Etude et implantation d'algorithmes pour le placement et l'ordonnancement d'applications Dataflow

Hamza Deroui

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

INSA RENNES

THESE DE DOCTORAT DE

L'INSTITUT NATIONAL DES SCIENCES APPLIQUEES RENNES
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Signal, Image, Vision

Par

Hamza DEROUI

Étude et implantation d'algorithmes pour l'ordonnancement d'applications Dataflow.

Thèse présentée et soutenue à Rennes, le 06/12/2019
Unité de recherche : IETR
Thèse N° : 19ISAR 29 / D19 - 29

Rapporteurs avant soutenance :

Emmanuel JEANNOT
Directeur de Recherche à l'INRIA Bordeaux Sud-Ouest
Matthieu MOY
Maître de conférences, HDR, à l'Université Claude Bernard Lyon 1

Composition du Jury :

Jocelyn SEROT
Professeur d'Université à l'Institut Pascal de Clermont Ferrand / Président du jury
Emmanuel JEANNOT
Directeur de Recherche à l'INRIA Bordeaux / Rapporteur
Matthieu MOY
Maître de conférences, HDR à l'Université Claude Bernard Lyon 1 / Rapporteur
Jean-François NEZAN
Professeur d'Université à l'INSA Rennes / Directeur de thèse
Karol DESNOS
Maître de conférences à l'INSA Rennes / Examinateur
Alix MUNIER-KORDON
Professeur d'Université à Sorbonne Université, LIP6 / Examinatrice

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

INSA RENNES

Intitulé de la thèse :

Étude et implantation d'algorithmes pour l'ordonnancement d'applications Dataflow.

Hamza DEROUI

En partenariat avec :

Contents

Acknowledgements

I would first, like to thank my thesis advisors Pr. Jean-François Nezan and Pr.Karol Desnos at the National Institute of Applied Sciences (INSA) of Rennes and Pr. Alix Munier-Kordon at Sorbonne University whose help cannot be overestimated and whose the door of their offices was always open whenever I had a question about my research or writing. Thank you for steering me in the right direction and unwavering support.

I would also like to extend my deepest gratitude to the experts who were involved in the validation survey for this research project: Dr. Emmanuel Jeannot senior researcher scientist at INRIA Bordeaux Sud-Ouest, Pr. Matthieu Moy at Claude Bernard Lyon 1 University, and Pr. Jocelyn Serot at Institut Pascal. Without their invaluable insight, the validation survey could not have been successfully conducted.

To all members of the VAADER team of the IETR, thank you for making me feel part of the team since the beginning. Special thanks to all my officemates and coworkers during these three years: Alexandre Sanchez, Pierre-Loup Cabarat, Julien Heulot, Justine Bonnot, Maxime Pelcat, Jean-Gabriel Cousin, Meriem Outtas, and Naty Sidaty. Thanks also to Frédéric Garesché for his IT support and many thanks to Corinne Calo, Aurore Gouin and Jocelyne Trémier for their administrative support.

To my parents and sisters, thank you for surrounding me with love and prayers all along with my school and professional career I will be forever in your debt.

Finally, many thanks to my wife Kaoutar for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis and for accepting nothing less than excellence from me.

Thank you from the bottom of my heart

CHAPTER 1

Introduction

1.1 General Context

1.1.1 Embedded Systems

Today, almost every device we use is an embedded system. Remarkably adaptable and versatile, embedded systems can be found at homes, at offices, in industries and automation systems. From the basic ones, like washing machines, home security systems, digital cameras, vending machines, to the highly complex ones, like self-driving cars, missile guidance systems, airplanes and satellites. Embedded systems have been one of the most remarkable technological advances.

Formally, an embedded system is a combination of computer hardware and software, designed to perform a dedicated function. The hardware part includes all the electronic elements of the embedded system, like microprocessors, timers, memory, input/output interfaces, display and sensors. The software part, includes the computer program which is a sequence of instructions executed by the processing elements of the embedded system. The computer program can be simple calculation instructions, a firmware, or an embedded Operating System (OS). Embedded systems can be stand alone devices or integrated as part of a larger embedded system that serves a more general purpose.

Historically, embedded systems date back to the 1960s. The first embedded system was developed by Charles Stark Draper at MIT in 1961 for the Apollo mission. The embedded system was designed to reduce the size and weight of the Apollo guidance computer; the digital system which helped astronauts collect real-time flight data. The Apollo guidance computer was the first computer to use the newly developed monolithic integrated circuits. In fact, it was considered as the riskiest item in the Apollo project. The first mass-produced embedded system was the D-17B, developed by Autonetics in 1965; a computer used in the Minuteman I missile guidance system. Besides, space and military industries, the Volkswagen 1600 was the first vehicle to use an embedded system in 1968. The integrated microprocessor was used to regulate the air-fuel mixture in the electronic fuel injection system of the vehicle, boosting its performance and fuel efficiency. By the late 1960s and early 1970s, embedded systems have come down in price and there has been a dramatic rise in processing power and functionality. Some of the important milestones are the release of the first commercial microcontroller in 1974 by Texas Instruments, the release

of the wildly used x86 processor series by Intel in 1978, the release of the first embedded OS (VxWorks) by Wind River in 1987, followed by Microsoft's Windows Embedded CE in 1996. The embedded Linux system, which is used today in almost all the embedded devices, has appeared in the late 1990s.

Thanks to the continuous technological advancement in semiconductor device fabrication, both the size and the cost per unit of Systems-on-Chips (SoC) have been decreasing tremendously. Complementary Metal–Oxide–Semiconductor (CMOS) technology for example, has made possible to miniaturize transistors to the level where today mobile processors includes billions of transistors. As consequences, embedded systems have become more sophisticated. The best examples are smart devices, like smartphones, smartwatches, smart fitness bands, and smart credit cards. Furthermore, with the emerging technologies such as sensors, robotics, Internet of Things (IoT), Artificial Intelligence (AI), and 5G technology, the smart device concept has been enlarged to smart buildings, smart cities, smart grids, and smart factories. Today, cyber-physical systems are capable of cooperating and communicating with each other, even making decisions on their own. Without any doubt, embedded systems can be considered as the heart of the fourth industrial revolution.

1.1.2 Designing Embedded Systems

Embedded Systems Development Constraints

Designing an embedded system is a hard task. Indeed, besides the complexity of the software and the hardware, many constraints must be satisfied which are often contradictory. These constraints are classified in [Des14] following 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.

Dataflow Programming

Dataflow Model of Computation (MoC) have been introduced as a simple programming model which enables the developer to naturally express the parallelism of the software. A dataflow MoC consists on modeling the application with directed graph such that, each vertex is an independent computational module, and each edge represents an explicit communication channel between two vertices. Dataflow graphs have gained popularity due to their simplicity 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. Furthermore, the compatibility of dataflow MoC with legacy code enables the developers to reuse previously developed and optimized programs, which increase their productivity. Many dataflow MoC have been proposed in the literature since the introduction of the first one by Kahn in 1974 [Kah74]. Each new dataflow model extends the semantics and the expressivity of the previous one with a new properties. For example, the dynamic dataflow models extends the static ones with control task and data which enables the application to dynamically change its behavior.

1.2 Contributions

In this thesis, we study the development process of applications modeled with the Interface-Based SDF (IBSDF) model, in the context of rapid prototyping. The IBSDF MoC extends the well know Synchronous Dataflow (SDF) MoC with a hierarchy mechanism that enables the specification of the internal behavior of actors with a SDF subgraph instead of host code. The hierarchy mechanism of the IBSDF model is based on interfaces that insulate each subgraph from its upper graph in term of schedulability. Additionally to the interfaces, the IBSDF model defines execution rules to ease the analysis of the graph.

The IBSDF graph is often transformed to a non hierarchical graph during the development process of the application in order to verify and analyze its behavior. This transformation is called flattening the hierarchy. The flattening process often results in an exponential increase of the graph's size which makes the graph hard and even impossible to process with a reasonable time and memory. In fact, the flattening process has become the bottleneck of the development process of complex applications with the IBSDF model.

Our contributions aim to propose new techniques for the evaluation of some important metrics like the maximum throughput and the minimum latency. These metrics needs to be evaluated as early as possible by the developer. Indeed, very fast evaluation of this property is mandatory for real-time feedback to the developer during the application development, for the mapping and scheduling of the application on Multiprocessor System-on-Chip (MPSoC), and for the MPSoC Design Space Exploration (DSE) i.e. the research of the best hardware for a specific application. The main contributions of this thesis are:

1. A new method named Schedule-Replace (SR) for the evaluation of the maximum throughput of an IBSDF graph when it is executed in the hierarchical execution mode.

2. A new method named Evaluate-Schedule-Replace (ESR) for the evaluation of the maximum throughput of an IBSDF graph when it is executed in the relaxed execution mode.

3. A new method named Hierarchical-Symbolic-Execution (H-SE) for the evaluation of the minimum achievable latency of the IBSDF graph in the hierarchical execution modes.

4. A new method named Hierarchical-Critical-Path Method (H-CPM) for the evaluation of the minimum achievable latency of the IBSDF graph in the relaxed execution modes.

Until today, the maximum throughput and the minimum latency of an IBSDF graph were evaluated using a classical approach which consists of first flattening the hierarchy of the graphs and then evaluating its performance as if it was a large SDF graph. Since the flattening process results in an exponential growth of actors and edges number, the classical approach can be used only for small IBSDF graphs. For large IBSDF graphs, the classical method either fails to return a result or takes hours to evaluate the graphs. In contrast to the classical approach, the new techniques of this thesis are based on a modular approach which enables the evaluation of large IBSDF graphs without flattening their hierarchy. As consequences, the new techniques are capable of evaluating the maximum throughput and the minimum latency of large IBSDF graphs in few seconds, while the classical approach fails to return a result.

1.3 Outline

This thesis summarizes all the work done in this perspective plus our contributions, it is organized in six chapters as follows: The first chapter gives a short presentation of the scope of the thesis. Chapter 2 formally defines the concept of dataflow programming and gives an overview of the different categories of dataflow MoC. For each category, we present the semantics and characteristics of some of its dataflow models. Chapter 3 describes the development process of dataflow graphs in the context of rapid prototyping. Chapter 4 presents the new techniques for the throughput evaluation of the IBSDF model. Chapter 5 in turn, presents the new techniques for the latency evaluation of the IBSDF model. Finally, a summary conclusion of the work carried out, followed by a potential future research perspectives.

CHAPTER 2

Dataflow Models of Computation

2.1 Introduction

The complexity of Multiprocessor System-on-Chip (MPSoC) architectures is increasing exponentially to meet the rising computation power demand of signal processing applications. As consequences, programming modern MPSoC with the traditional thread-based programming languages have became more and more complex, due to the increasing number of Processing Elements (PEs) and their heterogeneity.

In this context, dataflow Models of Computation (MoC) is gaining popularity as the most suitable models for designing complex signal processing applications for MPSoC architectures. Dataflow MoC are diagram-based models, which consist on representing an application with a directed graph of tasks called actors. The edges of the graph represent the data exchange between the actors. This decomposition of the application into a set of interconnected actors offers the developer a natural way to express the parallelism and the data dependencies between the actors.

Furthermore, dataflow MoC can be used to specify a wide range of signal processing applications such as video decoding, telecommunication, and computer vision applications. The expressive power and the diversity of dataflow MoC; e.g. static, dynamic, hierarchical and parametric models; combined with powerful dataflow compilers gives the developer the best tools to easily design complex applications and fully exploit the computation power and the specifications of modern MPSoC architectures.

In this chapter, we give an overview of the different categories of datafllow MoC. We start by providing formal semantics for dataflow MoC by presenting the Kahn Process Network (KPN) and the Dataflow Process Network (DPN) models in section 2. In the same section, we show how the different types of parallelism are expressed in dataflow models. Then, we discuss the properties of the dataflow MoC which can be used to compare the different models and categories. In section 3, we present the static dataflow models which are the most studied and used in the industry. This category models applications in which data values have no impact on the system's behavior. Static dataflow models includes Synchronous Dataflow (SDF), Homogeneous SDF (HSDF), Single-Rate SDF (srSDF) and Cyclo-Static Dataflow (CSDF) models. In contrast, section 4 presents the dynamic dataflow MoC which are capable of adapting their behavior according to the data values. This category includes Boolean DataFlow (BDF) and Scenario-Aware

Dataflow (SADF) models. Next, we introduce the category of hierarchical dataflow MoC which extends the semantics of basic dataflow models with a hierarchy mechanism. Section 5 focuses on the static hierarchical models, the Hierarchical SDF (Hierarchical SDF) and the Interface-Based SDF (IBSDF) models. While section 6 focuses on the parametric hierarchical models, the Parameterized SDF (PSDF) and Parameterized and Interfaced SDF (πSDF) models. Finally, we give examples of real signal processing applications modeled with IBSDF graphs in section 7.

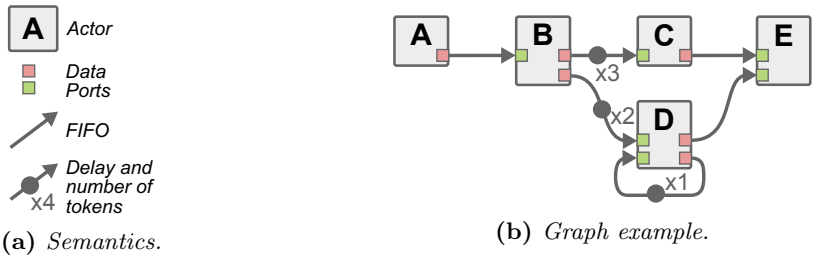
In this thesis, we are interested mainly in studying the behavior of the SDF model and its hierarchical extension, the IBSDF model.

2.2 Dataflow Programming

2.2.1 Semantics

Dataflow Programming is widely used for specifying the functionality of embedded systems. The first model to be introduced in the the context of parallel computation was the Computation graphs developed by Karp and Miller in the 60's [KM66]. In 1974, Kahn introduced the Kahn Process Network (KPN) [Kah74] as a parallel programming model that is *Turing complete*, meaning that the model can perform any computation described by an algorithm. Formally, the KPN MoC decomposes an application into tasks interconnected by directed First-In First-Out (Fifo) queues, forming a network of concurrent tasks. Each Fifo queue has an infinite memory and connects only one task to another, creating a data-dependency between the two tasks. Each task consumes (resp. produces) a number of data-tokens on its input Fifo queues (resp. output Fifo queues) at each execution. In 1995, Lee and Parks introduce the Dataflow Process Network (DPN) MoC [LP95] as a specialization of the KPN model. Formally, a DPN is a directed graph denoted $G = \langle A, F \rangle$, such that:

- A is the set of vertices of G. Each vertex $a \in A$ represents an indivisible computational task, also called an actor. 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.
  - $\mathbf{rate} : (R, P^{in}_{data} \cup P^{out}_{data}) \rightarrow \mathbb{N}$ 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.
- F is the set of edges of G. Each edge $e \in F$ represents an unbounded Fifo queue used to connect and transmit data tokens from an actor to another. Each Fifo queue $f \in F$ is defined as a tuple $\mathbf{f} = \langle \mathbf{prod}, \mathbf{cons}, \mathbf{src}, \mathbf{snk}, \mathbf{delay} \rangle$ where:
  - $\mathbf{prod} : F \rightarrow A$ and $\mathbf{cons} : F \rightarrow A$ associate producer and consumer actors to a Fifo.
  - $\mathbf{src} : F \rightarrow P^{out}_{data}$ and $\mathbf{snk} : F \rightarrow P^{in}_{data}$ associate source and sink ports to a Fifo.
  - $\mathbf{delay} : F \rightarrow \mathbb{N}$ corresponds to a number of data tokens present in the Fifo when the described application is initialized.



(a) *Semantics.* **(b)** *Graph example.*

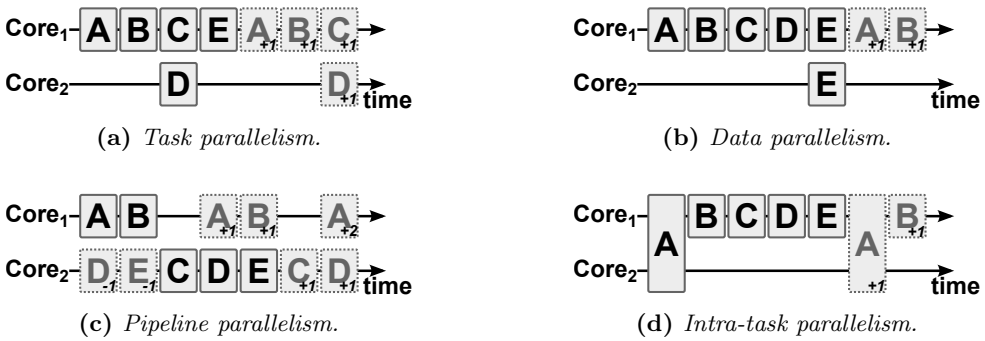
Figure 2.1 – *Dataflow Process Network (DPN) MoC.*

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 2.1b 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.

2.2.2 Expression of parallelism

Dataflow MoCs support four types of parallelism [ZDP+13], which are illustrated in figure 2.2 and defined as follows:

- **Task parallelism:** is when data-independent actors are executed in parallel. Two actors are called data-independent if there is no path in the graph between the two of them. For instance, figure 2.2a shows a parallel execution of actors C and D which are two data-independent actors in the DPN graph of figure 2.1b.
- **Data parallelism:** is when there is enough data tokens to execute the same actor many times simultaneously. Figure 2.2b shows an example of a case where actor E has enough data tokens to be executed twice simultaneously.
- **Pipeline parallelism:** is when several executions of the same dataflow graph overlap when it is possible. For example, in figure 2.2c, the next execution of the graph (actors with dotted borders and $+1$) starts before the end of the current execution. Similarly, the current graph execution overlaps the previous one (actors with -1).
- **Intra-task parallelism:** in the DPN model, the internal behavior of actors is described by a source code which can be sequential or parallel. For example, if the source code is written in a thread-based programming language then the actor may have an inner parallelism which enables its execution on several PE simultaneously. Figure 2.2d shows an example of Intra-task parallelism where actor A is a *parallel actor* executed on two PEs simultaneously.



(a) *Task parallelism.* **(b)** *Data parallelism.*

(c) *Pipeline parallelism.* **(d)** *Intra-task parallelism.*

Figure 2.2 – *Illustration of the four types of parallelism in dataflow MoC.*

2.3 Static Dataflow Models of Computation

Static dataflow models are the most studied category of dataflow-based MoC. Their popularity is due to their *decidability* property which enables the use of compile-time analysis to verify and evaluate key properties of signal processing applications. Indeed, static dataflow models are *non-reconfigurable* and *deterministic*, meaning that the production and consumption rates of each actor are known at design time and remain *static* at run-time [LP95]. However, static dataflow models do not express conditional states, data dependent iterations or recursion. For instance, the `if-then-else` statement has no equivalent in static dataflow models. Therefore, static dataflow MoC are not Turing complete models.

In the following, we present the semantics and characteristics of some of the most used static dataflow models in the industry.

2.3.1 Synchronous Dataflow (SDF) model

The SDF MoC is the simplest and the most commonly used static dataflow model. The SDF model was introduced in 1987 by Lee and Messerschmitt [LM87b]. Like, the DPN model, a SDF graph $G = \langle A, F \rangle$ decomposes an application into a set of actors A interconnected by a set of FIFO queue F to exchange data tokens. Each actor consumes (resp. produces) a fixed number of data-tokens from (resp. into) all its input FIFO (resp. output FIFO) at each execution. Thus, in contrast to the DPN model, each SDF actor has only one firing rule.

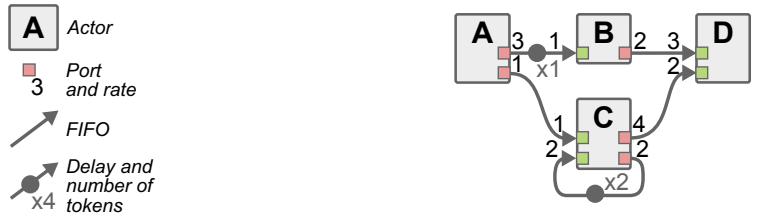
Formally, a Synchronous Dataflow (SDF) graph is a graph $G = \langle A, F \rangle$ respecting the Dataflow Process Network (DPN) MoC with the following restrictions:

- 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 $p \in P^{out}_{data}$ 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.

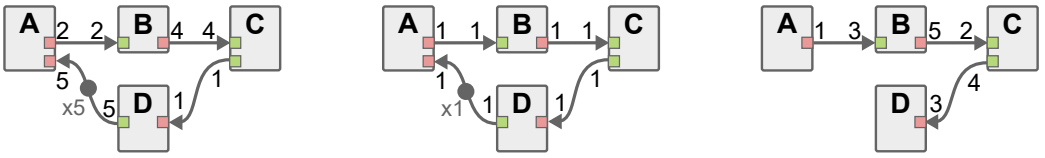
- **rate** : $A \times F \rightarrow \mathbb{N}$ is the production or consumption rate of actor $a \in A$ on FIFO $f \in F$. If a is both producer and consumer of f, then $rate(a, f)$ is the difference between the production and the consumption rates on f.

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.*



(a) *Single-Rate SDF (srSDF) model.* **(b)** *Homogeneous SDF (HSDF) model.* **(c)** *Directed Acyclic Graph (DAG) model.*

Figure 2.4 – *Specializations of the Synchronous Dataflow (SDF) model.*

2.3.2 Homogeneous SDF (HSDF), Single-Rate SDF (srSDF), and Directed Acyclic Graph (DAG) models

The HSDF, srSDF, and Directed Acyclic Graph (DAG) models are a specialization of the SDF MoC. Each model respects the semantics of the SDF MoC with some restrictions. In fact, each of the HSDF and the srSDF models add restrictions on the consumption and production rates of the SDF actors. The DAG model on the other hand, adds a restriction on the structure of the SDF graph. These models are illustrated in figure 2.4 and defined as follows:

- srSDF model: is a SDF graph such that for each FIFO queue, the production rate of its source actor is equal to the consumption rate of its target actor.
- HSDF model: is a SDF graph such that the consumption and production rates of the actors are all equal to 1. The HSDF model can also be seen as a specialization of the srSDF graph.
- DAG model: is a SDF graph which does not contain any cycle. The DAG model can also be as HSDF graph or a srSDF graph with no cycle.

The restrictions of the HSDF graph makes it is easy to analyze than other static dataflow graph. In fact, as we will see in the next chapter, several static dataflow models are converted to the HSDF and DAG models to be able to evaluate and analyze their performance accurately.

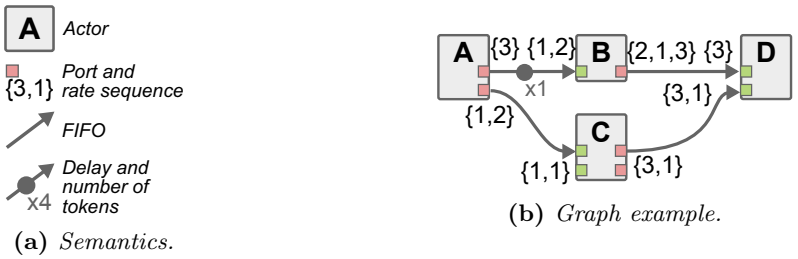
2.3.3 Cyclo-Static DataFlow (CSDF) model

The Cyclo-Static Dataflow (CSDF) model [BELP95] is an extension of the SDF model with consumption and production of actors decomposed into phases executed cyclically. Each actor has a fixed number of phases and each phase produces or consumes a specific number of data tokens.

Formally, 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 $p \in P_{data}^{in} \cup P_{data}^{out}$, 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 $p \in P_{data}^{in} \cup P_{data}^{out}$, the firing rule (i.e. the number of available tokens) and the production/consumption rates of p for the i^{th} firing of actor a is given by the $(i \bmod n + 1)^{th}$ element of $seq(p)$.

Figure 2.5 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 2.5 – *Cyclo-Static Dataflow (CSDF) MoC.*

2.4 Dynamic Dataflow Models

Static dataflow models like SDF and CSDF MoC are totally predictable and offer a great analyzability of their behavior at design time. However, they remain limited in terms of expressivity and thus cannot be used to model all applications.

In this context, many generalizations of the SDF MoC have been proposed over the years, which allow modeling dynamic and re-configurable applications. The main purpose of these new models is to preserve as much predictability and analyzability as possible while improving the expressivity of static dataflow MoC.

The Scenario-Aware Dataflow (SADF) and the Boolean DataFlow (BDF) models are two of the dynamic dataflow MoC which extend the semantics of static dataflow models by adding a control mechanism. By a *control-channel* or a control-actor, each of SADF and BDF models are capable of re-configuring and adapting the behavior of actors according to the data values. In the following, we present the semantics of each one of them.

2.4.1 Scenario Aware Dataflow (SADF)

The Scenario-Aware Dataflow (SADF) MoC [TGB+06] is a reconfigurable generalization of the SDF MoC, which is designed to enforce the *analyzability* of applications.

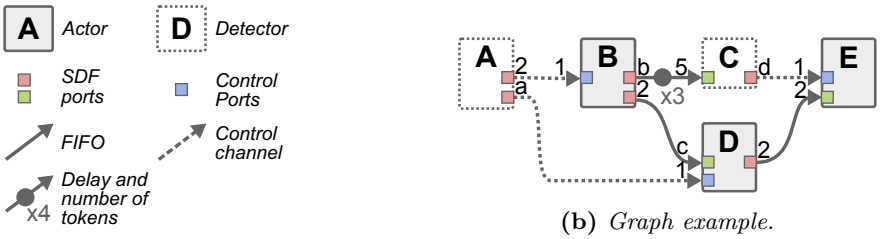
The SADF model extends the semantics of the SDF model with the concept of scenarios and by introducing a new type of actors and FIFO queues called respectively *detectors* and *control channels*. The detectors are special actors that control the behavior of other actors with the so-called *control tokens*. The control channels are used to transmit the control tokens from a detector to another actor. In the SADF model, each actor has a set of execution *scenarios* which define its behavior. Controlling an actor consists of choosing which execution scenario will be activated.

Formally, 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.

- $\mathbf{D} \subset A$ is the detectors set. 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



(a) *Semantics.* **(b)** *Graph example.*

Figure 2.6 – *Scenario-Aware Dataflow (SADF) MoC.*

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.

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

Figure 2.9 presents the graphical elements associated to the semantics of the SADF MoC and an example of SADF graph. The SADF graph of Figure 2.6b 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. Based on the definition of the SADF model given above, 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 a 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 a 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 Finite-State Machine (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 FSM.

2.4.2 Boolean Dataflow (BDF)

The BDF MoC [BL93] extends the semantics of the SDF MoC by introducing a new type of actors called switch and select. The BDF MoC uses the switch and select actors respectively as a demultiplexers and multiplexers to model conditional statements like the if-then-else patterns. Depending on the data values, the switch and select actors will forward the data tokens to different Fifo. Based on the result of a boolean value, the switch actor either forwards its input tokens to its first output port or its second output port, and thus acting like a demultiplexer logic cell. The select actor in turn, behaves as a multiplexer. The boolean value selects the input port whose tokens are then sent to the unique output port. Thus, the BDF model adds a control flow to the SDF MoC and makes the model Turing complete [BL93].

Figure 2.7 shows an example of a BDF graph which contains one switch and one select actors. In this BDF graph example, the switch actor (resp. select actor) will forward data

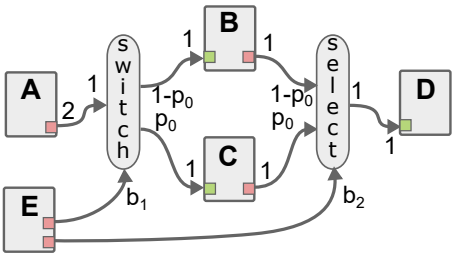


Figure 2.7 – *A Boolean DataFlow (BDF) graph example.*

tokens to one (resp. from one) of the actors B and C depending one the boolean data value sent by actor E.

2.5 Hierarchical Static Dataflow Models

With the increasing complexity of signal processing applications, the average number of actors of dataflow graphs has become very large. As consequences, designing and analyzing complex applications have become a difficult task for the designers, as well as for the development tools. In this context, new dataflow models with a modularity property have been proposed to ease the development of complex signal processing applications. The concept of modularity in dataflow modeling is based on decomposing the application into modules such that each module is a group of tasks. The module itself can be decomposed into sub-modules and thus creating a hierarchy of modules.

In this section we present two hierarchical dataflow MoC that extend the semantics of the SDF MoC with a hierarchy mechanism.

2.5.1 Hierarchical SDF

The Hierarchical SDF model [PL95] is the first hierarchical extension of the SDF MoC, which consists on associating an SDF graph to an actor as the description of its internal behavior. The associated actor is called a *hierarchical actor* which is described by an SDF subgraph. Similarly, an actor from the SDF subgraph can be associated in turn to another SDF subgraph. Thus, a Hierarchical SDF graph with multiple levels of hierarchy can be constructed. Figure 2.8 shows an example of a Hierarchical SDF graph composed of two hierarchical levels. The topgraph is composed of two regular actors A and B, and a hierarchical actor h. The hierarchical actor h is described by a SDF subgraph which is composed of two actors C and D.

The hierarchy mechanism of the Hierarchical SDF model is mainly used to ease the design phase of complex signal processing applications. In fact, during the actual execution of the application, each hierarchical actor is replaced by its subgraph which contains the actual actors to execute.

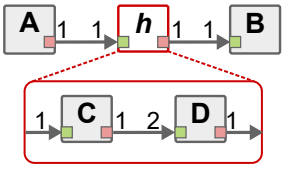
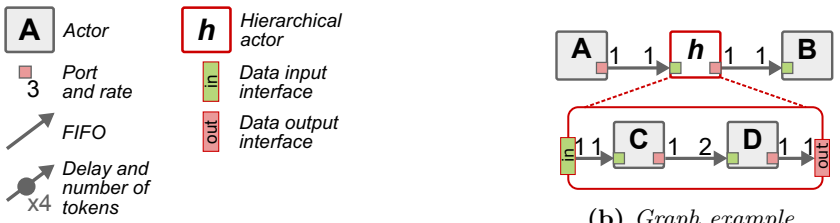


Figure 2.8 – *Hierarchical SDF graph*

Therefore, the Hierarchical SDF model is considered not compositional since all the actors are still data dependent even if they are organized into modules. Indeed, no subgraph in the Hierarchical SDF graph can be executed or analyzed independently from the whole graph.

2.5.2 Interface-based SDF (IBSDF)

Like the Hierarchical SDF model, the Interface-Based SDF (IBSDF) MoC [PBR09] offers the choice of specifying the internal behavior of actors with a host code, or with a dataflow subgraph. In contrast to the Hierarchical SDF model, the hierarchy of the IBSDF model is based on hierarchical interfaces. Practically, for each input Fifo queue of a hierarchical actor, an associated data input interface is added to the subgraph. Similarly, for each output Fifo queue of a hierarchical actor, an associated data output interface is added to the subgraph. Figure 2.9b shows the same Hierarchical SDF graph example of figure 2.8 modeled as an IBSDF graph where data interfaces are added to the subgraph. Figure 2.9a shows the graphical elements associated to the semantics of the IBSDF model.



(a) *Semantics.* **(b)** *Graph example.*

Figure 2.9 – *Interface-Based SDF (IBSDF) MoC.*

The main role of the data interfaces in an IBSDF graph is to transfer data tokens from the hierarchical actor to the subgraph and vice versa. Precisely, when the hierarchical actor is ready to execute, the data input interfaces transfer the data tokens consumed by the hierarchical actor from its input Fifo queues to its subgraph. Similarly, when the subgraph finishes processing all the data tokens received from its hierarchical actor, the data output interfaces transfer then the data tokens produced by the subgraph to its hierarchical actor. Once, the hierarchical actor receives data tokens from its subgraph, it produces them on its output Fifo queues in the topgraph, behaving like a regular actor.

Additionally, each of the data input and the data output interfaces has a special behavior which guarantees the compositionality of the IBSDF model:

- The input interfaces behave as circular buffers and reproduce the same data-tokens received from their hierarchical parent actor, as much as the subgraph needs to finish its execution

- The output interfaces receiving more data tokens than necessary from the subgraph will only transmit the number of data tokens defined by the hierarchical parent actor.

Thus, the hierarchical actor can behave exactly as a regular actor from the topgraph perspective. Indeed, the special behavior of the data interfaces guarantees that once a subgraph starts to execute, it will finishes and transfer data tokens to its hierarchical parent actor without requesting new data tokens from the topgraph. Hence, each subgraph in the IBSDF graph is data independent.

Therefore, unlike the naive hierarchy of the Hierarchical SDF model, the interface-based hierarchy of the IBSDF model introduces a compositionality that enables designing and analyzing each subgraph independently from the other subgraphs.

2.6 Hierarchical Parametric Dataflow Models

Like the dynamic dataflow models, the hierarchical parametric dataflow models add a dynamic property for the static hierarchical dataflow MoC. In this section we present two models, the PSDF MoC and the πSDF MoC which extends the semantics of the IBSDF MoC with the possibility to parameterize some attributes.

2.6.1 Parameterized SDF (PSDF)

In dataflow modeling, a meta-model is a model of model which consists on extending the semantic of an existing dataflow model with new elements that will enable new capabilities. The *Parameterized dataflow* for instance, is a meta-modeling framework introduced by Bhattacharya and Bhattacharyya in [BB01], which can be applied to many static dataflow models like the SDF and the CSDF models [BB01, DP18]. The *Parameterized dataflow* meta-model extends the semantics of the targeted MoC by adding dynamically reconfigurable hierarchical actors. Formally, 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.

Figure 2.10 shows an example of the PSDF model which is the resulting MoC when applying the *Parameterized dataflow* meta-model to the SDF MoC. The graphical semantics used in this figure are those 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.

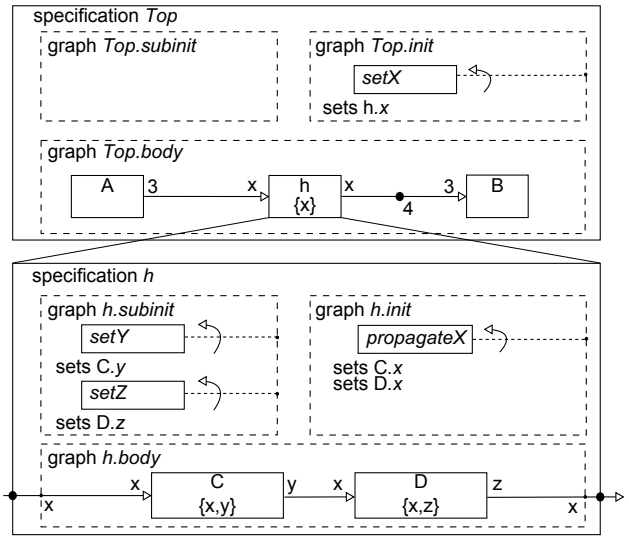


Figure 2.10 – *Example of Parameterized SDF (PSDF) graph*

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.

The execution of a PSDF graph follows the *runtime operational semantics* of the *Parameterized dataflow* meta-model defined by Bhattacharya and Bhattacharyya in [BB01]. The runtime operational semantics successive steps followed during the execution of a hierarchical actor whose internal behavior is specified with the three subgraphs, ϕ_i, ϕ_s, and ϕ_b subgraphs. A detailed description of this runtime operational semantics of the *Parameterized dataflow* meta-model can be found in [BB01]. Shortly, the execution of a PSDF graph starts by propagating the new values of the dynamic parameters through the hierarchy in order to set the consumption and production rates of the actors. The propagation of the parameters value is done by executing the init subgraph ϕ_i and the subinit subgraph ϕ_s of the hierarchical actors. Once the behavior of a hierarchical actor is defined, its body subgraph ϕ_b is executed.

