utilisés dans ce résumé.

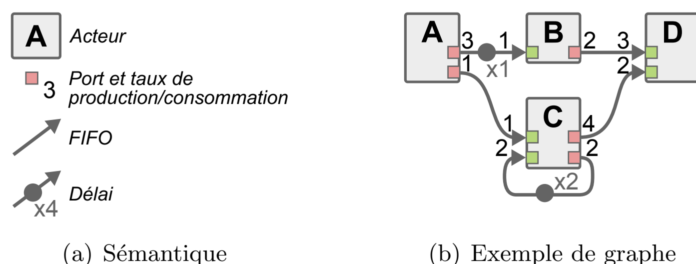
Modèle de flux de données synchrone (SDF)

Le modèle de flux de données synchrone (SDF) a été introduit par Lee et Messerschmitt en 1987 [LM87b]. Ce modèle est défini comme suit :

Definition A.2.1. *Un diagramme de flux de données synchrone (SDF) est un graphe orienté* $G = \langle A, F \rangle$ *tel que :*

- ***A*** *est l'ensemble des nœuds de* G*. Chaque nœud* $a \in A$ *représente un* ***acteur*** *: une entité de code séquentiel. Le comportement interne des acteurs ne fait pas partie du modèle de flux de données, il peut être décrit avec n'importe quel langage de programmation.*
- $\boldsymbol{F} \subseteq A \times A$ *est l'ensemble des arcs de* G*. Chaque arc* $f \in F$ *représente une file d'attente "premier arrivé, premier sorti" (FIFO) permettant la transmission de quanta de données, appelés* ***jetons de données****, entre les acteurs du graphe.*
- *Chaque acteur est associé à un ensemble de* ***règles de tirs*** *qui spécifie le nombre constant de jetons de données que cet acteur consomme et produit à chaque exécution sur chacune des FIFOs auxquelles il est connecté. Une nouvelle exécution d'un acteur peut débuter dès que suffisamment de jetons de données sont présent sur les FIFOs entrantes de cet acteur.*
- *Les* ***délais*** *(*$delay : F \to \mathbb{N}$*) sont des jetons de données contenus dans les FIFOs du graphe lors de son initialisation.*

Les pictogrammes associés à la sémantique du modèle SDF ainsi qu'un exemple de diagramme SDF sont présentés en Figure A.1.



(a) Sémantique (b) Exemple de graphe

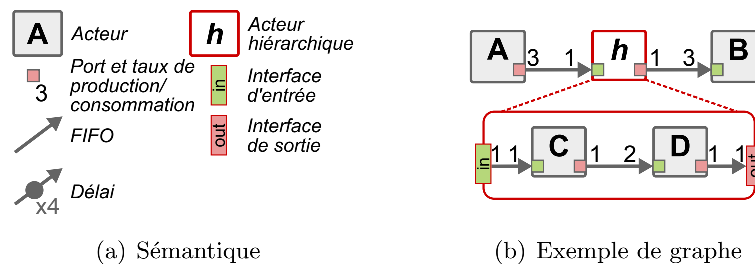
Figure A.1 – *Modèle de calcul SDF*

La popularité du modèle SDF est principalement due à sa capacité à exprimer le parallélisme des applications. Dans la Figure A.1(b) par exemple, les acteurs B et C peuvent être exécutés en parallèle puisqu'ils ne sont liés par aucune dépendance de données. Dans cette même figure, à chacune de ses exécutions, l'acteur A produit suffisamment de jetons de données pour déclencher 3 exécutions de l'acteur B. L'acteur B n'ayant pas de dépendance avec lui-même, contrairement à l'acteur C, il peut effectuer ces 3 exécutions en parallèle les unes des autres.

La popularité du modèle SDF est également due à sa grande analysabilité qui permet de vérifier certaines propriétés des applications modélisées lors d'une phase de compilation. Par exemple, il est possible de garantir qu'une application ne rencontrera jamais d'interblocage (ou étreinte fatale [HRM08]) [LM87b] lors de son exécution.

Modèle flux de données synchrone basé-interface (IBSDF)

Le modèle flux de données synchrone basé-interface (IBSDF) est une généralisation hiérarchique du modèle SDF proposée par Piat et al. dans [PBR09]. Dans le modèle IBSDF, le comportement interne d'un acteur $a \in A$ peut être décrit soit par du code séquentiel, soit par un diagramme de flux de données appelé sous-graphe de cet acteur. Les pictogrammes associés à la sémantique de l'IBSDF ainsi qu'un exemple de diagramme IBSDF sont présentés en Figure A.2.



(a) Sémantique (b) Exemple de graphe

FIGURE A.2 – *Modèle de calcul IBSDF*

Les interfaces hiérarchiques du modèle IBSDF ont pour rôle d'isoler les niveaux de hiérarchie les uns des autres. En particulier, si le nombre de jetons nécessaires pour l'exécution d'un sous-graphe est supérieur aux nombres de jetons fournis sur les ports de l'acteur hiérarchique, alors les interfaces d'entrée dupliqueront les jetons fournis autant de fois que nécessaire pour permettre l'exécution du sous-graphe. Par exemple, dans le diagramme IBSDF de la Figure A.2(b), deux exécutions de l'acteur C sont nécessaires pour fournir les jetons consommés par l'acteur D. Le port d'entrée de l'acteur h ne consommant qu'un seul jeton à la fois, ce jeton doit être dédoublé par l'interface d'entrée *in* pour permettre le lancement des deux exécutions de l'acteur C.

En pratique, les interfaces hiérarchiques font du modèle **IBSDF un modèle compositionnel**. Un modèle de flux de données est compositionnel si les propriétés d'un diagramme sont indépendantes des spécifications internes des éléments qui le composent [Ost95]. Par exemple, dans le diagramme de la Figure A.2(b), grâce aux interfaces, quel que soit le taux de consommation de l'acteur D, l'acteur hiérarchique h consommera toujours un unique jeton par exécution de son sous-graphe. Le graphe de plus haut niveau contenant les acteurs A, h, et B est donc bien indépendant de la spécification interne de l'acteur hiérarchique h.

A.3 Prototypage rapide avec PREESM

PREESM est un outil de prototypage rapide dont l'objectif est d'automatiser le déploiement d'applications modélisées par des diagrammes IBSDF sur des architectures multiprocesseurs hétérogènes [PDH+14]. PREESM est développé à l'Institut d'Électronique et de Télécommunications de Rennes (IETR) sous la forme de modules d'extension pour l'environnement de développement *Eclipse*. PREESM est un projet au code source ouvert, utilisé à des fins scientifiques, académiques, et industrielles.

Flux de travail typique de PREESM

La Figure A.3 présente les 3 parties principales de PREESM. Ces 3 parties se retrouvent dans la plupart des outils de prototypage rapide existants, tels que SynDEx [GLS99], SDF3 [Ele13], Ptolemy [BHLM94], ou MAPS [CCS+08].

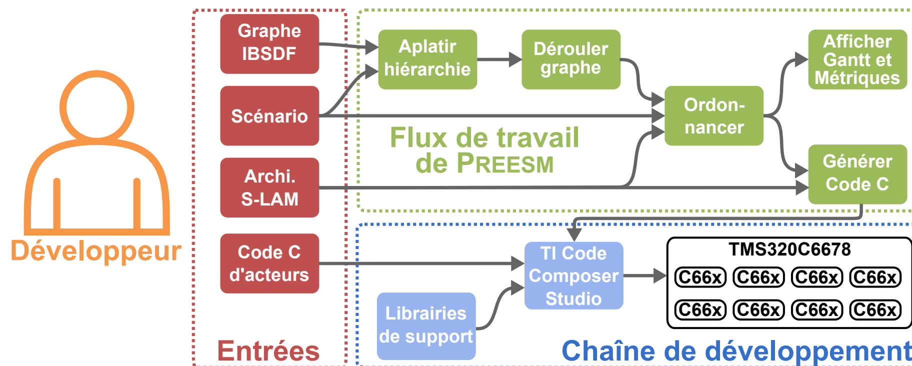


FIGURE A.3 – *Flux de travail typique de PREESM*

- **Entrées** : Cette partie de l'outil regroupe tous les modèles permettant au développeur de décrire les propriétés du système développé. Dans le contexte des systèmes embarqués, cette partie regroupe généralement un modèle dédié à la description des architectures matérielles visées, un modèle pour la description des applications à exécuter, et un modèle permettant de spécifier des contraintes de déploiement.

  Dans le cas de PREESM, le modèle d'architecture utilisé est un modèle de niveau système nommé S-LAM [PNP+09] et le modèle d'application est le modèle IBSDF. Dans PREESM, le modèle d'architecture et le modèle d'application sont complètement indépendants l'un de l'autre. Cette propriété permet de déployer une application sur plusieurs architectures, mais également d'utiliser une même architecture pour exécuter plusieurs applications. Pour chaque couple composé d'une application et d'une architecture cible, les contraintes de déploiement sont spécifiées par le développeur dans un *scénario*.

- **Flux de travail** : Dans PREESM, il est possible de personnaliser les opérations qui sont successivement appliquées par l'outil de prototypage rapide. Le flux de travail de PREESM est un graphe acyclique édité graphiquement par le développeur. Chacun des nœuds de ce graphe représente une tâche exécutée par l'outil, tel qu'une transformation de graphe, la simulation du système, ou la génération de code.

  Dans la Figure A.3, le flux de travail présenté contient 5 tâches :

  - *Aplatir hiérarchie* : Cette tâche permet de transformer un diagramme IBSDF hiérarchique en un diagramme SDF équivalent. En contrôlant la profondeur de hiérarchie devant être aplatie, le développeur peut adapter la granularité du graphe SDF obtenu. La granularité d'un graphe SDF caractérise la quantité de calcul effectuée par ses acteurs relativement au temps nécessaire pour transmettre les jetons de données sur les arcs du graphe.
  - *Dérouler graphe* : Cette tâche permet de révéler tout le parallélisme d'une application. Dérouler un graphe SDF consiste à transformer ce graphe en un graphe équivalent dans lequel les taux de production et de consommation de jetons de

données sont égaux sur chacune des FIFOs. Par exemple, le graphe SDF présenté en Figure A.4 est obtenu en aplatissant puis en déroulant le graphe IBSDF de la Figure A.2(b).

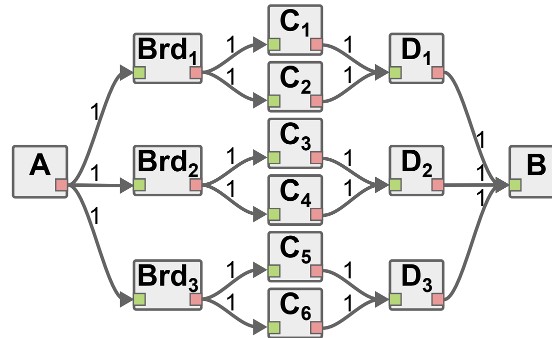


FIGURE A.4 – *Graphe SDF obtenu par aplatissement et déroulage du graphe IBSDF présenté en Figure A.2(b)*

  – *Ordonnancer* : Cette tâche a pour but d'assigner les acteurs du graphe SDF déroulé aux unités de traitement qui seront chargées de leurs exécutions. La tâche d'ordonnancement détermine également l'ordre d'exécution des acteurs sur chaque unité de traitement de l'architecture matérielle. L'objectif de cette tâche est de produire une solution maximisant les performances de l'application.

    L'ordonnancement d'une application est un problème complexe pour lequel il est impossible de trouver une solution optimale en un temps polynomial[Kwo97]. L'ordonnanceur utilisé dans PREESM permet au développeur de contrôler sa complexité et offre ainsi un compromis entre le temps d'exécution de la tâche d'ordonnancement et la qualité du résultat produit [PMAN09].

  – *Afficher Gantt et métriques* : Cette tâche a pour rôle de simuler le comportement du système généré par l'outil de prototypage rapide. Les résultats de simulations sont présentés sous la forme d'un diagramme de Gantt représentant l'activité des différentes unités de traitement de l'architecture au cours du temps. Le diagramme de Gantt est accompagné d'un ensemble de métriques permettant d'évaluer rapidement certains aspects du système simulé, tels que la répartition des calculs sur les unités de traitement, ou les performances de l'application par rapport à son déploiement sur une architecture monocoœur.

  – *Générer code C* : Cette dernière tâche a pour rôle de traduire les choix de conception faits par l'outil de prototypage rapide en code exécutable par l'architecture ciblée. Des architectures multiprocesseurs x86, c6x [Tex13] et ARM [HDN+12] sont actuellement supportées par la tâche de génération de code de PREESM.

- **Chaîne de développement** : Afin de compiler le code généré par l'outil de prototypage rapide, une chaîne de développement traditionnelle est nécessaire. L'utilisation de cette chaine de développement permet au développeur de profiter de toutes les optimisations de compilation développées pour l'architecture ciblée, en plus des optimisations réalisées par l'outil de prototypage rapide.

Comme présenté en Figure A.3, la tâche d'ordonnancement est au cœur de l'environnement de prototypage rapide. Ainsi, l'objectif principal de PREESM est d'optimiser les

performances du système en minimisant sa latence et en maximisant son débit. Les prochaines sections proposent de nouvelles techniques permettant d'étudier et d'optimiser les aspects mémoires des applications à différentes étapes du flux de travail du prototypage rapide.

A.4 Optimisation mémoire des diagrammes de flux de données

Les travaux de recherche existant sur les aspects mémoires des MPSoCs portent principalement sur la mise au point de techniques visant à minimiser la quantité de mémoire allouée pour l'exécution d'une application [MB00, Fab79]. Ces techniques ne peuvent généralement être appliquées que dans les dernières étapes du processus de conception d'un système embarqué, après l'étape d'ordonnancement des acteurs de l'application.

La méthode présentée dans cette thèse permet d'étudier et d'optimiser les caractéristiques d'une application dès les premières étapes d'un flux de conception, indépendamment de toute information sur l'architecture ciblée.

Graphe d'exclusion mémoire

Les techniques présentées dans cette thèse reposent sur un graphe modélisant les caractéristiques mémoires d'une application modélisée par un graphe IBSDF. Cette représentation intermédiaire prend la forme d'un graphe non-orienté appelé graphe d'exclusion mémoire, dont les nœuds représentent les objets à allouer en mémoire, et dont les arêtes représentent des exclusions entre les objets mémoires. Deux objets mémoires s'excluent s'ils représentent des données devant coexister durant l'exécution de l'application. De tels objets mémoires ne peuvent donc pas être alloués dans des espaces mémoire se chevauchant. À l'inverse, deux objets mémoires ne stockant pas des données valides simultanément ne s'excluent pas et peuvent être alloués dans des espaces mémoire superposés.

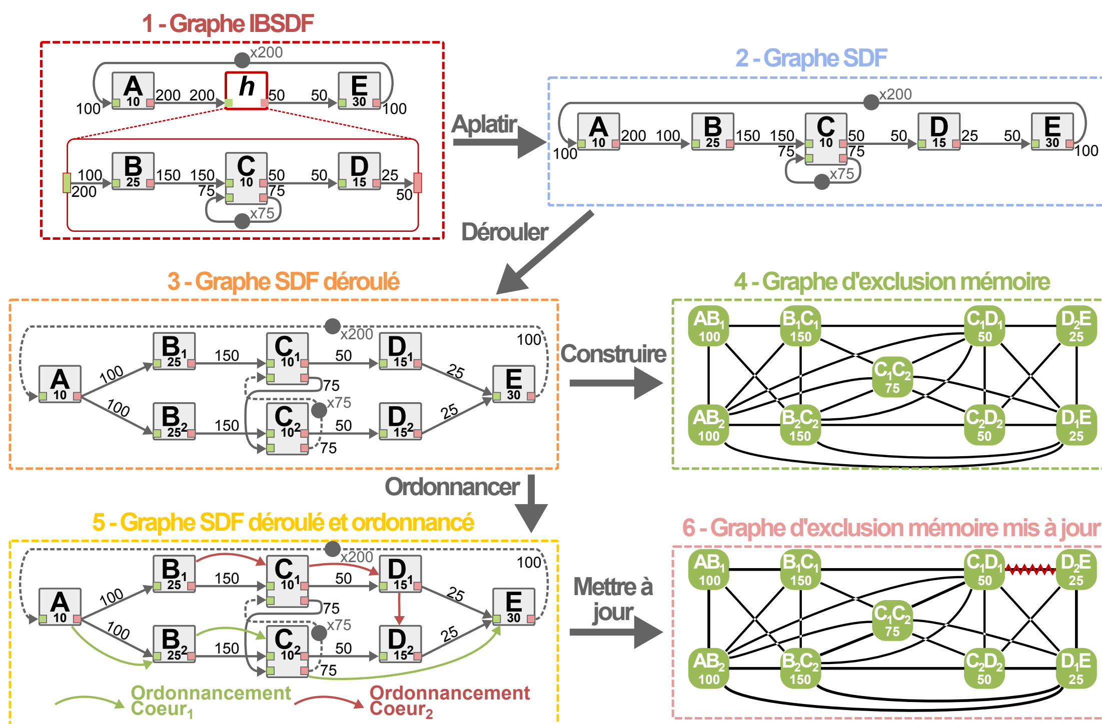


FIGURE A.5 – *Transformation d'un graphe IBSDF en graphe d'exclusion mémoire*

Le diagramme présenté en Figure A.5 illustre les transformations de graphe appliquées successivement pour construire le graphe d'exclusion mémoire. L'aplatissement et le déroulage du graphe IBSDF permettent de révéler le parallélisme de l'application, facilitant ainsi l'identification des objets mémoires pouvant coexister. Dans cet exemple, chaque FIFO du graphe SDF déroulé représente un objet de taille fixe devant être alloué en mémoire. Des exclusions sont ajoutées dans le graphe d'exclusion entre les objets mémoires qui ne sont pas liés par une relation d'antériorité. Par exemple, comme il n'existe aucun arc (ou chemin) liant les FIFOS AB_2 et C_1D_1, les objets mémoires correspondants à ces FIFOS sont liés par une exclusion dans le graphe d'exclusion mémoire.

Le graphe d'exclusion mémoire construit à partir du graphe SDF déroulé peut être mis à jour afin de prendre en compte l'ordonnancement de l'application. En effet, l'ordonnancement d'un graphe SDF résulte en la création de nouvelles relations d'antériorité entre certains objets mémoires de l'application. Par exemple, dans la Figure A.5, l'ordonnancement du graphe SDF sur deux unités de traitement crée une relation d'antériorité entre les FIFOS C_1D_1 et D_2E. Le graphe d'exclusion mémoire ainsi construit est utilisé pour borner les besoins mémoires de l'application et sert de base au processus d'allocation mémoire de PREESM.

Bornes mémoires

Borner la quantité de mémoire nécessaire pour l'implémentation d'une application sur une architecture multicœur permet au développeur de dimensionner adéquatement la mémoire du système développé.

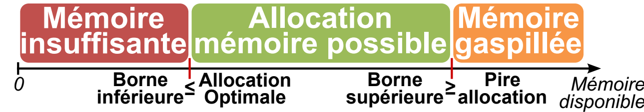


FIGURE A.6 – *Bornes mémoires*

La Figure A.6 illustre la place des bornes mémoires inférieure et supérieure d'une application sur un axe représentant la quantité de mémoire disponible dans une architecture. Si l'architecture ciblée possède moins de mémoire que la borne mémoire inférieure d'une application, il sera impossible d'implémenter cette application, en l'état, sur cette architecture. À l'inverse, si une architecture ciblée possède plus de mémoire que la borne mémoire supérieure d'une application, certaines parties de cette mémoire ne seront jamais utilisées pour l'implémentation de cette application. En effet, la borne mémoire supérieure représente la pire allocation possible pour une application : l'allocation utilisant le plus grand espace mémoire.