As an example, the PSDF graph of figure 2.10 starts its execution by initializing and propagating the value of the parameter x of the hierarchical actor h through the hierarchy. First the actor *setX* of the init subgraph *Top.init* of the topgraph is executed to initialize the parameter x, then the actor *propagateX* of the init subgraph *h.init* of the hierarchical actor is executed to propagate the value of x. After that, the body subgraph *top.body* of the topgraph is executed. At each execution of the hierarchical actor h, the value of the parameters x and y of its subgraph are first initialized by executing the *h.subinit* subgraph. Once the consumption and production rates of the subactors C and D are defined, the body subgraph *h.body* of the hierarchical actor h is then executed. If we assume that the parameters x, y, and z are always set to 1 then we obtain the following execution: $setX, propagateX, A, 3 \times (setY, setZ, C, D), B$ where the notation $3 \times ()$ means that the content of the parenthesis is executed 3 times.

Although the *Parameterized dataflow* meta-model enables a dynamic hierarchy property, it makes the PSDF MoC less predictable than the original SDF MoC. Indeed, contrary

to the SDF model, the consumption and production rates of the PSDF actors are unknown at design time, as they depend on dynamically set parameter values.

2.6.2 Parameterized and interfaced SDF (πSDF)

Similarly to the *Parameterized dataflow* meta-model, the Parameterized and Interfaced dataflow Meta-Model (PiMM) [DPN+13] extends the semantics of existing dataflow models with a parameterized hierarchy feature. However, the PiMM meta-model inherits its hierarchy semantics from the IBSDF model [PBR09] which is based on hierarchical interfaces.

The πSDF MoC for instance, is the semantic extension of the SDF MoC using the PiMM meta-model. Hence, the πSDF model extends the SDF model with a hierarchy mechanism which is based on interfaces like the IBSDF model, such that the consumption and production rates as well as the initial data tokens of the FIFO queues are dynamically set like the PSDF model.

Based on the definition given in [Heu15], the πSDF model extends the semantics of the SDF model with the following elements:

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

In [Des14], Desnos compares the semantics of the πSDF model with the semantics of each of the SDF and the IBSDF models. Figure 2.11, taken from [Des14], summarizes this comparison and shows the graphical elements associated to the semantics of each model.

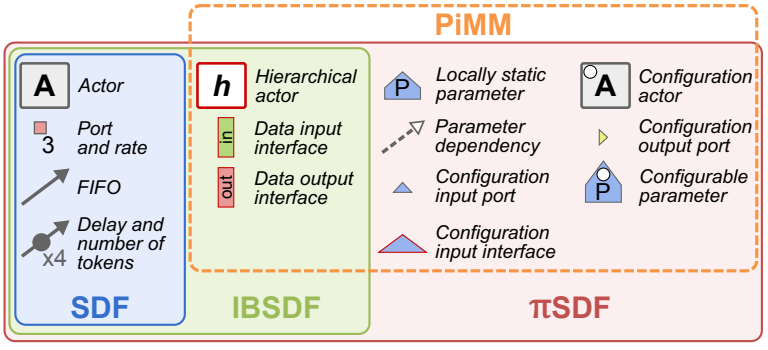


Figure 2.11 – *PiMM semantics [Des14]*

Multiple examples of real applications modeled with πSDF graph can be found in[Des14, Has18]. Figure 2.12 shows an example of a πSDF graph implementing an image filtering application. In this application, the *Read* and *Send* actor are respectively used for reading 1 pixel and sending a package of 3 pixels in a network. The purpose of the hierarchical actor *Filter* is to apply a filter on a $2D$ image for which the size is fixed by the parameter *size*. At each execution of the subgraph, actor *SetN* triggers a reconfiguration of the consumption and production rates of actor *Kernel* by assigning a new value to the parameter N. Reconfigurations enable a dynamic variation of the number of parallel execution of the *Kernel* actor.

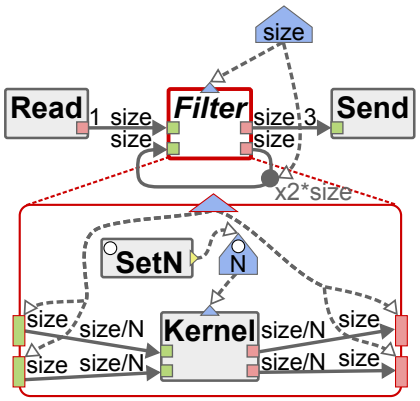


Figure 2.12 – *An example of the πSDF model of a image filter application [Des14]*

Compared to the *Parameterized dataflow* meta-model, the PiMM meta-model enables modeling complex applications with a concise model where parameters value is set via the configuration actors and propagated automatically using the configuration channels. In terms of the behavior analysis, the πSDF model is more predictable than the PSDF model thanks to its interface-based hierarchy inherited from the IBSDF MoC.

2.7 Conclusion

In this chapter we have presented several dataflow models. The static dataflow models like the SDF model are fully predictable which enables precise analysis. However, static dataflow models are semantically limited compared to dynamic dataflow models which extend the expressivity of existing model with dynamic parameters. Dynamic dataflow models like the SADF and the BDF models enable the application to adapt its behavior to the data values.

With the increasing complexity of signal processing applications, designing and analyzing large dataflow graphs became a hard tasks for the designer as well for the development tools. In this context, the hierarchical dataflow models have been introduced to add a compositionality to the existing dataflow models. Among the two presented hierarchical dataflow models, the IBSDF model combines the statics behavior of the SDF model and the compositionality of a hierarchical dataflow model. In fact, the IBSDF model enables the decomposition of the application into several modules organized in an interface-based hierarchy such that the behavior of each module can be specified and analyzed separately from the graph.

The parametric dataflow model extends the hierarchical dataflow models with dynamic parameters. The πSDF model for example extends the SDF MoC with a parametrized hierarchy mechanism. Thus the πSDF benefits from the modularity advantages of the interface-based hierarchy and from the dynamic adaptation of actors behavior through parameters.

In this thesis we are interested specially on the IBSDF model which we consider to be the most suitable model for the development of complex signal processing application in the context of rapid prototyping.

CHAPTER 3

Development Process of Dataflow Graphs

3.1 Introduction

In this chapter we present the different phases of the development process of dataflow-based applications in the context of rapid prototyping. We first start by defining what is rapid prototyping and what is the motivation behind it. Next, we present Parallel and Real-time Embedded Executives Scheduling Method (PREESM), a rapid prototyping framework. PREESM framework offers a simple user interface to easily develop, analyze and compile dataflow graphs while all the complexity of the process is abstracted in a simple workflow. In fact all the contributions of this thesis are integrated to PREESM as a performance analysis module.

After briefly presenting the compilation process of dataflow graphs, we detail some of the important phases like the verification, the transformation, the analysis, the mapping and scheduling phases. In section 3, we discuss the consistency property of static dataflow graphs and how to evaluate it for each of the SDF, the Hierarchical SDF, and the IBSDF graphs. In section 4, we define how to convert an SDF graph into each of an equivalent HSDF graph, an equivalent srSDF, and an equivalent DAG. In the same section we present how to flatten the hierarchy of an IBSDF graph into each of an equivalent flat srSDF graph and an equivalent flat DAG. These dataflow graph transformations are used during the compilation process in order to ease the analysis or to expose all the parallelism of the application. In section 5, we discuss the liveness property of dataflow graphs which is an important property to check for the validation of the design phase. Section 6 defines the different type of schedules used for the simulation of dataflow graphs executions. Two type of schedules are presented in this section, the As Soon As Possible (ASAP) schedule and the periodic schedule. In section 7, we discuss the mapping and scheduling phase, which is the most critical phase in the development process of signal processing applications. Section 8 presents Synchronous Parameterized Interfaced Dataflow Embedded Runtime (SPIDER), a run-time manager for parametric dataflow graphs like the πSDF graph. Lastly, we present in section 9 some of the tools used in this thesis for the numerical experiments and the benchmarks. Section 10 concludes this chapter.

3.2 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, a model of the targeted architecture, and a set of constraints for the deployment of the application on the architecture. 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

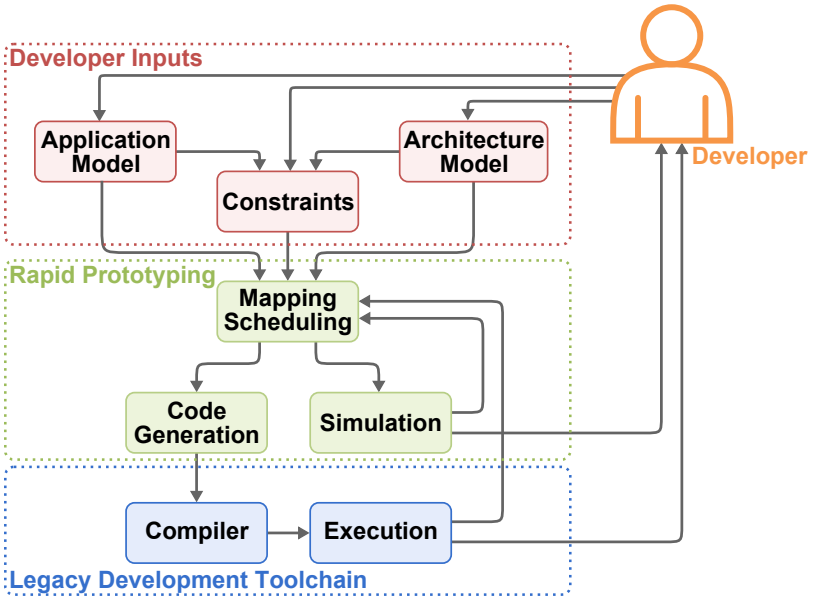


Figure 3.1 – *Overview of a rapid prototyping design flow. (source [Des14]).*

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.

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.

3.2.1 Preesm Rapid Prototyping Framework

The Parallel and Real-time Embedded Executives Scheduling Method (PREESM) is an Eclipse-based framework that provides dataflow-based methods to study and program embedded multicore platforms [PDH+14]. PREESM is an open-source framework developed by *VAADER* team of the *IETR Lab* at *INSA Rennes*. Many tutorials can be found on the Preesm website *https://preesm.org* for the easy initiation of C/C++ programmers to multi-core programming.

The PREESM framework focuses on providing high level rapid prototyping information on algorithm parallelism and latency. It also proposes detailed analyses on system memory requirements. Moreover, a platform adaptable C/C++ code generation is provided to transform the dataflow representation into a runnable code.

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

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

3.2.2 Preesm Typical workflow

A typical PREESM development workflow for an IBSDF graph consists of 6 main phases:

1. **Design phase**: The designer models the application with an IBSDF graph using the user interface of Preesm.

2. **Verification phase**: Verification of some necessary properties like the consistency and liveness.

3. **Conversion phase**: Flattening the hierarchy of the graph.

4. **Analysis phase**: Evaluate the performance of the graph.

5. **Mapping and scheduling phase**: Decides which actor to execute on which PE.

6. **Code generation**: Generates the actual source-code of the application which will be running on the targeted machine.

In the following, we present in detail some the important phases of the development process of dataflow graphs. We also simplifies the visual elements of the SDF and the IBSDF models, by ignoring the actor ports, and by representing the regular actors with circles and the hierarchical actors with squares.

3.3 Consistency Evaluation and Repetition Vector (RV)

Before analyzing the performance of a dataflow graph, a primarily step of the compilation consists on checking the deadlock freeness of the application; i.e. checking the consistency and the liveness of the graph. A dataflow graph is said to be consistent when its execution does not result in an accumulation of unconsumed data tokens. Meaning that a consistent graph can be executed indefinitely on MPSoC architectures with a sufficient bounded memory storage.

3.3.1 Consistency of a SDF graph

The consistency of a SDF graph $G = \langle A, F \rangle$ is checked by solving a system of linear equations defined by the matrix equation $\Gamma(G) * RV = 0$. The topology matrix $\Gamma(G)$ [LM87b], a $|F|$-by-$|A|$ matrix, represents the consumption and production rates of the actors on the FIFO queues. Each column of the topology matrix $\Gamma(G)$ is associated to an actor $a \in A$, and each row is associated to a FIFO queue $f \in F$. The matrix coefficients are defined as follows such that for each $a \in A$ and each $f \in F$:

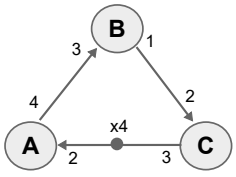
$$\Gamma_{a,f} = \begin{cases} prod(a,f) & \text{if } a \text{ is the source actor of } f \\ -cons(a,f) & \text{if } a \text{ is the target actor of } f \\ 0 & \text{otherwise} \end{cases}$$

The minimum solution vector of the system is called the Repetition Vector (RV). Each entry of the RV corresponds to the number of executions needed for an actor to restore the initial marking of the graph. If such vector exists, then the data-token production-consumption system is balanced and the graph is consistent.

Figure 3.2, the graph composed by the three actors A, B, and C represents a consistent SDF graph for which the repetition vector is $RV = [3, 4, 2]$.

Another method to verify the consistency of an SDF graph is by computing the rank of the topology matrix. Based on the theorem introduced by Lee in [LM87a], a consistent SDF graph is consistent if the *rank* of its topology matrix $\Gamma(G)$ is equal to the total number of its actors minus 1: $rank(\Gamma) = |A| - 1$.

In practice, the consistency of a SDF graph is checked by computing directly the RV of the graph using the polynomial algorithm of [BLM96a]. If the algorithm fails to compute the RV then the graph is not consistent.



(a) *SDF graph example.*

$$\begin{array}{c} \\ (A,B) \\ (B,C) \\ (C,A) \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 4 & -3 & 0 \\ 0 & 1 & -2 \\ -2 & 0 & 3 \end{bmatrix} \\ \Gamma \end{array} \times \begin{array}{c} \\ \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix} \\ RV \end{array} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

(b) *The Topology matrix.*

Figure 3.2 – *An SDF graph example and its corresponding topology matrix.*

3.3.2 Consistency of a Hierarchical SDF graph

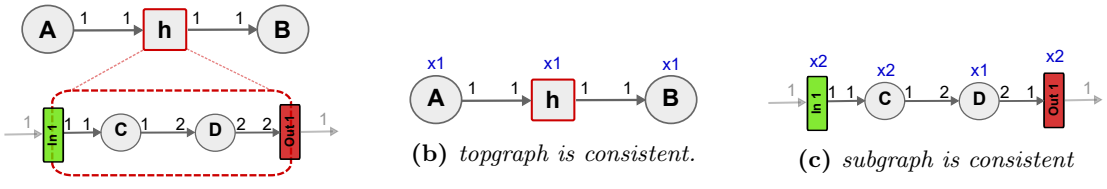
To evaluate the consistency of a Hierarchical SDF graph, the hierarchical actors must be replaced with their SDF subgraph description first. This process is called flattening the hierarchy of the graph, which results in an equivalent large non hierarchical SDF graph. For the Hierarchical SDF graph, this process is mandatory to be able to analyze the behavior of the graph. In fact, all the actors of a Hierarchical SDF graph are data dependent, even if each one of them belongs to a different level. Thus, in order to evaluate the consistency of the graph and to compute its RV, all the data dependencies of the actors must be revealed. Once the Hierarchical SDF graph is flattened, it is evaluated as an SDF graph using the classical methods described previously. Figure 3.3 shows a Hierarchical SDF graph example and its equivalent SDF graph after flattening its hierarchy. The equivalent SDF graph is consistent for which the RV is $RV = [2, 2, 1, 2]$. Thus, the whole Hierarchical SDF graph is consistent.

(a) *Hierarchical SDF graph example.* **(b)** *Equivalent flat graph.*

Figure 3.3 – *Flattening the hierarchical of a Hierarchical SDF graph to compute its RV.*

3.3.3 Consistency of an IBSDF graph

In contrast to the Hierarchical SDF model, the IBSDF MoC is a compositional model which enables the analysis of each subgraph independently. Indeed, with their special behavior, the input and output interfaces insulate each subgraph from the hierarchy in terms of data dependency. Thus, an IBSDF graph is consistent if each of its subgraphs is consistent, including the top-graph. Figure 3.4 shows the same hierarchical graph example of figure 3.3a but modeled with an IBSDF graph. Evaluating the consistency of the IBSDF graph example consists on checking the consistency of both the topgraph and the subgraph, but separately. Since both the graphs are consistent, the whole IBSDF graph example is consistent. The RV of the topgraph is $RV = [1, 1, 1]$ (see fig. 3.4b), and the RV of the subgraph is $RV = [2, 2, 1, 2]$ (see fig. 3.4c). More details on the consistency of the IBSDF model as well as a proof for the data dependency of the subgraphs can be found in [PBR09].



(a) *IBSDF graph example.* **(b)** *topgraph is consistent.* **(c)** *subgraph is consistent*

Figure 3.4 – *Evaluating the consistency of an IBSDF graph.*

In the IBSDF graph example, the behavior of the topgraph does not change regardless how many data tokens are required for actor D to execute. Indeed, the input interface duplicates the data tokens received from the hierarchical actor H as many times as needed

for the execution of actor D. Unlike the Hierarchical SDF graph example where actor A of the topgraph A must be executed twice to produce enough data tokens for actor D. Actor B in turn is executed twice to consume all the data tokens produced by actor D.

This example demonstrates the compositionality of the IBSDF model and it capability of insulating the different modules of the hierarchy in terms of data dependency. Furthermore, each module can be reused and integrated to other applications without changing the behavior of the module itself or the behavior of the new applications.

In this thesis, to simplify the computations and to preserve the semantics of the IBSDF model during the compilation process, we make the following modifications on the interface once the repetition factor of the IBSDF graph is computed:

1. For each input interface in, we set its production rate on its output Fifo queue f equal to $Prod(in, f) \times RV(in)$.
2. For each output interface out, we set its consumption rate on its input Fifo queue f equal to $Cons(out, f) \times RV(out)$.
3. For each input and output interface, we set its repetition factor equal to 1.

Figure 3.5 shows the new production and consumption rates of the interfaces of the IBSDF graph example after computing its RV.

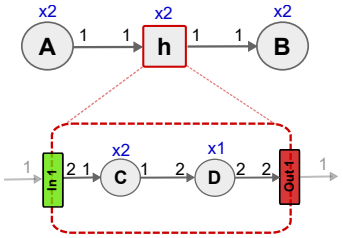


Figure 3.5 – *The IBSDF graph example after computing its RV*

3.4 Dataflow graphs Conversion

Dataflow graphs are constantly converted at many stages of the compilation into equivalent simplified and more expressive graph versions. The HSDF and the DAG conversions are the most used transformations for static dataflow models. The main purpose of this two conversions is to expose all the parallelism in a dataflow graph in order to precisely analyze, map, and schedule the application. In the following we present the different graph transformations for the SDF and the IBSDF models.

3.4.1 SDF graph conversions

SDF graph to HSDF graph

The HSDF graph conversion, also called the expansion, is one of the most used transformations for SDF graphs. It consists on duplicating each SDF actor according to its repetition factor. This transformation is also based on token ordering. If an SDF actor a produces $p = Prod(a, f)$ data token on a Fifo queue f then the same Fifo queue f will be duplicated p times for each instance of the actor a in the resulting HSDF graph. Thus, each actor in the equivalent HSDF graph has a repetition factor equals to 1, and a consumption and production rates equal to 1 on all its input and output Fifo queues.

Converting an SDF graph into an HSDF graph enables to explicitly express the data-parallelism as task-parallelism. Thus, the equivalent HSDF graph enables a precise analysis

of the SDF graph behavior. As an example, the equivalent HSDF graph is used to evaluate the exact maximum throughput of the application using a polynomial algorithm, while the problem complexity is still unknown for the SDF model.

An algorithm for the HSDF graph conversion is described in [BLM96a]. Figure 3.6b shows the equivalent HSDF graph of the SDF graph example of figure 3.6a. As the figures shows, each SDF actor was duplicated according to its repetition factor such that its consumption and production rate on each Fifo queue is 1.

SDF graph to srSDF graph

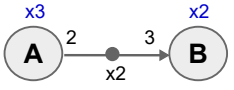
Similar to the previous transformation, converting an SDF graph to an equivalent srSDF graph consists on duplicating the actors according to their repetition factor. However, the srSDF graph conversion uses less Fifo queues to connect the duplicated actors. Indeed, if two or more Fifo queues are connecting the same two actors in the HSDF graph, then they are merged in one Fifo queue with a production and a consumption rates equal to the number of merged Fifo queues.

The equivalent srSDF graph is often used instead of the equivalent HSDF graph, since it express the same data dependencies between the actors but with less Fifo queues. Figure 3.6c shows the equivalent srSDF graph of the SDF graph example of figure 3.6a. As the figure shows, the equivalent srSDF graph exposes all the parallelism of the SDF graph using less Fifo queues than the equivalent HSDF graph of figure 3.6b.

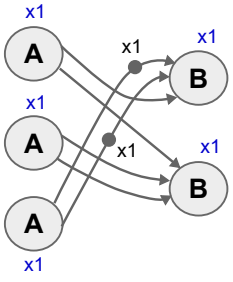
SDF graph to DAG

The equivalent DAG of an SDF graph is obtained by first converting the SDF graph to an equivalent srSDF graph. Next, deleting all the Fifo queues of the equivalent srSDF graph that contain initial data-tokens. If the SDF graph is consistent and live, then the resulting DAG should not contain any cycle. The equivalent DAG is used essentially for finding the Critical-Path (CP) to evaluate the latency, and for the mapping/scheduling problem i.e. the allocation of the necessary memory space and PEs for actors executions in a fixed order. Figure 3.6d shows the equivalent DAG of the SDF graph example of figure 3.6a.

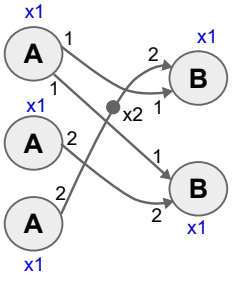
The space and the time complexity of the conversions HSDF, srSDF, and DAG for an SDF graph are relative to the RV of the graph. In fact, for a SDF graph $G = \langle A, F \rangle$, the total number of actors in each of the equivalent HSDF graph, srSDF graph, and



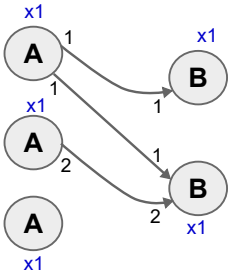
(a) *SDF graph example.*



(b) *The equivalent HSDF graph.*



(c) *The equivalent srSDF graph.*



(d) *The equivalent DAG.*

Figure 3.6 – *The illustration of some of the conversions of a SDF graph.*

DAG is $\sum_{a \in A} RV(a)$. As consequences, if the RV consist of large numbers then the time complexity of the conversion as well as the number of actors may grow exponentially.

SDF graph to Normalized SDF graph

A normalized SDF graph is a SDF graph such that each actor has the same production and consumption rates on all its input and output Fifo queues. The *normalized rate* of an actor a is noted Z_a. The normalization process is an essential transformation for the SDF graph in order to evaluate its liveness (schedulability) using a sufficient condition [MMK09] and to compute a periodic schedule if its exists. An algorithm for the normalization process of SDF graph is described in [Les17].

3.4.2 Flattening the hierarchy of an IBSDF graph

The flattening process is the transformation of a hierarchical dataflow graph into a flat non-hierarchical graph. This process is often used for the IBSDF graph during the development process in order to analyze its behavior using the classical methods of SDF graphs. The two most used conversions are the *flat srSDF graph* and the *flat DAG* conversion which guarantee a equivalent non hierarchical graph with the same behavior as the IBSDF graph.

In the following, we present how to flatten the hierarchy of an IBSDF graph into both an equivalent flat srSDF graph and an equivalent flat DAG.

Into an equivalent flat srSDF graph

The *equivalent flat srSDF graph* is constructed by repeating the following steps, starting from the top-graph of the hierarchy until the IBSDF graph is completely flattened:

- **Step 1:** Convert the top-graph to an equivalent srSDF graph.
- **Step 2:** Replace each instance of a hierarchical actor in the top-graph by the equivalent srSDF graph of its subgraph.
- **Step 3:** Go to *Step 1* and repeat the process until no hierarchical actor remains in the resulted flat graph.

Figure 3.7 shows an example of an IBSDF graph composed of two hierarchical levels. The hierarchical actor B of the topgraph ABC is described by the subgraph DEF. The equivalent flat srSDF graph of this IBSDF graph example is obtained by first, converting the topgraph to an equivalent srSDF graph as shown in Figure 3.8a. Next, by replacing $B1$, $B2$, and $B3$, the three instances of the hierarchical actor B, with the equivalent srSDF graph of the subgraph shown in figure 3.8b. The resulting flat srSDF graph contains 47 actors $(2A + 3 \times (1In + 2D + 6E + 4F + 1out) + 3C)$ and 170 edges.

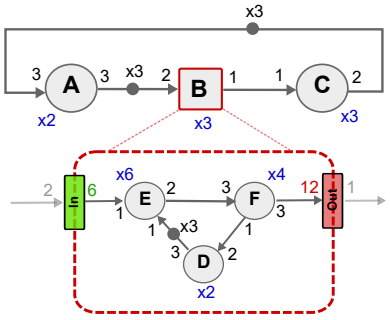
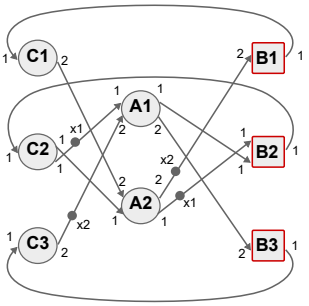
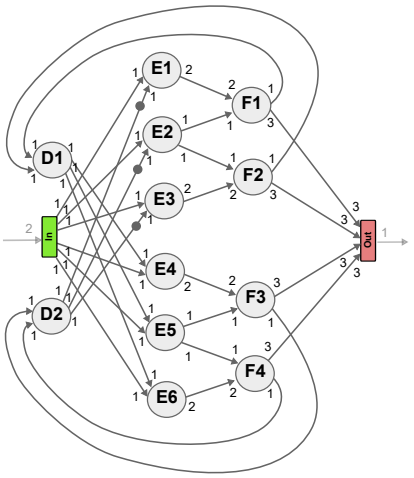


Figure 3.7 – *An IBSDF graph example composed of two hierarchical levels.*



(a) *The equivalent srSDF graph of the topgraph ABC.*



(b) *The equivalent srSDF graph of the subgraph DEF.*

Figure 3.8 – *The equivalent flat srSDF graph of the IBSDF graph example of figure 3.8 is obtained by replacing each instance of the hierarchical actor B with the srSDF graph version of its subgraph.*

Into an equivalent flat DAG

The equivalent flat DAG of an IBSDF graph is obtained simply by flattening first, the hierarchy of the IBSDF graph into an equivalent flat srSDF graph. Next, by deleting all the Fifo queues of the equivalent flat srSDF graph that contains data tokens. If the IBSDF graph is consistent and live, the resulting flat DAG should not contain a cycle.

The space and time complexity issue of the dataflow conversions is much more pronounced in the IBSDF graph conversions due to the flattening process. Indeed, each actor a in the IBSDF graph is duplicated $RV(a)$ times multiplied by the product of the repetition factor of all its hierarchical parent actors. Hence, the total number of actors in both the equivalent flat srSDF graph and the equivalent flat DAG grows exponentially. In fact, the flattening process is becoming the bottleneck in the development process of complex applications using the IBSDF model.

3.5 Liveness Evaluation

The consistency property ensures that a dataflow graph can be executed on an MPSoC architecture with a bounded memory storage. The liveness property on the other hand, ensures that the initial marking of a dataflow graph is sufficient to execute the graph indefinitely without a lack of data-tokens causing a deadlock. The liveness property must be verified specially when the dataflow graph has cycles in it. In the following we present the different methods to evaluate the liveness of the HSDF graph, the SDF graph, and the IBSDF graph.

3.5.1 Liveness evaluation of HSDF graph

An HSDF graph is live if each of its cycles contains ate least one data token. In graph theory, identifying all the elementary cycles in an graph is a exponential problem. Hence, the HSDF graph is instead converted to an equivalent DAG by deleting all the Fifo queue that contains data tokens. If the resulted equivalent DAG still contains a cycle then the

HSDF graph is not live. Thus, the HSDF graph is live only if its equivalent DAG does not contain any cycle.

3.5.2 Liveness evaluation of SDF graph

There are three methods to verify the liveness of an SDF graph. The first two methods are exact methods, however their time and space complexity depend on the size of the RV of the SDF graph. Thus, for large SDF graphs with a RV composed of large numbers, the liveness evaluation may takes an exponential time. The third method is based on a sufficient condition for the liveness of SDF graphs, which can be verified with a polynomial time algorithm. However, if the sufficient condition is not satisfied, the method cannot conclude if the SDF graph is live or not.

In the following we detail the three methods, starting by the exact ones.

HSDF conversion based method

The first method consists of first converting the SDF graph into an equivalent HSDF graph, then checking its liveness using the approach described previously. This simple method allows defining exactly whether an SDF graph is live or not. However, the size of the equivalent HSDF graph may grow exponentially since it depends on the RV of the SDF graph. Thus, the HSDF conversion-based method is considered as a not efficient due to its potentially exponential space-complexity.

Symbolic execution based method

The second method is based on a symbolic execution of the SDF graph. A symbolic execution consists on simulating the execution of the SDF graph without actually executing the source code of the actors. Indeed, from the dataflow graph perspective, the execution of an actor is the result of two consecutive actions: consume data token from the input Fifo queues and then produce data tokens on the output Fifo queues. This technique is commonly used for the analysis of the SDF graph behavior.

Hence, an SDF graph $G = \langle A, F \rangle$ is live if its symbolic execution returns a sequence S of actors executions in which each actor $a \in A$ is executed $RV(a)$ times. Moreover, an infinite execution of the graph is obtained simply by repeating infinitely the sequence S. In case where the SDF graph is deadlock (not live), the symbolic execution will reach a deadlock state, where no actor has enough data tokens to execute, and stops.

Like the previous method, the symbolic execution defines exactly weather an SDF graph is live or not. However, simulating the entire execution of the SDF graph may take an exponential-time. In fact, up today, there is no polynomial-time algorithm for the liveness evaluation of SDF graphs [MMK09, BLDMK14]. Hence, the next method consists on verifying if the initial marking of the SDF graph satisfies a sufficient condition of liveness using a polynomial algorithm. This approach was proved to be faster than a symbolic execution [BLDMK14].

Sufficient Condition (SC)

The sufficient condition of liveness was introduced in [MMK09] first for the SDF graph, and then extended for other static dataflow models. The sufficient condition uses the normalized version of the SDF graph, which is defined as follows:

A SDF graph is live if the initial marking of each cycle μ in its normalized version satisfies the

$$\sum_{f=(a,b)\in\mu} M_0^*(f) > \sum_{f=(a,b)\in\mu} (Cons(b,f) - gcd_f)$$

where gcd_f of a FIFO queue $f = (a, b)$ is the greatest common divisor of $Prod(a, f)$ and $Cons(b, f)$, and $M_0 * (f)$ is the number of useful data tokens present on f [MMK09]. The number of useful data tokens on a FIFO queue f is computed as follows :

$$M_0^*(f) = \lfloor \frac{M_0(f)}{gcd_f} \rfloor \times gcd_f$$

In practice, the sufficient condition of liveness is verified for an SDF graph using the polynomial algorithm described by the following steps:

- step 1: Normalize the SDF graph.
- step 2: For each FIFO queue $f = (a, b)$, set its values equals to $M_0^*(f) + gcd_f - Cons(b, f)$
- step 3: Use the algorithm to verify the existence of a negative cycle
- step 4: If no negative cycle exists then the SDF graph is live. Otherwise, the liveness of the graph cannot be concluded.

3.5.3 Liveness evaluation of IBSDF graph

Like the consistency evaluation, an IBSDF graph is live if all of its subgraph are live, including the topgraph of the hierarchy. Indeed, as the main advantage of the interface-based hierarchy is the ability to analyze each subgraph independently from the hierarchy. Therefore, the liveness of the IBSDF graph is evaluated by checking the liveness of each subgraph independently, using the previous methods of SDF graph.

In contrast to the Hierarchical SDF model, the hierarchy of the IBSDF graph does not need to be flattened in order to check the liveness of the graph. Furthermore, the compositionality of the IBSDF model is said to be deadlock free.

3.6 Simulating a Dataflow graph

Dataflow graphs are often simulated during the development process to analyze their behavior under certain conditions. For example, simulating the execution of a dataflow graph on an MPSoC architecture with unlimited resources is a common approach for the evaluation of certain performance metrics.

Simulating the execution of a dataflow graph consists on defining the start date of each execution of each actor of the dataflow graph. Depending on the type of the used schedule, the behavior of the dataflow graph may changes. However, regardless which schedule is used, the start dates of the actors executions must satisfy all the data dependencies between the actors as well as the execution duration of the actors themselves. To guarantee an accurate result, the dataflow community tends to use the Worst-Case Execution Time (WCET) possible for each actor, which is measured or defined by the designer. In the following we present two type of schedules, the ASAP schedule and periodic schedule.

3.6.1 ASAP Schedule

The ASAP schedule [GGS+06] is the most used schedule. It consists of executing actors as soon as there is enough data tokens on their input FIFO queues. This type of schedule, allows the application to be executed as fast as possible. Therefore, the ASAP schedule is used to compute some performance metrics like the maximum throughput and the minimum latency, which will be discussed in the next chapters.

One drawback of the ASAP schedule when simulating one iteration of a SDF graph $G = \langle A, F \rangle$, is that each execution $i \in [1, RV(a)]$ of each actor a has an independent start date $S(a, i)$. Hence, for one iteration of SDF graph where each actor a is executed $RV(a)$ times, the resulting ASAP schedule is defined by $\sum_{a \in A} RV(a)$ start dates. For large SDF graphs, this may become an issue.

3.6.2 Periodic Schedule

The periodic schedule was introduced in [BNHMMK12] for a subclass of Petri Net (PN) named Weighted Event Graphs (WEG) as an alternative to ASAP schedule. Since the semantic of both models SDF graph and WEG are equivalent, the periodic schedule was also used for scheduling SDF graphs.

Formally, a periodic schedule σ of an SDF graph $G = \langle A, F \rangle$ is a cyclic schedule [MK09], in which each actor $a \in A$ has a periodic execution defined by an initial start date $S^{\sigma}_{\langle a,0 \rangle}$ and a period w^{σ}_{a}. Hence, the execution start date of any instance i of an actor $a \in A$ can be calculated as follow:

$$\forall i \in \mathbb{N}^*, \quad S^{\sigma}_{\langle a,i \rangle} = S^{\sigma}_{\langle a,0 \rangle} + (i-1) \cdot \omega^{\sigma}_{a}$$

This property of the periodic schedule has a huge impact on the size of the schedule. In fact, the periodic schedule describes the execution of multiple iterations of a SDF graph $G = \langle A, F \rangle$ by only two informations per actor. However, the problem of the periodic schedule is to find a period w^{σ}_{a} for each actor a such that all the precedence constraints (data dependencies) are always satisfied. For some dataflow graphs, a periodic schedule may not exists. Therefore, it was proved in [MMK09] that a periodic schedule exists for an SDF graph if it satisfies the sufficient constraint of liveness. Furthermore, one of the main drawbacks of the periodic schedule is that the data parallelism of the actors is not exploited, which does not guarantee a fast execution of the graph like with the ASAP schedule.

3.7 Mapping and Scheduling Dataflow Graphs

The mapping and scheduling phase is the most critical phase in the development process of signal processing applications in which, many decisions are made based on the end-user demands alongside with the architecture specifications that will define the end behavior and performance of the designed application. In fact, the mapping process consists in choosing the different Processing Elements (PEs) that will execute each actor of the application, while the scheduling process consists on choosing the execution order of the actors on the selected PEs. Hence, the goal of the mapping and scheduling phase is to find a solution that satisfies budget constraints like number of cores and memory capacity, while optimizing performance criteria such as throughput and latency, or system efficiency criteria like energy consumption. Despite the fact that the mapping and scheduling problems are a NP-hard problems, the optimal solution differs from one case to another depending

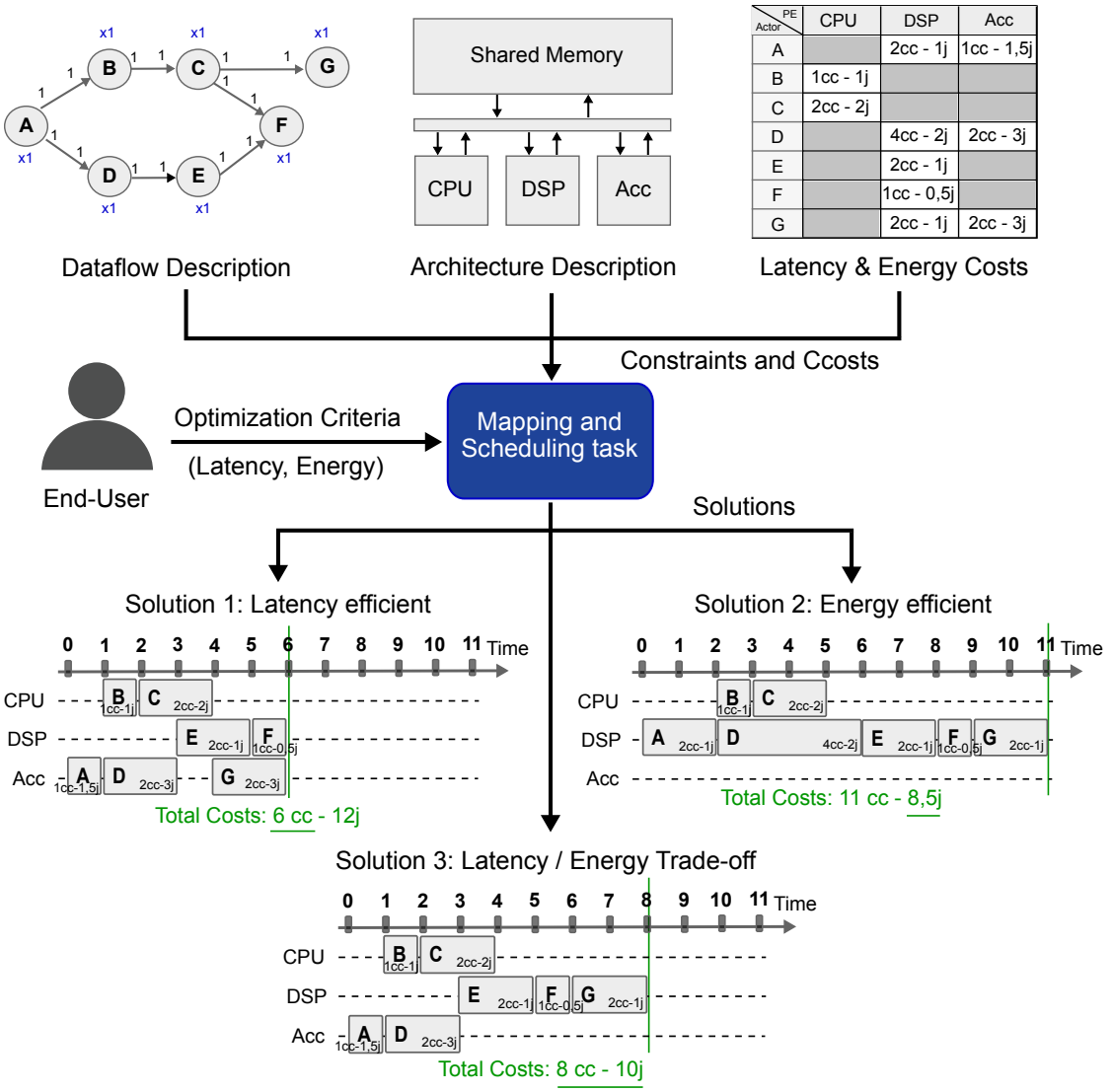


Figure 3.9 – *Mapping and scheduling a dataflow application on an MPSoC architecture.*

on the most important criteria to optimize. In most cases, the developer must optimize a trade-off between two or more criteria. Figure 3.9 illustrates an example of mapping and scheduling a dataflow graph of 7 actors on an MPSoC architecture composed of 3 PEs: a Central Processing Unit (CPU), a Digital Signal Processor (DSP), and one accelerator. As the figure shows, the mapping and scheduling process has four inputs: the dataflow graph description, the architecture description, the mapping costs, and the end-user demands. The dataflow graph description defines some of the application constraints like the precedence constraints between the actors, which must be satisfied during the scheduling process. Actors F and G for instance, cannot start to execute before actor C finishes its execution. The architecture description in turn, defines the budget constraints like the number of PEs, as well as the Network on Chip (NoC) topology and the communication cost between the PEs. In this example, the three processors are fully connected via a communication bus. The data transfer between the PEs is done through a shared memory for which the communication cost is neglected to simplify this example. At the other hand, the end-user defines the performance criteria which are the most critical to the application and must be optimized during the mapping and scheduling phase. In this example, the latency

and the energy consumption are the performance criteria to optimize. Finally, the costs of mapping each actor on each PE are used to evaluate the quality of the feasible solutions in order to choose the best one. The represented table in the figure example shows the possible mapping choices and their costs in terms of duration and energy consumption for each actor of the application. For example, executing (mapping) actor A on the DSP will take a total duration of 2 clock-cycles (cc) for which it will consume 1 joules (j). Another mapping choice for actor A is the accelerator on which it takes $1cc$ but consumes $0,5j$ more than on the DSP. Based on the internal behavior of the actors, some of the mapping choices are not possible which are represented in the table with shaded boxes. In this example, three solutions wich are represented with a *Gantt chart* are showned to illustrate the complexity of the optimization process of the mapping and scheduling phase. The first solution minimizes the duration of the application down to its minimum achievable multi-core latency $6cc$. However, it consumes a high amount of energy $12j$, which is 40% higher than the minimum energy consumption of the application $(8,5j)$. In contrast, the second solution decreases the energy consumption of the application down to its minimum value, but it results in an increase of the latency by 80% $(11cc)$ compared to the first solution. The third solution at the other hand, optimizes the trade-off between the latency and the energy consumption of the application. The proposed solution has a total duration of $8cc$ which is 30% higher than the minimal value, and a total energy consumption of $10j$ which is 17% higher than the minimal value. From both performance criteria perspective, the third solution may be considered as the best mapping and scheduling choice. However, in case of unpredictable changing constraints at run-time like decreasing the power consumption due to a high rise of the system temperature, the second solution may be considered as a fast alternative solution. Indeed, remapping and rescheduling a dataflow graph at run-time adds a considerable execution-time overhead which may not be acceptable for certain applications.

In this context, many taxonomies have been developed over the years [CK88, LH89, KA99, SSKH13], which classify the mapping and scheduling methodologies into two main categories: compile-time methodologies and run-time methodologies. In the following, we present each one of them.

3.7.1 Scheduling Methodologies

Run-time Methodologies

The run-time methodologies consist on mapping and scheduling applications at run-time. These methodologies are suitable for dynamic and parametric dataflow MoC [DPN+13] since the relation between the actors and their execution time are unknown at compile-time. Hence, a run-time management process is required to handle the execution of the application. Real-Time Operating System (RTOS) like Spider [HPD+14] can performs the complete mapping and scheduling process at run-time. However, the RTOS execution requires a dedicated PE which induces a run-time overhead in both forms execution-time and extra hardware. Therefore, typically greedy heuristic algorithms with low time-complexity and local view of the system are used to avoid excessive run-time overheads. Nevertheless, run-time methodologies yield a great flexibility to the system, giving it the possibility to adapt its behavior to dynamically changing constraints. In [LH89] and [SSKH13], run-time methodologies are respectively called fully-dynamic scheduling strategies and on-the-fly mappings.

Compile-time Methodologies

In the other hand, the compile-time methodologies which are called fully-static scheduling strategies in [LH89], are suitable for static dataflow MoC. In fact, the static behavior of the actors as well as their Worst-Case Execution Time (WCET) are completely known at compile-time, which offers a global view of the system. Hence, better mapping and scheduling decisions are made by compile-time methodologies. Moreover, more complex constraints can be handled at compile-time and more solutions can be explored to satisfy the end-user demands. Even if compile-time methodologies are completely intolerant of random behavior, the performance of the application remains predictable [LH89]. For this reason, [Des14] and [PAPN13] argue that as much mapping and scheduling decisions as possible should be shifted to compile-time in the context of rapid prototyping.

Hybrid Methodologies

A third category of emerging methodologies consists on combining the compile-time techniques with the run-time management. For example, the static-assignment and the self-timed scheduling strategies introduced in [LH89]. The static-assignment consists on mapping the actors at compile-time while scheduling their executions at run-time. The self-timed execution in turn, consists on mapping and scheduling the actors at compile-time while the actual timing of their firings is managed at run-time. Hence, actors are executed successively without any delays, which accelerates the execution of the application. Therefore, the self-timed execution is considered as the most suitable scheduling strategy for static dataflow models [SB09a]. A different approach introduced in [SSKH13] as an hybrid method, consists on exploring mapping alternatives at compile-time for different predictable fault scenarios and using them at run-time when needed. The mapping and scheduling process of figure 3.9 illustrates this approach. The second solution for example, represents a mapping alternative for the case where the accelerator becomes defective due to aging for instance. As the mapping and scheduling approaches become more diverse, many taxonomy refinements have been proposed to provide a common and rich terminology as well as classification mechanisms. For example, in [CK88], the refinement is based on both, the category of the algorithms (enumeration, graph-theory, or mathematical programming) and their optimality (optimal or sub-optimal). In [KA99], the mapping and scheduling approaches are organized following the problem's class: the Bounded Number of Processors (BNP) class, the Unlimited Number of Clusters (UNC) class, and the Arbitrary Processor Network (APN) class. In [SSKH13], the refinement is rather based on both the type of the architecture (homogeneous or heterogeneous) and the type of the run-time management (centralized or distributed).

3.7.2 Solving the mapping and scheduling problem

In the literature the mapping and scheduling problem is proved as a NP-Hard problem. List-scheduling based algorithm are the common used heuristics for mapping and scheduling dataflow graph. A list-scheduling based algorithm consists of scheduling the list of actors that are ready to execute based on a calculated priority. The difference between the multiple variants of the list-scheduling approach lies on the way of constructing the priority list and whether if the order of the actors in the list remains static or changes dynamically during the mapping and scheduling phase. List-scheduling algorithm are capable of mapping and scheduling up to 10000 actor [KA97].

3.8 SPIDER: a Run-time Manager for dataflow graphs

3.8.1 Overview of SPIDER

Parametric dataflow graphs like the πSDF graphs, are capable of changing their behavior at run-time by changing dynamically the values of their parameters. Hence, a run-time manager is necessary for managing their executions on the MPSoC architectures. In fact, the parametric dataflow graph must be completely (or partially) recompiled each time the values of its parameters change. Thus, the run-time manager must be able to quickly recompute both the RV and the mapping and scheduling solution, without causing an excessive run-time overhead.

In this context, Spider was introduced in [HPD+14] as a run-time manager specially for the execution of the πSDF graph. It was developed by the *VAADER* team at the *IETR* lab of *INSA Rennes*, and is available online at *https://github.com/preesm/spider*. Spider is designed to be a low-level run-time, enabling an efficient and dynamic reconfiguration of applications on MPSoC. Moreover, Spider is capable of performing the whole compilation process at run-time.

The run-time manager Spider takes as an input a πSDF graph developed in Preesm framework. If all the parameters of the πSDF graph are available then Spider executes the flattening process. If not, this transformation is performed when the reconfigurable parameters are set during the processing of the graph. Once the πSDF graph is processed, Spider performs then the mapping and scheduling and sends executions commands to the PEs of the MPSoC.

3.8.2 SPIDER Structure

Spider is composed of one Global RunTime (GRT) and multiple Local RunTime (LRT), organized in a master/slave approach. The GRT is the master of the run-time system which manages all the processing part of the πSDF graph like recomputing the RV, mapping and scheduling the πSDF actors. The GRT is usually placed on a general purpose PE. On the other hand, the LRT represents the slaves which are mapped onto the PEs of the MPSoC. The LRT are lightweight slaves used to locally manage the execution of actors on the PE on which they are mapped. Figure 3.10 shows the master/slave structure of the run-time manager Spider.

The master/slave structure of Spider enables both centralized and distributed management. Indeed, the GRT manages the scheduling and the memory parts while the LRT manages the local execution of the actor on the PEs. However, low latency communications are required to transmit the control commands between the GRT and the LRT. Therefore, Spider use two kinds of job queues to manage the communications between the GRT and the LRT, which are described as follows:

- Data channels that are used for the data path of data tokens exchange. Such data tokens are the memory buffers where the computation of actors is carried out.
- Control queues that are used for the sending of computation commands, reconfiguration parameters (partial or global) and the profiling of the dynamically scheduled re-configurable dataflow graph.

In the following, we describe the operations of Spider used during the execution of a parametric dataflow graph.

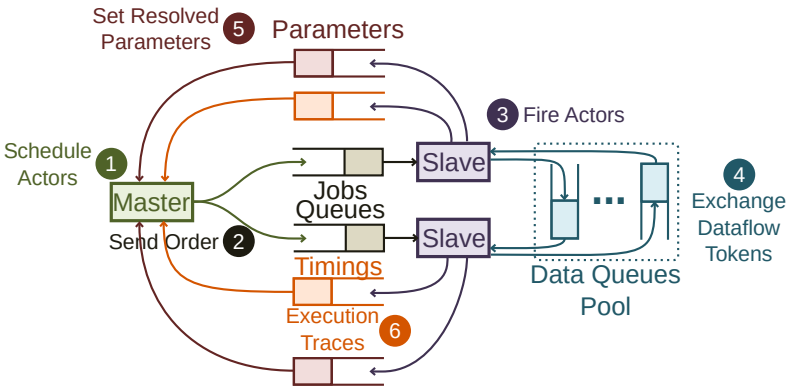


Figure 3.10 – *Spider run-time structure and operations.*

3.8.3 SPIDER Operations

The execution steps followed by Spider to run an actor are numbered in Figure 3.10. First (Step 1), the GRT schedules an actor on a PE of the architecture, and sends the execution order (Step 2) through the dedicated job queue of the LRT of this PE. A job is a message that embeds all the required data to execute one instance of an actor: a job ID, location of actor data and code, and identifiers of preceding actors in a graph execution. When an LRT starts an actor execution (Step 3), it waits for data tokens to be available in the input Fifo queues specified in the job message, among a pool of data Fifo. On actor completion, data tokens are written in output Fifo queues (Step 4), and the LRT sends new parameter values (Step 5) if any, and execution traces back to the GRT for reconfiguration, monitoring and debugging purposes (Step 6). Each LRT is associated with a job counter that stores the integer job ID of the last executed job. As the job IDs increases monotonically both with scheduling order and data dependencies between jobs, these job counters can be used for synchronization purposes between the different LRT, to check whether an LRT already executed a given job.

3.9 Tools

Among the external tools used in this thesis for the numerical experiments and the benchmarks, there are SDF^3 and Turbine tools which are presented in the following.

3.9.1 SDF3 tool

Synchronous Dataflow For Free (SDF^3) [SGB06] is an open source popular software written in C++. It provides a random dataflow generator and handles SDF and CSDF models. SDF^3 also includes extensive libraries to analyze and transform SDF and CSDF graphs. The library includes, among other things, tools to evaluate consistency, compute repetition vectors and compute all possible optimal trade-offs between the storage-space allocated to the Fifo queues of an SDF graph and the maximal throughput. SDF^3 handles sdf3 (xml) files format which is commonly used by the dataflow community.

3.9.2 Turbine tool

Turbine [BLDMK14] is an open source datflow graph generator developed by the *LIP6 lab* at Sorbonne University and available online at *https://github.com/bbodin/turbine*. It isimplemented in Python and uses the *NetworkX* package to handle the graph data structure.

It manipulates simple files (non-xml) in order to facilitate self-made instances, however the xml format of SDF[3] is also supported. Turbine generates live SDF, CSDF and Phased Computation Graph (PCG) graphs up to 10,000 actors in less than 30 seconds. Random dataflow graphs are generated in three steps. The first step generates a random graph with actors and Fifo queues. The second step computes consumption and production rates of the graph actors. The last step computes a live initial marking based on the sufficient condition of liveness.

3.10 Conclusion

In this chapter, we presented the different phases of the development process of signal processing applications modeled with SDF and IBSDF graphs. The verification phase for example, which consists on evaluating the consistency and the liveness of the graph to determine if the modeled application is deadlock free or not. The mapping and scheduling phase, a crucial phase in the development process, which try to find a solution that satisfies both the user demands and the MPSoC architecture constraints. During these phases, both the SDF and the IBSDF graphs are converted to more expressive dataflow models like the srSDF graph and the DAG. These transformations are necessary in order to reveal all the parallelism of the application, which will be used first for the evaluation of some key performance indicators and then for the mapping and scheduling phase.

However, converting a dataflow graph into an equivalent srSDF graph or an equivalent DAG results in an exponential increase of the number of actors and edges. This issue is more pronounced when flattening the hierarchy of the IBSDF graph into equivalent non hierarchical graphs, like the equivalent flat srSDF graph and the equivalent flat DAG. Such large graphs are hard to analyze, map and schedule. In fact, the flattening process has become the bottleneck in the development process of complex applications using the IBSDF model.

In the next chapters, we present new methods for the evaluation of key performance metrics for the IBSDF graph. In contrast to the classical methods which rely on the flattening process, the new methods evaluate rapidly the IBSDF graph without any conversion. In the context of rapid prototyping, a fast evaluation of the key performance indicators is mandatory for real-time feedback to the developer during the application development and for the mapping and scheduling of the application on MPSoC architecture.

CHAPTER 4

Throughput Evaluation of IBSDF graph

4.1 Introduction

For signal processing applications, the maximum throughput is one of the required properties to be evaluated as early as possible by the developer. The maximum throughput is the maximum amount of data tokens that the application can produce at each unit of time. The evaluation of this property is mandatory for the development process, especially when a real-time constraint must be satisfied. The value of the maximum throughput with other key properties enables the developer to validate the design phase and continue the development process of the application. If the evaluated throughput does not reach an acceptable value, the design phase is not validated. And so, the developer must reconfigure the graph until the throughput reaches the required value. Without an automatic process or guiding from the development tool, the developer may iterate many times on re-configuring the graph until the throughput value is validated. For this purpose, a very fast evaluation of the maximum throughput is essential for a real-time feedback to the developer during the application development, and for MPSoC Design Space Exploration (DSE) i.e. the research of the best hardware for a specific application.

In this chapter we are interested specifically in evaluating the maximum throughput of IBSDF graphs on MPSoC architectures with unlimited resources, meaning an unlimited number of PEs, unlimited memory capacity, and a high-performance NoC. The only criteria taken into account is the Worst-Case Execution Time (WCET) of actors, which is predefined or measured by running each actor on a specific PE. The evaluation of the maximum throughput in these perfect conditions enables the developer to measure the maximum potential of the modeled application, regardless of the target architecture. In general, cases where the target architecture is not defined yet, an early evaluation of this maximum throughput helps the developer to limit the search space and choose the adequate architecture available in the market. For critical applications when the high performance is the first priority, the evaluation of the maximum throughput allows to pre-define the specifications of a new architecture prototype.

The chapter is organized into seven sections including the introduction. In the following section, we present the state-of-the-art methods for the throughput evaluation problem of SDF graphs. In section 3 and 4, we present respectively the two execution modes of IBSDF graphs and how to evaluate their throughput using a classical approach based on

SDF methods. In section 5, we discuss the time and space complexity of the classical approach and how to reduce it by avoiding a flattening process. In the same section, we introduce two new methods, Schedule-Replace (SR) and Evaluate-Schedule-Replace (ESR) techniques which are our first major contribution in this thesis. Both introduced methods evaluate the throughput of IBSDF graphs without completely flattening their hierarchy. In section 6, we compare the performance of the new methods with the classical approach, on both real and synthetic set of IBSDF graphs. Section 7 concludes this chapter.

4.2 SDF State-Of-The-Art Methods

The throughput evaluation problem of SDF graphs is one of the basic problems in dataflow graphs performance analysis. Even though, its complexity is still unknown and is not well solved for certain graphs. Almost, all of the SDF methods presented in this section are based on either converting the SDF graph into an expressive equivalent one or simulating a schedule of the original SDF graph. The conversion-based methods are known as the simple approach to computing the exact maximum throughput of a SDF graph. However, their conversion process may result in an equivalent graph with exponential numbers of actors and edges, which makes the throughput computation a hard task. As an alternative to conversion-based methods, the simulation of an ASAP schedule of a SDF graph allows computing its exact maximum throughput without any conversion. Indeed this alternative approach has a lower space complexity than conversion-based methods but, simulating many iterations of a SDF graph may take too long to evaluate the throughput. As consequences, if the Repetition Vector (RV) of a SDF graph is too large, it is not possible to evaluate the throughput with the previous methods in a reasonable amount of time or memory. Therefore, computing an approximation of the throughput using another type of schedules like the periodic one is suitable for a fast evaluation of large SDF graphs.

In theory, the state-of-the-art methods are restricted to strongly-connected graphs. In practice, this does not represent a restriction, because each consistent SDF graph can be transformed into a strongly connected graph by modeling the capacity of its Fifo via backward edges [15]. Furthermore, some methods consider a restricted parallelism in which actors are not allowed to be executed more than once in parallel. This is specified directly in the SDF graph by adding a self-loop edge with one data token for each actor.

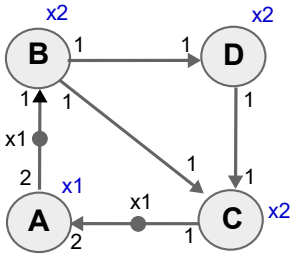


Figure 4.1 – *An example of a SDF graph composed of* 4 *actors: A, B, C, and D.*

In the following, we present each of the SDF state-of-the-art methods for the throughput evaluation under no resource constraint. And to illustrate the methods algorithm, we take the SDF graph of figure 4.1 as the main graph example for this section. The graph is composed of 4 actors, A, B, C, and D. One iteration of the graph equals to 1 execution of actor A and 2 executions of each of the remaining actors. To simplify the illustration of the different schedules, we define the duration of each actor as 1 unit of time.

4.2.1 HSDF based method

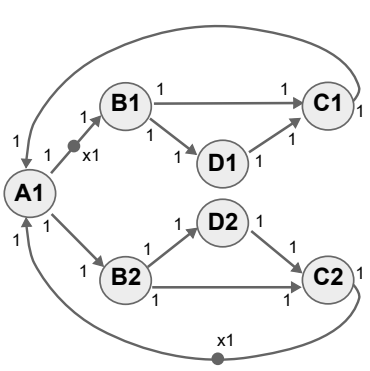
The HSDF based method is the first method in the literature to compute the maximum throughput of SDF graphs based on a conversion. As defined in [SB09b] the method consists of the following steps:

- **Step 1**: convert the SDF graph to its equivalent HSDF graph.
- **Step 2**: compute the Maximum Cost-to-Time Ratio (MCR) value of the equivalent HSDF graph obtained in step 1.
- **Step 3**: calculate the maximum throughput of the original SDF graph as 1 over the MCR value computed in Step 2.

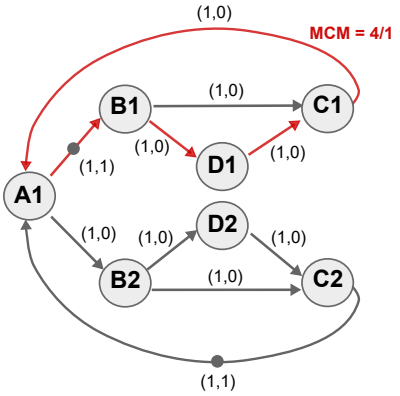
In this method, the SDF graph is transformed into an equivalent HSDF graph to show explicitly all the parallelism and all the data dependencies between the graph actors. Once the equivalent HSDF graph is obtained, the second step consists of finding the cycle $c \in C(G)$ that maximize the ratio: $r(c) = L(c)/D(c)$ where $L(c) = \sum_{a \in c} l(a)$ the total execution time of the cycle actors, and $D(c) = \sum_{e \in c} d(e)$ the total number of data tokens present on the cycle edges. This problem is called the Maximum Cost-to-Time Ratio (MCR) [DIG99], also referred to as Maximum Cycle Mean (MCM) problem. The MCR is a well-known problem in combinatorial optimization which can be solved by a polynomial-time algorithm [Das04]. The MCR value ($\max_{c \in C(G)} r(c)$) can be shown ([17],[20, Lemma 7.3] to be equal to the average time between two executions of any of the HSDF actors. Based on that, the final step consists of calculating the maximum throughput of the original SDF graph as:

$$Th_{\max}(G) = \frac{1}{MCR(G_H)}$$

This equation derives directly from the throughput definition and represents the exact maximum value that the throughput can not exceed, due to the data dependence between the actors of the cycle with the MCR value.



(a) *The equivalent HSDF graph of the SDF graph example of figure 4.1*



(b) *The equivalent bi-valued graph for the MCM computations*

Figure 4.2 – *Computing the throughput of the SDF graph example with the HSDF based method*

Figure 4.12 shows an illustration of computing the throughput of the SDF graph example of figure 4.1 using the HSDF-based method. In the first step, the SDF graph example is converted to a HSDF graph (fig.4.2a) in which each actor is duplicated according to the RV. Then, the second step computes the MCR value of the HSDF graph as shown in figure 4.2b. The red cycle composed of actors $A1$, $B1$, $D1$, and $C1$ represents the cycle

with the maximum ratio of a value equals to 4. Hence the maximum throughput of the SDF graph example is 1/4 data token per time unit.

The time complexity of both HSDF conversion and MCR problem are polynomial. However, this method is defined by the literature as inefficient. Indeed, converting a SDF graph to its equivalent HSDF graph may result in an exponential growth of both actors and edges number. As consequences, the exponential space complexity of the HSDF conversion makes the MCR problem hard to solve in a reasonable amount of time and memory.

4.2.2 Max-plus Algebra-based method

In [dGKBS12], the author introduced the Max-plus Algebra-based method, a second method based on a transformation to compute the exact maximum throughput of SDF graphs. As the first method, the Max-plus Algebra-based method computes the MCR of an equivalent graph to the original SDF graph. In contrast, the second method uses a Linear Constraint Graph (LCG) conversion instead of HSDF conversion. The LCG expresses the SDF graph as a linear time-invariant max-plus system with less number of actors and edges than the HSDF graph. From the numerical experiments of [dGKBS12], the Max-plus Algebra-based method was concluded to be efficient than the HSDF based method. However, for large SDF graphs, both conversion based methods may fail due to the large size of the equivalent graphs.

4.2.3 State-Space Exploration method

In contrast to conversion based methods, the State-Space Exploration (SSE) method [GGS+06] is able to compute the exact maximum throughput of SDF graphs without any conversion. The method consists of simulating an ASAP schedule of a SDF graph to measure its maximum throughput. In [GGS+06], the author demonstrates first how the ASAP schedule of a SDF graph, also called the self-timed execution, consists of a transient phase followed by a periodic phase in which a same set of executions is repeated periodically. Then from a relation between the traditional definition of the throughput and actors behavior in the ASAP schedule, the author defines the maximum throughput of a SDF graph by the following equation:

$$Th_{\max}(G) = \frac{Nb\ of\ graph\ iterations\ in\ one\ period}{The\ duration\ of\ one\ period}$$

In practice, the SSE algorithm simulates symbolically the ASAP schedule of a SDF graph until it reaches the periodic phase. At each clock cycle of the simulation, the algorithm stores the state of the graph (the number of data tokens present in each edge) and compares it to the previously stored states. If the current state is equal to one of the previously explored states, it means that the graph has reached the periodic phase. Starting from that state that belongs to the periodic phase, the algorithm simulates one last iteration to define the maximum throughput value of the graph.

Figure 4.3 shows the ASAP schedule of the SDF graph example of figure 4.1. Form the schedule we can see that at $t3$ the graph reaches its periodic phase, in which one period takes 4 times unit and includes one iteration of the graph. Based on the definition above, the maximum throughput of the SDF graph example is then 1/4.

In order to reduce the space complexity of the state-space exploration method, the implementation of its algorithm has been optimized to not store all the explored states. However, the numerical experiments of [dGKBS12] have shown that the state-space exploration method may fail for SDF graphs that start with a long transient phase. Compared

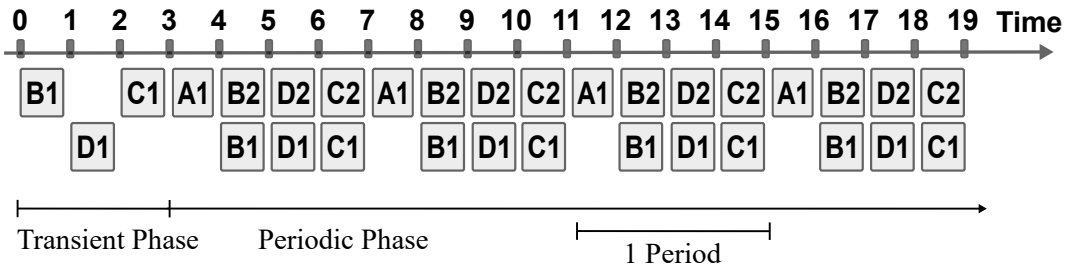


Figure 4.3 – *ASAP schedule of the SDF graph example of figure 4.1.*

with conversion based methods, the state-space exploration method is faster than the HSDF based method but slower than the max-plus algebra based method. Nevertheless, the state-space exploration method does not require any conversion and operates directly on the original SDF graph.

4.2.4 Periodic schedule based method

Computing the maximum throughput of large SDF graphs with the previous methods can be time and space consuming. In some cases, the previous methods may fail to return a result due to the large number of actors and their big repetition factor. Hence, an approximation of the maximum throughput which can be rapidly computed is often efficient for a rapid prototyping of signal processing applications. In this context, using the properties of the periodic schedule is the fast way to compute an approximation of the maximum throughput of large SDF graphs.

As presented in the previous chapter, a periodic schedule σ of an SDF graph $G = \langle A, F \rangle$ is a cyclic schedule [RV10], in which each actor $a \in A$ has a periodic execution defined by its initial start date $S^{\sigma}_{\langle a,0 \rangle}$ and a period ω^{σ}_{a}. Based on this property, the throughput of each actor $a \in A$ is defined as 1 over its execution period ω^{σ}_{a}. Hence, the throughput $Th^{\sigma}(G)$ of a periodic schedule σ of a SDF graph $G = \langle A, F \rangle$ is equal to:

$$Th^{\sigma}(G) = \frac{1}{\max_{a \in A}\{\omega^{\sigma}_{a}\}}$$

Among all the feasible periodic schedules of a SDF graph, there is one that minimizes the most the execution period of all the actors and thus maximizes the throughput of the graph. That schedule is called the optimum periodic schedule σ^* of the SDF graph. Therefore, the periodic schedule based method consists of computing the throughput of the optimum periodic schedule to estimate the maximum throughput of a SDF graph. In practice, the algorithm consists of the following steps:

- **Step 1**: Normalize the SDF graph $G = \langle A, F \rangle$ to define the Z_a of each actor $a \in A$.
- **Step 2**: Verify the existence of a periodic schedule for G.
- **Step 3**: Find the optimum periodic schedule σ^* of G by computing the minimum *average-token-flow-time* K^*.
- **Step 4**: Compute the execution period $\omega^{\sigma^*}_{a}$ of each actor $a \in A$ as $\omega^{\sigma^*}_{a} = K^* \times Z_a$.
- **Step 5**: Compute the throughput of the optimum periodic schedule σ^* of G as $Th^{\sigma^*}(G) = 1/\max_{a \in A}\{\omega^{\sigma^*}_{a}\}$.

The first step is mandatory for the computations, it simplifies the calculations and enables some useful properties. The goal of this step is to compute a normalized consumption and production rate Z_a for each actor $a \in A$. In this way, each actor of the graph

has the same consumption and production rate value on all of its input and output edges respectively. Once the SDF graph G is normalized, the second step verifies that G admit a periodic schedule. As presented in the previous chapter, a periodic schedule exists for G only if the sufficient condition of liveness is satisfied. In the case of G do not fulfill that condition, the periodic schedule based method cannot be used and the algorithm will fail to return a result. Step 3 consists of computing the minimum *average-token-flow-time* K^* of G that defines the optimum periodic schedule of the graph. Based on the system of precedence constraint that describes a periodic schedule, K^* is obtained by solving the following minimization problem:

$$\begin{cases} minimize & K^\sigma \\ \forall e(a,b) \in E & S^\sigma(b,0) - S^\sigma(a,0) \ \geq \ l(a) - K^\sigma \ H(e) \\ \forall a \in A & S^\sigma(a,0) \geq 0 \end{cases}$$

where the value $H(e)$ of an edge $e(a,b)$ is defined as $M_0(e) - (Z_b - gcd_e)$. Notice that, the dual problem of this minimization problem is nothing more than the formulation of a MCR problem:

$$\begin{cases} Maximize & \sum\limits_{e(a,b)\in E} l(a)\ x_e \\ \forall a \in A & \sum\limits_{e\in A^+(a)} x_e - \sum\limits_{e\in A^-(a)} x_e = 0 \\ & \sum\limits_{e\in A} H(e)\ x_e = 1 \\ \forall e \in E & x_e \geq 0 \end{cases}$$

In fact, since G is strongly connected and satisfies the sufficient condition ($\forall c \in C(G), \quad H(c) > 0$), a linear transformation can be applied to express explicitly the solution K^* of the main problem as a MCR problem:

$$\begin{cases} minimize & K^\sigma \\ \forall c \in C(G) & K^\sigma \times \sum\limits_{e\in c} H(e) \ \geq \ \sum\limits_{a\in c} l(a) \end{cases} \qquad \text{for which} \quad K^* = \max_{c\in C(G)} \frac{L(c)}{H(c)}$$

where $L(c) = \sum_{a\in c} l(a)$ the total execution time of the cycle, and $H(c) = \sum_{e\in c} H(e)$ the sum of the cycle edges value.

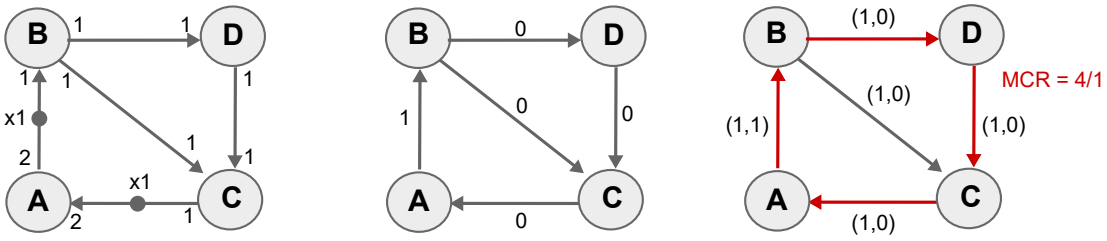
Once the value of K^* is defined, step 4 calculates the minimum execution period $\omega_a^{\sigma^*}$ of each actor $a \in A$ based on the property $\omega_a^{\sigma^*} = k^* \times Z_a$. Finally, the last step calculates the throughput value of the optimum periodic schedule which represents an estimation of the maximum throughput of G.

Figure 4.4 illustrates the different steps of computing the throughput of the SDF graph example of figure 4.1 using the periodic schedule based method. The first step computes a normalized consumption and production rate for each actor which turns out to be the same configuration as the original SDF graph example (fig 4.4a). Hence, the normalized rates of actors A, B, C, and D are respectively $Z_A = 2$, $Z_B = 1$, $Z_C = 1$, and $Z_D = 1$. Figure 4.4b shows the valued graph associated with the normalized SDF graph example, in which each edge e has an associated value $H(e)$. From the valued graph, the second step concludes that for any cycle c in the graph, $H(c) > 0$ and thus a periodic schedule exists for the SDF graph example. The third step computes K^*, the solution of the following

minimization problem to define the optimum periodic schedule σ^* of the graph:

$$\begin{cases} minimize \quad K^\sigma \\ S^\sigma(B,0) - S^\sigma(A,0) \geq 1 - 1 \times K^\sigma \\ S^\sigma(C,0) - S^\sigma(B,0) \geq 1 - 0 \times K^\sigma \\ S^\sigma(D,0) - S^\sigma(B,0) \geq 1 - 0 \times K^\sigma \\ S^\sigma(C,0) - S^\sigma(D,0) \geq 1 - 0 \times K^\sigma \\ S^\sigma(A,0) - S^\sigma(C,0) \geq 1 - 0 \times K^\sigma \\ S^\sigma(A,0), S^\sigma(B,0), S^\sigma(C,0), S^\sigma(D,0) \geq 0 \end{cases}$$

To simplify the computations, figure 4.4c shows the associated bi-valued graph of the system, in which each edge $e(i,j)$ has a bi-value $(l(i), H(e))$. From that graph, the MCR value equals to $4/1$ and thus $K^* = 4$. Then, step four defines the execution period ω^{σ^*} of the actors using their normalized rate Z and the K^* value: $\omega_A = 4 \times 2 = 8$, $\omega_B = 4$, $\omega_C = 4$, and $\omega_D = 4$. Finally, since ω_A is the maximum execution period, step five concludes that the maximum throughput of the SDF graph example is $1/\omega_A = 1/8$. Figure 4.5 illustrates the resulted periodic schedule of the graph. Compared to the ASAP schedule of figure 4.3, the throughput of the optimum periodic schedule is twice lower than the exact maximum throughput of the graph.