La borne mémoire supérieure d'une application se calcule simplement en sommant les tailles de tous les objets mémoires contenus dans son graphe d'exclusion mémoire. La borne mémoire inférieure se calcule en trouvant le clique de poids maximum dans le graphe d'exclusion mémoire. Plusieurs méthodes pour résoudre ce problème sont présentées dans le Chapitre 4 de cette thèse.

A.5 Optimisation mémoire des acteurs

Les graphes d'exclusion mémoire ne modélisent que les opportunités de réutilisation mémoire révélées en analysant le graphe IBSDF d'applications. Les graphes IBSDF ne contenant pas d'information sur le comportement interne des acteurs, il est supposé qu'un acteur

conserve un accès à toutes ses entrées et sorties durant tout son temps d'exécution. Pour relaxer cette contrainte, il est nécessaire de fournir à l'outil de prototypage rapide des informations sur les dépendances de données internes aux acteurs.

Annotations pour les ports d'acteurs

La première contribution présentée dans le Chapitre 5 est un jeu d'annotations pouvant être associées aux ports des acteurs par le développeur. Ces annotations permettent de spécifier le comportement de l'acteur vis-à-vis des données disponibles sur ses ports. Ces 3 annotations sont décrites ci-après :

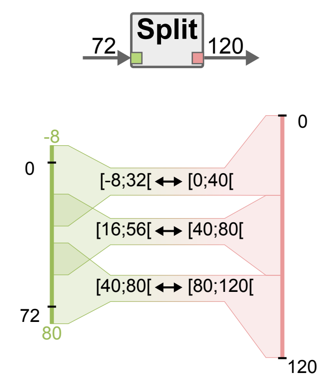
- **Lecture seule** : L'acteur possédant un port d'entrée en lecture seule ne peut que lire des données à partir de ce port. Comme pour une variable `const` en langage C, ou pour une variable `final` en langage Java, il est impossible pour un acteur d'écrire dans la mémoire tampon associée à un port en lecture seule.
- **Écriture seule** : L'acteur possédant un port de sortie en écriture seule ne peut qu'écrire des données dans ce port. Un acteur n'est donc pas autorisé à lire les données stockées dans la mémoire tampon associée à un port en écriture seule, même si ces données ont été écrites par l'acteur lui-même.
- **Inutilisé** : L'acteur possédant un port d'entrée inutilisé n'accèdera jamais aux données stockées dans la mémoire tampon associée à ce port. Un port d'entrée inutilisé est semblable à un fichier `/dev/null` dans les systèmes d'exploitation Unix, tout jeton de données transmis sur ce port est immédiatement détruit.

Scripts mémoires

La seconde contribution présentée dans le Chapitre 5 est un système de scripts permettant au développeur de spécifier des opportunités de réutilisation mémoire pour l'allocation des mémoires tampons associées aux ports d'entrée et de sortie d'un acteur. En d'autres termes, les scripts mémoires permettent au développeur d'autoriser explicitement l'allocation des objets mémoires associés à ses ports dans des espaces mémoires se chevauchant.

```
// Mémoires tampons d'entrée: E
// Mémoires tampons de sortie: S
// Paramètres: h, l, h_jointure, nb_tranche
h_tranche ← h/nb_tranche + 2 * h_jointure;
taille_tranche ← h_tranche * l;
pour i ← 0 a nb_tranche − 1 faire
    idxSrc ← (i * h/nb_tranche − h_jointure) * l;
    intervalSrc ← [idxSrc, idxSrc + taille_tranche];
    intervalDst ← [i * taille_tranche, (i + 1) * taille_tranche];
    Associe E.intervalSrc avec S.intervalDst;
fin
```

(a) Script mémoire pour l'acteur *Split*



(b) Associations créées pour l'acteur *Split*

Figure A.7 – *Script mémoire pour les acteurs* Split

Le script mémoire présenté en Figure A.7(a) est associé à l'acteur *Split* qui a pour rôle de découper une image de hauteur h et de largeur l en $nb_{tranche}$ tranches. Les pixels

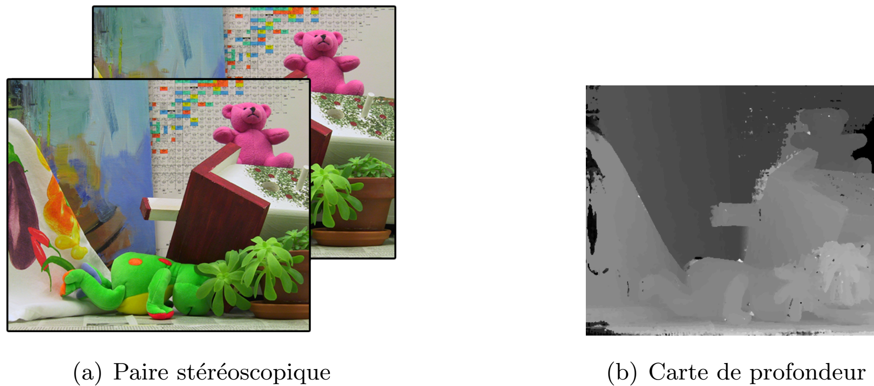
de deux tranches d'image successives se chevauchent sur une bande de $h_{jointure}$ pixels de haut.

L'objectif principal du script mémoire est de créer des associations entre des intervalles d'octets issus des mémoires tampons d'entrée et des intervalles issus des mémoires tampons de sortie. Par exemple, dans le script présenté en Figure A.7(a), une association d'intervalles est créée pour chaque tranche produite par l'acteur. La Figure A.7(b) illustre les intervalles associés pour une exécution de l'acteur *Split* avec les paramètres suivants : $h = 9$, $l = 8$, $nb_{tranche} = 3$, et $h_{jointure} = 1$.

En combinant les informations données par le développeur sous forme d'annotations et de scripts mémoires, l'outil de prototypage rapide parvient à minimiser automatiquement la quantité de mémoire allouée pour une application.

A.6 Etude de cas : application d'appariement stéréoscopique

L'appariement stéréoscopique



(a) Paire stéréoscopique (b) Carte de profondeur

Figure A.8 – *Exemple de paire stéréoscopique tirée de la base de donnée [SS02]*

L'appariement stéréoscopique est un algorithme dont l'objectif est de mettre en correspondance les pixels de deux images d'une même scène capturées par deux caméras séparées par une faible distance (cf. Figure A.8(a)). Le résultat produit par l'algorithme est une carte de profondeur permettant d'évaluer la distance entre la caméra et les éléments de la scène capturée (cf. Figure A.8(b)).

Le graphe IBSDF modélisant l'application d'appariement stéréoscopique est présenté dans le Chapitre 6. Cette application constitue un cas d'étude intéressant de par l'importance des calculs qu'elle suppose, mais également pour son fort degré de parallélisme. En effet, les acteurs nécessitant le plus de calcul peuvent s'exécuter jusqu'à soixante fois en parallèle. De plus, le déploiement de cette application sur une architecture multiprocesseur embarquée, où les ressources mémoires sont souvent limitées, présente un réel défi.

Résultats d'expérimentation

Le Tableau A.1 présente les caractéristiques mémoires de l'application d'appariement stéréoscopique dans 4 scénarios différents. Chaque scénario correspond à une mise à jour spécifique du graphe d'exclusion mémoire. Les résultats sont présentés pour : un graphe d'exclusion mémoire n'ayant pas été mis à jour avec les informations d'ordonnancement, un graphe d'exclusion mis à jour avec les informations d'ordonnancement, et un graphe

d'exclusion mis à jour avec un minutage de l'application (qui ajoute de nouvelles informations d'antériorités). Les résultats présentés pour chaque scénario sont les bornes d'allocations inférieures et supérieures, ainsi que les empreintes mémoires allouées pour le déploiement de l'application sur deux architectures multiprocesseurs : le processeur généraliste Intel i7-3610QM [Int13] et le processeur de traitement du signal Texas Instrument TMS320C6678 [Tex13].

	Bornes (Mo)		Allocations[2] (Mo)	
Scénarios	Supérieure	Inférieure	i7	C6678
Pré-ordonnancement[1]	1453	1314	+0	+0.051
Pré-ordonnancement	170	100	+0.164	+0.679
Post-ordonnancement	170	80	+0	+0.014
Post-minutage	170	68	+0	+0.342

1 : Scripts mémoires non-appliqués dans ce scénario.
2 : Relativement à la borne inférieure.

Table A.1 – *Résultats d'allocation dans différents scénarios*

Les deux premières lignes du Tableau A.1 présentent les résultats obtenus avant l'ordonnancement, sans puis avec les scripts mémoires. L'utilisation des scripts mémoires permet donc une réduction de l'empreinte mémoire de 92% pour cette application, passant de 1314Mo à 100Mo. Les mises à jour suivantes du graphe d'exclusion mémoire avec les informations d'ordonnancement et de minutage permettent des réductions successives de 19% et 17% de l'empreinte mémoire allouée.

Sur cette application, les techniques d'optimisation mémoire présentées dans cette thèse résultent en une empreinte mémoire 15% plus faible que celle obtenue avec une technique de dimensionnement de Fifos [SGB06] qui est une référence dans la littérature sur l'optimisation mémoire pour les graphes de flux de données.

A.7 Vers plus de dynamisme : le modèle PiSDF

Le modèle SDF étudié dans les sections précédentes a une expressivité limitée. En effet, dans le modèle SDF les taux de production et de consommation de jetons de données des acteurs sont des constantes dont la valeur ne peut être modifiée durant l'exécution de l'application.