(a) *A normalized version of the SDF graph example.*

(b) *Verifying the existence of a periodic schedule.*

(c) *Computing the K^* value of the optimum periodic schedule.*

Figure 4.4 – *Computing the throughput of the SDF graph example of figure 4.1 using the periodic schedule based method.*

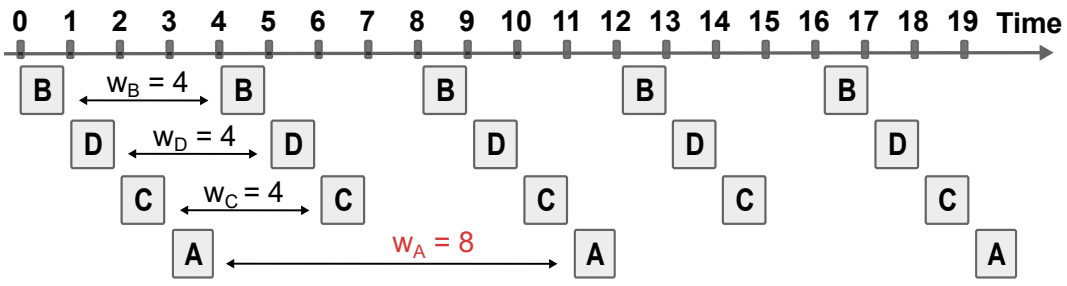


Figure 4.5 – *Optimum periodic schedule of the SDF graph example of figure 4.1.*

4.2.5 K-Iter method

Recently, [BMKdD16] introduced an iterative algorithm named K-Iter based on a k-periodic schedule to compute the throughput of both CSDF and SDF graphs. The K-perioidc schedule consists of defining the execution period of K instances of each actor, instead of one execution period per actor like the periodic schedule. The experimental results of [BMKdD16] show that the K-Iter method is more efficient than all the previous methods in computing throughput of large SDF graphs. However, the algorithm has failed to return a result for certain synthetic graphs.

4.3 Execution Modes of IBSDF graph

Before diving into the throughput computation of IBSDF graphs, we first present in this section the two execution modes of the IBSDF model. The first mode is the hierarchical execution, which maintains the insulation property of the hierarchical levels during the execution of the IBSDF graph. The second mode is a relaxed execution, which breaks the insulation property to accelerate the execution of the IBSDF graph and thus increasing its throughput.

4.3.1 Hierarchical Execution

Additionally to the semantics of the IBSDF graph, there are three firing rules that have been introduced in [PBR09]. The objective of these rules is to maintain the insulation property of the interface-based hierarchy during the execution of the IBSDF graph. The three firing rules are defined as follows:

- **Rule 1**: A subgraph can start its execution only when enough data tokens are available on all the input Fifo of its hierarchical parent actor.
- **Rule 2**: A subgraph can transmit data tokens in the output Fifo of its hierarchical parent actor only when it finishes its iteration.
- **Rule 3**: A subgraph executes only one iteration at a time for each firing of its parent actor.

The first rule guarantees that no actor of a subgraph in the hierarchy starts to execute before the firing of its hierarchical parent actor. In this way, the execution of an IBSDF graph always starts from its top graph in a hierarchical way. When a hierarchical actor of the topgraph is ready to be fired, only at that time, its subgraph starts to execute. The second and the third rules guarantee an atomic behavior for the hierarchical actors. Indeed, a hierarchical actor will produce data tokens only when its subgraph have finished one complete iteration.

With these three firing rules, a hierarchical actor behaves exactly like a regular SDF actor consuming and producing data tokens. Therefore, the insulation of the subgraphs is maintained during the execution of the IBSDF graph. Moreover, each subgraph can be evaluated and scheduled independently from the hierarchy, which eases the analysis of the IBSDF graph.

Figure 4.6 shows an example of an IBSDF graph for which a hierarchical execution is shown in figure 4.7. The IBSDF graph example is composed of two hierarchy levels, in which actor C is a hierarchical actor described by the IBSDF graph $EFGH$. An iteration of the topgraph is composed of 1 execution of actor A and 2 executions of each of the

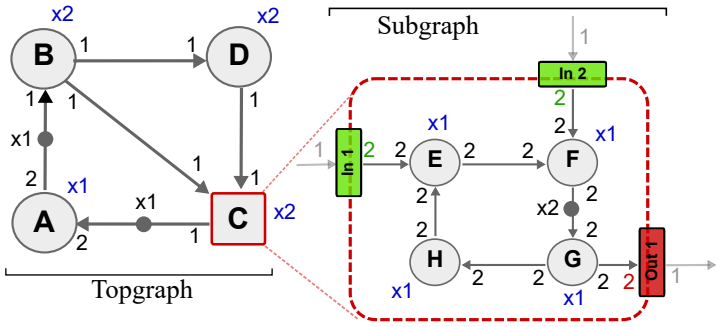


Figure 4.6 – *Example of an IBSDF graph composed of two levels of hierarchy, in which actor C is a hierarchical actor described by the IBSDF subgraph EFGH.*

remaining actors B, C, and D. One iteration of the subgraph equals to 1 execution of each of its actors. Applying the firing rules described above, the ASAP schedule of the IBSDF graph example results in the hierarchical execution shown in figure 4.7. As the schedule demonstrates, the subgraph $EFGH$ start to execute at time 2 only when actor C receives enough data tokens from its precedent actors B and D (Rule 1). Following the second rule, actor C produces data tokens only when its subgraph finishes its execution at time 6, which results in firing actor A. Based on the graph configuration, each execution of actor A results in two executions of actor C. Thus by respecting the third rule, a complete subgraph iteration is executed for each of the two firings of actor C simultaneously.

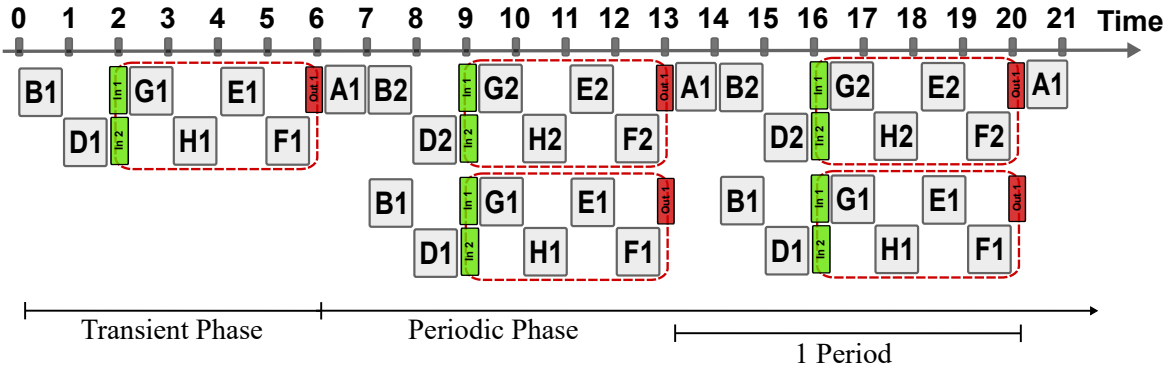


Figure 4.7 – *ASAP schedule of the equivalent flat srSDF graph of figure* **??**.

4.3.2 Relaxed Execution

The firing rules were introduced mainly to ease the analysis of the IBSDF graph, especially the liveness evaluation. Indeed, with the new added atomic property of the hierarchical actors, each subgraph can be analyzed independently from the hierarchy. Thence, the IBSDF graph is consistent and live if each of its subgraphs is consistent and live, including the topgraph. However, the firing rules may slow down the execution of the IBSDF graph by constraining the firings of the subgraph actors. Hence, the firing rules may stop the application to reach its maximum throughput.

In this context, the second mode is used to increase the performance of the IBSDF graph. The relaxed execution consists of relaxing the firing rules once the consistency and the liveness of the IBSDF graph have been verified. Hence, each actor of the hierarchy is free to execute as soon as there are enough data tokens on its input Fifo including input and output interfaces. Therefore, the IBSDF graph can reach its maximum throughput.

Figure 4.9 shows the ASAP schedule of the IBSDF graph of figure 4.6 in the relaxed execution mode. To better understand the behavior of the IBSDF graph in this execution mode, figure 4.8 shows its equivalent flat-srSDF graph. As these two figures show, subgraph actors $G1$ and $G2$ are free to execute at $t0$ simultaneously with actor B because the first rule has been relaxed. Similarly, by relaxing the second rule, both output interfaces $Out1$ are free to transmit data tokens immediately after the execution of $G1$ and $G2$. And so, actor $A1$ can be executed right after that at $t1$ simultaneously with sub-actor $H1$, even if the subgraph has not finished yet its iteration. Again, relaxing the first rule allows the input interface $In1$ to transmit immediately the data tokens received from actor $B2$ to the subgraph actor $E2$. Hence, the input interface $In1$ does not wait until the input interface $In2$ receives data tokens from actor $D2$ to execute. Thus, subgraph actor $E2$ and actor $D2$ start to execute simultaneously at $t3$ once the actor $B2$ finishes its execution.

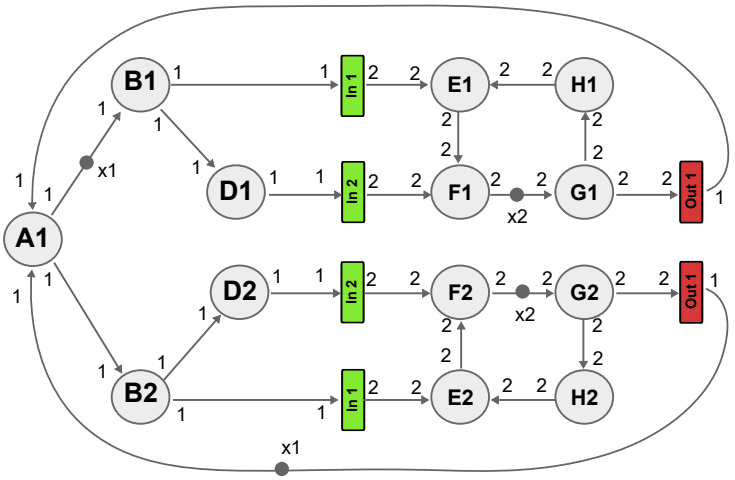


Figure 4.8 – *The equivalent flat srSDF graph of the IBSDF graph of Figure 4.6.*

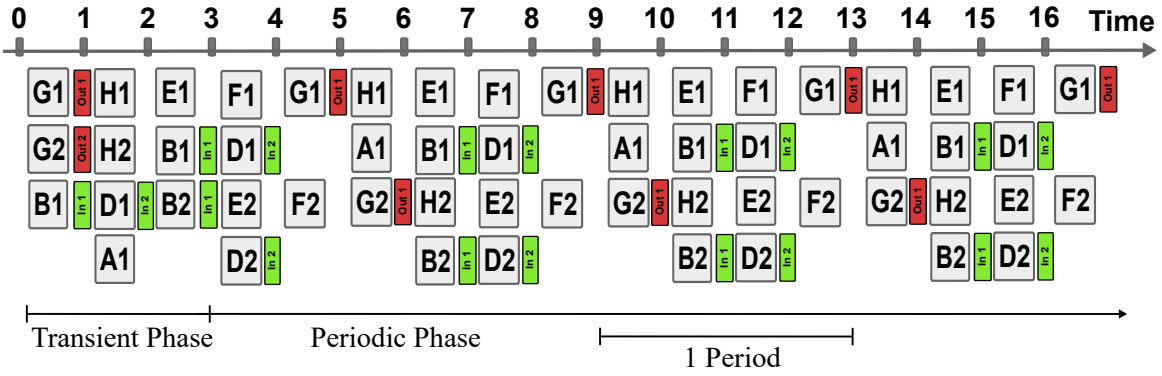


Figure 4.9 – *ASAP schedule of the IBSDF graph of figure 4.6 under the relaxed execution mode.*

Compared to the previous execution mode (fig 4.7), the relaxed execution mode reduces the duration of one period of the ASAP schedule down to 4 times unit, instead of 7. Which means that the execution of the IBSDF graph of figure 4.6 speeds up by a factor of 1.75 when the firing rules are relaxed. However, a relaxed execution may use more PEs than in a hierarchical execution in order to fully exploit the parallelism of the application. Comparing the two schedules of figures 4.7 and 4.9, the relaxed execution of the IBSDF graph example uses up to 4 PEs in parallel instead of 2 PEs in the hierarchical execution.

4.4 Throughput Evaluation by Flattening the Hierarchy

Now that the two execution modes of the IBSDF graph have been presented along with the SDF state-of-the-art methods, this section discusses how to compute the exact maximum throughput of IBSDF graphs. Firstly, we present the classical approach which is based on SDF methods to evaluate the throughput of IBSDF graphs under a relaxed execution. Lastly, we show how to model the firing rules of the IBSDF graph in order to evaluate its throughput under a hierarchical execution.

4.4.1 Classical Approach

The classical approach is based on a simple technique that has been used since the introduction of the IBSDF model to compile it with the existing methods. The technique consists of first flattening the hierarchy of the IBSDF graph into an equivalent flat-srSDF graph. Then, computing the throughput of the resulted flat graph with SDF methods as if it was a large SDF graph. From SDF state-of-the-art methods, we define two classical methods for the IBSDF graph:

- **Flat-SSE**: which is based on the State-Space Exploration (SSE) method and consists of simulating an ASAP schedule of the equivalent flat-srSDF graph to measure its exact maximum throughput. As an example of application, figure 4.9 shows the ASAP schedule of the equivalent flat-srSDF of the IBSDF graph example (figure 4.8). One period of the schedule in the periodic phase is composed of 1 iteration of the graph and takes 4 times unit. Thus, the maximum throughput of the original IBSDF graph in a relaxed execution is 1/4 data tokens per time unit.

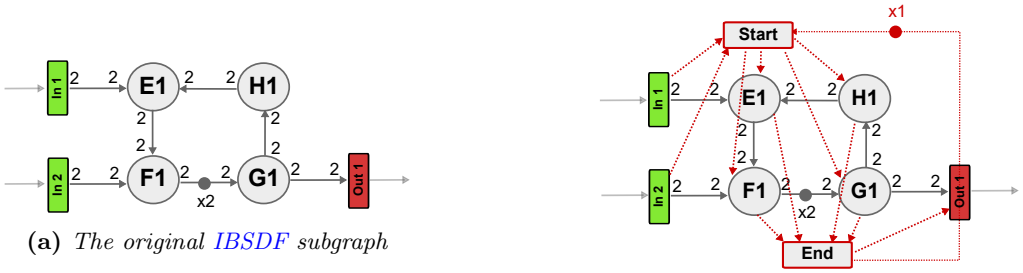
- **Flat-Periodic**: which consists of computing the optimal periodic schedule of the equivalent flat-srSDF graph to define its maximum throughput. Notice that a periodic schedule always exists for a consistent and live srSDF graph. Moreover, all of the actors of a srSDF graph have the same execution period $\omega_a = K$ when applying a periodic schedule. Hence, the throughput of a srSDF graph is equal to $1/K$, where K is the MCR value of its normalized version. Taking the fact that normalizing a srSDF graph results in an HSDF graph, the Flat-periodic method is nothing more than the HSDF-based method applied to the equivalent flat-srSDF graph. In practice, the Flat-Periodic method converts an IBSDF graph first to a flat-srSDF graph, then to an HSDF graph, and finally computes its exact maximum throughput as 1 over its MCR value.

Using the K-Iter algorithm on the flat-srSDF graph results in computing a *1-Periodic* schedule of the graph, which is exactly the same algorithm as the Flat-Periodic schedule. However, we did not investigate on developing a new flattening process based on the LCG conversion of the Max-Plus Algebra-based method, since the new flattening process will nearly have the same space-complexity as the one based on the srSDF conversion.

4.4.2 Modeling the Firing Rules

Unfortunately, the classical flattening process of the IBSDF graph results in an equivalent flat-srSDF graph that does not express explicitly the firing rules of the hierarchical execution mode. As consequences, the classical methods cannot evaluate the throughput of the hierarchical execution of an IBSDF graph, without adapting either their algorithms of the equivalent flat-srSDF graph. Since the flat graph is used at each phase of the development process, we have adapted its structure to support the hierarchical execution

mode. This also enables us to use the equivalent flat srSDF graph as the input graph of any implementation of SDF methods in the existing prototyping tools.



(a) *The original IBSDF subgraph*

(b) *The IBSDF subgraph with the firing rules*

Figure 4.10 – *Modeling the firing rules for the subgraph EFGH of the IBSDF graph of figure 4.6.*

The firing rules are modeled in the equivalent flat-srSDF graph by adding two extra actors *Start* and *End*, and extra edges to control the execution of the subgraph. Each firing rule is modeled as follows:

- **Modeling Rule 1**: The *Start* actor enforces *Rule 1* by imposing a precedence relationship between all input interfaces of the subgraph on one side, and all subgraph actors on the other side.
- **Modeling Rule 2**: The *End* actor enforces *Rule 2* by imposing a precedence relationship between all subgraph actors on one side, and all output interfaces of the subgraph on the other side.
- **Modeling Rule 3**: *Rule 3* is enforced by adding precedence relationships between the *Start* and the *End* actor.

Figure 4.10b shows the resulted srSDF subgraph of figure 4.6 after adding the extra actors and the extra edges to the original subgraph (figure 4.10a). Figure 4.11 shows the resulted flat-srSDF graph with the modeled firing rules.

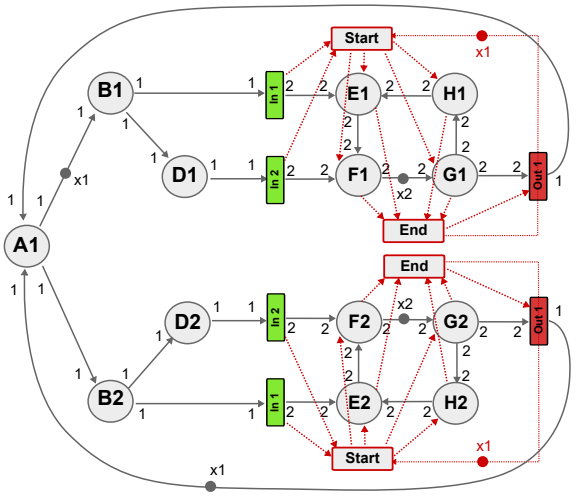


Figure 4.11 – *The equivalent flat srSDF graph of the IBSDF graph of Figure 4.6 with firing rules.*

4.5 Throughput Evaluation without Flattening the Hierarchy

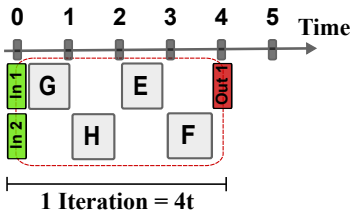
In the previous section, we have presented the classical approach, which consists of reusing SDF methods to compute the throughput of IBSDF graphs after flattening their hierarchy into a flat-srSDF graph. This approach is a simple method that does not require too much development and can easily be integrated into existing prototyping tools. However flattening the hierarchy of an IBSDF graph may result in an exponential growth of actors and edges number in the equivalent flat-srSDF graph, for which SDF methods fail to return a result in a reasonable amount of time and memory. This issue of the flattening process has motivated us to deeply study the throughput evaluation problem of IBSDF graphs, and develop new methods specifically for this hierarchical dataflow model. In the following, we introduce two new methods for IBSDF graphs, Schedule-Replace (SR) technique for the hierarchical execution mode and Evaluate-Schedule-Replace (ESR) method for the relaxed execution mode. Each method takes advantage of both the interface-based hierarchy of the model and the properties of the targeted execution mode. Hence, both methods compute the throughput of IBSDF graphs without completely flattening their hierarchy.

4.5.1 The Schedule-Replace (SR) technique

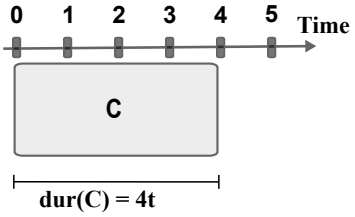
The Schedule-Replace (SR) technique computes the throughput of IBSDF graphs under a hierarchical execution. The technique is based on constructing an ASAP schedule of the IBSDF graph in a bottom-up approach and computes its throughput as follows:

- **Phase 1** Starting from the bottom level of the hierarchy up to the topgraph, for each level:
  - **Step 1**: Compute the duration of the hierarchical actors by **scheduling** their subgraph using a symbolic execution of an ASAP schedule.
  - **Step 2**: **Replace** each hierarchical actor with a regular actor that has the same duration as the subgraph execution, computed in step 1.
  - **Step 3**: Move to the upper level and repeat step 1 and step 2 until the topgraph is reached.
- **Phase 2** Compute the throughput of the resulted topgraph as follows:
  - **Step 1**: Convert the resulting topgraph to a srSDF graph and add a self-loop edge for each actor which was originally hierarchical.
  - **Step 2**: Compute the throughput of the resulted srSDF graph with SDF State-Of-The-Art methods [GGS+06, BNHMMK12].

Based on the firing rules of the IBSDF graph that define the hierarchical execution mode, a hierarchical actor behaves exactly like a regular SDF actor consuming and producing data tokens. Indeed, the execution of its subgraph which is defined by a bloc of sub-actors execution can be seen as the execution of a regular SDF actor that require multiple resources to execute. As an example, figure 4.12a shows the execution of the subgraph $EFGH$ of the hierarchical actor C in the hierarchical execution mode. The execution of the subgraph can be abstracted with the execution of a regular SDF actor (fig-4.12b) that requires 2 PE to execute and has a duration of 4 times unit.



(a) *The execution of the subgraph EFGH of the hierarchical actor C.*



(b) *The equivalent SDF actor of the hierarchical actor C.*

Figure 4.12 – *Abstracting the execution of the subgraph EFGH of the IBSDF graph example.*

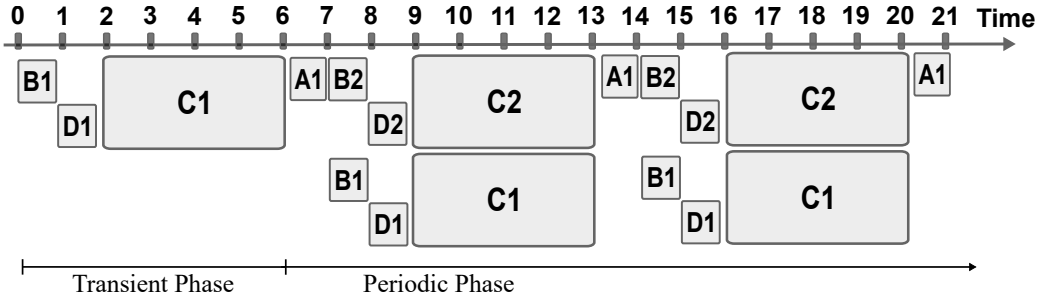


Figure 4.13 – *An abstracted hierarchical execution of the IBSDF graph example.*

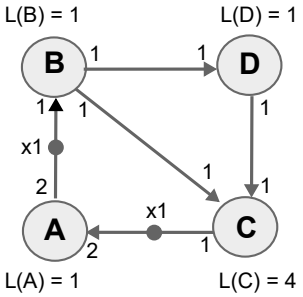
Hence, for each hierarchical actor h described by a subgraph G, we define its equivalent SDF actor h^* with a duration $l(h^*)$ and a number of resources $R(h^*)$ such that:

- $l(h^*)$ equals the duration of one iteration of the subgraph G
- $R(h^*)$ equals the maximum number of sub-actors executed in parallel

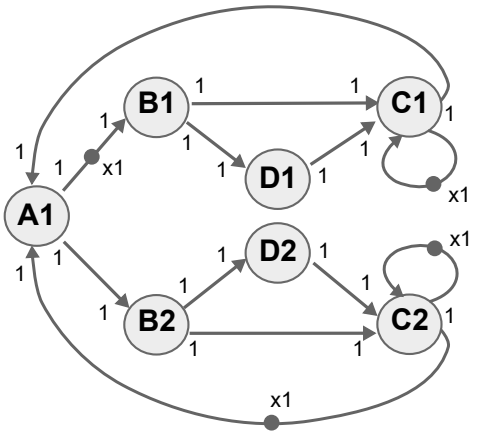
Figure 4.13 shows the ASAP schedule of the IBSDF graph example under a hierarchical execution, in which each execution of the hierarchical actor C is represented by one bloc abstracting the execution of the subgraph.

Therefrom, the SR technique consists of **scheduling** one iteration of a subgraph to measure its duration, then **replacing** its hierarchical parent actor with an equivalent SDF actor. Thanks to the compositionality feature of the interface-based hierarchy, each subgraph can be scheduled independently. And so, starting from the bottom level of the hierarchy up to the topgraph, the complete execution of an IBSDF graph can be abstracted in its topgraph by repeating the process of schedule/replace for each hierarchical actor. Hence, the first phase of the SR technique consists of abstracting the complete execution of an IBSDF graph in its topgraph using the schedule/replace process. Then, the final phase computes the maximum throughput of the resulted topgraph to determine the throughput of the original IBSDF graph. To ensure the IBSDF firing rules and to respect the data dependencies of the subgraphs, a self-loop edge containing 1 data token is added to each instance of a hierarchical actor in the equivalent srSDF topgraph. Figure 4.14a shows the resulted topgraph of the IBSDF graph example after applying the first phase of the SR technique. Figure 4.14b shows the resulting srSDF graph of the second phase in which its throughput is evaluated using SDF methods and represents the throughput of the original IBSDF graph example.

Algorithm 1 shows an implementation of the SR technique, in which the abstraction process of the first phase is done by a recursive function. Even if the SR technique converts the topgraph into a srSDF graph to evaluate its throughput, the time and space complexity of this method remains very low compared to the classical approach. Indeed,



(a) *The resulted topgraph after applying the first phase of SR technique.*



(b) *The equivalent srSDF graph of the resulted topgraph after adding the self-loop edges.*

Figure 4.14 – *Evaluating the throughput of the IBSDF graph example with the SR technique.*

the schedule/replace process is done one time for each hierarchical actor plus, the size of the resulted topgraph to evaluate is very small compared to the size of the equivalent flat-srSDF graph. Moreover, the SR technique is able to compute the throughput of large IBSDF graphs since it analyzes the hierarchy level by level.

Algorithm 1 Schedule-Replace (SR) technique pseudo-code

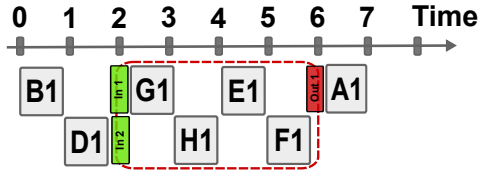
```
function Evaluate(IBSDF graph)                          ▷ First phase
    for all actors a ∈ IBSDF.TopGraph do
        if a is hierarchical actor then
            Process(a)
        end if
    end for                                             ▷ Second phase
    srSDF = ConvertToSRSDF(IBSDF.TopGraph)
    for all actors a ∈ srSDF do
        if a is hierarchical actor then
            AddSeflLoopEdge(a)
        end if
    end for
    throughput = computeThroughput(srSDF)
    return throughput
end function

procedure Process(hierarchical actor H)
    for all actors a ∈ H.SubGraph do
        if a is hierarchical actor then
            Process(a)
        end if
    end for
    schedule = ASAPschedule(H.SubGraph)
    regularActor = createNewActor(schedule.duration)
    ReplaceHierarchicalActor(H, regularActor)
end procedure
```

4.5.2 The Evaluate-Schedule-Replace (ESR) method

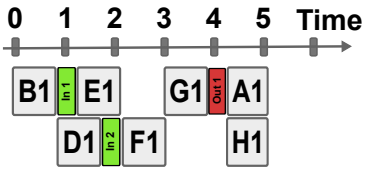
The Evaluate-Schedule-Replace (ESR) method computes the throughput of IBSDF graphs under a relaxed execution. This method is based on the Schedule-Replace (SR) technique and consists of the following steps:

- **Phase 1** Re-time the IBSDF subgraphs to synchronize their execution and to reveal the hidden delays in the hierarchy.
- **Phase 2** Starting from the bottom level of the hierarchy up to the topgraph, for each level:
  - **Step 1** Construct a replacement graph for each hierarchical actor of the current level by **scheduling** and **evaluating** its subgraph.
  - **Step 2** Convert the graphs of the current level to srSDF graphs and **replace** each instance of a hierarchical actor by its replacement graph.
  - **Step 3** Move to the upper level and repeat step 1 and step 2 until the topgraph is reached.
- **Phase 3** Compute the throughput of the resulted srSDF graph of the topgraph with SDF State-Of-The-Art methods [GGS+06, BNHMMK12].

Similarly to the SR technique, the ESR method analyzes the IBSDF graph level by level to compute its throughput. However, the new method replaces a hierarchical actor by a small graph modeling the behavior of its subgraph in a relaxed execution. Indeed, relaxing the firing rules enables both input and output interfaces of the subgraph to execute at a different time as soon as they are ready to be fired. This behavior of the subgraph interfaces makes the execution of a hierarchical actor different to the one of a regular SDF actor. As consequences, a hierarchical actor cannot be replaced with a regular SDF actor in a relaxed execution of the IBSDF graph. As an example, figure 4.15 shows a comparison between the behavior of the subgraph in each execution mode of the IBSDF graph example. In the hierarchical execution (figure 4.15a), the subgraph behaves as one bloc of actors execution in which both input interfaces $In1$ and $In2$ start at the same time. As concerns the output interface $Out1$, it starts to execute until the end of the subgraph iteration. In the relaxed execution mode where the firing rules are ignored (figure 4.15b), the subgraph behaves differently. In fact, the input interfaces start at a different time and the output interface starts before the end of the subgraph iteration. This behavior of the subgraph interfaces is what actually makes the relaxed execution faster than the hierarchical one because it compresses the execution of the entire IBSDF graph. But, it is not possible anymore to abstract the subgraph execution with a regular actor as we did in figure 4.13 for the hierarchical execution of the IBSDF graph example.



(a) *The execution of the subgraph under a hierarchical execution*



(b) *The execution of the subgraph under a relaxed execution*

Figure 4.15 – *A comparison between the behavior of the subgraph in each execution mode*

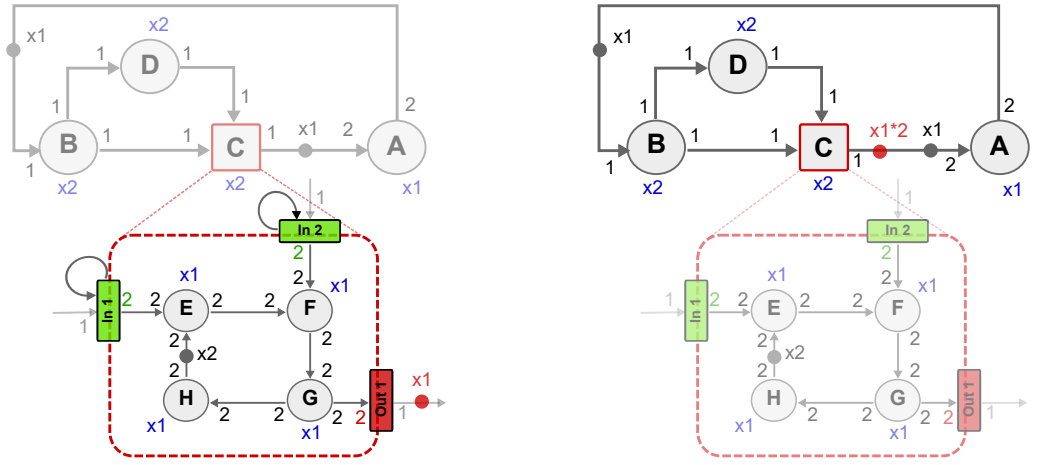
Therefore, the core algorithm 2 of the ESR method consists of first, representing the behavior of the subgraph with a small graph that models the time difference between the execution of the interfaces. Then replacing the hierarchical parent actor of the subgraph with its replacement graph which we named the equivalent subgraph execution model. In the following, we explain and illustrate each step of the ESR algorithm taking the IBSDF graph of figure 4.6 as the graph example.

Algorithm 2 Evaluate-Schedule-Replace (ESR) Pseudocode

```
ListReplacementGraphs = new LIST(Actor, Graph)
                                              ▷ Evaluate an IBSDF graph
function EVALUATE(IBSDF)
    SYNCHRONIZESUBGRAPHS(IBSDF)
    for all actors a ∈ IBSDF.TopGraph do
        if a is hierarchical actor then
            PROCEED(a)
        end if
    end for
    srSDF = CONVERTTOSRSDF(IBSDF.TopGraph)
    for all actors a ∈ srSDF do
        if a is hierarchical actor then
            REPLACE(a, ListReplacementGraphs)
        end if
    end for
    throughput = COMPUTETHROUGHPUT(srSDF)
    return throughput
end function
                                              ▷ Proceed a hierarchical actor
procedure PROCEED(H)
    for all actors a ∈ H.SubGraph do
        if a is hierarchical actor then
            PROCEED(a)
        end if
    end for
    srSDF = CONVERTTOSRSDF(H.SubGraph)
    for all actors a ∈ srSDF do
        if a is hierarchical actor then
            REPLACE(a, ListReplacementGraphs)
        end if
    end for
    HSDF = CONVERTTOHSDF(srSDF)
    K = COMPUTEMCR(HSDF)
    DAG = CONVERTTODAG(HSDF)
    schedule = ASAP&ALAPSCHEDULE(DAG)
    replacementGraph = CONSTRUCTGRAPH(schedule,K)
    ListReplacementGraphs.add(H,replacementGraph)
end procedure
```

ESR main algorithm

The first phase of the ESR method is mandatory to analyze independently the subgraphs. It consists of moving up the data tokens that are ready to be transmitted from the subgraphs to their upper graphs. Starting from the bottom level of the hierarchy up to the topgraph, each subgraph is executed until it stops. Each time an output interface is executed, the data tokens to transmit are moved from the subgraph to its upper graph. A temporary empty self-loop edge is added to actors without input edges and to input interfaces for preventing an infinite execution of subgraphs during this phase. At the end of phase 1, the current state of the subgraphs is saved as the new initial marking. In the IBSDF graph, each instance of a hierarchical actor is described by an instance of its subgraph. Therefore each data token moved from a subgraph to an upper level is multiplied by the repetition factor of the hierarchical actor parent of the subgraph. Figure 4.17 shows the synchronization of the IBSDF subgraph of figure 4.6. Figure 4.16b represent the final state of the IBSDF graph which will be used for the throughput evaluation.



(a) *Add empty self-loop edges to input interfaces and execute the subgraph until it stops.*

(b) *Move the data tokens produced by output interfaces to the upper level and duplicate them.*

Figure 4.16 – *Synchronization of the execution of the IBSDF subgraph of the figure. 4.6*

The second phase represents the core algorithm of the ESR method. It consists of evaluating the maximum execution behavior of a subgraph, based on its structure. Modeling it by a relaxed subgraph execution model that abstracts the subgraph structure and shows only the execution time difference between its input and output interfaces. Finally, replacing the hierarchical parent actor of the subgraph by its relaxed subgraph execution model. This process is repeated for each level starting from the bottom level up to the topgraph. At the end of *Phase* 2, the topgraph abstracts the complete hierarchy of the IBSDF graph. *Phase* 3 computes the throughput of the topgraph to determine the IBSDF graph throughput.

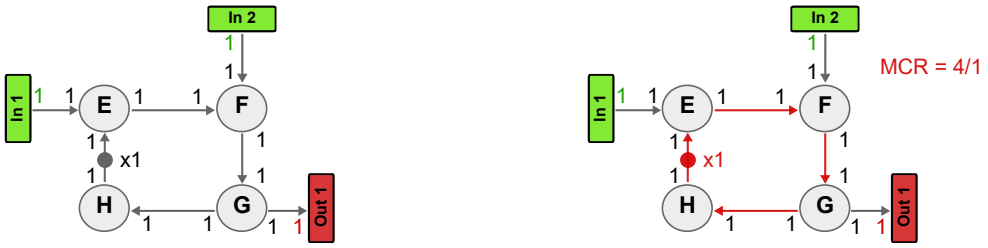
Constructing the Equivalent Subgraph Execution Model

Despite the precedence relationship of a hierarchical actor, the subgraph cannot exceed the maximum relaxed execution defined by its own structure. The purpose of the equivalent subgraph execution model is to abstract the subgraph structure by representing only the maximum throughput of the subgraph and the execution time difference between its interfaces. Which is all required for the throughput evaluation.

The equivalent subgraph execution model is constructed as follows:

1. Convert the subgraph to a srSDF graph and compute the minimum execution period of its actors using the periodic schedule [BNHMMK12]
2. Schedule the srSDF subgraph using an ASAP schedule followed by an As Late As Possible (ALAP) schedule
3. Construct the equivalent subgraph execution model using the schedule and the minimum execution period obtained in the previous steps

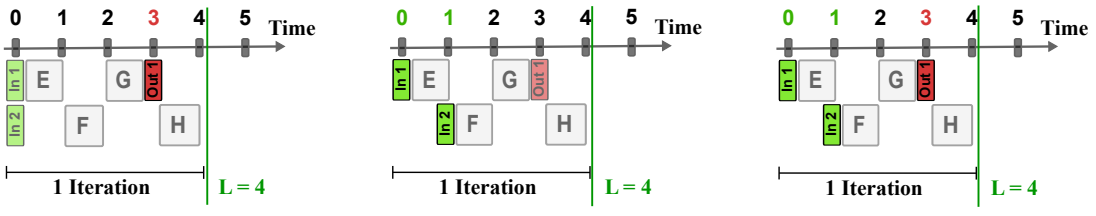
Step 1 consists of computing the average-token-flow-time K of the srSDF subgraph using the periodic schedule [BNHMMK12]. The K value represents the minimum waiting time to respect between two successive firings of subgraph actors due to their data dependencies. And $1/K$ represents the maximum throughput of the subgraph. Figure 4.17b shows the MCR of the equivalent HSDF subgraph (figure 4.17a), which equals 1/4. Thence, $k = 4$ and means that each subgraph actor needs to wait 4 clock-cycles between two successive firings.



(a) *The equivalent HSDF subgraph EFGH* **(b)** *The MCR of the HSDF subgraph*

Figure 4.17 – *Computing the average-token-folw-time K of the IBSDF subgraph of figure. 4.6*

Step 2 consists of scheduling the subgraph by an ASAP schedule followed by an ALAP schedule to define the minimum time difference between its interfaces execution. The ASAP schedule allows defining the minimum time that output interfaces take to transmit data tokens when the subgraph starts an iteration. Scheduling the subgraph with ALAP schedule allows to define the maximum delay for input interfaces to execute without affecting the execution of output interfaces. To use the existent algorithm of ASAP and ALAP schedules, the HSDF subgraph of step 1 is converted to a DAG. The DAG version of the HSDF subgraph is obtained simply by removing all Fifo that contain initial marking. Figures 4.18a and 4.18b show respectively the ASAP and ALAP schedule of the IBSDF subgraph of figure 4.6. The final schedule (figure 4.18c) highlights only the execution of the interfaces, which is all required for step 3.



(a) *ASAP schedule* **(b)** *ALAP schedule* **(c)** *Final schedule*

Figure 4.18 – *Scheduling the IBSDF subgraph of figure. 4.6 with an ASAP schedule followed by an ALAP schedule to measure the time difference between the execution of the subgraph interfaces.*

Step 3 constructs the equivalent subgraph execution model by representing the final schedule with an equivalent srSDF graph. The start time of each interface is represented by an actor ($TimeLineActor$) with a null duration. The time difference between two consecutive executions of interfaces in the timeline is represented by an actor ($TransitionActor$) with a duration equals to that time difference. Each interface is connected to the associated actor of its execution start time. The execution duration of the subgraph interfaces is set to 0. A final actor, named $PeriodActor$, is added to model the minimum waiting time of the subgraph actors by creating a cycle with the $TimelineActors$. The duration of the $PeriodActor$ is computed as $numerator(K) - \sum L(TransitionActor)$. The maximum throughput of the subgraph is modeled by adding $denominator(K)$ data tokens to the cycle created by the $PeriodActor$. Thus the MCR value of the cycle represents the exact value of K.

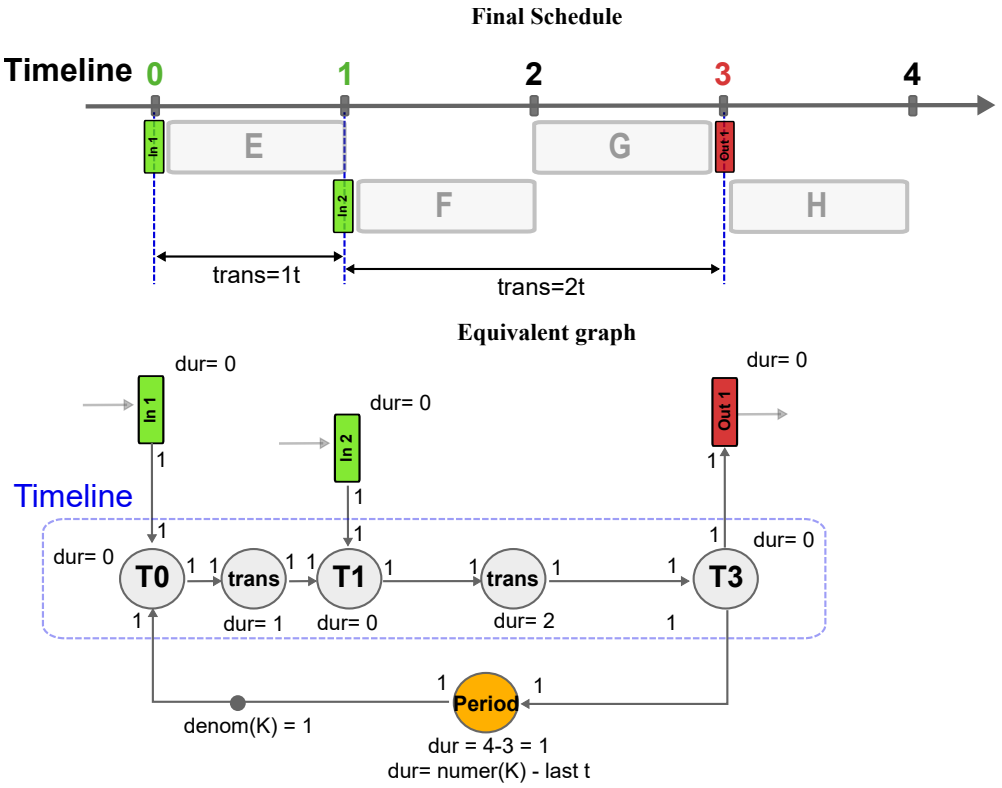


Figure 4.19 – *Constructing the equivalent subgraph execution model of the IBSDF subgraph of figure. 4.6.*

Figure 4.20 shows the equivalent subgraph execution model of the IBSDF subgraph of figure 4.6. The three interfaces $in1$, $in2$, and $out1$ are connected respectively to the timeline actors $T0$, $T1$ and $T3$ with a null duration. The time difference between the firing of both input interfaces $in1$ and $in2$ is represented by the first transition actor $Trans$ with a duration equal to 1. The second transition actor $Trans$, with a duration equal to 2, models the time difference between the firing of the input interface $in2$ and the output interface $out1$. The period actor $period$ is connected to the time actor $T0$ in one side, and to $T3$ on the other side. The duration of the period actor is computed as $K = 4$ minus the total duration of the transition actors $\sum L(Trans) = 3$, which equals 1. The maximum throughput of the subgraph which equal to $1/K = 1/4$ is modeled by adding 1 data token on the FIFO between the period actor $Period$ and $T0$.

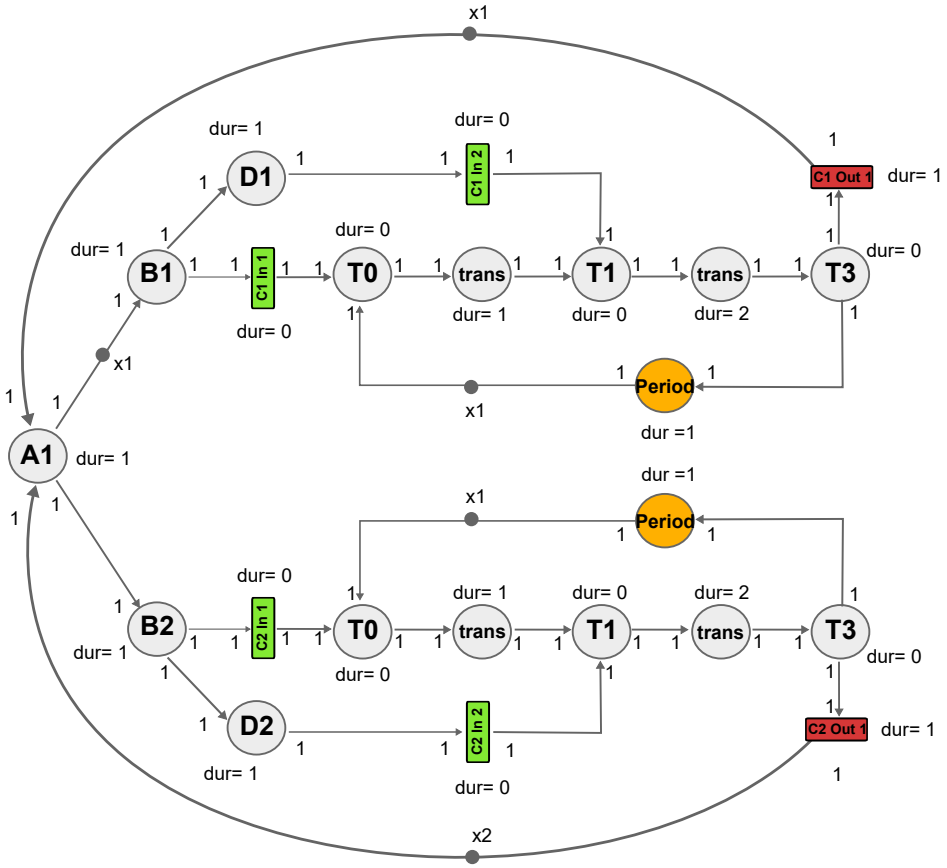


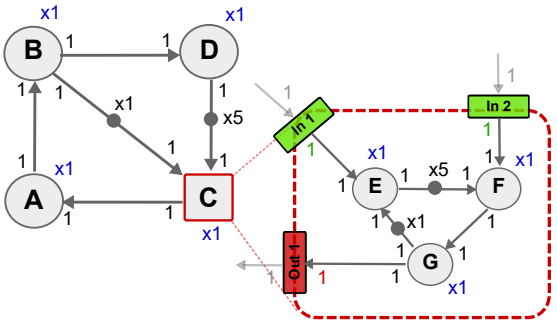
Figure 4.20 – *The resulted srSDF topgraph of the ESR method.*

Figure 4.20 shows the resulted srSDF topgraph of the ESR method. The graph is obtained by first converting the IBSDF topgraph of figure 4.6 to a srSDF graph. Then replacing each instance of the hierarchical actor C with the equivalent subgraph execution model of its subgraph (figure 4.19).

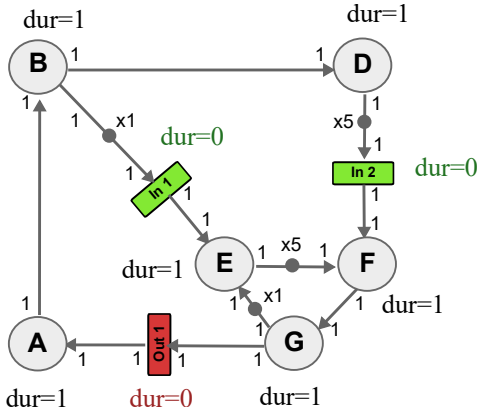
A special case of IBSDF graphs

For a particular case of IBSDF graphs, the ESR method is not capable of computing the exact maximum throughput. Especially when the initial marking of the graph removes the data dependence between output and input interfaces in a subgraph iteration. For instance, the initial marking of the IBSDF graph in figure 4.21a allows executing 5 times the output interface $out1$ without the need of executing the input interface $in1$. That makes the output interface $out1$ independent from the input interface $in1$ in terms of data tokens for 5 subgraph iterations.

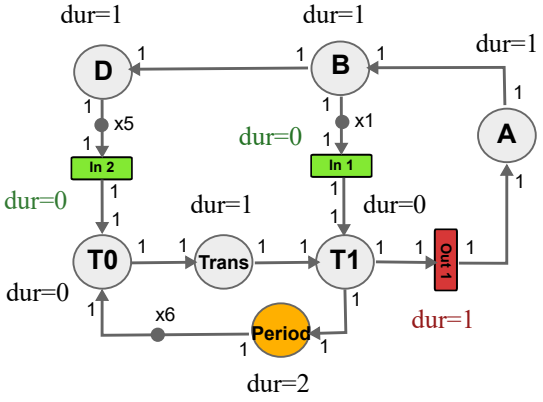
Figure 4.21b shows the equivalent relaxed srSDF graph of the IBSDF graph of figure 4.21a. The MCR of the relaxed srSDF graph is the value of the cycle (A, B, D, $in2$, F, G, $out1$) and so the throughput is $1/MCR = 5/5 = 1$. Using ESR method, the throughput is 1/3 computed as 1 over the MCR value of its resulted srSDF graph (fig. 4.21c) which is the value of the cycle (A, B, $in1$, $T1$, $out1$). The throughput computed with the ESR method represents 33.3% of the maximum throughput.



(a) *A Particular case example of IBSDF graph*



(b) *The equivalent relaxed srSDF graph without the extra-edges.*



(c) *The resulting srSDF graph of the ESR method.*

Figure 4.21 – *An IBSDF graph for which the ESR method fails to return the exact value.*

4.6 Experimental Results

4.6.1 Experimental Setup

In this section, we compare the performance of the new techniques Schedule-Replace (SR) and Evaluate-Schedule-Replace (ESR) with the classical methods Flat-State-Space Exploration (SSE) and Flat-Periodic which are based on flattening the hierarchy. The numerical experiments consist of measuring the running time for each method to compute the throughput, including the srSDF conversion when relevant. The maximum throughput value computed by the classical methods is used to compare the quality of SR and ESR techniques. Experimental results are summarized in Table 4.2,4.3 and 4.4. The difference in the throughput value between the two techniques ESR and SR shows how much the execution of IBSDF graphs can speed up by relaxing the firing rules.

To extensively test our methods, we have used a set of IBSDF graphs composed of two categories. The first category is a set of real signal processing applications modeled by IBSDF graphs and available in [Pre]. The second category is a set of synthetic IBSDF graph generated randomly using the IBSDF graph generator based on Turbine tool [BLDMK14]. Table 4.1 shows the characteristics of the benchmark. The number of hierarchical levels of the real applications does not exceed 4 levels, which is not big enough to show all the potential of the new methods. Therefore, the synthetic graphs are generated with a higher number of levels than the real applications. The last synthetic IBSDF graph *Graph* 6 has 10 hierarchical levels, which enable us to compare the performance of the new methods on large IBSDF graphs.

Table 4.1 – *Description of the benchmark set*

IBSDF Graph			Equivalent Flat-srSDF graph			
			Hierarchical Exec. mode		Relaxed Exec. mode	
Name	Levels	Actors	Actors	FIFOs	Actors	FIFOs
Crypto	2	10	34	85	34	49
Large FFT	2	10	267	1300	267	777
LTE	4	18	250	641	272	337
Stereo Matching	2	41	1604	5829	1606	3143
Graph 1	3	15	503	1654	513	816
Graph 2	5	20	17727	80976	17730	56703
Graph 3	6	24	84440	338391	87440	196019
Graph 4	5	150	653289	3253811	654566	2034980
Graph 5	8	240	39 E10	-	39 E10	-
Graph 6	10	100	31 E15	-	31 E15	-

Both methods SR and ESR have been integrated to the open-source Preesm framework [PDH+14] as well as the Periodic schedule based method. For the SSE method [GGS+06], we have used the open source implementation of SDF^3 [SGB06]. We also used the mathematical programming solver Gurobi [gur] for the computation of the optimum periodic schedule [BNHMMK12]. All the methods were tested on one core of an Intel i5-6300 processor clocked at 2.4 GHz and with 8GB of RAM.

4.6.2 Results

Table 4.1 shows the description of both, the IBSDF graphs and their equivalent Flat-srSDF graph. From the size of the equivalent Flat-srSDF graphs of each execution mode, we can clearly see that the flattening process results in an exponential growth of actors and edges number. In fact, it was not possible to convert the last two synthetic IBSDF graphs *Graph* 5 and *Graph* 6 with the available RAM. As consequences, the classical methods have failed to compute the throughput of these two graphs in both execution modes.

Table 4.2 compares the total execution time of the SR method with the classical methods Flat-SSE and Flat-Periodic, to compute the throughput of the benchmark set under the hierarchical execution mode. As the table shows, the Flat-SSE is faster than Flat-Periodic and SR techniques for small IBSDF graphs. However, the execution time of both classical methods increases exponentially as the number of levels and the size of the equivalent srSDF graph grow. In contrast, the SR technique was able to compute the throughput of all the IBSDF graphs in a few milliseconds since the technique avoids flattening the hierarchy. For the Stereo-Matching application, the SR technique is 3 times faster than Flat-Periodic method and 70 times faster than Flat-SSE method.

The results for the synthetic graphs confirm that the periodic schedule based method is suitable for large SDF graphs than State-Space Exploration (SSE) method. In fact, even if the Flat-Periodic method took 4 minutes to compute the throughput of *Graph* 4 compared to 61 milliseconds with SR technique, it remains better than FLat-SSE which failed to return a result in less than 5 minutes.

Table 4.2 – *Performance comparison between the Schedule-Replace technique and the srSDF conversion based methods.*

IBSDF Graph		Total Execution Time		
Name	Levels	Flat-SSE	Flat-Periodic	Schedule-Replace (SR) technique
Crypto	2	**4 ms**	8 ms	38 ms
Large FFT	2	**29 ms**	48 ms	36 ms
LTE	4	**22 ms**	32 ms	32 ms
Stereo	2	3676 ms	151 ms	**37 ms**
Graph 1	3	493 ms	67 ms	**34 ms**
Graph 2	5	>5 min	3060 ms	**34 ms**
Graph 3	6	>5 min	14600 ms	**34 ms**
Graph 4	5	>5 min	234000 ms	**61 ms**
Graph 5	8	-	-	**61 ms**
Graph 6	10	-	-	**72 ms**

Table 4.3 – *Performance comparison between Classical Approaches, Schedule-Replace (SR) technique, and the Evaluate-Schedule-Replace (ESR) method.*

IBSDF graph	Flat-SSE	Flat-Periodic	ESR technique		SR
Name	Exec.Time	Exec.Time	% of Opt.	Exec.Time	% of Opt.
Crypto	**1 ms**	6 ms	100%	45 ms	100%
Large FFT	44 ms	**33 ms**	100%	74 ms	100%
LTE	152 ms	**20 ms**	100%	58 ms	100%
Stereo Matching	4320 ms	**80 ms**	100%	130 ms	99.18%
Graph 1	11984 ms	**30 ms**	100%	59 ms	34.48%
Graph 2	>1h	1190 ms	100%	**70 ms**	55.05%
Graph 3	>1h	2319 ms	100%	**90 ms**	20.70%
Graph 4	>1h	55407 ms	100%	**306 ms**	12.49%
Graph 5	-	-	??	**560 ms**	1%
Graph 6	-	-	??	**1930 ms**	0.x%

Table 4.3 compares the total execution time of the ESR method with the classical methods Flat-SSE and Flat-Periodic, to compute the throughput of the benchmark set under the relaxed execution mode. The optimality of the ESR technique is calculated based on the exact value of the maximum throughput computed with the classical methods.

Based on the results, the classical methods are faster than ESR technique for small graphs, but again, their execution time increases exponentially as the number of hierarchical levels grows. However, this time the Flat-Periodic method took less time to compute the throughput of the equivalent Flat-srSDF graphs. Which is explained by the fact that the equivalent flat-srSDF graph has less number of actors and edges in the relaxed execution mode than the hierarchical one since the extra actors and extra edges are not added to the flat graph. Surprisingly, this is not the case for the Flat-SSE method. We assume that the equivalent flat-srSDF graph has more states to explore in a relaxed execution since the number of actors to execute at each cycle is higher. Thus the Flat-SSE method takes a longer time to return a result.

Table 4.4 – *Graphs description*

IBSDF graph			Resulted srSDF topgraph			
			SR technique		ESR method	
Name	Levels	Actors	Actors	FIFOs	Actors	FIFOs
Crypto	2	10	10	17	30	37
Large FFT	2	10	4	5	8	8
LTE	4	18	3	4	7	7
Stereo Matching	2	41	39	63	56	78
Graph 1	3	15	15	26	23	32
Graph 2	5	20	12	30	24	42
Graph 3	6	24	12	35	22	45
Graph 4	5	150	90	196	95	201
Graph 5	8	240	150	441	174	465
Graph 6	10	100	50	124	68	142

Like the SR technique, the ESR method has shown to be efficient for large IBSDF graphs compared to the classical methods. Indeed, the throughput of all the benchmark set has been computed in less than 2 seconds with the ESR method. As an example, the Flat-Periodic method took almost 1 minute to compute the maximum throughput of *Graph* 4, compared to 0.3 seconds with ESR method. Furthermore, the ESR technique has been able to compute the exact maximum throughput of both real and synthetic IBSDF graphs.

The last column of table 4.3 shows the difference value between the computed throughput of SR technique and the one of ESR method. Based on the result, we can see how much the execution of an IBSDF graph can speed up by relaxing the firing rules. Or in other words, how much the execution of an IBSDF graph can slow down by constraining the subgraph execution with the firing rules. Moreover, the results show that the IBSDF graph throughput in a hierarchical execution decreases as the number of its hierarchical levels grows. As the worst case example, the throughput of *Graph* 6 under a hierarchical execution represents less than 1% of the one under a relaxed execution.

Form a space complexity perspective, even if the new methods SR and ESR convert the topgraph into a srSDF graph, the size of the resulted graph remains very small compared to the flat-srSDF graph. Table 4.4 shows the size of the resulted srSDF topgraph of each of SR and ESR methods. As the table shows, the complete execution of the hierarchy of each IBSDF graph in the benchmark set is abstracted in a small srSDF topgraph.

4.7 Conclusion

In this chapter, we have presented the two execution modes of the IBSDF graph. The hierarchical execution mode which maintains the insulation property of the IBSDF model through three firing rules. And, the relaxed execution mode which breaks the firing rules to accelerate the execution of IBSDF graphs to reach their maximum throughput. Based on the SDF state-of-the-art methods we have introduced two classical methods that flatten the hierarchy of an IBSDF graph into an equivalent flat-srSDF graph and evaluate its exact maximum throughput as if it was a large SDF graph. Since the classical flattening process does not support the hierarchical execution mode, we have shown how to model the

firing rules explicitly in the equivalent flat-srSDF graph via extra actors and extra edges. Despite the fact that the classical methods reuse existing implementation of SDF methods and do not necessitate any development, unfortunately the flattening process result in an exponential growth of both actors and edges number of the equivalent flat-srSDF graph. This exponential space-complexity makes the the throughput evaluation a hard task for the classical methods. In this context, we have introduced two new methods, each one evaluate the throughput of an IBSDF graph under an execution mode without completely flattening the hierarchy of the graph. Both methods take advantage of the interface-based hierarchy of the IBSDF graph and evaluate its throughput level by level. The Schedule-Replace (SR) technique which is the new method for the hierarchical execution mode, consists of abstracting the execution of a subgraph with a regular SDF actor that has the same duration. Scheduling one iteration of the subgraph allows to measure its duration and thus defining the duration of the replacement actor. The SR technique repeats that process of scheduling and replacing subgraphs for each hierarchical level of the graph starting from the bottom level. At the end of that abstraction process, the new technique evaluates only the throughput of the topgraph to define the maximum throughput of the original IBSDF graph. For the relaxed execution mode, we have introduced the new method Evaluate-Schedule-Replace (ESR) which is based on the same process of the SR technique. In contrast the ESR method replace a subgraph with an Equivalent Subgraph Execution Model (ESEM) which models the behavior of the subgraph in the relaxed execution mode. The size of the Equivalent Subgraph Execution Model (ESEM) remains very small since it depends only on the number of the subgraph interfaces. Thus, like SR technique, the ESR method abstracts the complete execution of the IBSDF graph in its topgraph that has an equivalent throughput. To confirm our theoretical studies, we have tested the introduced method on a real applications modeled with IBSDF graphs, and on a synthetic IBSDF graph generated randomly with a higher number of hierarchical levels. The experimental results have shown that the new methods SR and ESR outperform the classical methods in term of time and space-complexity. Moreover, the new methods are capable of evaluating the throughput of IBSDF graphs with high number of hierarchical levels since they analyze the graph level by level. In the next chapter, we discuss how to efficiently evaluate the minimum latency of IBSDF graphs using the same approach of the new techniques introduced in this chapter.

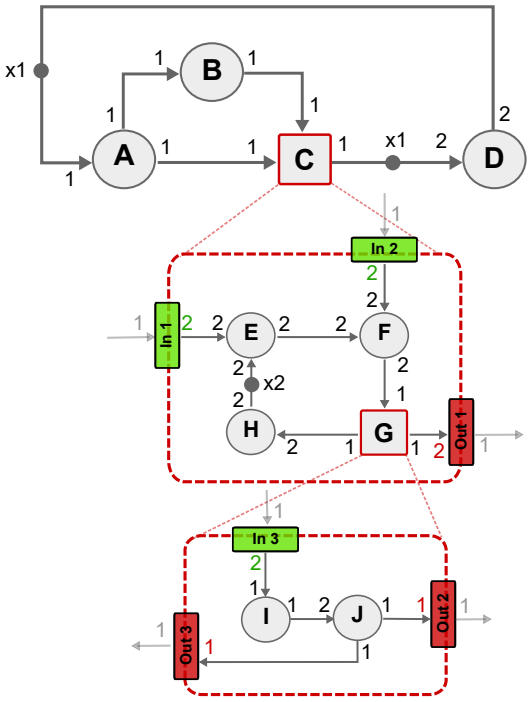
CHAPTER 5

Latency Evaluation of IBSDF graph

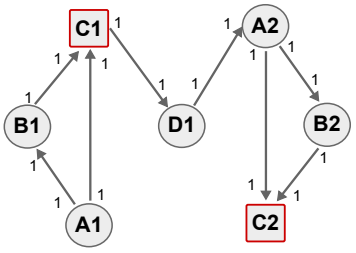
5.1 Introduction

In this chapter we evaluate a second key performance indicator for signal processing applications which is the multi-core latency. The latency of an application is defined as the required time for the application to process all the input data and return a result. From the dataflow graph perspective, the latency is the duration of one complete iteration of the graph. The *multi-core latency* refers to the duration of the application when it is executed on a multi-core architecture. The multi-core latency of dataflow-based Model of Computation (MoC) has been the subject of many researches in the past and it is still studied today with the rising of multi/many-core architectures [Les17]. One part of the researches focuses on evaluating the minimum possible multi-core latency of a dataflow graph, either considering the specifications of the multi-core architecture or not [GSB+07, SGC16, CH17]. Another part aims to compute a new initial marking for the dataflow graph to minimize its latency. This last process is often called *re-timing* the application [OS01]. In scheduling problems area, the latency is termed the *makespan* [Das04] and it is used during the optimization as an objective function to minimize or as a constraint to satisfy in case of real-time applications [RA01, BHR09, DLC15, BS12, KKB17]. Our work focus on evaluating the minimum possible multi-core latency of applications modeled with IBSDF graphs regardless the specifications of the architecture, i.e. the minimum achievable latency of the IBSDF graph when it is running on an architecture with unlimited number of Processing Element (PE) and memory capacity so that all the parallelism of the application is enabled. The only criteria taken into account is the Worst-Case Execution Time (WCET) of actors, which is predefined or measured.

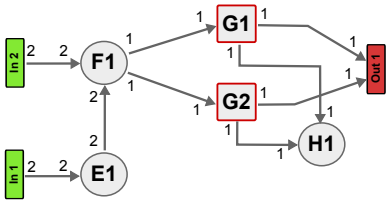
The minimum multi-core latency of basic models like the SDF graph has been wildly studied and several methods have been introduced to evaluate its latency at low complexity [GSB+07, Das04]. However, the latency evaluation of the IBSDF graph still relies on a classical approach which consists on flattening the complete hierarchy of the graph and then evaluate it with the state-of-the-art methods of SDF graph. This approach has been proven to be inefficient due to the exponential growth of actors number during the flattening process [DDNMK17b]. Therefore, we developed two new methods to evaluate the latency of IBSDF graphs at low complexity. Both methods take advantage of the



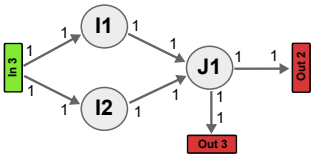
(a) *An IBSDF graph of three hierarchical levels. Actors C and G are hierarchical actors described respectively by the subgraphs $EFGH$ and IJ.*



(b) *The equivalent DAG of the topgraph.*



(c) *The equivalent DAG of the subgraph $EFGH$.*



(d) *The equivalent DAG of the subgraph IJ.*

Figure 5.1 – *The IBSDF graph which will serve as the graph example in this chapter.*

interface-based hierarchy of the IBSDF model to hierarchically compute the multi-core latency of the graph without completely flattening its hierarchy.

This chapter is composed of six sections, including the introduction as the first section. In section 2, we present how to compute the mono-core latency of an IBSDF graph and to visualize its value from the hierarchy perspective. In section 3, we present the classical approach for the multi-core latency evaluation of IBSDF graphs based on the state-of-the-art methods of the SDF model. In section 4, we present our new methods to evaluate the multi-core latency of IBSDF graphs at low complexity. Each presented method concerns a specific execution mode of the IBSDF model. In section 5, we prove our theoretical study with numerical experiments in which we compare the performance of the new methods with the performance of the classical approach. Section 6 concludes the chapter.

In this chapter, we will take the IBSDF graph of sub-figure 5.1a as the graph example on which we will illustrate each introduced method. The IBSDF graph example is composed of three hierarchical levels. Actors C and G are both hierarchical actors described respectively by the subgraphs $EFGH$ and IJ. The duration of the regular actors is set to 1 clock-cycle (cc) to simplify the illustrations of the chapter. The duration of the hierarchical actors on the other hand, depends on the execution of the subgraphs. Each of the actors A, B, C, G, and I have a repetition factor of 2. The remaining actors have a repetition factor of 1. The sub-figures 5.1b, 5.1c, and 5.1d show the equivalent DAG of each subgraph of the IBSDF graph example. Based on these sub-figures, the entire hierarchy of the IBSDF graph example is flattened into an equivalent flat DAG by replacing each instance of a hierarchical actor with the equivalent DAG of its subgraph, starting from the topgraph equivalent DAG. The resulting equivalent flat DAG of the IBSDF graph example has 23 actors, 50 edges and 18 interfaces which will be illustrated further in section 5.3.

5.2 Mono-Core Latency Evaluation

5.2.1 For SDF graph

Computing the mono-core latency value is as important as evaluating the multi-core latency. It allows measuring the speedup of the modeled application when it is executed on a multi-core architecture. The latency value of a mono-core execution of a dataflow graph is the duration of one complete iteration of the graph when it is executed on one Processing Element (PE). For SDF graphs, the mono-core latency value is computed as the sum of actors duration multiplied by there repetition factor [BLM96a]:

$$MonoCoreLatency(G_{SDF}\langle A, F\rangle) = \sum_{a \in A} (dur(a) \times RV(a))$$

This equation derives from the fact that a mono-core execution of the graph is an execution sequence with no empty slots in which, each actor $a \in A$ is executed exactly $RV(a)$ times. A feasible sequence can be obtained by scheduling one iteration of the SDF graph on one PE. It can also be obtained by ordering the actors of the equivalent DAG of the SDF graph using a topological order. From the mono-core latency evaluation perspective, scheduling an iteration of the graph is not mandatory for the evaluation since any feasible sequence has the same duration, which is the sum of actors duration multiplied by their repetition factor.

5.2.2 For IBSDF graph

Similarly, the mono-core latency of an IBSDF graph can be computed without scheduling an iteration of the graph. Indeed executing an IBSDF graph on one PE results in an execution sequence in which each actor in the hierarchy is executed following its repetition factor. Regardless of the execution mode used, hierarchical or relaxed, the duration of the resulted mono-core execution sequence is the same.

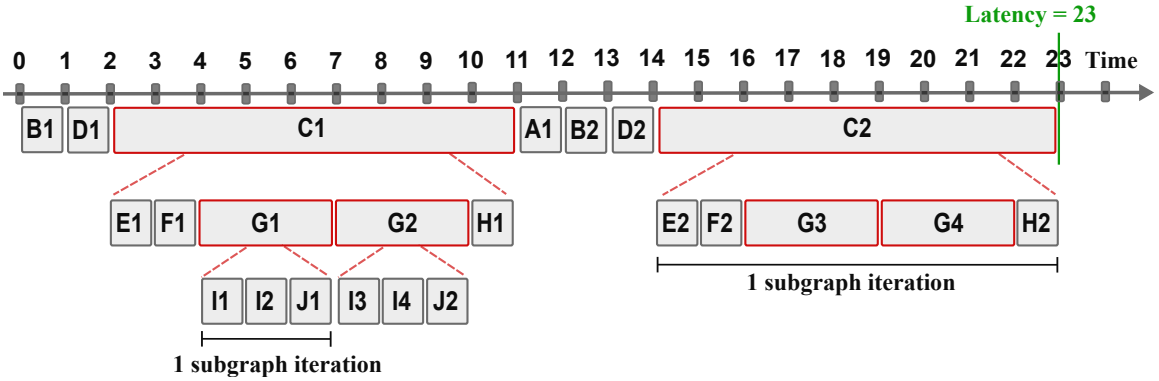


Figure 5.2 – *A mono-core execution of the IBSDF graph example of figure 5.1a. Each execution of a hierarchical actor abstracts the execution of its subgraph.*

Figure 5.2 shows a mono-core execution of the IBSDF graph example of figure 5.1a. As the figure shows, it takes 23 clock-cycles (cc) to execute a complete iteration of the entire hierarchical graph. Thus, the minimum latency of a mono-core execution of the IBSDF graph example is 23 cc. As the figure shows, each execution of the hierarchical actor C abstracts the mono-core execution of the subgraph $EFGH$, which takes 9 cc to execute a complete iteration. Similarly, each execution of the hierarchical actor G abstracts the mono-core execution of the subgraph IJ that takes 3 cc.