Pour remédier à cette limitation tout en préservant la compositionalité du modèle IBSDF, un nouveau méta-modèle de flux de données nommé PiMM est introduit dans le Chapitre 7. Dans le contexte des modèles de calcul de type flux de donnée, un méta-modèle est un ensemble d'éléments de sémantique pouvant être ajoutés à la sémantique d'un modèle existant afin de lui apporter de nouvelles capacités. En plus du mécanisme de composition hiérarchique inspiré du modèle IBSDF, le méta-modèle PiMM apporte un mécanisme de reconfiguration dynamique au modèle auquel il est appliqué.

Sémantique du PiMM

L'ensemble des éléments de sémantique du méta-modèle PiMM est présenté en Figure A.9. Comme indiqué dans cette figure, le modèle résultant de l'application du PiMM au modèle SDF se nomme le πSDF. En plus des interfaces hiérarchiques issues du modèle IBSDF, la sémantique du modèle πSDF contient également des paramètres et des dépendances de paramètres. Un paramètre est un nœud du graphe associé à une valeur entière pouvant être

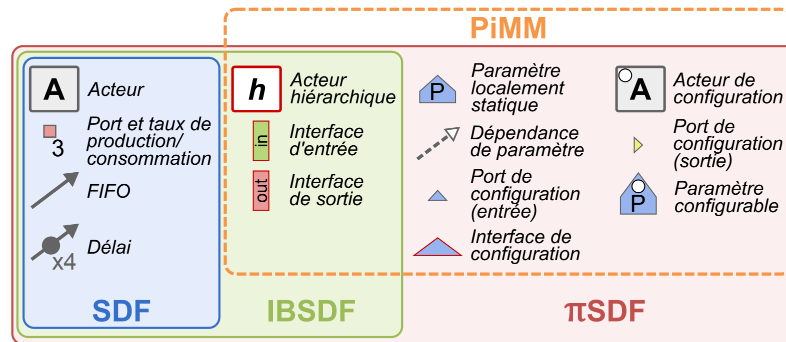


Figure A.9 – *Sémantique du méta-modèle PiMM*

utilisée pour calculer les taux de production et de consommation des acteurs du graphe. Les dépendances de paramètres ont pour rôle de relier les paramètres aux acteurs qu'ils influencent.

Il existe deux types de paramètres dans le méta-modèle PiMM : les paramètres localement statiques et les paramètres configurables. Les paramètres localement statiques ont une valeur constante durant toute la durée d'exécution du graphe auquel ils appartiennent. Les paramètres configurables en revanche se voient assigner une nouvelle valeur à chaque nouvelle itération du graphe auquel ils appartiennent. Les acteurs de configuration sont des acteurs spéciaux qui ont la capacité d'assigner une nouvelle valeur aux paramètres configurables. En contrepartie, les acteurs de configurations doivent être exécutés exactement une fois au début de chaque itération du graphe auquel ils appartiennent. Cette restriction permet de garantir que les paramètres reconfigurables recevront une nouvelle valeur à intervalles réguliers, renforçant ainsi la prédictibilité du modèle et des applications décrites.

Exemple d'application

La Figure A.10 présente un graphe de flux de données d'une application de traitement d'image utilisant la sémantique du modèle πSDF. Dans cette application, les acteurs *Read* et *Send* sont respectivement utilisés pour lire une source de pixels, et pour envoyer des paquets de 3 pixels sur un réseau. L'acteur hiérarchique *Filter* a pour rôle d'appliquer un filtre à une image 2D dont la taille est fixée par le paramètre localement statique *size*.

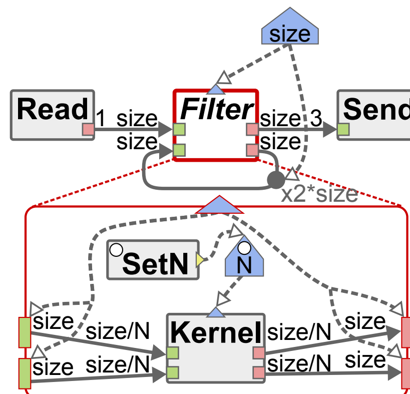


Figure A.10 – *Graphe πSDF d'une application de traitement d'image*

Dans le sous-graphe de l'acteur *Filter*, l'acteur de configuration *SetN* a pour rôle d'assigner une nouvelle valeur au paramètre configurable N. Ce paramètre influence les taux de production et de consommation de l'acteur *Kernel* qui sont utilisés pour déterminer le nombre d'exécutions de l'acteur. Ainsi, pour chaque nouvelle image traitée par l'acteur hiérarchique *Filter*, le nombre d'exécutions simultanées de l'acteur *Kernel* peut être modifié dynamiquement pour s'adapter, par exemple, au nombre de processeurs disponibles dans l'architecture.

A.8 Conclusion

Les contributions présentées dans cette thèse sont résumées en Figure A.11. Le principal objectif de ces contributions est de faciliter l'étude et l'optimisation des caractéristiques mémoires d'applications modélisées par des graphes de flux de données IBSDF. Toutes les techniques présentées dans cette thèse sont implémentées dans l'outil de prototypage rapide PREESM.

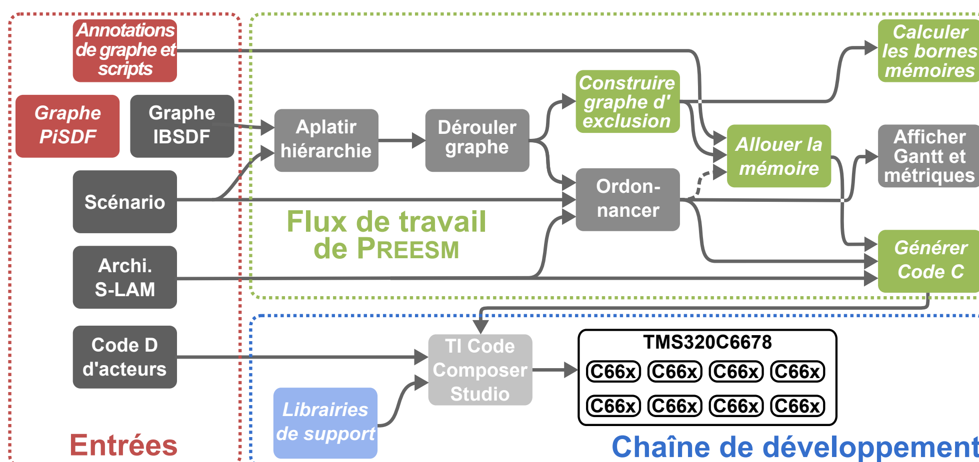


FIGURE A.11 – *Flux de travail typique de PREESM incluant les nouvelles contributions présentées dans cette thèse.*

Perspectives

Le travail réalisé durant cette thèse a permis d'identifier de nombreux axes intéressants pour de futurs travaux de recherche. Un premier axe possible serait de rendre compatible les méthodes d'allocation mémoire avec des architectures à mémoire distribuée. En effet, les méthodes d'allocation basées sur le graphe d'exclusion mémoire ne sont, pour l'instant, adaptées qu'à des architectures multiprocesseurs possédant une mémoire partagée entre toutes les unités de traitement. Il serait également intéressant de développer un nouvel ordonnanceur capable de minimiser à la fois la latence de l'application et l'empreinte mémoire nécessaire à son déploiement. Enfin, de futurs travaux de recherche pourront porter sur l'analyse avancée des propriétés des diagrammes πSDF, ou sur l'extension des techniques d'optimisation mémoire pour supporter le dynamisme offert par le modèle πSDF.

List of Figures

2.1 Dataflow Process Network (DPN) MoC 14
2.2 Illustration of the 4 sources of parallelism in dataflow MoCs 15
2.3 Synchronous Dataflow (SDF) MoC 19
2.4 Single-rate SDF graph derived from the SDF graph of Figure 2.3(b) 21
2.5 Directed Acyclic Graph (DAG) derived from the SDF graph of Figure 2.3(b) 22
2.6 Cyclo-Static Dataflow (CSDF) MoC 23
2.7 SDF graph equivalent to the CSDF graph of Figure 2.6(b). 23
2.8 Hierarchical SDF graph 25
2.9 Non-compositionality of naive hierarchical SDF MoC 25
2.10 Interface-Based SDF (IBSDF) MoC 26
2.11 Flattening and single-rate transformations of the IBSDF graph of Figure 2.10(b) 27
2.12 Example of Parameterized SDF (PSDF) graph 29
2.13 Example of Schedulable Parametric Dataflow (SPDF) graph 32
2.14 Scenario-Aware Dataflow (SADF) MoC 33

3.1 Overview of a rapid prototyping design flow. 36
3.2 Example of heterogeneous architecture model 38
3.3 Example of application model: a homogeneous SDF graph. 39
3.4 Gantt charts for 2 mappings of the application from Figure 3.3 on the architecture from Figure 3.2. 41
3.5 Example of typical PREESM workflow 44
3.6 Screenshot of the IBSDF graph editor of PREESM 46
3.7 Example of the impact of scheduling choice on FIFO size 50
3.8 Example of memory reuse opportunity 50

4.1 Example of IBSDF graph 56
4.2 SDF graph resulting from the flattening of Figure 4.1 57
4.3 Single-rate SDF graph derived from IBSDF graph of Figure 4.1 (Directed Acyclic Graph (DAG) if dotted FIFOs are ignored) 57
4.4 Memory Exclusion Graph (MEG) derived from the IBSDF graph of Figure 4.1 60
4.5 Memory Bounds 61
4.6 Cliques examples 65
4.7 Example of optimal memory allocation for the MEG of Figure 4.4. 67