Algorithm 3 Mono-core Latency Evaluation of IBSDF graph

```
function MonoCoreLatency(IBSDF)
    Latency = 0
    for all actors a ∈ IBSDF do
        if actor a is hierarchical then
            a.duration = MonoCoreLatency(a.subgraph)
        end if
        Latency = Latency + (a.duration × a.repetitionFactor)
    end for
    return Latency
end function
```

Therefore, the mono-core latency of a consistent and live IBSDF graph can be computed hierarchically based on the same approach as the Schedule-Replace (SR) technique [DDNMK17a] introduced in the previous chapter as follows :

- **Phase 1** Starting from the bottom level of the hierarchy up to the topgraph, for each level:
  - **Step 1**: Compute the duration of the hierarchical actors by evaluating the mono-core latency of their subgraph using the SDF method.
  - **Step 2**: Set the duration the hierarchical actors based on the mono-core latency of their subgraph, computed in step 1.
  - **Step 3**: Move to the upper level and repeat step 1 and step 2 until the topgraph is reached.
- **Phase 2** Compute the mono-core latency of the resulted topgraph using SDF method.

Algorithm 3 shows an implementation of the SR-based technique to compute the mono-core latency of an IBSDF graph recursively. By executing algorithm 3 on the IBSDF graph example, the computation starts by evaluating the mono-core latency of the subgraph IJ to define the duration of the hierarchical actor G. Using the SDF method, the mono-core latency of the subgraph IJ is computed as $dur(I) \times RV(I) + dur(J) \times RV(J)$ which is equal to 3 clock-cycles (cc). Then, the duration of actor G is set, and the same process is repeated for the subgraph $EFGH$ to define the duration of the hierarchical actor C. After evaluating the mono-core latency of the subgraph $EFGH$, the duration of actor C is defined as 9 cc. A final evaluation of the mono-core latency of the topgraph $ABCD$ allows defining the mono-core latency of the entire IBSDF graph.

The time complexity of evaluating the mono-core latency of both graphs SDF and IBSDF is linear $\mathcal{O}(|\mathcal{A}|)$, where $\mathcal{A}$ is the set of all atomic and hierarchical actors in the graph. This low complexity enables the developer to instantly evaluate the mono-core latency of the designed graph as soon as the duration of the atomic actors is defined.

5.2.3 The Mono-Core Latency from the hierarchy perspective

Usually, when evaluating the mono-core latency of a SDF graph, only the latency value is (significant) to the developer giving him an approximation of the expected performance on a mono-core architecture. However, for a hierarchical model like the IBSDF graph, it might be interesting to give a deep view of the mono-core latency value from the hierarchy perspective, answering the following questions:

- how the mono-core latency is distributed among the hierarchy?
- what is the contribution percentage of each actor to the mono-core latency?
- which actors are critical to the performance of the application?

In order to answer these questions, we introduce a new parameter for the IBSDF actors called the total repetition factor. This new parameter represents the total number of executions of each actor in the hierarchy during a complete iteration of the entire IBSDF graph. Let us take actor I of the IBSDF graph of figure 5.1a as an example. The actor I has a repetition factor of 2, meaning that it will be executed 2 times at each firing of its parent actor G. However, actor G, in turn, has its own repetition factor $RV(G) = 2$ and a hierarchical parent actor C for which $RV(C) = 2$. Thus, in one iteration of the entire IBSDF graph, the actor I is executed a total number of $RV(I) \times RV(G) \times RV(C) = 8$. Formally, we define the total repetition factor $TotalRV(a)$ of an actor $a \in \mathcal{A}$ as the product of its own repetition factor and the repetition factor of all its hierarchical parent actors $\mathcal{P}(a)$, from the level of the actor back to the top-graph:

$$TotalRV(a) = RV(a) \times \prod_{p \in \mathcal{P}(a)} RV(p)$$

Using this new parameter in simple statistics like multiplying the duration of an actor by its total repetition factor, it allows to define its total PE usage which can also be represented as a percentage of the mono-core latency and thus defining the critical actor that represents a bigger percentage of the mono-core latency:

$$\frac{totalRV(a) \times Dur(a)}{MonoCoreLatency(G_{IBSDF})} \times 100$$

Table 5.1 shows the mono-core latency value of the IBSDF graph of figure 5.1a computed earlier but, this time from the hierarchy perspective. The first and second columns of the table respectively represent the name and the type of each actor of the graph. The third and fourth columns respectively represent the repetition factor and the total repetition factor of each actor. The fifth and sixth columns show respectively the duration and the total PE usage of each actor. Based on the mono-core latency value, the last column shows the total PE usage of each actor as the percentage of the mono-core latency of the application. As the table shows, actors I and J have the highest total repetition factor among all the regular actors. In fact, flattening the hierarchy of the graph will result in an equivalent flat graph where the total duplication number of actors I and J represent more than 50% of the total number of actors. In terms of PE usage, the actor I and J represent respectively on their own 34.80% and 17.40% of the total PE usage of the application. This is higher than the PE usage of all the atomic actors of the top graph combined. Since the hierarchical actor C abstracts both subgraphs G_{EFGH} and G_{IJ}, it represents itself 78.26% of the mono-core latency.

Table 5.1 – *The mono-core latency value of the IBSDF graph example of figure 5.1a from the hierarchy perspective.*

Actor	Type	RV	TotalRV.	Dur.	TotalRV. × Dur.	%of Latency
Level 1						
A	regular	1	1	1	1	4.35%
B	regular	2	2	1	2	8.70%
C	**hierarchical**	**2**	**2**	**9**	**18**	**78.26%**
D	regular	2	2	1	2	8.70%
Level 2						
E	regular	1	2	1	2	8.70%
F	regular	1	2	1	2	8.70%
G	**hierarchical**	**2**	**4**	**3**	**12**	**52.17%**
H	regular	1	2	1	2	8.70%
Level 3						
I	regular	2	8	1	8	34.80%
J	regular	1	4	1	4	17.40%

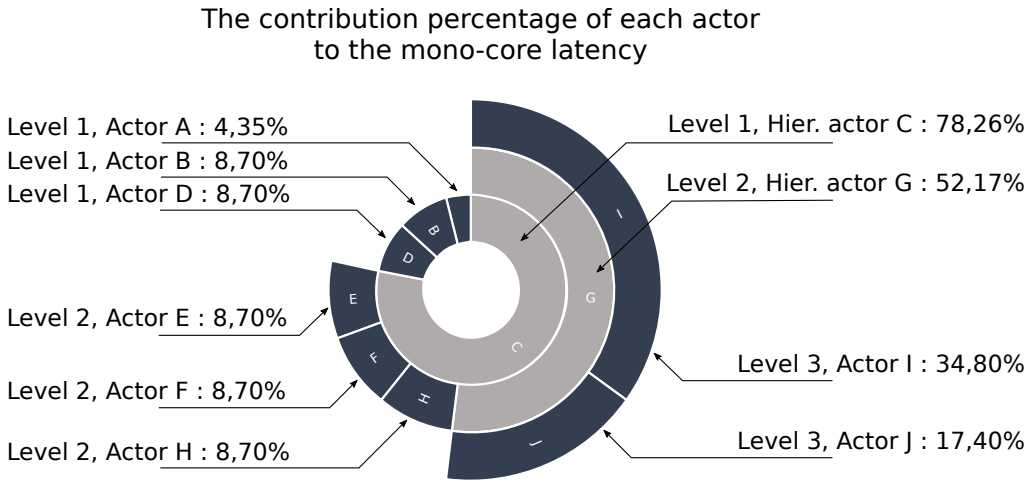


Figure 5.3 – *A multilevel pie chart showing the contribution percentage of each actor of the IBSDF graph example (fig.5.1a) to the mono-core latency.*

Figure 5.3 resumes the result of table 5.1 in a multilevel pie chart, which allows to visually identify the critical actors in the hierarchy. Each hierarchical level is represented by a concentric ring, where the center ring corresponds to the top level of the graph. Each segment of a concentric ring represents an actor. If the actor is hierarchical, then a set of child segments is added in the next concentric ring to represent its subactors. The size of each segment shows how much an actor contributes to its hierarchical parent actor and thus to the mono-core latency of the application. From the pie chart of figure 5.3, it is clear that actors I and J constitute the biggest percentage of the mono-core latency of the graph and thus the highest PE usage of the application.

This new representation of the mono-core latency from a hierarchy perspective is crucial for a precise analysis of the application performance. It enables the developer to identify the critical actors of the IBSDF graph who the performance of the application depends on. Moreover, hierarchically evaluating the latency allows defining the maximum duration of each hierarchical actor in case of its subgraph is executed on one PE.

5.3 Multi-Core Latency Evaluation by flattening the hierarchy

As defined in the introduction, the multi-core latency of an application is the minimum possible duration of one iteration of its equivalent dataflow graph when it is executed on a multi-core architecture with unlimited resources, such that all the task and data parallelisms of the application are enabled. Computing the multi-core latency of a dataflow graph is essential to evaluate the theoretical speedup of the application when it is running at maximum performance compared to a mono-core execution.

A simple way to evaluate the multi-core latency of an IBSDF graph is by flattening its hierarchy into a flat graph and evaluate it like if it was a large SDF graph. This classical approach used previously for the throughput evaluation problem [DDNMK17a] has shown to be easy to implement and to integrate into existing prototyping tools [PDH+14]. Practically, the classical approach consists of: first, flattening the hierarchy of the IBSDF graph into an equivalent flat graph, then adding extra actors and extra edges to model the firing rules in the case of a hierarchical execution. Finally, using one of the state-of-the-art methods of the SDF model to evaluate the latency of the resulted flat graph as a large SDF graph. In the literature, the multi-core latency of a SDF graph which is often called the minimum achievable latency is computed using two methods: the Critical-Path Method (CPM) [Kel61] and the Symbolic-Execution (SE) method [GSB+07]. In the following, we introduce, illustrate and compare each of these two methods on the IBSDF graph example of figure 5.1a.

5.3.1 Critical-Path Method (CPM)

The Critical-Path Method (CPM) [Kel61] is a general technique used in the project management area, in conjunction with the Project Evaluation and Review Technique (PERT) for scheduling project activities [Ker]. In the embedded system area, the CPM is used for finding the CP of signal processing applications that are modeled with a dataflow graph. The critical path of an application is the longest path of dependent actors in the equivalent DAG, where the length of the path is computed as the sum of its actors' duration. From the execution perspective, the critical path represents the longest execution sequence of actors who cannot be executed in parallel due to their data dependency. Therefore, the length of the critical path represents the minimum possible duration of a graph iteration despite the infinite number of the available PEs, i.e., the minimum achievable multi-core latency of the application.

For SDF graphs, the CPM consists of first converting the graph into an equivalent DAG and then finding its critical path as follows:

- **Step 1:** Add dummy actors S and T with a null duration to the equivalent DAG. Then, connect actor S to all DAG actors with no input Fifo. Similarly, connect actor T to all DAG actors with no output Fifo.
- **Step 2:** Compute a topological order for the DAG actors using [Kah62, Tar76].
- **Step 3:** For each actor, set the length of its output edges equal to its duration and its distance from actor S to negative infinity.
- **Step 4:** Compute the longest path from actor S to actor T by relaxing the distance of the actors following the topological order and based on the length of the edges.

Similarly for the IBSDF graph, applying the CPM consists of converting the hierarchical graph into an equivalent flat DAG and computing its critical path following the steps described previously. However, the equivalent flat DAG is obtained by first flattening the

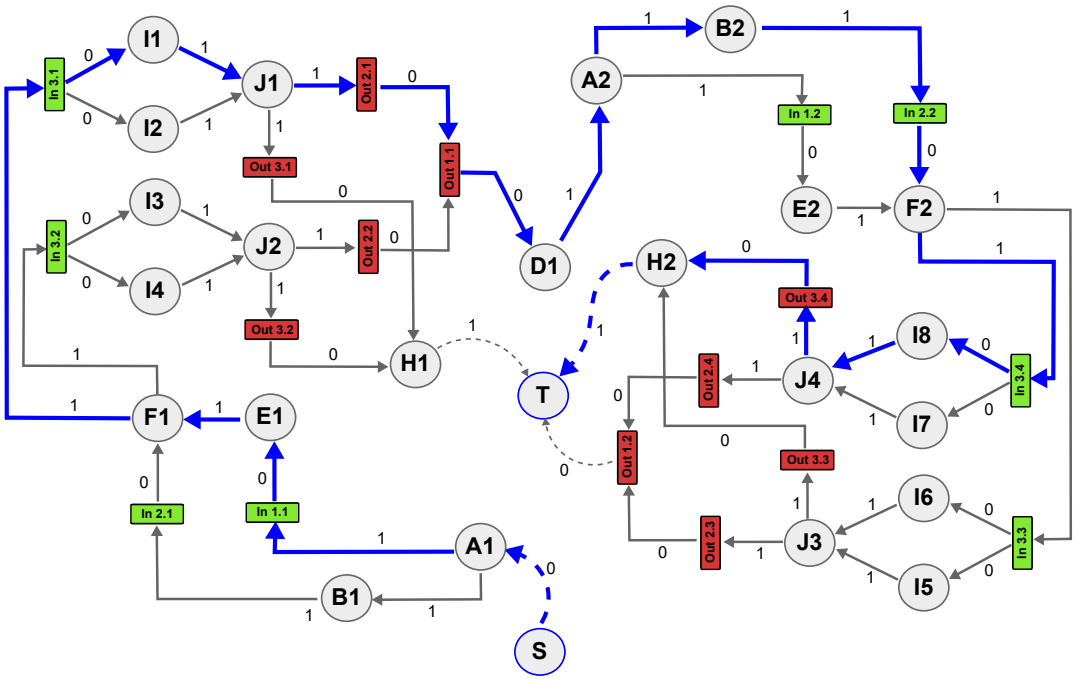


Figure 5.4 – *The longest path of the equivalent flat DAG of the IBSDF graph example of Figure 5.1a. The longest path is represented with blue colored edges and has a total length of* 12 *clock-cycles (cc).*

hierarchy of the IBSDF graph into an equivalent flat srSDF graph and then removing all the edges that contain data tokens. Figure 5.4 shows the resulted equivalent flat DAG of the IBSDF graph of figure 5.1a after adding the dummy actors S and T, and setting the length of the edges based on the duration of their source actor. The critical path of the equivalent flat DAG is the longest path from actor S to actor T, which is represented in the figure by the blue colored edges. The total duration of the critical path is 12 clock-cycles (cc), and thus the minimum achievable multi-core latency of the IBSDF graph is 12 cc, considering an architecture with an infinite number of PEs.

In graph theory, the longest path problem is known as an NP-Hard problem for general graphs [Law01] i.e., a problem for which it is hard to find an optimal solution in a reasonable amount of time. Luckily it has a linear time solution for DAG [CLRS09] using the algorithm described previously. Precisely, its overall time-complexity is: $\mathcal{O}(|A|+|F|)$ where $G_{DAG} = \langle A, F \rangle$ is the input DAG. This low time-complexity is vital for a fast evaluation of the multi-core latency using the CPM. However, the size of the resulting equivalent DAG of both SDF graph transformation and IBSDF graph flattening process may increase exponentially according to the repetition factor of the actors. This hidden exponential time and space complexity can be revealed by expressing the overall complexity of the algorithm in SDF graph and IBSDF graph terms. For the SDF graph, the overall time-complexity is $\mathcal{O}(\sum_{a \in A} RV(a) + \beta \times |F|)$ where $G_{SDF} = \langle A, F \rangle$ is the original SDF graph and β is the average duplication number of the edges in the equivalent DAG. For the IBSDF graph, the overall time-complexity is $\mathcal{O}(\sum_{G_{sub} \in G_{IBSDF}} (\prod_{p \in \mathcal{P}(G_{sub})} RV(p) \times (\sum_{a \in A^-} RV(a) + \beta \times |F|))$ where G_{IBSDF} is the original IBSDF graph, $G_{sub} = \langle A^-, F \rangle$ is an IBSDF subgraph for which A^- and F respectively represent the set of its regular actors and edges. $\mathcal{P}(G_{sub})$ represents the set of all the hierarchical parent actors of G_{sub} up to the topgraph. Lastly, β represents the average duplication number of the G_{sub} edges in its equivalent DAG.

5.3.2 Symbolic-Execution (SE)

The Symbolic-Execution (SE)-based method [GSB+07] remains a common technique to evaluate properties and metrics of signal processing applications based on simulating their behavior [Gha08]. For the latency evaluation of SDF graphs, the SE-based method consists of simulating one complete iteration of the graph and measuring its total duration. In the context of multi-core latency, an infinite number of available PEs is considered during the simulation so that each actor is executed as soon as it is ready. Therefore, by adopting an unconstrained ASAP schedule, the method is able to simulate the fastest possible execution of a dataflow graph. Thus, the resulting duration of the graph represents the minimum achievable multi-core latency of the application.

In contrast to the Critical-Path Method (CPM), the SE-based method evaluates the multi-core latency of an SDF graph without converting it to an equivalent DAG. Practically, the SE algorithm consists of the following steps:

- **Phase 1**: Initialization phase
  - **Step 1**: Identify all the actors that are ready to execute based on the initial marking of the graph.
  - **Step 2**: Calculate the maximum number of executions $n(a)$ for each ready actor a based on the number of data tokens present on its input FIFO queues. In case of an actor a without input FIFO queues, set $n(a) = RV(a)$.
  - **Step 3**: For each ready actor a, remove $n(a) \times cons(a, e_i)$ data tokens from each one of its input FIFO queue e_i. Then, set the start date of the actor executions to 0 and the finish date equals the duration of the actor. Finally, add the actor to the list of the currently running actors.
- **Phase 2**: Simulation phase
  - **Step 1**: Select the actor a which has the minimum finish date form the list of the currently running actors.
  - **Step 2**: Add $n(a) \times prod(a, e_o)$ data tokens on each one of the output FIFO queue e_o of actor a and verify if the target actor a_t of the output FIFO queue is ready to execute. If so, fire the target actor as follows:
    1. Compute the maximum number of executions $n(a_t)$ of the target actor.
    2. Remove $n(a_t) \times cons(a_t, e_i)$ data tokens from each one of the input FIFO queue e_i of the target actor.
    3. Set the start date of the target actor a_t equals to the finish date of actor a. Then, set the finish date of the target actor equals to its start date plus its duration. Finally, add the target actor to the list of the currently running actors.
  - **Step 3**: Remove actor a from the list of currently running actors. Return to the first step and select another actor from the list to repeat the process until all the actors of the graph have been executed according to their repetition factor.

Like the State-Space Exploration (SSE) method [GGS+06] introduced in the previous chapter for the throughput evaluation of SDF graphs, the SE algorithm simulates the execution of the graph symbolically without actually executing the source code of the actors. From the dataflow graph perspective, the execution of an actor is the result of respectively removing (consuming) and adding (producing) data tokens from and to all of

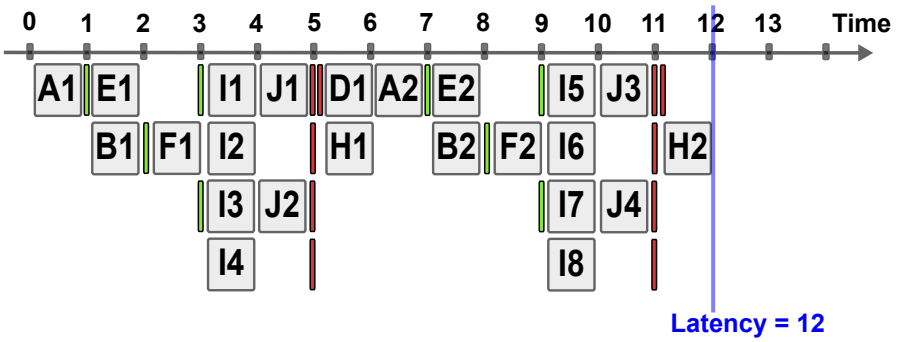


Figure 5.5 – *An unconstrained ASAP schedule of the IBSDF graph example of Figure 5.1a under the relaxed execution mode. The duration of one complete iteration of the graph is* 12 *cc.*

the input and output Fifo queues of the actor. In contrast to the SSE algorithm which simulates many iterations of the graph to measure its throughput, the SE algorithm on the other hand simulates only one iteration of the graph to measure its latency. Moreover, the simulated time of the graph execution is tracked by the start and the finish date values of the actors executions instead of simulating all the clock cycles of the graph execution. Thus, at the end of the simulation the finish date of the last executed actor represents the total duration of the graph. A quite similar implementation of the SE algorithm can be found in [BLM96b] which has an overall time-complexity of $\mathcal{O}(\sum_{a \in A} RV(a) \times f_i \times f_o + |F|)$, where $G_{sdf} = \langle A, F \rangle$ is the SDF graph and f_i (f_o) is the maximum overall actors of the number of input (output) Fifo that are incident to any actor.

Applying the SE-based method on the IBSDF graph consists of first flattening the hierarchy of the graph into either an equivalent flat srSDF graph or a flat DAG. Then, simulating one iteration of the equivalent flat graph to measure the minimum possible duration of the original IBSDF graph. In this context, the flattening process is mandatory to simulate the execution of the hierarchical actors in the relaxed execution mode that enables applications to run at full potential without any execution rule. Thus, the simulation of an unconstrained ASAP schedule of the equivalent flat graph results in executing each actor as soon as it receives data tokens from its predecessor actors. As an example, figure 5.5 shows the resulting ASAP schedule of one simulated iteration of the equivalent flat DAG of the IBSDF graph of fig. 5.1a. Based on the initial marking of the IBSDF graph, actor A is the first actor to execute, enabling its successor actors B and E to fire at the same time after it finishes the execution. Similarly, the simulation continues to fire each actor as soon as it receives data tokens from its predecessors, and stops when all the actors of the flat graph are fired once. As the resulting schedule shows, the minimum duration of one iteration of the equivalent flat DAG is 12 clock-cycles (cc). Thus the multi-core latency of the IBSDF graph of figure 5.1a is 12 cc.

Compared to the CPM, the SE algorithm dramatically reduces the space complexity of the evaluation of SDF graphs. However, it remains the same for the IBSDF graph since the hierarchy is flattened in both methods. In terms of time-complexity, the SE-based may take slightly more time than CPM because a complete iteration of the graph must be simulated to measure the total duration of the graph. Therefore, the time-complexity of both methods remains exponential for the IBSDF graph. From an end-to-end compilation perspective, the CPM saves time by finding the critical path of the graph at the same time as evaluating the latency. Which enables the identification of the critical actors of the application who may need some optimization to increase the application performance.

5.4 Multi-Core Latency Evaluation without Flattening the Hierarchy

Computing the multi-core latency of an IBSDF graph with the classical approach requires flattening the entire hierarchy into an equivalent flat graph. Such transformation often results in a large graph with an exponential number of actors and edges. As consequences, the time and space complexity of SDF state-of-the-art methods increase exponentially [DDNMK17a]. For some IBSDF graphs it is even impossible to flatten completely their hierarchy with a reasonable amount of memory, which makes the classical approach useless in those cases. Furthermore, in the context of rapid prototyping, the developer must be able to evaluate if possible in real time the multi-core latency of the application as the design process goes forward. For this purpose, the classical approach is defined again as unsuitable for fast evaluation of IBSDF graphs performance.

In this section, we introduce two new methods that take advantage of the interface-based hierarchy and evaluate the multi-core latency of IBSDF graphs without completely flattening their hierarchy. The first method, named Hierarchical-Symbolic-Execution (H-SE) computes the exact multi-core latency of an IBSDF graph in the hierarchical execution mode. The second method, named Hierarchical-Critical-Path Method (H-CPM) computes in turn the exact multi-core latency of an IBSDF graph in the relaxed execution mode. By avoiding the flattening process, both methods dramatically reduce the time and space complexity of the multi-core latency evaluation of IBSDF graphs, making them suitable for the context of rapid prototyping. In the following, we define each of Hierarchical-Symbolic-Execution (H-SE) and Hierarchical-Critical-Path Method (H-CPM) method and illustrate their algorithm on the IBSDF graph example of figure 5.1a.

5.4.1 Hierarchical-Symbolic-Execution (H-SE)

The Hierarchical-Symbolic-Execution (H-SE) method extends the symbolic execution of SDF graphs by supporting the simulation of the hierarchical execution of IBSDF graphs. The new method takes advantage of both the interface-based hierarchy of the IBSDF model and the properties of the hierarchical execution mode to simulate one complete iteration of the graph without any flattening of the hierarchy. Thus compared to the classical approach, the H-SE method dramatically reduces both space and time complexity of the IBSDF graph simulation:

- **In terms of space-complexity**: According to the execution rules of the IBSDF model, no subgraph can start to execute until its hierarchical parent actor is ready. As consequences, the hierarchical execution of an IBSDF graph always starts from the tograph. Moreover, in this context, the hierarchical actors behave exactly like the regular actors consuming and producing data tokens. Based on these two properties, the H-SE method starts by simulating the execution of the topgraph. Each time a hierarchical actor is ready to execute, the simulation launches a symbolic execution of the subgraph and waits until it finishes a complete iteration to resume the execution of the topgprah. The same simulation process is applied for each hierarchical actor in the hierarchy. Hence, a complete iteration of the entire IBSDF graph is simulated without flattening the hierarchy. Thus the space-complexity is reduced to $\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in G_{IBSDF}}(|A|+|F|))$.
- **In terms of time-complexity**: The main purpose of the H-SE method is to measure the minimum duration of the IBSDF graph to define its multi-core latency. Based on this fact, the H-SE algorithm takes advantage of the Schedule-Replace

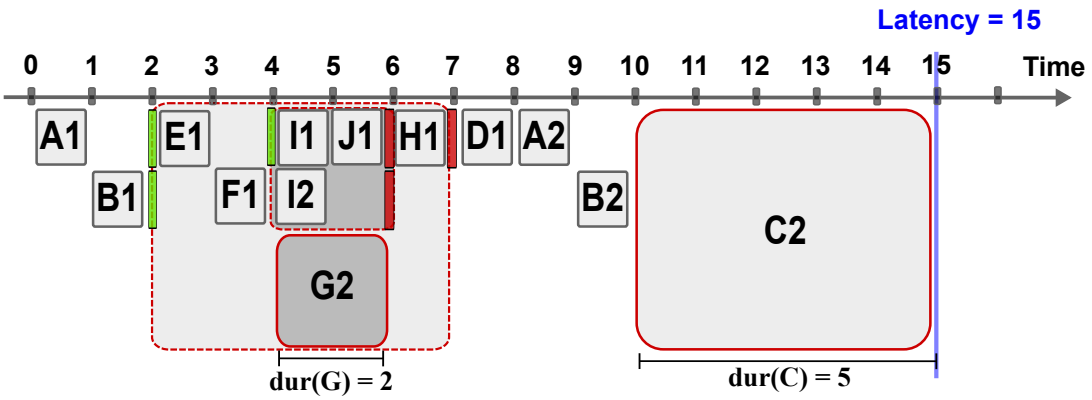


Figure 5.6 – *The resulting symbolic execution of the IBSDF graph example of Figure 5.1a under the hierarchical execution mode. The duration of one complete iteration of the graph is* 15 *cc.*

(SR) technique introduced in the previous chapter to reduce the time complexity of the simulation. In fact, simulating an unconstrained ASAP schedule of a subgraph allows measuring the minimum duration of its hierarchical parent actor. Therefore, each hierarchical actor can be considered as a regular actor during the simulation once its minimum duration was measured. This technique enables the H-SE algorithm to avoid the simulation of the same subgraph each time its hierarchical parent actor is fired. Thus, each subgraph in the hierarchy is simulated only one time during the symbolic execution of an entire iteration of the IBSDF graph. And so, the time-complexity of simulating the execution of an IBSDF graph is reduced to: $\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in IBSDF}(\sum_{a\in A} RV(a) \times f_i \times f_o + |F|))$.

Algorithm 4 represents a recursive implementation of the H-SE method, based on the Schedule-Replace (SR) technique. By applying the algorithm on the IBSDF graph example of figure 5.1a, the H-SE method starts by simulating one iteration of the subgraphs IJ and $EFGH$ to measure the minimum duration of the hierarchical actors G and C respectively. Finally, one iteration of the topgraph is simulated to measure the duration of the IBSDF graph. During the simulation, both hierarchical actors G and C are considered as regular actors to avoid simulating their subgraphs multiple times. Figure 5.6 shows the resulting symbolic execution of the IBSDF graph example in the hierarchical execution mode. As the figure shows the minimum duration of the topgraph is 15 clock-cycles (cc). Thus the multi-core latency of the IBSDF graph example in the hierarchical execution mode is 15 cc, representing a speedup by a factor of $1,5$ compared to its mono-core execution (23 cc).

Algorithm 4 Hierarchical-Symbolic-Execution (H-SE) algorithm

```
function HierarchicalSymbolicExecution(IBSDF subgraph)
    for all actors a ∈ IBSDF subgraph do
        if actor a is hierarchical then
            a.duration = HierarchicalSymbolicExecution(a.subgraph)
        end if
    end for
    latency = SymbolicExecution(IBSDF subgraph)
    return latency
end function
```

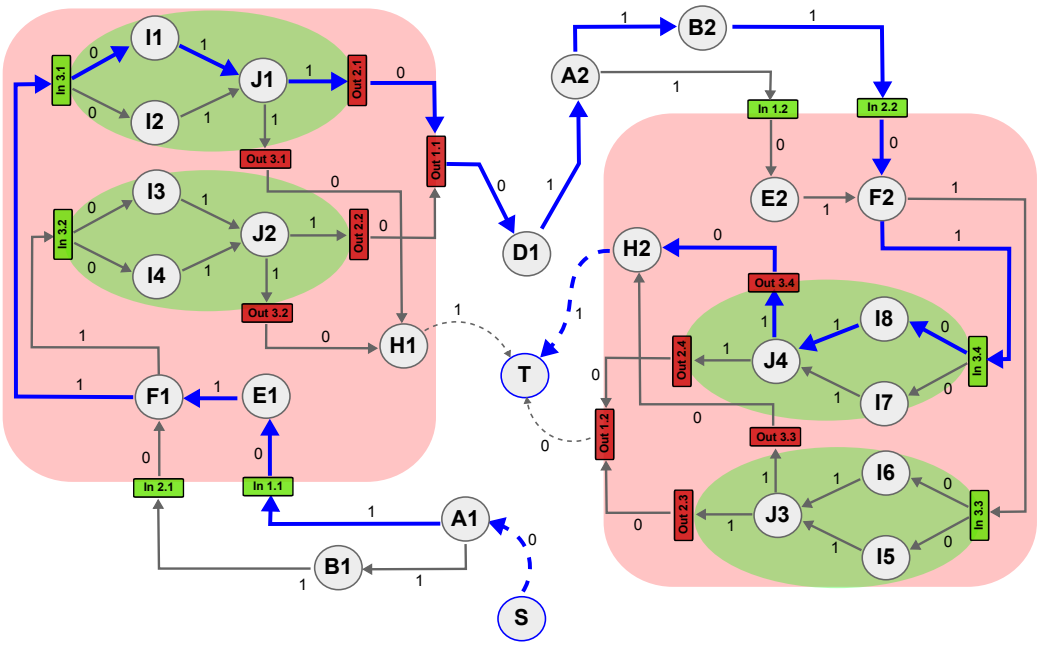


Figure 5.7 – *The equivalent flat DAG of figure 5.4 in which the similar parts are highlighted with the same color. The red and green highlighted parts correspond respectively to the subgraphs EFGH and IJ.*

Although the Hierarchical-Symbolic-Execution (H-SE) method evaluates the multi-core latency of IBSDF graphs at very low complexity, the method remains limited to the hierarchical execution mode. This limitation is due to multiple facts, the first one is that a hierarchical actor has a different behavior than a regular actor when the execution rules are ignored. As consequences, a hierarchical actor cannot be considered as a regular actor during the simulation of a relaxed execution. Therefore, the second fact is that a relaxed execution of an IBSDF graph cannot be simulated correctly without flattening the complete hierarchy of the graph. However, the H-SE method can be used as a fast approximation method for the computation of the minimum multi-core latency of IBSDF graphs under a relaxed execution mode.

5.4.2 Hierarchical-Critical-Path-Method (H-CPM)

The lack of a method to compute the exact multi-core latency of an IBSDF graph under the relaxed execution mode at low complexity has motivated us to study the idea of adapting the state-of-the-art Critical-Path Method (CPM) for the IBSDF model. By analyzing the weak points of the CPM algorithm alongside with the structure of the equivalent flat DAG and the properties of the IBSDF model, we have made the two following observations leading us to develop a new method:

- **Redundant computations**: The CPM algorithm is based on visiting all the actors and all the edges of a DAG in order to find the longest path. In the case of a DAG resulting from the flattening of an IBSDF graph, exploring the entire DAG becomes a weak point of the CPM algorithm. Indeed, visiting an exponential number of actors and edges leads to an exponential time complexity algorithm. Hence, the only way to reduce the time complexity of the CPM algorithm is by minimizing the number of actors and edges to visit. Luckily, this is possible based on our observation

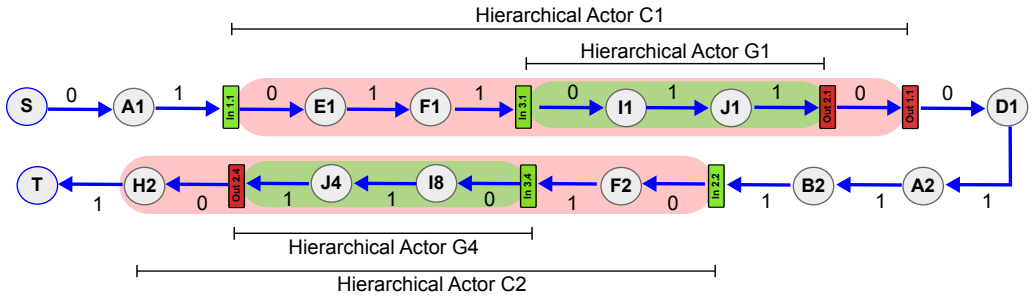
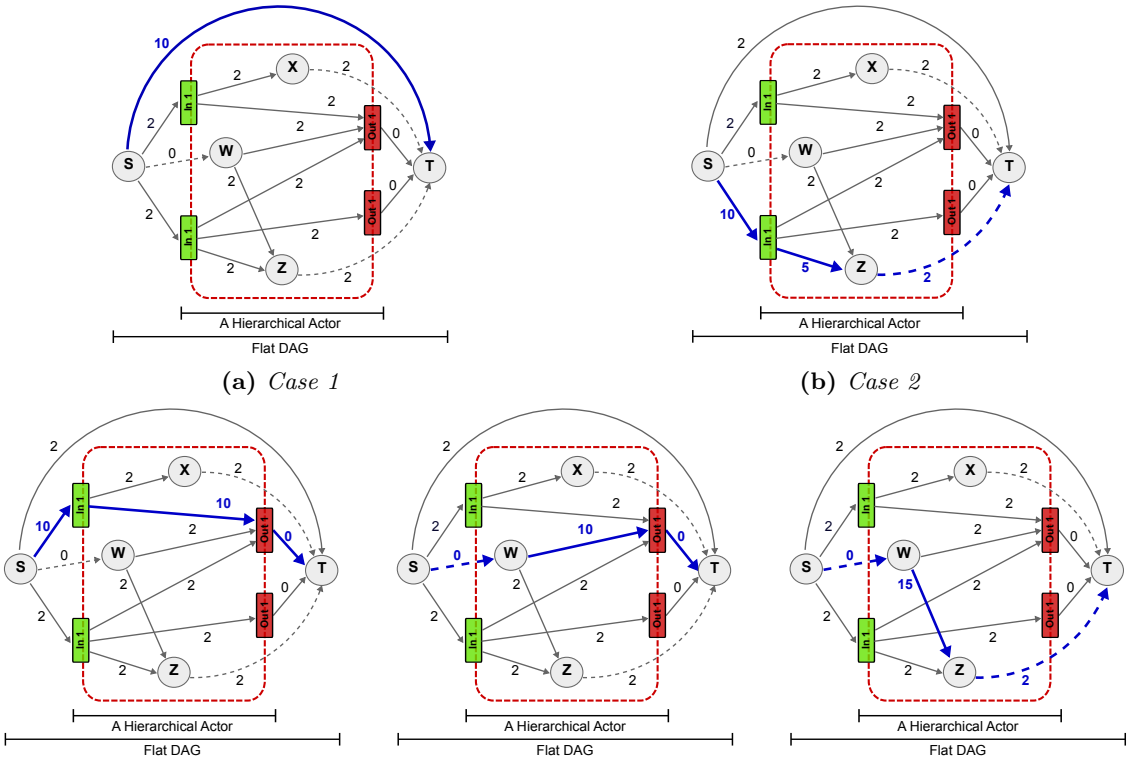


Figure 5.8 – *The composition of the critical path of the equivalent flat DAG of Figure 5.7.*

of the structure of the equivalent flat DAG. Indeed, the main process of flattening the hierarchy of an IBSDF graph is based on duplicating each subgraph multiple time according to the total repetition factor of its hierarchical parent actor. As consequences, multiple parts of the resulting flat DAG are exactly the same, which end up being explored multiple times by the CPM algorithm. Figure 5.7 shows the equivalent flat DAG of the IBSDF graph example of figure 5.1a, in which the similar parts are highlighted with the same color. The red highlighted parts of the flat DAG correspond to the duplication of the equivalent flat graph of the subgraph $EFGH$. Similarly, the green highlighted parts of the flat DAG represent the same equivalent flat graph of the subgraph IJ. By applying the CPM algorithm, both subgraphs are explored multiple times during the computation. Therefore, the exploration may be decomposed following the IBSDF subgraphs to avoid processing similar parts of the equivalent flat DAG multiple times. Hence, the number of visited actors and edges is minimized and the time-complexity of the CPM algorithm is reduced.