4.8 Four techniques for the computation of memory bounds. 68
4.9 Runtime of the three MWC algorithms for random MEGs of density 0.80. . 69
4.10 Simplified development flow of PREESM with possible stages for memory allocation. 70
4.11 Update of the DAG from Figure 4.3 with scheduling information. 71
4.12 MEG updated with scheduling information from Figure 4.11 and 4.13. . . 71
4.13 Example of timed schedule for the DAG of Figure 4.3. 72
4.14 Loss of runtime flexibility with timed allocation for DAG of Figure 4.11. . . 74
4.15 Performance of pre-scheduling allocation algorithms for 50 graphs. 77
4.16 Performance of post-scheduling allocation algorithms for 50 graphs. 77
4.17 Performance of timed allocation algorithms for 50 graphs. 78

5.1 Image processing SDF graph 84
5.2 Single-rate SDF graph derived from the SDF graph of Figure 5.1. 85
5.3 Automatically inserted actors and corresponding MEG. Diagram within actors represent the copy operations realized by the actor. 85
5.4 *Median* actor internal behavior with in-place results. 87
5.5 Memory reuse opportunities for custom actors. 88
5.6 Memory script for *Fork* actors 92
5.7 Memory script for *Broadcast* actors 92
5.8 Memory script for *Split* actors 93
5.9 Single-rate SDF graph from Figure 5.2 for $h = 9$, $w = 8$, and $n = 3$. r and w mark *read-only* and *write-only* ports respectively. Red letters uniquely identify the FIFOs. 95
5.10 Match tree associated to buffers of actors *RGB2Gray*, *Split*, and *Fork*. . . . 96
5.11 Matching patterns with inapplicable matches. 97
5.12 Divisibility of buffers 98
5.13 Folding the match tree formed by *RGB2Gray-Split-Fork* 103
5.14 Allocation resulting from match tree folding vs. MEG-based allocation . . . 104
5.15 Update of the MEG derived from the single-rate SDF graph of Figure 5.9. . 105
5.16 Removing exclusion between *write-only* and *unused* memory objects of a MEG. 106

6.1 Stereo Matching Example from [SS02] database 108
6.2 Stereo-matching IBSDF graphs.[1] 109
6.3 Memory script for *RGB2Gray* actors 111
6.4 Cache coherence solution without memory reuse 112
6.5 Cache coherence issue with memory reuse 113
6.6 Multicore cache coherence solution 113
6.7 Cache Alignment Issue 115
6.8 Throughput of the stereo matching application depending on the number of cores. 119
6.9 Memory footprint of the stereo matching application depending on the number of targeted C6x cores. 119
6.10 Dynamic allocation: Heap size after N iterations. Each blue line represents the heap size for an execution of the stereo matching application. 121

7.1 Image processing example: SDF graph 124
7.2 Image processing example: IBSDF graph 126
7.3 Image processing example: PSDF graph 126

7.4 PiMM semantics 127
7.5 Image processing example: Static πSDF graph 129
7.6 Image processing example: πSDF graph 130
7.7 πSDF description of a configurable adder 134
7.8 πCSDF description of a configurable adder 135
7.9 πSDF model of an FIR filter 136
7.10 πSDF model of the bit processing part of the LTE PUSCH decoding. . . . 137
7.11 PSDF model of the bit processing part of the LTE PUSCH decoding. . . . 138
7.12 Quasi-static schedule for graph in Figure 7.10 139

8.1 Preesm typical workflow including new contributions from this thesis. . . . 144

A.1 Modèle de calcul SDF 150
A.2 Modèle de calcul IBSDF 151
A.3 Flux de travail typique de Preesm 152
A.4 Graphe SDF obtenu par aplatissement et déroulage du graphe IBSDF présenté en Figure A.2(b) 153
A.5 Transformation d'un graphe IBSDF en graphe d'exclusion mémoire 154
A.6 Bornes mémoires 155
A.7 Script mémoire pour les acteurs *Split* 156
A.8 Exemple de paire stéréoscopique tirée de la base de donnée [SS02] 157
A.9 Sémantique du méta-modèle PiMM 159
A.10 Graphe πSDF d'une application de traitement d'image 159
A.11 Flux de travail typique de Preesm incluant les nouvelles contributions présentées dans cette thèse. 160

List of Tables

2.1 Features comparison of presented dataflow MoCs 34

3.1 Costs for the execution of actors from Figure 3.3 on processing elements from Figure 3.2. 40

4.1 Algorithm proceeding for the MEG of Figure 4.4 67
4.2 Performance of Maximum-Weight Clique algorithms on random MEGs . . . 68
4.3 Properties of the test graphs 69
4.4 Performance of MWC algorithms on MEGs derived from the test graphs of Table 4.3 69
4.5 Properties of the test graphs 75
4.6 Pre-scheduling memory allocation 76
4.7 Post-scheduling memory allocation 77
4.8 Timed memory allocation 78
4.9 Comparison of MEG allocation results with FIFO dimensioning technique from SDF3 [Ele13]. 80

6.1 MEGs characteristics and allocation results 117
6.2 Comparison of the stereo matching performance with static and dynamic allocations 119

7.1 Features comparison of different dataflow MoCs 139

A.1 Résultats d'allocation dans différents scénarios 158

Acronyms

πCSDF Parameterized and Interfaced CSDF. 135, 136, 163

πSDF Parameterized and Interfaced SDF. 127–139, 141, 145, 158–160, 163

AAA Algorithm-Architecture Adequation. 35, 42

ABC Architecture Benchmark Computer. 45

ADF Affine Dataflow. 24, 34, 138, 139

ALU Arithmetic Logic Unit. 74

APEG Acyclic Precedence Expansion Graph. 23

Blodi BLOck DIagram compiler. 11

BF Best-Fit. 49, 74–78, 118

CAL CAL Actor Language. 14

CB Code Block. 137

CPU Central Processing Unit. 37, 38, 40, 43, 45, 67, 111, 112, 114, 117–120

CSDF Cyclo-Static Dataflow. 18, 23, 24, 28, 34, 42, 43, 89, 135, 161, 167

DAG Directed Acyclic Graph. 22, 23, 44, 55–62, 69–74, 79, 128, 161, 162, 169

DIF Dataflow Interchange Format. 46

DMA Direct Memory Access. 37

DPN Dataflow Process Network. 13, 14, 16–18, 23, 30–32, 34, 43, 46, 138, 139, 161

DSP Digital Signal Processing. 4, 6, 11, 12, 14, 17, 24, 29, 38, 40, 42, 43, 45, 111, 112, 117–120, 123, 134, 145

DSSF Deterministic SDF with Shared Fifos. 27, 28, 34, 138, 139

EIDF Enable-Invoke Dataflow. 43

eNodeB base station. 136, 137

Fifo First-In First-Out queue. 12–16, 18–22, 24, 27, 33, 39, 50, 51, 56–62, 69, 75, 79, 80, 83–85, 89, 94–98, 101, 104–106, 112, 113, 124, 125, 127, 128, 131, 133, 138, 141, 145, 150, 153, 155, 158, 161, 162, 165, 167

FF First-Fit. 49, 74–78

FFT Fast Fourier Transform. 24, 87

FIR Finite Impulse Response. 24, 133, 134, 136, 163

fps Frames per second. 40, 117–119

FSM Finite-State Machine. 12, 33

GPU Graphics Processing Unit. 37, 107

IBSDF Interface-Based SDF. 25–27, 31, 34, 42–46, 50, 51, 55–58, 60, 61, 68, 69, 72, 75, 79, 80, 83, 84, 106, 108–111, 116, 118, 123, 125, 126, 128, 129, 132, 138, 139, 143, 145, 151–155, 157, 158, 160–163

IDE Integrated Development Environment. 42, 43

IETR Institute of Electronics and Telecommunications of Rennes. 6, 7

KPN Kahn Process Network. 14, 43

LIDE DSPCAD Lightweight Dataflow Environment. 43, 46

LTE Long Term Evolution. 68–70, 75, 123, 133, 136–138, 163

MAC Medium Access Control. 137

MAPS MPSoC Application Programming Studio. 43, 152

MCSE methodology for the design of electronic systems. 11

MEG Memory Exclusion Graph. 55–77, 79, 80, 84–86, 96, 104–106, 117, 118, 161, 162, 165

MIMD Multiple Instructions, Multiple Data. 37

MoC Model of Computation. 5, 7, 11–19, 21–34, 38, 39, 42, 43, 46, 51, 57, 83, 86, 89, 90, 110, 123–127, 131, 133, 135, 137–139, 143, 144, 149, 161, 165

MPSoC Multiprocessor System-on-Chip. 4–7, 38, 42, 43, 47, 51, 61, 73, 74, 83, 104, 107, 108, 112, 141, 143, 144, 147–149, 154, 168

MWC Maximum-Weight Clique. 64, 65, 67–70, 162, 165

Orcc Open RVC-CAL Compiler. 43, 46

Preesm the Parallel and Real-time Embedded Executives Scheduling Method. 6, 7, 42–47, 56, 58, 59, 65, 67, 70, 74, 80, 83, 84, 91, 95, 114, 116, 118, 119, 143–145, 151–153, 155, 160–163

PDAG Parameter dependency Directed Acyclic Graph. 128, 129, 133, 140

PiMM Parameterized and Interfaced dataflow Meta-Model. 123–129, 131, 133–136, 138, 140, 141, 143–145, 158, 159, 163

PSDF Parameterized SDF. 28–31, 34, 43, 126, 127, 132, 133, 138, 139, 141, 161–163

PUSCH Physical Uplink Shared Channel. 123, 124, 136–138, 163

RV Repetition Vector. 20, 21, 25, 30, 125, 127, 138, 140

S-LAM System-Level Architecture Model. 6, 43–45, 152

SADF Scenario-Aware Dataflow. 14, 28, 32–34, 42, 138, 139, 141, 161

SDF Synchronous Dataflow. 14, 18–34, 39, 42–46, 49–51, 55–58, 60, 62, 69, 70, 75, 76, 79, 84–87, 89–91, 95, 96, 101, 104–106, 112, 114, 118, 123–127, 132, 135, 137–139, 141, 150–153, 155, 158, 161–163, 167–169

SDF3 SDF For Free. 42, 43, 46, 75, 76, 79, 80, 152, 165

SIMD Single Instruction, Multiple Data. 37

SoC System-on-Chip. 4

SPDF Schedulable Parametric Dataflow. 28, 31, 32, 34, 138, 139, 161

UE User Equipment. 136, 137

UML Unified Modeling Language. 11

VHDL VHSIC Hardware Description Language. 12–14

Personal Publications

[DEA+14] K. Desnos, S. El Assad, A. Arlicot, M. Pelcat, and D. Menard. Efficient Multicore Implementation of An Advanced Generator of Discrete Chaotic Sequences In *International Conference for Internet Technology and Secured Transactions (ICITST)*, 2014.