- **Critical-path composition**: By identifying the different parts of the equivalent flat DAG, we have been able to clearly see the composition of its critical-path from the hierarchy perspective. As figure 5.8 shows, the critical-path is a composition of multiple sub-paths from all over the hierarchy. The green and red highlighted sub-paths are respectively paths from the subgraph of the hierarchical actors G and C, while all the remaining sub-paths belong to the topgraph. Moreover, each sub-path represents the longest possible path between the input and the output interface of the hierarchical actor from which the critical-path is passing through. In general, we distinguish five possible cases for a hierarchical actor as a part of the critical-path of an IBSDF graph. Each configuration of the subgraph presented in figure 5.9 illustrates one of the five cases. Actors S and T represent the dummy actors used to compute the critical-path of the IBSDF graph. Actors X and Z of the subgraph represent actors without output edges who are directly connected to the dummy actor T. Actor W in turn, represents an actor without input edges who is directly connected to the dummy actor S. The edges in the other hand, abstract the longest path between their source and target actors. The five illustrated cases are as follows:

  - **case 1**: the critical path is composed only by regular actors without passing through a hierarchical actor. Sub-figure 5.9a for example.
  - **case 2**: the hierarchical actor constitutes the end of the critical-path. In this case, the critical-path passes into the hierarchical actor through one of its input interfaces and stops at the furthest (deeper) actor that has no output edges (actor W for example in sub-figure 5.9b).



(a) *Case 1* **(b)** *Case 2*

(c) *Case 3* **(d)** *Case 4* **(e)** *Case 5*

Figure 5.9 – *The five possible cases of a hierarchical actor as a part of the global critical-path.*

- **case 3**: the hierarchical actor represents a sub-path in the critical path. In this case, the critical-path goes into (res. goes out from) the hierarchical actor through one of its input interfaces (res. one of its output interfaces) as sub-figure 5.9c shows. The chosen sub-path between the two interfaces is the longest possible one.
- **case 4**: the hierarchical actor constitutes the beginning of the critical-path. In this case, the critical-path starts from the subgraph of the hierarchical actor with an actor that has no input edges, goes out from one of the output interfaces of the hierarchical actor and stops elsewhere in the hierarchy. Sub-figure 5.9d shows an example of case 4 where the critical path starts with actor W that has no input edges.
- **case 5**: the entire critical path is a subpath from the subgraph of the hierarchical actor. As sub-figure 5.9e shows, the critical-path starts from (resp. ends with) actor W (resp. actor Z) where the path between the two actors is the longest one in the hierarchy.

When the critical-path passes through multiple hierarchical actors, the resulting path may be a multiple combinations of these cases. From figure 5.8, we can see that the critical-path of the IBSDF graph example is a combination of cases 2 and 3. In fact, it starts by an actor from the topgraph, passes through the hierarchical actors C and G, goes back to the topgraph, and finally passes through the hierarchical actors C and G a second time where it ends with an actor from the subgraph $EFGH$ that has no output edges.

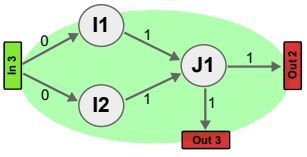
Based on these two observations, we successfully adapted the SDF Critical-Path Method (CPM) for the IBSDF model. The new method is named Hierarchical-Critical-Path Method (H-CPM) and consists on hierarchically computing the critical path of an entire IBSDF graph, without flattening its hierarchy. Thus, the H-CPM is able to evaluate at low complexity the exact multi-core latency of an IBSDF graph under a relaxed execution. In the following, we present in detail the H-CPM algorithm and illustrate it on the IBSDF graph example of figure 5.1a.

H-CPM Main algorithm

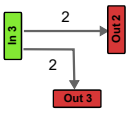
From the graph theory perspective, the H-CPM algorithm can be seen as a decomposition technique for the longest-path problem of the equivalent flat DAG. The algorithm aims to decompose the main problem into sub-problems which are much more easy to handle in terms of space and time complexity. The decomposition technique of the H-CPM algorithm follows the same schema as the Schedule-Replace (SR) technique. It consists on taking advantage of the interface-based hierarchy to evaluate the IBSDF graph level by level such that each evaluated subgraph is abstracted in the upper graph.

The first phase of the H-CPM algorithm consists on abstracting the entire hierarchy of the IBSDF graph in the topgraph. The abstraction process is based on computing the Longest-Path Equivalent Graph (LPEG) of each subgraph in the hierarchy starting from the bottom level up to the topgraph. The LPEG is a weighted graph composed only by input and output interfaces and by actors with no input or output edges. Each weighted edge of the LPEG abstracts the longest path between its source and target actors in the subgraph for which the weight represents the length of the abstracted path. The main purpose of constructing the equivalent LPEG of a subgraph is to abstract the structure of the graph in a small weighted graph with few nodes and edges, so that it is no longer needed to explore multiple times the entire subgraph to compute the longest paths between the interfaces. As an example, figure 5.10 shows the equivalent Longest-Path Equivalent Graph (LPEG) of the subgraph IJ of the IBSDF graph example of figure 5.1a. The longest path from the input interface $In1$ to each of the output interface $Out1$ and $Out2$ has a length of 2. Therefore, the two equivalent LPEG edges have a weight of 2. Hence, by replacing each instance of the hierarchical actor G with the equivalent LPEG of its subgraph IJ, the H-CPM algorithm avoids exploring the same subgraph IJ multiple times. Figure 5.11a shows the resulting equivalent DAG of the subgraph $EFGH$ after replacing the hierarchical actor G.

Furthermore, by repeating the same process at each level, all the longest sub-paths of the entire hierarchy are abstracted in the equivalent LPEG of the hierarchical actors of the topgraph. Thus, the second phase of the H-CPM algorithm consists on computing only the critical-path of the topgraph to define the multi-core latency of the entire IBSDF graph. As figure 5.11b shows, computing the LPEG of the equivalent DAG of the subgraph $EFGH$

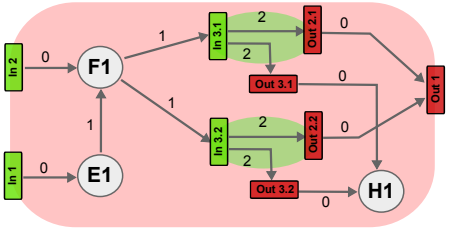


(a) *The equivalent DAG of the subgraph IJ.*

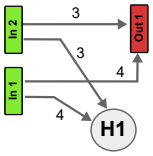


(b) *The LPEG of the subgraph IJ.*

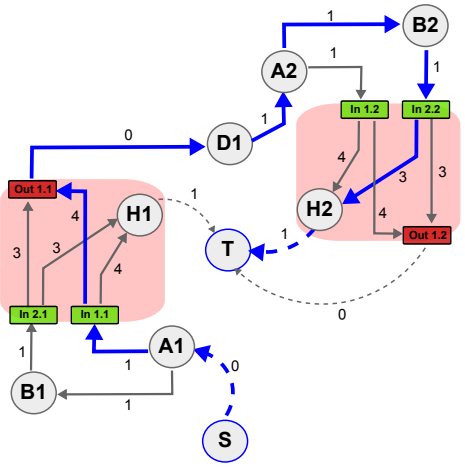
Figure 5.10 – *Abstracting the longest paths between the input and output interfaces of the subgraph IJ in the Longest-Path Equivalent Graph (LPEG).*



(a) *The equivalent DAG of the subgraph EFGH after replacing the two instances of the hierarchical actor G by the LPEG of its subgraph IJ.*



(b) *The LPEG of the subgraph EFGH which abstracts the longest sub-paths in the entire hierarchy in 4 weighted edges.*



(c) *The equivalent DAG of the topgraph ABCD after replacing the two instances of the hierarchical actor C by the LPEG of its subgraph EFGH.*

Figure 5.11 – *An illustration of the H-CPM algorithm on the IBSDF graph example of Fig. 5.1a.*

allows to abstract the entire hierarchy in a small graph of 4 weighted edges, which represent all the possible sub-paths in the hierarchy including the subpath that ends with actor H that has no output edges. Then, replacing the hierarchical actor C of the topgraph allows computing the critical path of the entire IBSDF graph by only exploring the equivalent DAG of the topgraph $ABCD$. As figure 5.11c shows, the computed critical path is the same as the one computed with the classical approach in figure 5.7, except that the sub-paths are abstracted by weighted edges. At this stage, the multi-core latency of the IBSDF graph has been evaluated. The final phase of the H-CPM algorithm aims to retrieve the uncompressed form of the computed critical-path. The uncompressed form is obtained by repetitively replacing each LPEG edge in the critical-path by the original sub-path until the complete critical-path is unfolded. The following steps resume the H-CPM algorithm:

- **Phase 1**: Starting from the bottom level of the hierarchy up to the topgraph:
  - **Step 1**: *Construct* the equivalent LPEG of the current subgraphs.
  - **Step 2**: Move to the upper graph and *Replace* the hierarchical parent actors of the current subgraphs with their equivalent LPEG.
  - **Step 3**: Repeat *steps 1* and *step 2* until the topgraph is reached.
- **Phase 2**: Compute the critical-path of the equivalent DAG of the topgraph to define the multi-core latency of the entire hierarchical graph.
- **Phase 3**: Unfold the compressed critical-path to retrieve its original form.

Compared to the classical approach, each subgraph of the IBSDF graph example is explored once during the entire exploration. Moreover, at each iteration of the H-CPM algorithm, a subgraph is removed from the memory once its equivalent LPEG is constructed. Thus, the H-CPM algorithm is able to reduce both time and space complexity of the multi-core latency evaluation of IBSDF graphs. Precisely, the space-complexity is reduced from the size of the entire equivalent flat DAG of an IBSDF graph, down to the

size of the equivalent DAG of only the largest subgraph in the hierarchy plus the size of the LPEG of its hierarchical actors. The time-complexity in turn, is reduced from the complexity of exploring the entire equivalent flat DAG where the subgraphs are duplicated many times, down to the complexity of constructing the equivalent LPEG of each subgraph in the hierarchy. In the following, we present how to construct the equivalent LPEG and discuss its space and time-complexity.

Constructing the LPEG

The main purpose of the Longest-Path Equivalent Graph (LPEG) is to abstract the structure of an IBSDF subgraph with a small weighted graph. Thus in the upper graph, the hierarchical parent actor of the abstracted subgraph can be replaced without increasing exponentially the total number of actors and edges. For this purpose, the LPEG represents only the longest sub-paths between the input and the output interfaces of the subgraph, which are relevant to the critical-path computation. Each longest sub-path between two interfaces is abstracted in the LPEG via an edge with a weight equal to the total length of the abstracted sub-path. Thanks to the properties of the interface-based hierarchy of the IBSDF model, the LPEG of each subgraph can be constructed independently from its upper levels. Based on that, the LPEG of each IBSDF subgraph is constructed as follows:

- **Phase 1**: Convert the subgraph into an equivalent DAG and replace its hierarchical actors by their equivalent LPEG which have been constructed before. Create a new empty LPEG and add to it all the input and output interfaces of the subgraph.
- **Phase 2**: For each input interface of the equivalent DAG:
  - **Step 1**: Compute the longest distance from the input interface to all the remaining nodes of the equivalent DAG.
  - **Step 2**: Add a new edge between the input interface and each output interface in the new LPEG, such that the weight of the edges is equal to the longest distance between the two interfaces.
  - **Step 3**: If the furthest node from the input interface is an actor with no output edges then, add the actor to the new LPEG and connect it to the input interface through a weighed edge with the same distance.
- **Phase 3**: For each output interface of the equivalent DAG:
  - **Step 1**: Compute the longest distance from the output interface back to all the remaining nodes of the equivalent DAG.
  - **Step 2**: If the furthest node from the output interface is an actor with no input edges then, add the actor to the new LPEG and connect it to the output interface through a weighed edge with the same distance.
- **Phase 4**: Compute the local critical-path of the equivalent DAG. If the local critical-path is a path from an actor with no input edges to an actor with no output edges then, save it for the latency evaluation to cover *case 5* of sub-section 5.4.2.

The size of the equivalent LPEG remains small compared to the size of the equivalent DAG of a subgraph. In the worst case scenario, the LPEG is composed of interfaces such that each interface is connected to a distinct actor with no input or output edges. So, for an IBSDF graph where f_i (resp. f_o) is the maximum number of input interfaces (resp. output interfaces) of a hierarchical actor, the size of the largest LPEG will be $2 \times (f_i + f_o)$ nodes and $f_i \times f_o + (f_i + f_o)$ edges. By considering $f_i = f_o = f$, the size of the largest equivalent LPEG of a subgraph in the worst case is $4 \times f$ nodes and $(f + 1)^2 - 1$ edges.

Constructing the equivalent LPEG of a subgraph requires exploring many times its equivalent DAG. In fact, for each interface, the algorithm explores the equivalent DAG to computes the longest distance from the interface to all the remaining nodes, plus one last exploration of the graph to compute its local critical-path. Computing the longest distance from a node to all its connected nodes in a DAG is based on the same algorithm as Critical-Path Method (CPM) which has a linear time-complexity of $\mathcal{O}(|V|+|E|)$, where V is the set of the nodes and E is the set of the edges of the DAG [CLRS09]. Thus, for an IBSDF subgraph $G_{sub} = \langle A, F \rangle$ with f_i input interfaces and f_o output interfaces, the time-complexity of the LPEG construction is $\mathcal{O}((f_i + f_o + 1) \times (\sum_{a \in A} RV(a) + \beta \times |F|))$ where β is the average duplication number of the edges in the equivalent DAG.

Therefore, for an IBSDF graph of n_s subgraphs, the overall time-complexity of the H-CPM algorithm is $\mathcal{O}(n_s \times (f_i + f_o + 1) \times (\sum_{a \in A} RV(a) + \beta \times |F|))$, where f_i (resp. f_o) is the maximum number of input interfaces (resp. output interfaces) of a subgraph and $\sum_{a \in A} RV(a) + \beta \times |F|$ is the size of the equivalent DAG of the largest IBSDF subgraph.

In terms of space-complexity, at each iteration the H-CPM algorithm process the equivalent DAG of a subgraph in which the hierarchical actors are replaced by the equivalent LPEG of their subgraph. Hence, for an IBSDF graph where f_i (resp. f_o) is the maximum number of input interfaces (resp. output interfaces) of a subgraph and n_h is the maximum number of hierarchical actors in a subgraph, the space complexity in the worst case is the size of the equivalent DAG of the largest subgraph in the hierarchy plus n_h times the maximum size of an LPEG: $\mathcal{O}(\sum_{a \in A} RV(a) + \beta \times |F|) + n_h \times (f_i + f_o + 3 \times (f_i + f_o)))$

Tables 5.2 and 5.3 compare the time and space complexity of the new methods with the classical approaches. As the tables show, the high complexity of the classical approaches comes mainly form the product $\prod_{p \in \mathcal{P}(G_{sub})} RV(p)$ which represents the total duplication number of a subgraph G_{sub} in the entire flat DAG of the IBSDF graph according to the repetition factor of all its hierarchical parent actors $\mathcal{P}(G_{sub})$.

Table 5.2 – *The time-complexity of the multi-core latency evaluation methods.*

Method	Time-complexity
Classical-methods	
CPM-based	$\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in G_{IBSDF}} (\prod_{p \in \mathcal{P}(G_{sub})} RV(p) \times (\sum_{a \in A} RV(a) + \beta \times \lvert F\rvert))$
SE-based	$\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in G_{IBSDF}} (\prod_{p \in \mathcal{P}(G_{sub})} RV(p) \times (\sum_{a \in A} RV(a) \times f_i \times f_o + \lvert F\rvert))$
New methods	
H-SE	$\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in G_{IBSDF}} (\sum_{a \in A} RV(a) \times f_i \times f_o + \lvert F\rvert))$
H-CPM	$\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in G_{IBSDF}} ((f_i \times f_o + 1) \times (\sum_{a \in A} RV(a) + \beta \times \lvert F\rvert)))$

Table 5.3 – *The space-complexity of the multi-core latency evaluation methods.*

Method	Space-complexity
Classical-methods	
CPM-based	$\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in G_{IBSDF}} (\prod_{p \in \mathcal{P}(G_{sub})} RV(p) \times (\sum_{a \in A} RV(a) + \beta \times \lvert F\rvert))$
SE-based	$\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in G_{IBSDF}} (\prod_{p \in \mathcal{P}(G_{sub})} RV(p) \times (\sum_{a \in A} RV(a) + \beta \times \lvert F\rvert))$
New methods	
H-SE	$\mathcal{O}(\sum_{G_{sub}=\langle A,F\rangle \in G_{IBSDF}} (\lvert A\rvert + \lvert F\rvert))$
H-CPM	$\mathcal{O}(\sum_{a \in A} RV(a) + \beta \times \lvert F\rvert) + n_h \times (f_i + f_o + 3 \times (f_i + f_o)))$

Table 5.4 – *Description of the benchmark set.*

IBSDF graph				Subgraphs details (average value)						
Name	Levels	Actors	subgraphs	Actors	f_i	f_o	RV	$TotalRV$	FIFOs	β
Crypto	2	13	2	7	1	1	2	3	8	2
Large FFT	2	13	2	7	1	1	15	15	6	49
LTE	4	28	5	6	1	1	3	11	5	4
Stereo Match.	2	50	4	13	1	1	56	56	14	97
Graph 1	3	22	3	8	1	1	8	38	7	9
Graph 2	5	29	5	6	2	2	5	972	7	8
Graph 3	6	35	6	6	2	2	6	4060	7	10
Graph 4	5	166	5	34	2	2	25	4469	48	38
Graph 5	8	266	8	33	2	2	38	10×10^8	47	57
Graph 6	10	131	10	13	2	2	406	7×10^{15}	16	442

5.5 Experimental Results

5.5.1 Experimental Setup

In this section, we confirm our theoretical study by comparing the performance of the the classical approaches Critical-Path Method (CPM) and Symbolic-Execution (SE) with the new developed methods Hierarchical-Symbolic-Execution (H-SE) and Hierarchical-Critical-Path Method (H-CPM). The numerical experiments consists on measuring the running time and memory usage for each method to compute the multi-core latency of each IBSDF graph of the benchmark set. The measured running time includes the time for the flattening process or DAG conversions when relevant. The benchmark set used in this numerical experiments is the same one used in the experiments of the previous chapter, which is composed of two categories of IBSDF graphs. The first category is the set of real DSP applications modeled as IBSDF graphs for which a source code is available in [Pre]. The second category is the set of the synthetics IBSDF graphs which are randomly generated using the IBSDF graph generator that is based on Turbine tool [BLDMK14]. The second category is used mainly to complete the benchmark set of real applications with large IBSDF graphs in order to provide an extensive performance comparison. In fact, till today the number of the hierarchical levels in real applications remains small due to the hardness of compiling large graphs using the classical approaches. Table 5.4 shows the characteristics of the two categories of the benchmarks set: the real applications (Crypto, Large FFT, LTE [PAPN13], and Stereo Matching [Heu15]) and the synthetic graphs (Graph 1 to 6). The first three columns represent respectively the name, the number of hierarchical levels, the total number of actors, and the total number of subgraphs for each evaluated IBSDF graph. The rest of the columns represent the characteristics of the subgraphs. Each column show the average value overall the subgraphs of the hierarchy including the topgraph. The displayed characteristics are the number of actors, the maximum number of input (f_i) and output (f_o) Fifo queues of the actors, the repetition factor (RV) and total repetition factor ($TotalRV$) of the actors. Lastly, the number of edges and their duplication number in the equivalent DAG of the subgraph. As concerns the actors duration, it was generated randomly for the synthetic actors and measured for the real ones. The measurements was done using the automated measurement module of the open-source Preesm framework [PDH+14], which consists on running the source code of the actors multiple times to estimate their Worst-Case Execution Time (WCET). The

Table 5.5 – *Performance comparison between the classical approaches Flat-CPM and Flat-SE, and the new methods H-SE and H-CPM.*

IBSDF graph	Flat-CPM	Flat-SE	H-SE		H-CPM
	Exec.Time	Exec.Time	Appr./Exact	Exec.Time	Exec.Time
Crypto	0.65 ms	1.15 ms	1.00	0.50 ms	**0.50 ms**
Large FFT	4.66 ms	6.11 ms	1.00	0.50 ms	**3.73 ms**
LTE	2.52 ms	2.85 ms	1.00	0.50 ms	**1.49 ms**
Stereo Matching	18.20 ms	23.56 ms	1.00	0.87 ms	**15.00 ms**
Graph 1	2.40 ms	3.32 ms	1.80	0.93 ms	**1.83 ms**
Graph 2	108.16 ms	134.00 ms	1.76	0.99 ms	**3.91 ms**
Graph 3	441.40 ms	415.55 ms	3.73	1.00 ms	**5.11 ms**
Graph 4	6.25 s	6.64 s	7.7	1.20 ms	**48.14 ms**
Graph 5	-	-	5.5	4.74 ms	**83.32 ms**
Graph 6	-	-	8×10^6	12.81 ms	**308.60 ms**
Max. Memory usage	**1,4 GB**	**1,4 GB**	**11 MB**		**163 MB**

algorithm implementation of the new methods was integrated to Preesm framework as well as the implementation of the classical approaches. All the methods are written in Java language and were tested on one core of an Intel i5-6300 processor clocked at 2.40 GHz, with 8GB of RAM. For an accurate experimental results, each method was tested 100 times for each IBSDF graph of the benchmark set.

5.5.2 Results

In the experiments, all the IBSDF graphs were evaluated under the relaxed execution mode to compute the minimum possible value of their multi-core latency. The exact minimum value computed by each of the classical approaches and the H-CPM is used to evaluate the quality of the approximation value computed with the H-SE technique. Table 5.5 summarizes the experimental results. Each column of the table represents the average execution time of each method for each IBSDF graph of the benchmark set. The Flat-CPM and Flat-SE refer respectively to the CPM-based and SE-based classical approaches. For the H-SE method, a second column shows the ratio between the approximation and the exact value of the multi-core latency. This ratio indicates also the speedup of the applications when changing the execution mode from hierarchical to relaxed.

From the results, we observe that the two new methods H-CPM and H-SE are faster than the classical approaches Flat-CPM and Flat-SE. Although H-SE technique is the fastest one among the four methods, the minimum multi-core latency can be over estimated up to 8×10^6 times the exact value. In terms of accuracy, the results demonstrate the efficiency of the H-CPM algorithm in computing the exact multi-core latency of the graphs, even if its time-complexity is not polynomial. For example, the H-CPM took only 48.2 millisecond to evaluate the exact multi-core latency of *Graph 4* and 0.3 seconds for *Graph 6*, while the two classical approaches took more than 6 seconds to evaluate *Graph 4* and failed for *Graph 6*. On the other hand, the H-SE method took only 1.2 millisecond to evaluate *Graph 4* and 12.8 milliseconds for *Graph 6*, but the multi-core latency was over estimated by 7.7 times the exact value for *Graph 4* and 8×10^6 times for *Graph 6*.

Since the classical approaches rely on the flattening process, they have failed to evaluate the multi-core latency of *Graph 5* and *Graph 6*. Indeed, it was not possible to flatten the hierarchy of these two last IBSDF graphs with the available RAM memory. Table 5.6

Table 5.6 – *The size of the equivalent DAG of each of the entire IBSDF graph and the largest subgraph in the hierarchy.*

IBSDF graph	The equivalent DAG of the entire IBSDF graph				The equivalent DAG of the largest subgraph			
Name	Actors	FIFOs	f_i	f_o	Actors	FIFOs	f_i	f_o
Crypto	34	40	2	4	10	12	2	2
Large FFT	267	776	256	256	264	773	3	1
LTE	272	336	22	22	54	98	2	1
Stereo Matching	1606	3140	380	380	1143	1901	2	2
Graph 1	513	813	5	8	78	120	2	1
Graph 2	17730	48723	15	15	38	122	3	2
Graph 3	87440	193579	15	15	62	147	3	2
Graph 4	654566	2030272	119	45	1535	4783	4	2
Graph 5*	39×10^{10}	78×10^{10}	119	561	2552	6653	3	2
Graph 6*	31×10^{18}	79×10^{18}	1190	1190	29753	46405	2	2

shows the actual size of the resulting equivalent flat DAG of each IBSDF graph of the benchmark set. For *Graphs 5* and *6*, the size of their equivalent flat DAG was only estimated based on the repetition factor RV and the total repetition factor $TotalRV$ of their actors. From the estimated number of actors and edges, we assume that it will require more than 100 petabytes of RAM memory to completely flatten *Graphs 5* and *6*. We also observe that the size of the equivalent flat DAG grows exponentially as the number of hierarchical levels grows. This explains why the execution time of the classical approaches Flat-CPM and Flat-SE in table 5.5 increases exponentially as the number of hierarchical levels grows. In contrast, the new methods H-SE and H-CPM have been able to evaluate all the benchmark set in less than a half second. The time efficiency of both methods goes back to the decomposition technique that reduces the time and space complexity of the evaluation. Indeed, each of H-SE and H-CPM evaluates only one subgraph per iteration instead of evaluating the entire equivalent flat DAG once. The second part of table 5.6 shows the size of the equivalent DAG of the largest subgraph of each IBSDF graph of the benchmark set. As the table shows, the size of the equivalent DAG of a subgraph to evaluate at each iteration remains very small compared to the size of the equivalent flat DAG of the entire IBSDF graph. This also explains the low memory usage of the new methods during the numerical experiments. The last row of table 5.5 shows the maximum memory usage of each method during the latency evaluation of the benchmark set. The maximum memory usage of the classical approaches is about $1,4$ Gigabytes (GB) of RAM memory. This value corresponds to the amount of RAM memory used to flatten the entire hierarchy of the large synthetic IBSDF graph 4. In the other hand, the H-SE method has consumed 11 Megabytes (MB) of RAM memory to evaluate the benchmark set. This low memory usage is due to the fact that the H-SE method simulates the execution of the entire IBSDF graph without any DAG conversion at any level. Since the H-CPM converts one subgraph to an equivalent DAG at each iteration, it consumed 163 Megabytes (MB) of RAM memory to evaluate the benchmark set, which is more than the H-SE method but much more less than the classical approaches.

Figure 5.12 shows for each method, the percentage of the total execution time spent on each of DAG conversion phases and computation phases for each IBSDF graph. From the first two charts, we observe that the flattening phase takes between 38% and 76% of the total execution time of the classical approaches. This confirms that flattening the

hierarchy can be a long process that takes more time than the computation phase itself. For H-CPM the execution time spent on DAG conversions is reduced to 47%, down to 12% of the total execution time. The H-SE in the other hand, spends 100% of the total execution time in the computation phase since it does not require any DAG conversion.

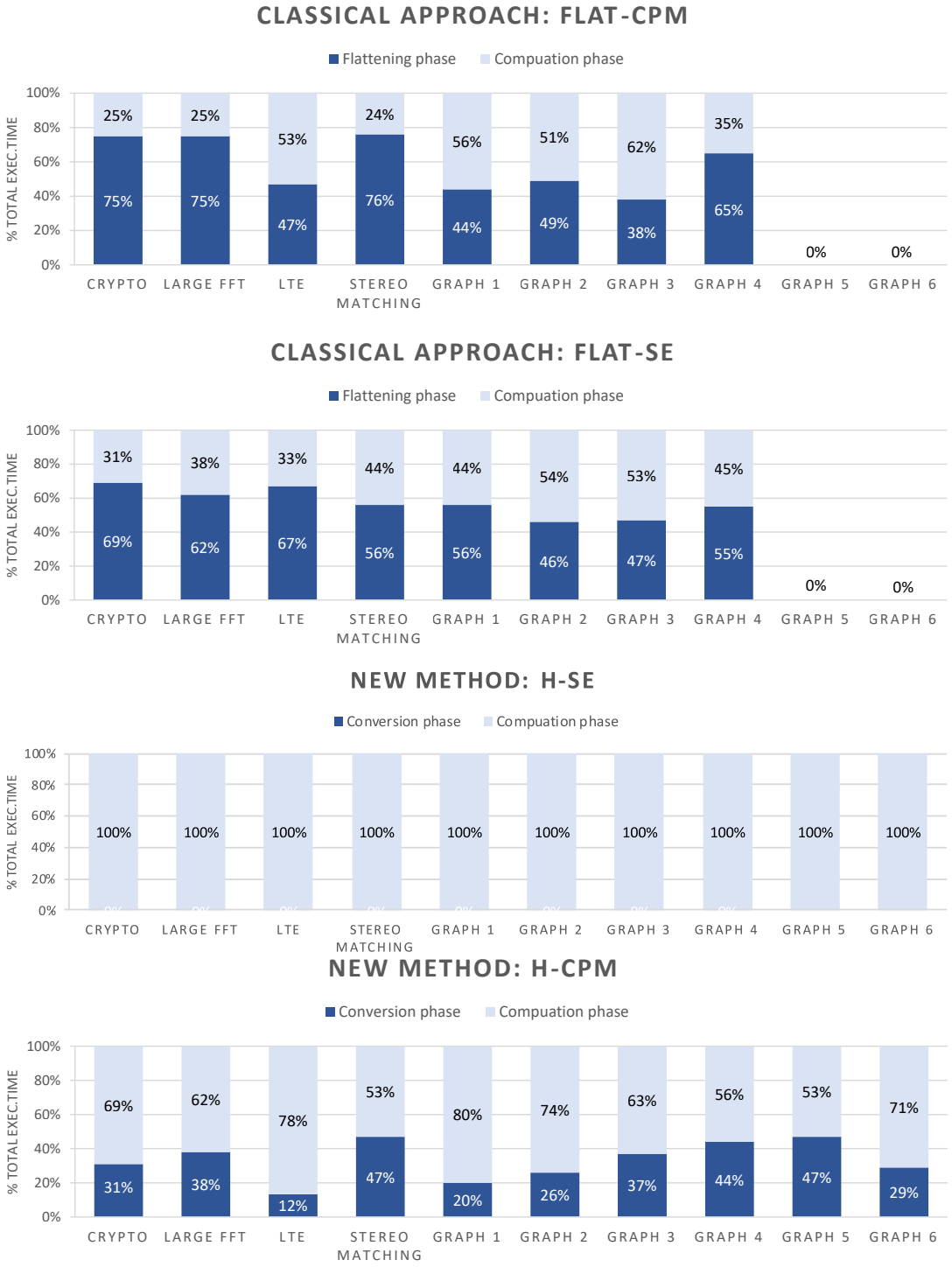


Figure 5.12 – *The percentage of the total execution time spent on each of the conversion phase and the computation phase of each of Flat-CPM, Flat-SE, H-SE, and H-CPM methods.*

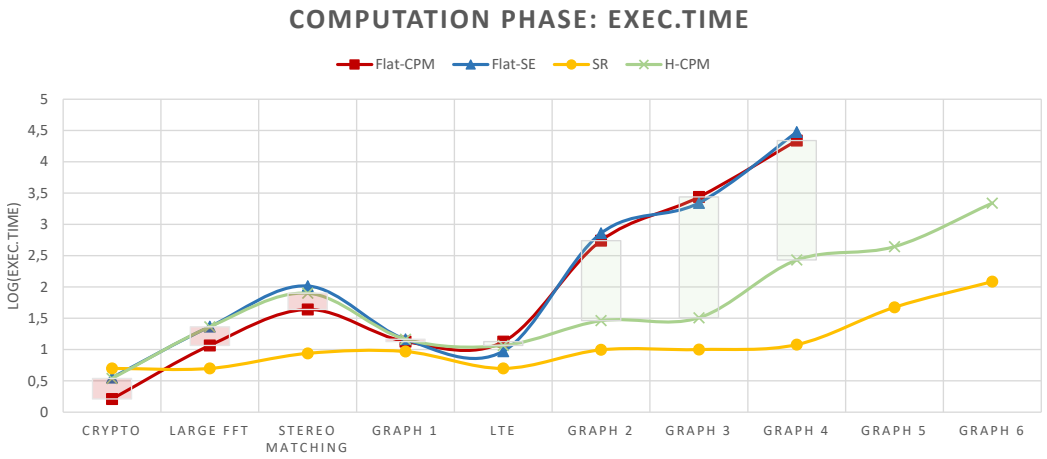


Figure 5.13 – *A comparison between the execution time of the computation phase of each of Flat-CPM, Flat-SE, H-SE, and H-CPM methods.*

Table 5.7 – *The computation phase speedup and the overall speedup of the HCPM over the classical approaches Flat-CPM and Flat-SE.*

IBSDF graph	Computation phase speedup of the H-CPM compared to		Overall speedup of the H-CPM compared to	
	Flat-CPM	Flat-SE	Flat-CPM	Flat-SE
Crypto	0.47	1.03	1.30	2.30
Large FFT	0.50	1.00	1.25	1.64
Stereo Matching	0.55	1.30	1.21	1.57
Graph 1	0.92	1.00	1.31	1.81
LTE	1.15	0.81	1.69	1.91
Graph 2	19.06	25.01	27.66	34.27
Graph 3	85.01	68.41	86.38	81.32
Graph 4	81.14	110.84	129.83	137.93

For a fair comparison, the chart of figure 5.13 compares only the execution time of the computation phase of each method. Due to the large difference between the original values of the execution time, we show in this chart the logarithm value to reduce that large difference. From the displayed chart, we can see that the computation phase of the classical approach Flat-CPM is faster than the one of H-CPM for the IBSDF graphs with a small number of hierarchical levels like *Crypto*, *Large FFT*, and *Stereo Matching* applications. This can be explained by the fact that the equivalent flat DAG of these graphs is relatively small so that computing the longest path in one iteration is faster than constructing it in a hierarchical way in which an LPEG must be constructed for each subgraph. However, as the number of levels grows, the H-CPM becomes significantly faster than classical approaches. Despite of the fact that the H-SE method is an approximation technique, it remains the fastest method among the four methods.

Table-5.7 summarizes the computation phase speedup and the overall speedup of the H-CPM over the two classical approaches Flat-CPM and Flat-SE. As the table shows, the speedup of the H-CPM increases significantly as the number of levels grows. For *Graph 4*, the H-CPM is 130 times faster than the two classical approaches on the overall execution, and 81 times faster on the computation phase.

5.6 Conclusion

In this chapter we have presented how to evaluate the latency of an IBSDF graph despite the specifications of the targeted architecture. We have first demonstrated how to compute the mono-core latency of an IBSDF graph, which is mandatory for the speedup calculation. Based on the properties of the interface-based hierarchy of the IBSDF model, we have been able to hierarchically compute and analyze the mono-core latency of IBSDF graphs. Indeed, each subgraph of an IBSDF graph is insulated in terms of data-dependency, which allows to analyze its structure and behavior independently from its upper levels. As a result, we defined the maximum duration of each hierarchical actor in the hierarchy by computing the mono-core latency of its subgraph. Using this technique in a bottom-up approach, we computed the mono-core latency of the entire IBSDF graph in a hierarchical way. Moreover, based on this hierarchical evaluation and based on a new parameter named the total repetition factor, we have been able to compute the contribution percentage of each actor in the hierarchy to the total mono-core latency of the IBSDF graph. By doing that, we answered questions like: how the mono-core latency is distributed among the hierarchy ? which actors are critical to the performance of the application ?