[DPN+13] K. Desnos, M. Pelcat, J.-F. Nezan, S.S. Bhattacharyya, and S. Aridhi. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, pages 41–48. IEEE, 2013. 7

[DPNA12] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. Memory bounds for the distributed execution of a hierarchical synchronous data-flow graph. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XII)*, 2012. 6, 61

[DPNA13] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. Pre-and post-scheduling memory allocation strategies on mpsocs. In *Electronic System Level Synthesis Conference (ESLsyn)*, 2013. 6, 70, 79

[DPNA14] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs In *Journal of Signal Processing Systems*, Springer, 2014. 7

[ABP+14] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and M. Abid MARTE to ΠSDF transformation for data-intensive applications analysis In *Design and Architectures for Signal and Image Processing (DASIP)*, 2014.

[HDN+12] J. Heulot, K. Desnos, J.-F. Nezan, M. Pelcat, M. Raulet, H. Yviquel, P.-L. Lagalaye, J.-C. Le Lann An experimental toolchain based on high-level dataflow models of computation for heterogeneous mpsoc. In *Design and Architectures for Signal and Image Processing (DASIP)*, 2012. 6, 22, 42, 45, 46, 153

[HPD+14] J. Heulot, M. Pelcat, K. Desnos, J.-F. Nezan, and S. Aridhi. SPIDER: A Synchronous Parameterized and Interfaced Dataflow-Based RTOS for Multicore DSPs In *European Embedded Design in Education and Research Conference (EDERC)*, 2014.

[PDH+14] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi. PREESM: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming In *European Embedded Design in Education and Research Conference (EDERC)*, 2014.

[ZDP+13] Z. Zhou, K. Desnos, M. Pelcat, J.-F. Nezan, W. Plishker, and S.S. Bhattacharyya. Scheduling of parallelized synchronous dataflow actors. In *System on Chip (SoC), 2013 International Symposium on*, pages 1–10, Oct 2013. 14, 15, 41, 46

[ZPB+14] Z. Zhou, W. Plishker, S.S. Bhattacharyya, K. Desnos, M. Pelcat, and J.-F. Nezan. Scheduling of Parallelized Synchronous Dataflow Actors for Multicore Signal Processing In *Journal of Signal Processing Systems*, Springer, 2014.

Bibliography

[ABB+13] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. Dupont de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. Morey Chaisemartin, T.H. Nguyen, X. Raynaud, and R. Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18(0):1624 – 1633, 2013. 2013 International Conference on Computational Science. 14

[ABBB13] O.J. Arndt, D. Becker, C. Banz, and H. Blume. Parallel implementation of real-time semi-global matching on embedded multi-core architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, 2013. 107

[Ada14] Adapteva. Epiphany-IV E64G401 product page, 2014. http://www.adapteva.com/epiphanyiv/. 37, 144

[Ash90] P.J. Ashenden. *The VHDL cookbook*. Department of Computer Science, University of Adelaide, 1990. 13

[BAH09] M. Bouchard, M. Angalović, and A. Hertz. About equivalent interval colorings of weighted graphs. *Discrete Appl. Math.*, October 2009. 49, 63, 64, 75

[Bar09] M. Barr. Real men program in C. *Embedded Systems Design*, 22(7):3, 2009. 5

[BASC+13] R. Ben Atitallah, E. Senn, D. Chillet, M. Lanoe, and D. Blouin. An efficient framework for power-aware design of heterogeneous MPSoC. *Industrial Informatics, IEEE Transactions on*, 9(1):487–501, 2013. 42

[BB01] B. Bhattacharya and S.S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *Signal Processing, IEEE Transactions on*, 2001. 16, 28, 29, 30, 123, 125, 129, 131, 132, 133, 138, 140

[BDM09] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009. 37, 47

[BE81] C.S. Burrus and P. Eschenbacher. An in-place, in-order prime factor FFT algorithm. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 29(4):806–817, Aug 1981. 87

[BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, Feb 1996. 21, 23, 135

[BHLM94] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, Vol. 4:pp. 155–182, 1994. 42, 152

[BKV+08] F. Balasa, P. Kjeldsberg, A. Vandecappelle, M. Palkovic, Q. Hu, H. Zhu, and F. Catthoor. Storage estimation and design space exploration methodologies for the memory management of signal processing applications. *Journal of Signal Processing Systems*, 53:51–71, 2008. 49

[BL93] J.T. Buck and E.A. Lee. 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. 18, 31

[BL06] S.S. Bhattacharyya and W.S. Levine. Optimization of signal processing software for control system implementation. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 1562–1567. IEEE, 2006. 17, 43

[BMKDdD12] B. Bodin, A. Munier-Kordon, and B. Dupont de Dinechin. K-Periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In *Embedded Computer Systems (SAMOS)*, 2012. 117

[BML96] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. *Software synthesis from dataflow graphs*, volume 360. Springer, 1996. 19

[BMMKU10] M. Benazouz, O. Marchetti, A. Munier-Kordon, and Pascal Urard. A new approach for minimizing buffer capacities with throughput constraint for embedded system design. In *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS*, 2010. 50, 56

[Bod13] B. Bodin. *Analyse d'Applications Flot de Données pour la Compilation Multiprocesseur*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2013. 14

[Bou09] J. Boutellier. *Quasi-static scheduling for fine-grained embedded multiprocessing*. PhD thesis, University of Oulu, 2009. 138

[Bru07] P. Brucker. *Scheduling algorithms*, volume 3. Springer, 2007. 41

[BTV12] A. Bouakaz, J.-P. Talpin, and J. Vitek. Affine data-flow graphs for the synthesis of hard real-time applications. *ACSD*, 2012. 24, 138

[Cal93] J.P. Calvez. *Embedded Real-Time Systems*. Wiley Series in Software Engineering Practice. Wiley, 1993. 11

[CCS+08] J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, Ra. Leupers, G. Ascheid, H. Meyr, T.i Isshiki, and H. Kunieda. MAPS: an integrated framework for MPSoC application parallelization. In *Proceedings of the 45th annual Design Automation Conference*, pages 754–759. ACM, 2008. 43, 152

[CDG+14] L. Cudennec, P. Dubrulle, F. Galea, T. Goubier, and R. Sirdey. Generating code and memory buffers to reorganize data on many-core architectures. *Procedia Computer Science*, 29:1123–1133, 2014. 21, 84, 89

[CH89] J.E. Cooling and T.S. Hughes. The emergence of rapid prototyping as a real-time software development tool. In *Software Engineering for Real Time Systems, 1989., Second International Conference on*, pages 60–64, Sep 1989. 35

[Chu32] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932. 12, 149

[CJVDP08] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008. 39

[Den74] J.B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974. 13

[Des13] K. Desnos. PREESM tutorial: Memory footprint reduction, july 2013. http://preesm.sourceforge.net/website/index.php?id=memory-footprint-reduction. 114

[DGCDM97] E. De Greef, F. Catthoor, and H. De Man. Array placement for storage size reduction in embedded multimedia systems. *ASAP*, 1997. 6, 47, 49, 74, 76, 77, 78, 87, 89, 149

[DPN+13] K. Desnos, M. Pelcat, J.-F. Nezan, S.S. Bhattacharyya, and S. 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. 7

[DPNA12] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. Memory bounds for the distributed execution of a hierarchical synchronous data-flow graph. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XII), 2012 International Conference on*, 2012. 6, 61

[DPNA13] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. Pre-and post-scheduling memory allocation strategies on MPSoCs. In *Electronic System Level Synthesis Conference (ESLsyn)*, 2013. 6, 70, 79

[DPNA14] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. Memory analysis and optimized allocation of dataflow applications on shared-memory MPSoCs. *Journal of Signal Processing Systems, Springer*, 2014. 7

[DZ13] K. Desnos and J. Zhang. PREESM project - stereo matching, December 2013. svn://svn.code.sf.net/p/preesm/code/trunk/tests/stereo. 110

[EAN13] S. El Assad and H. Noura. Generator of chaotic sequences and corresponding generating system, February 6 2013. http://www.google.com/patents/EP2553567A1?cl=en. 80

[EJ03] J. Eker and J. Janneck. CAL Language Report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003. 14

[Ele13] Electronic Systems Group - TU Eindhoven. SDF For Free (SDF3), March 2013. http://www.es.ele.tue.nl/sdf3/. 42, 75, 76, 80, 86, 89, 152, 165

[Emb13] Embedded Vision Alliance. Embedded vision alliance website, December 2013. http://www.embedded-vision.com. 107

[EMD09] W. Ecker, W. Müller, and R. Dömer. *Hardware-dependent Software*. Springer, 2009. 5

[Fab79] J. Fabri. *Automatic storage optimization*. Courant Institute of Mathematical Sciences, New York University, 1979. 49, 63, 64, 89, 154

[FGG+13] J. Forget, C. Gensoul, M. Guesdon, C. Lavarenne, C. Macabiau, Y. Sorel, and C. Stentzel. *SynDEx v7 User Manual*. INRIA Paris-Rocquencourt, December 2013. http://www.syndex.org/v7/manual/manual.pdf. 21, 89

[FGP12] P. Fradet, A. Girault, and P. Poplavko. SPDF: A schedulable parametric data-flow MoC. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 769–774. IEEE, 2012. 31, 32, 138

[Fly72] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. 37

[FWM07] S. Fischaber, R. Woods, and J. McAllister. SoC memory hierarchy derivation from dataflow graphs. In *Signal Processing Systems, 2007 IEEE Workshop on*, pages 469–474, 2007. 89

[GC04] Y. Gong and C.F.N. Cowan. A novel variable tap-length algorithm for linear adaptive filters. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 2, pages ii–825–8 vol.2, May 2004. 134

[GG21] F.B. Gilbreth and L.M. Gilbreth. Process charts: First steps in finding the one best way to do work. In *Annual Meeting of the American Society of Mechanical Engineers*, 1921. 11

[GG11] Viliam Geffert and Jozef Gajdoš. In-place sorting. In I. Černá, T. Gyimóthy, J. Hromkovič, K. Jefferey, R. Královič, M. Vukolić, and S. Wolf, editors, *SOFSEM 2011: Theory and Practice of Computer Science*, volume 6543 of *Lecture Notes in Computer Science*, pages 248–259. Springer Berlin Heidelberg, 2011. 87

[GGS+06] A.H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, and M. Theelen, B.and Mousavi. Throughput analysis of synchronous data flow graphs. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 25–36. IEEE, 2006. 19

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999. 14, 42, 152

[Gol94] S. Golson. State machine design techniques for Verilog and VHDL. *Synopsys Journal of High-Level Design*, 9:1–48, 1994. 12, 149

[GS03] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *MEMOCODE*, 2003. 35, 42

[GSLD11] T. Goubier, R. Sirdey, S. Louise, and V. David. SigmaC: A programming model and language for embedded manycores. In Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou, editors, *Algorithms and Architectures for Parallel Processing*, volume 7016 of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin Heidelberg, 2011. 14

[HDN+12] J. Heulot, K. Desnos, J.-F. Nezan, M. Pelcat, M. Raulet, H. Yviquel, P.-L. Lagalaye, and J.-C. Le Lann. An experimental toolchain based on high-level dataflow models of computation for heterogeneous MPSoC. *Design and Architectures for Signal and Image Processing (DASIP)*, 2012. 6, 22, 42, 45, 46, 153

[Hig93] N. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993. 135, 136

[HKB05] C.-J. Hsu, M.-Y. Ko, and S.S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems (SCOPES'05)*, pages 37–49. ACM, 2005. 86, 89

[HRM08] J. Hallyday, A. Rodriguez, and C. Maé. Etreintes fatales. *Album "Ça ne finira jamais" - Warner*, October 2008. 150

[IET14a] IETR. Graphiti GitHub project page, June 2014. https://github.com/preesm/graphiti. 45

[IET14b] IETR. PREESM GitHub project page, June 2014. https://github.com/preesm/preesm. 42, 43, 144

[Int13] Intel. i7-3610QM processor product page, December 2013. 37, 111, 114, 158

[JHM04] W.M. Johnston, J.R. Hanna, and R.J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004. 16

[Joh73] D.S Johnson. *Near-optimal bin packing algorithms.* PhD thesis, Massachusetts Institute of Technology, 1973. 49, 74, 78

[Joh97] G.W. Johnson. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control.* McGraw-Hill School Education Group, 2nd edition, 1997. 11

[Kah74] G. 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, 12, 13, 14, 148

[Kal14] Kalray. MPPA256 manycore processor product page, 2014. http://www.kalray.eu/products/mppa-manycore/mppa-256/. 6, 37, 43, 144, 145, 149

[Kil64] J.S. Kilby. Miniaturized electronic circuits, June 23 1964. US Patent 3,138,743. 4

[KLC10] D. Kim, V.W. Lee, and Y.-K. Chen. Image processing on multicore x86 architectures. *Signal Processing Magazine, IEEE*, 27(2):97–107, March 2010. 114

[KLV61] J.L. Kelly, C. Lochbaum, and V.A. Vyssotsky. A block diagram compiler. *Bell System Technical Journal*, 40(3):669–676, 1961. 11

[KS02] J.T. Kim and D.R. Shin. New efficient clique partitioning algorithms for register-transfer synthesis of data paths. *Journal of the Korean Phys. Soc*, 40, 2002. 49, 75, 88

[KSB+12] H. Kee, C.C. Shen, S.S. Bhattacharyya, I. Wong, Y. Rao, and J. Kornerup. Mapping parameterized cyclo-static dataflow graphs onto configurable hardware. *Journal of Signal Processing Systems*, 2012. 28, 29, 126

[Kwo97] Y.-K. Kwok. *High-performance algorithms of compile-time scheduling of parallel processors*. PhD thesis, Hong Kong University of Science and Technology, 1997. 41, 45, 56, 153

[LH89] E.A. Lee and S. 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. 40

[LM87a] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, 100(1):24–35, 1987. 20

[LM87b] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987. 14, 18, 19, 21, 24, 79, 124, 125, 138, 139, 140, 141, 150

[LP95] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 1995. 13, 14, 15, 17, 18, 138

[Mag05] D.P. Magee. Matlab extensions for the development, testing and verification of real-time DSP software. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, pages 603–606, New York, NY, USA, 2005. ACM. 89

[Mat96] Inc MathWorks. *SIMULINK dynamic system simulation for MATLAB: modeling, simulation, implementation*. Number vol. 1 in SIMULINK Dynamic System Simulation for MATLAB: Modeling, Simulation, Implementation. MathWorks, 1996. 11

[MB00] P.K. Murthy and S.S. Bhattacharyya. Shared memory implementations of synchronous dataflow specifications. In *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, 2000. 49, 56, 154

[MB04] P.K. Murthy and S.S. Bhattacharyya. Buffer merging: a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Trans. Des. Autom. Electron. Syst.*, 9(2):212–237, April 2004. 51, 58, 89, 90

[MB10] P.K. Murthy and S.S. Bhattacharyya. *Memory management for synthesis of DSP software*. CRC Press, 2010. 50, 79

[MFK+11] A.R. Mamidala, D. Faraj, S. Kumar, D. Miller, M. Blocksome, T. Gooding, P. Heidelberger, and G. Dozsa. Optimizing MPI collectives using efficient intra-node communication techniques over the Blue Gene/P supercomputer. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 771–780, May 2011. 89

[MNMZ14] A. Mercat, J.-F. Nezan, D. Menard, and J. Zhang. Implementation of a stereo matching algorithm onto a manycore embedded system. In *International Symposium on Circuits and Systems 2014, Proceedings*, 2014. 47, 55

[Moo65] G.E. Moore. Cramming more components onto integrated circuits, 1965. 5, 37

[MPZ+03] E.N. Malamas, E.G.M. Petrakis, M. Zervakis, L. Petit, and J.-D. Legat. A survey on industrial vision systems, applications and tools. *Image and vision computing*, 21(2):171–188, 2003. 107

[Nie14] P. Niemeyer. BeanShell website, 2014. http://www.beanshell.org. 91

[NL04] S. Neuendorffer and E.A. Lee. Hierarchical reconfiguration of dataflow models. In *MEMOCODE*, 2004. 16, 17, 28, 30, 129, 131

[NSD05] H. Nikolov, T. Stefanov, and E. Deprettere. Modeling and FPGA implementation of applications using parameterized process networks with non-static parameters. In *FCCM Proceedings*, 2005. 12

[Ö01] Patric R. J. Östergård. A new algorithm for the maximum-weight clique problem. *Nordic J. of Computing*, 8(4):424–436, December 2001. 65, 68

[Ost95] J.S. Ostroff. Abstraction and composition of discrete real-time systems. *Proc. of CASE*, 95:370–380, 1995. 17, 125, 139, 151

[PAPN12] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan. *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*. Springer, 2012. 6, 12, 22, 29, 41, 42, 45, 47, 69, 72, 75, 79, 84, 86, 118, 123, 125, 137

[Par95] T.M. Parks. *Bounded scheduling of process networks*. PhD thesis, University of California, 1995. 50, 79, 117

[PBL95] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee. *A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs*. Electronics Research Laboratory, College of Engineering, University of California, 1995. 22, 79, 101

[PBPR09] J. Piat, S.S. Bhattacharyya, M. Pelcat, and M. Raulet. Multi-core code generation from interface based hierarchy. *DASIP 2009*, 2009. 27, 49

[PBR09] J. Piat, S.S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *SiPS Proceedings*, 2009. 16, 17, 24, 25, 26, 123, 125, 128, 132, 138, 151

[PDH+14] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi. Dataflow-based rapid prototyping for multicore DSP systems. Technical Report PREESM/2014-05TR01, IETR, INSA-Rennes, 2014. 42, 151

[Pia10] J. Piat. *Modélisation flux de données et optimisation pour architecture multi-cœurs de motifs répétitifs*. PhD thesis, INSA de Rennes, September 2010. 21, 25

[PMAN09] M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan. Scalable compile-time scheduler for multi-core architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1552–1555, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association. 41, 42, 45, 153

[PNP+09] M. Pelcat, J.-F. Nezan, J. Piat, J. Croizer, and S. Aridhi. A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems. In *Proceedings of DASIP conference*, 2009. 37, 43, 152

[PPL95] T.M. Parks, J.L. Pino, and E.A. Lee. 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. 23

[PSK+08] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S.S. Bhattacharyya. Functional DIF for rapid prototyping. In *Rapid System Prototyping, 2008. RSP'08. The 19th IEEE/IFIP International Symposium on*, pages 17–23. IEEE, 2008. 43

[QC93] R.W. Quong and S.-C. Chen. Register allocation via weighted graph coloring. ECE Technical Reports. 232, Register Allocation via Weighted Graph Coloring, june 1993. http://docs.lib.purdue.edu/ecetr/232/. 88

[Rai92] S. Raina. Virtual shared memory: A survey of techniques and systems. *University of Bristol, Bristol, UK*, 1992. 47

[Rau06] M. Raulet. *Optimisations Mémoire dans la Méthodologie AAA pour Code Embarqué sur Architectures Parallèles*. PhD thesis, INSA Rennes, may 2006. 49, 64, 89

[Rei07] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007. 39

[RG13] P. Razaghi and A. Gerstlauer. Multi-core cache hierarchy modeling for host-compiled performance simulation. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pages 1–6, May 2013. 42

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004. 11

[RK08] M. Radetzki and R.S. Khaligh. Accuracy-adaptive simulation of transaction level models. In *Proceedings of the conference on Design, automation and test in Europe*, pages 788–791. ACM, 2008. 41

[Ros84] J.B. Rosser. Highlights of the history of the lambda-calculus. *Annals of the History of Computing*, 6(4):337–349, Oct 1984. 12

[Roy99] S. Roy. Stereo without epipolar lines: A maximum-flow formulation. *International Journal of Computer Vision*, 34(2-3):147–161, 1999. 108

[Sav98] J.E. Savage. *Models of computation*, volume 136. Addison-Wesley Reading, MA, 1998. 12, 13, 149

[SB09] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2009. 22, 45

[SGB06] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd annual Design Automation Conference*, pages 899–904. ACM, 2006. 50, 56, 79, 145, 158

[SGTB11] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, 2011. 16, 18, 30, 33, 141

[SK01] R. Szymanek and K. Kuchcinski. A constructive algorithm for memory-aware task assignment and scheduling. In *CODES Proceedings*, 2001. 49

[SL90] G.C. Sih and E.A. Lee. Scheduling to account for interprocessor communication within interconnection-constrained processor networks. In *ICPP (1)*, pages 9–16, 1990. 23

[Smi82] A.J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982. 47

[SS02] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1-3):7–42, 2002. 108, 157, 162, 163

[SWC+11] C.-C. Shen, L.-H. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S.S. Bhattacharyya. The DSPCAD lightweight dataflow environment: introduction to LIDE version 0.1. Technical report, U. of Maryland, 2011. 43, 89

[SWNP12] N. Siret, M. Wipliez, J.-F. Nezan, and F. Palumbo. Generation of efficient high-level hardware code from dataflow programs. In *Proceedings of Design, Automation and test in Europe (DATE)*, 2012. 43

[SZ00] R. Szeliski and R. Zabih. An experimental comparison of stereo algorithms. In *Vision Algorithms: Theory and Practice*, pages 1–19. Springer, 2000. 108

[TBG+13] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E.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. 17, 24, 25, 27, 138

[Tex08] Texas Instruments. TMS320C64x+ DSP image/video processing library, May 2008. 87

[Tex13] Texas Instruments. TMS320C6678 product page, December 2013. http://www.ti.com/product/tms320c6678. 6, 45, 112, 114, 145, 149, 153, 158

[Tex14] Texas Instruments. 66AK2H14 Multicore DSP+ARM KeyStone II System-on-Chip (SoC) Produce Page, 2014. http://www.ti.com/product/66ak2h14. 38

[TGB+06] B.D. Theelen, M. Geilen, T. Basten, J. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware dataflow model for combined long-run average and worst-case performance analysis. In *MEMOCODE*, 2006. 12, 14, 32, 33, 138, 141

[Til14] Tilera. TILE-Gx processor family page, 2014. http://www.tilera.com/products/processors/TILE-Gx_Family. 37, 144

[URND06] F. Urban, M. Raulet, J.-F. Nezan, and O. Déforges. Automatic DSP cache memory management and fast prototyping for multiprocessor image applications. In *14th European Signal Processing Conference, Eusipco*, 2006. 112, 113

[Wag07] D. Wagner. *Handheld augmented reality*. PhD thesis, Graz University of Technology, 2007. 107

[WL91] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6):30–44, 1991. 48

[YLJ+13] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet. Orcc: Multimedia development made easy. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 863–866, New York, NY, USA, 2013. ACM. 43

[YM08] K. Yamaguchi and S. Masuda. A new exact algorithm for the maximum weight clique problem. In *23rd International Conference on Circuit/Systems, Computers and Communications (ITC-CSCC'08)*, 2008. 65, 68

[ZDP+13] Z. Zhou, K. Desnos, M. Pelcat, J.-F. Nezan, W. Plishker, and S.S. Bhattacharyya. Scheduling of parallelized synchronous dataflow actors. In *System on Chip (SoC), 2013 International Symposium on*, pages 1–10, Oct 2013. 14, 15, 41, 46

[Zha13] J. Zhang. *Prototyping methodology of image processing applications on heterogeneous parallel systems.* PhD thesis, INSA de Rennes, December 2013. 6, 22, 42, 46

[ZNPC13] J. Zhang, J.-F. Nezan, M. Pelcat, and J.-G. Cousin. Real-time GPU-based local stereo matching method. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 209–214. ECSI, 2013. 108, 109, 110

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs

Nom Prénom de l'auteur : DESNOS KAROL

Membres du jury :
- Monsieur TAKALA Jarmo
- Monsieur ARIDHI Slaheddine
- Monsieur NEZAN Jean-François
- Monsieur PELCAT Maxime
- Monsieur SIRDEY Renaud
- Madame MUNIER-KORDON Alix
- Monsieur BHATTACHARYYA Shuvra

Président du jury : Jarmo TAKALA

Date de la soutenance : 26 Septembre 2014

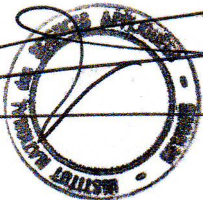
Reproduction de la these soutenue

Thèse pouvant être reproduite en l'état
Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 26 Septembre 2014

Signature du président de jury

Le Directeur,



M'hamed DRISSI

Résumé

Le développement d'applications de traitement du signal pour des architectures multi-cœurs embarquées est une tâche complexe qui nécessite la prise en compte de nombreuses contraintes. Parmi ces contraintes figurent les contraintes temps réel, les limitations énergétiques, ou encore la quantité limitée des ressources matérielles disponibles. Pour satisfaire ces contraintes, une connaissance précise des caractéristiques des applications à implémenter est nécessaire. La caractérisation des besoins en mémoire d'une application est primordiale car cette propriété a un impact important sur la qualité et les performances finales du système développé. En effet, les composants de mémoire d'un système embarqué peuvent occuper jusqu'à 80% de la surface totale de silicium et être responsable d'une majeure partie de la consommation énergétique. Malgré cela, les limitations mémoires restent une contrainte forte augmentant considérablement les temps de développements.

Les modèles de calcul de type flux de données sont couramment utilisés pour la spécification, l'analyse et l'optimisation d'applications de traitement du signal. La popularité de ces modèles est due à leur bonne analysabilité ainsi qu'à leur prédisposition à exprimer le parallélisme des applications. L'abstraction de toute notion de temps dans les diagrammes flux de données facilite l'exploitation du parallélisme offert par les architectures multi-cœurs hétérogènes.

Dans cette thèse, nous présentons une méthode complète pour l'étude des caractéristiques mémoires d'applications de traitement du signal modélisées par des diagrammes flux de données. La méthode proposée couvre la caractérisation théorique d'applications, indépendamment des architectures ciblées, jusqu'à l'allocation quasi-optimale de ces applications en mémoire partagée d'architectures multi-cœurs embarquées. L'implémentation de cette méthode au sein d'un outil de prototypage rapide permet son évaluation sur des applications récentes de vision par ordinateur, de télécommunication, et de multimédia. Certaines applications de traitement du signal au comportement très dynamique ne pouvant être modélisé par le modèle de calcul supporté par notre méthode, nous proposons un nouveau méta-modèle de type flux de données répondant à ce besoin. Ce nouveau méta-modèle permet la modélisation d'applications reconfigurables et modulaires tout en préservant la prédictibilité, la concision et la lisibilité des diagrammes de flux de données.

Abstract

The development of embedded Digital Signal Processing (DSP) applications for Multiprocessor Systems-on-Chips (MPSoCs) is a complex task requiring the consideration of many constraints including real-time requirements, power consumption restrictions, and limited hardware resources. To satisfy these constraints, it is critical to understand the general characteristics of a given application: its behavior and its requirements in terms of MPSoC resources. In particular, the memory requirements of an application strongly impact the quality and performance of an embedded system, as the silicon area occupied by the memory can be as large as 80% of a chip and may be responsible for a major part of its power consumption. Despite the large overhead, limited memory resources remain an important constraint that considerably increases the development time of embedded systems.

Dataflow Models of Computation (MoCs) are widely used for the specification, analysis, and optimization of DSP applications. The popularity of dataflow MoCs is due to their great analyzability and their natural expressivity of the parallelism of a DSP application. The abstraction of time in dataflow MoCs is particularly suitable for exploiting the parallelism offered by heterogeneous MPSoCs.

In this thesis, we propose a complete method to study the important aspect of memory characteristic of a DSP application modeled with a dataflow graph. The proposed method spans the theoretical, architecture-independent memory characterization to the quasi-optimal static memory allocation of an application on a real shared-memory MPSoC. The proposed method, implemented as part of a rapid prototyping framework, is extensively tested on a set of state-of-the-art applications from the computer-vision, the telecommunication, and the multimedia domains. Then, because the dataflow MoC used in our method is unable to model applications with a dynamic behavior, we introduce a new dataflow meta-model to address the important challenge of managing dynamics in DSP-oriented representations. The new reconfigurable and composable dataflow meta-model strengthens the predictability, the conciseness and the readability of application descriptions.

INSA RENNES

N° d'ordre : **14ISAR 22 / D14 - 22**

Institut National des Sciences Appliquées de Rennes

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

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