Next, we studied the minimum achievable latency of the IBSDF graph which represents the minimum multi-core latency of the graph when it is running on an architecture with unlimited resources. Based on the state-of-the-art methods of the SDF model, we presented the classical approach that evaluates the IBSDF graph as if it was a large SDF graph. The method consists on flattening the hierarchy of the IBSDF graph into an equivalent flat graph and then evaluate its multi-core latency using SDF methods. As simple as it is, the approach is considered as inefficient in the context of rapid prototyping. Indeed, flattening the hierarchy of an IBSDF graph results in an exponential growth of both actors and edges number of the equivalent flat graph. As consequences, the exponential space-complexity of the flattening process makes the latency evaluation a hard task for the state-of-the-art methods of the SDF model. Furthermore, some IBSDF graphs cannot be flattened due to the exponential size of the resulted graph, which simply makes SDF methods fail to return a result.

In this context where the classical approach fails to evaluate the latency of IBSDF graphs, we have developed two new methods to compute the multi-core latency of the IBSDF graph at low complexity. Both methods take advantage of the interface-based hierarchy to evaluate the IBSDF graph without flattening its hierarchy. The first method named Hierarchical-Symbolic-Execution (H-SE) adapts the state-of-the-art method Symbolic-Execution (SE) of the SDF model for the IBSDF model. The H-SE method evaluates the multi-core latency of the IBSDF graph in the hierarchical execution mode. The method simulates hierarchically the fastest execution of the IBSDF graph to measure the minimum duration of its entire iteration. So, instead of simulating the entire iteration of the equivalent flat graph as the SE-based classical approach does, the H-SE starts by simulating the multi-core execution of each subgraph to define the minimum duration of its hierarchical parent actor. Thus, in a bottom-up approach, the duration of the entire hierarchy is abstracted by the minimum duration of the hierarchical actors of the topgraph. As a result, simulating only one iteration of the topgraph in which the hierarchical actors are considered as regular actors with a minimum duration, allows to measure the duration of the entire hierarchical graph. Thus the H-SE method is able to evaluate the multi-core latency of the IBSDF graph in the hierarchical execution mode without any flattening of the hierarchy.

The second method named Hierarchical-Critical-Path Method (H-CPM), adapts the state-of-the-art Critical-Path Method (CPM) of the SDF model for the IBSDF model. The H-CPM consists on evaluating the multi-core latency of the IBSDF graph in the relaxed execution mode, which represents the minimum achievable latency of the application regardless the unlimited resources that an architecture can have. Compared to the CPM-based classical approach that computes the critical-path of the entire equivalent flat graph, the H-CPM constructs hierarchically the critical-path of the IBSDF graph without flattening completely its hierarchy. To do so, we have first demonstrated that the critical-path of an IBSDF graph is a composition of multiple sub-paths from the IBSDF subgraphs. Then, we have introduced a new graph model named Longest-Path Equivalent Graph (LPEG) that abstracts the longest path between each input and output interface of an IBSDF subgraph through weighted edges. By using this abstraction model in a bottom-up approach, the H-CPM abstracts with a set of equivalent LPEG all the possible subpaths in the hierarchy that may compose the critical path of the IBSDF graph. As a result, computing only the critical-path of the topgraph in which the hierarchical actors are replaced with the equivalent LPEG of their subgraph, allows to evaluate the critical-path of the entire IBSDF graph. Thus the H-CPM is able to evaluate the multi-core latency of the IBSDF graph in the relaxed execution mode without flattening its hierarchy or exploring the entire equivalent flat graph.

Finally, to prove the efficiency of our new low-complexity methods, we have compared their performance with the classical approaches on a benchmark set composed of real and synthetic IBSDF graphs. The experimental results have shown that the flattening process of the classical approach may take up to 70% of the total execution time of the evaluation. Moreover, the classical approach have failed to evaluate the multi-core latency of some synthetic IBSDF graphs due to the exponential size of their equivalent flat graph. At the other hand, the new methods H-SE and H-CPM have been able to evaluate all the benchmark set since they avoid the flattening process. The H-SE method in particular has demonstrated to be extremely fast in computing the multi-core latency of IBSDF graphs in the hierarchical execution mode. The H-SE method can also be used as an approximation method for the relaxed execution mode of the IBSDF graph. However, the experimental results have shown that the approximation value can be up to 8×10^6 times more than the exact value of the multi-core latency of an IBSDF graph in the relaxed execution mode. The H-CPM in turn, has shown to be also faster than the classical approach in evaluating the latency of IBSDF graphs under the relaxed execution mode. Indeed, the new method has computed the exact value of all the benchmark set in less than a half second.

As a conclusion for this chapter, we have developed new efficient methods for the latency evaluation of IBSDF graphs, which are suitable for the context of rapid prototyping. Indeed, very fast evaluation of this metric enables the developer to validate the design of the application before reaching the final phases of the development process. Moreover, evaluating the latency in a hierarchical way has enabled a better analysis of the hierarchy which was not possible with the classical approach. Furthermore, the new methods can be included into Design Space Exploration (DSE) methods [KAL11, FS14] to evaluate πSDF graphs [DPN+13]; a parametric extension of the IBSDF model. A πSDF graph can have multiple configurations depending on all the possible values of its parameters. Hence, fast evaluation methods are mandatory in order to explore the latency of each possible configuration of a πSDF graph.

CHAPTER 6

Conclusion

6.1 Summary

The complexity of Digital Signal Processor (DSP) applications has been increasing exponentially over the last twenty years. In parallel, the complexity of Multiprocessor System-on-Chip (MPSoC) architectures is increasing exponentially to meet the rising computation power demand. Modern MPSoC architectures like the many-cores architectures, already embed hundreds of Processing Element (PE) in one single chip, and plan to integrate up to thousand PE in the near future. Thus, new programming models and languages must be found to exploit all the parallel processing power of these new devices.

Dataflow Model of Computation (MoC) are one of the emerging programming models, designed for the development of complex signal processing applications for MPSoC architectures. Dataflow programming has gained popularity over the years as a simple model which naturally expresses the parallelism of an application. In fact, a dataflow model decomposes the computations of an application into a graph of tasks called actors. Thus, a simple analysis of the graph allows to determine automatically which tasks could be executed in parallel. As a promising programming model, more dataflow MoC are being proposed, each one extends the expressivity of the previous one.

The Interface-Based SDF (IBSDF) MoC is one of the recent extensions of the classical Synchronous Dataflow (SDF) model. The IBSDF model is a compositional model which enables the developer to decompose an application into a set of modules organized in a hierarchy. Each module has its own actors which communicate with other modules via interfaces. In fact, the interface-based hierarchy of the IBSDF model enables the design and the analysis of each module (subgraph) independently from the hierarchy.

The development process of signal processing applications modeled with IBSDF graphs consists of first verifying the ability of the graph to run on an architecture with bounded memory, which is called the consistency and liveness property. Next, analyzing the maximum theoretical performance of the application based on the graph description. Then, mapping and scheduling the graph on the targeted MPSoC architecture. If the user requirement like the performance of the application is satisfied then a last phase of the development process consists on generating the source code which will be actually executed on the targeted MPSoC architecture. If the user requirement is not satisfied then

the developer must iterate the whole process to optimize the performance of the application.

Therefore, in the context of rapid prototyping, a fast evaluation of the maximum performance of the application is essential for a real-time feedback to the developer during the application development. The contributions presented in this thesis address the problem of flattening the hierarchy of the graph during the development process. Flattening the hierarchy, which is the transformation of a hierarchical graph to a non hierarchical graph, is mandatory in the classical development process. Such transformation results in an exponential growth of the graph size, which makes the application hard to process. Each presented contribution propose a new method for the performance evaluation of the IBSDF graph without flattening its hierarchy.

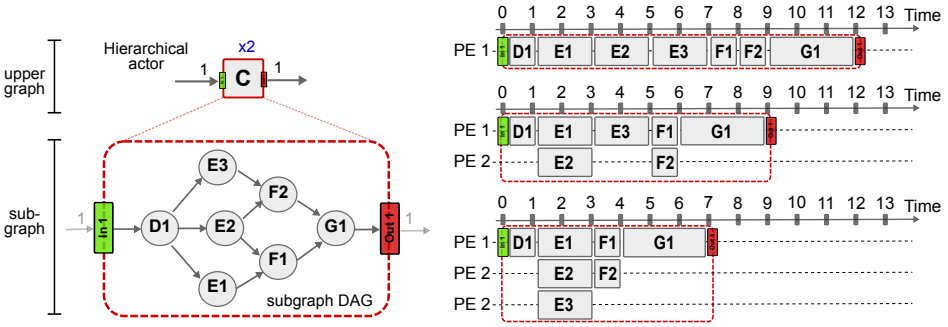
In Chapter 4, new techniques for the maximum throughput evaluation of IBSDF graphs are presented. The first method named Schedule-Replace (SR) computes the maximum throughput of an IBSDF graph when it is executed in the hierarchical execution mode. The second method named ESR computes the maximum throughput of the IBSDF graph when it is executed in the relaxed execution mode; an execution mode where applications reaches their maximum performance. Both proposed methods evaluate the IBSDF graph in a modular way without flattening its hierarchy. The numerical experiments have shown that the new techniques are capable of evaluating large IBSDF graph in less than 2 seconds. While the classical approaches take much more time or fail to return a result.

In chapter 5, new techniques for the minimum achievable latency evaluation of IBSDF graphs are presented. The first technique named Hierarchical-Symbolic-Execution (H-SE) evaluates the exact minimum latency of the IBSDF graph when it is executed in the hierarchical execution mode. The second technique named Hierarchical-Critical-Path Method (H-CPM) evaluates the exact minimum latency of the IBSDF graph when it is executed in the relaxed execution mode. Both techniques evaluate the latency of IBSDF graphs without flattening their hierarchy. The H-SE method consists on simulating the execution of the IBSDF graph in a hierarchical way. The H-CPM method in turn, consists on constructing the critical path of the entire IBSDF graph in a modular way. Compared to the classical method Critical-Path Method (CPM) which first flatten the IBSDF graph and then compute its critical path, the new technique is capable of evaluating large IBSDF graphs in few milliseconds.

These contributions were implemented as part of the Preesm software rapid prototyping framework. Our new developed techniques will enable the developer to rapidly evaluate the maximum performance of the modeled application at early stage of the development process. Thus, the developer can validate the design phase before reaching the mapping and scheduling phase, which avoids him repeating the entire development process. If the evaluated performance does not reach an acceptable value, the new techniques enables the developer to analyze level by level in order to identify which subgraph needs to be optimized.

6.2 Future Work

The work presented in this thesis opens many opportunities for future research on dataflow MoC and rapid prototyping.



(a) *A hierarchical actor example.*

(b) *Scheduling the subgraph multiple time on different numbers of PEs from 1 to 3.*

Actor C	
#PE	Dur
1	12
2	9
3	7

(c) *profile list.*

Figure 6.1 – *Constructing the equivalent moldable tasks of a hierarchical actor by scheduling its subgraph multiple times on different numbers of PE*

6.2.1 Modular Mapping and scheduling

From our contributions on evaluating the throughput and the latency of IBSDF graphs, we concluded that the flattening process must be avoided as much as possible, and use modular approaches instead. Indeed, modular approaches, like the Schedule-Replace (SR) for the throughput evaluation and the Hierarchical-Critical-Path Method (H-CPM) for the latency evaluation, have been demonstrated to be highly efficient for large IBSDF graphs. These modular-based algorithms take advantage of the insulation property of the IBSDF model to evaluate each subgraph independently from the hierarchy. Thus, the entire IBSDF graph is explored and evaluated hierarchically without flattening its hierarchy.

Similarly, we describe in the following the idea of a new approach for mapping and scheduling IBSDF graphs on homogeneous architectures in a modular way without flattening their hierarchy. The new approach is based on the idea of considering a hierarchical actor as a *moldable task*. In the literature, a moldable task is defined as a multiprocessor task which requires a certain number of PEs to execute. The moldable task is free to run on any number of PEs from 1 to m, according to its maximum parallelism. However, once the moldable task is executed, the number of chosen PEs cannot change during the execution. According to each number of PE, the moldable task takes a certain duration known at compile-time. Based on this description, a hierarchical actor is equivalent to a moldable task, when the IBSDF graph is executed in the hierarchical execution mode. Indeed, the execution of a hierarchical actor can be seen as the execution of a regular actor encapsulating the actual execution of the subgraph, which takes a certain duration and requires a certain number of PEs. Moreover, a list of profiles can be build by scheduling multiple times the subgraph of the hierarchical actor on different numbers of PEs. Each profile of the hierarchical actor corresponds to a pair of number of PEs and the corresponding duration that the subgraph takes to finish its execution. Thus, each hierarchical actor of the IBSDF graph can be converted to a moldable task.

Figure 6.1, shows an example of a hierarchical actor for which a list of profiles is build based on scheduling its subgraph on different numbers of PEs. The subgraph of the hierarchical actor C (fig. 6.1a) has a maximum parallelism of 3, which can be obtained by simulating an ASAP schedule of the subgraph. Hence, the subgraph is scheduled multiple times on different numbers of PEs from 1 to 3 (fig. 6.1b). Each obtained schedule corresponds to the execution of the hierarchical actor C as a large actor that encapsulates the execution of the subgraph. Thus, the profile list of the hierarchical actor C is obtained, which defines its equivalent moldable task (fig. 6.1c).

The new approach takes advantage of this equivalence between a hierarchical actor and a moldable task to reduce both the time and the space complexity of the mapping and scheduling problem of IBSDF graphs. In fact, the profile list of each hierarchical actor abstracts all the mapping and scheduling choices of the subgraph. Hence, instead of replacing the hierarchical actor by its subgraph and dealing with all the mapping and scheduling choices of each sub-actor, the hierarchical actor is considered as a moldable task and thus the problem is reduced to one question: which profile should be used for the hierarchical actor? This problem is known as the moldable task-graph scheduling problem. Thus, scheduling an IBSDF graph becomes a problem of scheduling hierarchical moldable tasks, which has not been studied yet in the literature.

In the context of rapid prototyping, this new approach will enables the evaluation of the expected performance of the application on the many-core architecture at compile-time, before reaching the code generation and the execution phases of the development process.

ANNEXE A

French Summary

A.1 Introduction

La complexité des applications DSP (Digital Signal Processor) augmente de façon exponentielle pendant les vingt dernières années. Parallèlement, la complexité des architectures MPSoC (Multiprocessor System-On-Chip) augmente de façon exponentielle pour répondre à la demande croissante de puissance de calcul. Les architectures MPSoC modernes, comme les architectures multi-cœurs, intègrent déjà des centaines d'éléments de traitement (PE) dans une seule puce, et prévoient d'intégrer jusqu'à des milliers de PE dans un avenir proche. Il faut donc trouver de nouveaux modèles et langages de programmation pour exploiter toute la puissance de traitement parallèle de ces nouveaux appareils.

Le modèle de calcul de flux de données (MoC) est l'un des modèles de programmation émergents, conçu pour le développement d'applications complexes de traitement du signal pour les architectures MPSoC. La programmation par flux de données a gagné en popularité au fil des ans en tant que modèle simple qui exprime naturellement le parallélisme d'une application. En fait, un modèle de flux de données décompose les calculs d'une application en un graphique de tâches appelé acteurs. Ainsi, une simple analyse du graphe permet de déterminer automatiquement quelles tâches peuvent être exécutées en parallèle. En tant que modèle de programmation prometteur, d'autres MoC de flux de données sont proposés, chacun prolongeant l'expressivité du précédent.

Le modèle IBSDF (Interface-Based Synchronous Dataflow) est l'une des récentes extensions du modèle SDF (Synchronous Dataflow) classique. Le modèle IBSDF est un modèle compositionnel qui permet au développeur de décomposer une application en un ensemble de modules organisés selon une hiérarchie. Chaque module a ses propres acteurs qui communiquent avec les autres modules via des interfaces. En effet, la hiérarchie basée sur les interfaces du modèle IBSDF permet la conception et l'analyse de chaque module (sous-graphe) indépendamment de la hiérarchie.

Le processus de développement d'applications de traitement du signal modélisées avec des graphes IBSDF consiste d'abord à vérifier la capacité du graphe à s'exécuter sur une architecture à mémoire limitée, qui est appelée la propriété de consistance et de vivacité. Puis, l'analyse de la performance théorique maximale de l'application sur la base de la description graphique. Ensuite, le placement et l'ordonnancement du graphe sur l'architecture MPSoC ciblée. Si les besoins de l'utilisateur comme les performances de l'application sont

satisfaits, une dernière phase du processus de développement consiste à générer le code source qui sera alors exécuté sur l'architecture MPSoC visée. Si les exigences de l'utilisateur ne sont pas satisfaites, le développeur doit alors recommencer l'ensemble du processus pour optimiser la performance de l'application.

Par conséquent, dans le contexte du prototypage rapide, une évaluation rapide des performances maximales de l'application est essentielle pour un feed-back en temps réel au développeur lors du développement de l'application. Les contributions présentées dans cette thèse abordent le problème de l'aplatissement de la hiérarchie du graphe pendant le processus de développement. L'aplatissement de la hiérarchie, qui est la transformation d'un graphe hiérarchique en un graphe non hiérarchique, est obligatoire dans le processus de développement classique. Une telle transformation entraîne une croissance exponentielle de la taille du graphe, ce qui rend l'application difficile à traiter. Nos contributions visent à proposer de nouvelles techniques pour l'évaluation rapide de deux métriques importantes : le débit maximal et la latence minimal de l'application. Ces deux métriques doivent être évalués le plus tôt possible par le développeur afin de valider ou non la phase de conception de l'application. Précisément, nous avons développé :

1. Une méthode nommée *Schedule-Replace (SR)* pour l'évaluation du débit maximal d'un graphe IBSDF lorsqu'il est exécuté en mode d'exécution hiérarchique.

2. Une méthode nommée *Evaluate-Schedule-Replace (ESR)* pour l'évaluation du débit maximal d'un graphe IBSDF lorsqu'il est exécuté en mode d'exécution relâché.

3. Une méthode nommée *Hierarchical-Critical-Path Method (H-CPM)* pour l'évaluation de la latence minimale du graphe IBSDF dans les deux modes : le mode d'exécution hiérarchique et le mode d'exécution relâché.

Contrairement à la technique classique qui repose sur le processus d'aplatissement, les nouvelles techniques évaluent le graphe IBSDF de manière modulaire, sans aplatir sa hiérarchie. Ainsi, les nouvelles techniques ont permis d'évaluer de grands graphes IBSDF en moins de 2 secondes, alors que la méthode classique prenait 5 minutes ou échouait pour certains graphes IBSDF à cause de leur grande taille.

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 résumée. 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 les contributions de cette thèse. La section A.4 présente les nouvelles techniques SR et ESR pour l'évaluation du débit maximal d'un graph IBSDF. La section A.5 présente la nouvelle technique H-CPM pour l'évaluation de la latence minimale d'un graphe IBSDF. Par la suite, la section A.6 présente les résultats des expérimentations numériques qui ont pour but de comparer les performances des nouvelles techniques avec celle de la méthode classique sur un jeu de donnée de graphes IBSDF généré aléatoirement. Enfin, la section A.7 conclut ce résumé.

A.2 Modèles de flot de données

Les modèles de flux de données sont des modèles basés sur des diagrammes, qui consistent à représenter une application avec un graphe orienté de tâches appelées acteurs. Les arêtes du graphe représentent l'échange de données entre les acteurs. Cette décomposition de l'application en un ensemble d'acteurs interconnectés offre au développeur un moyen naturel d'exprimer le parallélisme et les dépendances de données entre les acteurs. Dans ce

qui suit, nous présentons les propriétés des deux modèles de flux de données utilisée dans cette thèse : le modèle Synchronous Dataflow (SDF) et le modèle Interface-Based SDF (IBSDF).

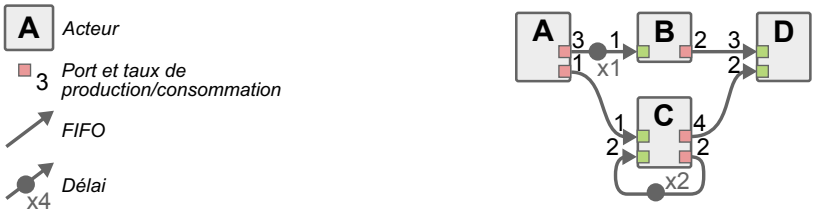
A.2.1 Synchronous Dataflow (SDF)

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.*
- ***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 FIFO auxquelles il est connecté. Une nouvelle exécution d'un acteur peut débuter dès que suffisamment de jetons de données sont présent sur les FIFO entrantes de cet acteur.*
- *Les **délais** (delay : $F \to \mathbb{N}$) sont des jetons de données contenus dans les FIFO 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.*

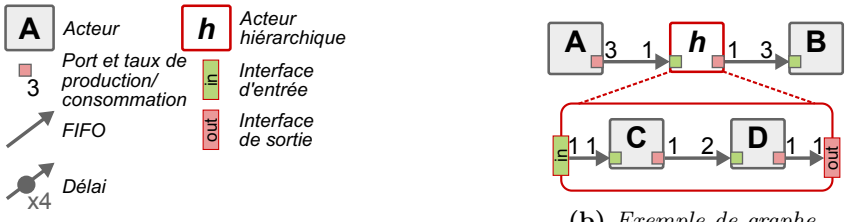
Figure A.1 – *Modèle de calcul SDF.*

La popularité du modèle SDF est principalement due à sa capacité à exprimer le parallélisme des applications. Dans la Figure A.1b 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 [LM87b] lors de son exécution.

A.2.2 Interface-Based SDF (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.2b, 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. Par exemple, dans le diagramme de la Figure A.2b, 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 PREESM : un outil de prototypage rapide

PRESSM est un framework basé sur Eclipse qui fournit des méthodes basées sur le flux de données pour étudier et programmer des plates-formes multi-cœurs embarquées [PDH+14]. Ce framework est open-source et de nombreux tutoriels sont disponibles sur le site web de Preesm pour l'initiation facile des développeurs C/C++ à la programmation multi-cœurs.

Le framework Preesm se concentre sur la fourniture d'informations de prototypage rapide de haut niveau sur le parallélisme et la latence des algorithmes. Il propose également des analyses détaillées sur les besoins en mémoire du système. De plus, une génération de code C/C++ adaptable à la plate-forme est fournie pour transformer la représentation du flux de données en un code exécutable.

Un workflow de développement Preesm typique pour un graphe IBSDF se compose de 6 phases principales :

1. Phase de conception : Le concepteur modélise l'application à l'aide d'un graphe IBSDF en utilisant l'interface utilisateur de Preesm
2. Phase de vérification : Vérification de certaines propriétés nécessaires comme la consistance et la vivacité.
3. Phase de conversion : Aplatissement de la hiérarchie du graphe
4. Phase d'analyse : Évaluer les performances du graphe.
5. Phase de placement et d'ordonnancement : décide quel acteur exécuter sur quel PE.
6. Génération de code : génère le code source réel de l'application qui s'exécutera sur la machine cible.

A.4 Evaluation du Débit maximal

Dans cette section, nous nous intéressons à l'évaluation du débit maximal des graphes IBSDF sur des architectures MPSoC avec des ressources illimitées, c'est-à-dire un nombre illimité de PE, une capacité mémoire illimitée et un réseau sur puce (NoC) à haute performance. Le seul critère pris en compte est le temps d'exécution dans le pire des cas (WCET) des acteurs, qui est prédéfini ou mesuré en faisant exécuter chaque acteur sur un PE donné. Formellement, le débit maximal est la quantité maximale de jetons de données que l'application peut produire à chaque unité de temps. L'évaluation du débit maximal permet au développeur de mesurer le potentiel maximal de l'application modélisée, quelle que soit l'architecture cible. La figure A.3 montre un exemple de graphe IBSDF qui servira d'exemple pour l'évaluation du débit dans cette section. Dans ce qui suit, nous présentons les deux modes d'exécution du graphe IBSDF, puis la méthode classique pour l'évaluation du débit, et enfin les nouvelles techniques d'évaluation du débit que nous avons développé.

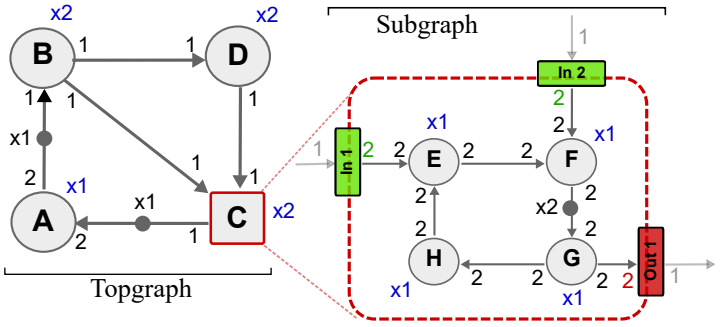
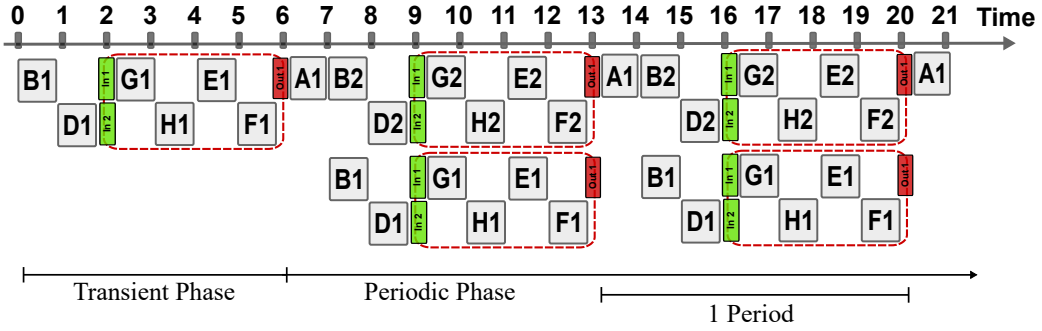


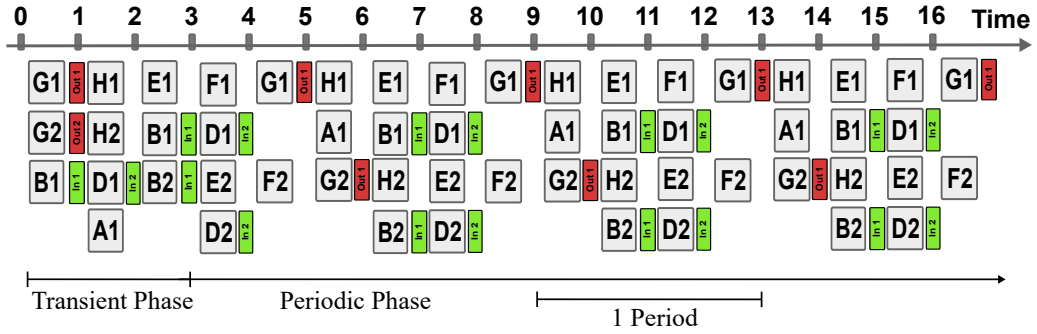
Figure A.3 – *Exemple d'un graphe IBSDF.*

A.4.1 Modes d'exécution d'un graph IBSDF

Le graphe IBSDF a deux modes d'exécution. Le premier mode est l'exécution hiérarchique, qui maintient la propriété d'isolation des niveaux hiérarchiques pendant l'exécution du graphe IBSDF. Le second mode est une exécution relâchée, qui casse l'isolation pour accélérer l'exécution du graphe IBSDF et ainsi augmenter son débit. La figure montre une exécution hiérarchique du graphe IBSDF exemple de la figure A.3. Tandis que la figure montre une exécution relâchée du même graphe. Comme le montrent les figures, l'exécution du graphe IBSDF est accélérée lorsque l'exécution du graphe est relâchée. Cependant, l'exécution relâchée peut consommer plus de PE que l'exécution hiérarchique.



(a) *Exécution hiérarchique du graphe IBSDF de la figure A.3.*



(b) *Exécution relâchée du graphe IBSDF de la figure A.3.*

Figure A.4 – *Modes d'exécution d'un graphe IBSDF.*

A.4.2 Méthode classique pour l'évaluation du débit

La méthode classique consiste d'abord à aplatir la hiérarchie du graphe IBSDF en un graphe non hiérarchique aplati équivalent. Ensuite, calculer le débit du graphe plat obtenu avec les méthodes du graphe SDF comme s'il s'agissait d'un large graphe SDF. A partir de l'état de l'art sur les méthodes du graphe SDF, nous définissons deux méthodes classiques pour le graphe IBSDF :

1. **Flat-SSE** : La méthode simule un ordonnancement au plus tôt ASAP du graphe aplati équivalent qui se présente sous la forme d'une phase transitoire suivie d'une phase périodique. Le débit est ensuite calculé comme 1 sur la durée d'une itération dans la phase périodique. Cette méthode est efficace lorsque l'exécution ne commence pas par une longue phase transitoire. Comme les deux figures A.4a et A.4b le montrent, le débit du graphe IBSDF de la figure A.3 est de 1/7 sous le mode d'exécution hiérarchique et 1/4 sous le mode d'exécution relâchée, puisque la durée d'une itération dans les deux modes est 7 et 4 respectivement.

2. **Flat-Periodic** : Pour les grands graphes IBSDF, le calcul du débit maximum avec la méthode précédente peut échouer en raison de la grande taille. Par conséquent, la methode *Flat-Periodic* consiste à calculer un ordonnancement périodique optimal du graphe plat équivalent pour définir une approximation de son débit maximal.

Cependant, le processus d'aplatissement d'un IBSDF consiste à convertir le topgraphe en un graphe plat équivalent, puis à remplacer chaque instance d'un acteur hiérarchique par le graphe plat équivalent de son sous-graphe. Ce processus se traduit souvent par une croissance exponentielle du nombre d'acteurs. Ainsi, les méthodes classiques ne parviennent pas à évaluer les graphes IBSDF.

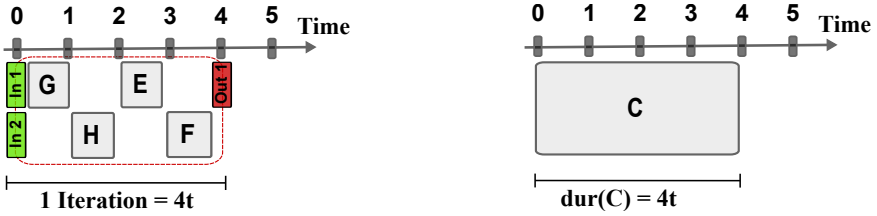
A.4.3 Nouvelle Méthode : Schedule-Replace (SR)

La technique Schedule-Replace (SR) calcule le débit des graphes IBSDF sous une exécution hiérarchique. La technique est basée sur la construction d'un ordonnancement ASAP du graphe IBSDF selon une approche ascendante et calcule son débit de la manière suivante :

- **Phase 1** : En partant du niveau inférieur de la hiérarchie jusqu'au niveau supérieur, pour chaque niveau :
  - **Etape 1** : Calculez la durée des acteurs hiérarchiques en ordonnançant leur sous-graphe en utilisant une exécution symbolique d'un ordonnancement ASAP.
  - **Etape 2** : Remplacer chaque acteur hiérarchique par un acteur régulier ayant la même durée que l'exécution du sous-graphe, calculée à l'étape 1.
  - **Etape 3** : Passer au niveau supérieur et répéter les étapes 1 et 2 jusqu'à ce que le top-graphe soit atteint.
- **Phase 2** : Calculez le débit du topographe obtenu comme s'il s'agissait d'un graphe SDF.

La méthode SR consiste à faire abstraction de la hiérarchie en remplaçant les acteurs hiérarchiques par des acteur régulier. En effet, grâce à la propriété d'isolation du modèle IBSDF l'exécution du sous-graph d'un acteur hiérarchique est équivalente à l'exécution d'un acteur régulier qui consomme un certain nombre de PE et prend une certaine durée d'exécution. La figure A.5 montre l'acteur équivalent à l'exécution du sous-graphe $EFGH$ de l'acteur hiérarchique C du graphe IBSDF de la figure A.3.

La figure A.6 montre l'ordonnancement ASAP du graphe IBSDF sous une exécution hiérarchique, dans laquelle chaque exécution de l'acteur hiérarchique C est représentée par un bloc qui fait abstraction de l'exécution du sous-graph $EFGH$.



(a) *Exécution du sous-graphe EFGH du graphe IBSDF de la figure A.3.* **(b)** *l'acteur régulier équivalent à l'exécution du sous-graph EFGH.*

Figure A.5 – *Remplacement d'un acteur hiérarchique par un acteur régulier.*

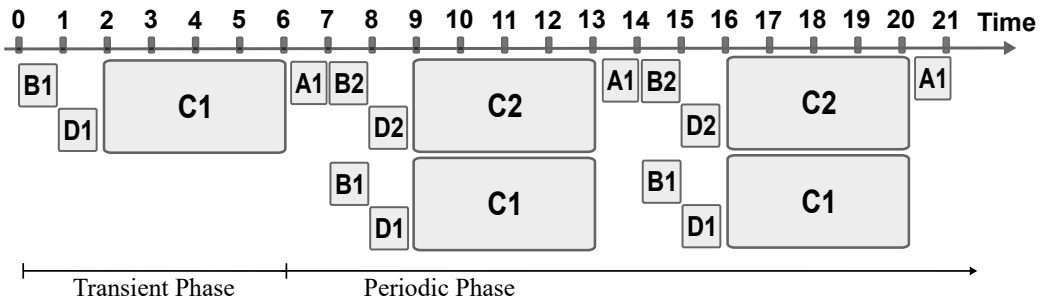


Figure A.6 – *Exécution hiérarchique du graphe IBSDF de la figure A.3.*

A.4.4 Nouvelle Méthode : Evaluate-Schedule-Replace (ESR)

La méthode Evaluate-Schedule-Replace (ESR) calcule le débit des graphes IBSDF sous une exécution relâchée. Cette méthode est basée sur la technique SR et comprend les étapes suivantes :

- **Phase 1** : Reprogrammer les sous-graphies IBSDF pour synchroniser leur exécution et révéler les retards cachés dans la hiérarchie.
- **Phase 2** : En partant du niveau inférieur de la hiérarchie jusqu'au niveau supérieur, pour chaque niveau :
  - **Étape 1** : Construire un graphe de remplacement pour chaque acteur hiérarchique du niveau actuel en ordonnançant et en évaluant son sous-graphe.
  - **Étape 2** : Convertir les sous-graphes du niveau courant en graphes aplatis et remplacer chaque instance d'un acteur hiérarchique par son graphe de remplacement.
  - **Étape 3** : Passer au niveau supérieur et répéter les étapes 1 et 2 jusqu'à ce que le top-graphe soit atteint.
- **Phase 3** : Calculez le débit du topographe obtenu comme s'il s'agissait d'un graphe SDF.

Tout comme la technique SR, la méthode ESR analyse le graphe IBSDF niveau par niveau pour en calculer le débit. Cependant, la nouvelle méthode remplace un acteur hiérarchique par un petit graphe qui modélise le comportement de son sous-graphe dans une exécution détendue. En effet, le mode d'exécution relâchée permet aux interfaces d'entrée et de sortie du sous-graphe de s'exécuter à un moment différent dès qu'elles sont prêtes à être exécutées. Ce comportement des interfaces du sous-graphe rend l'exécution d'un acteur hiérarchique différente de celle d'un acteur SDF classique. En conséquence, un acteur hiérarchique ne peut pas être remplacé par un acteur SDF régulier dans une exécution relâchée du graphe IBSDF. A titre d'exemple, la figure A.7 montre une comparaison entre le comportement du sous-graphe dans chaque mode d'exécution du graphe IBSDF de la figure . Dans l'exécution hiérarchique (figure A.7a), le sous-graphe se comporte comme un bloc d'exécution d'acteurs dans lequel les deux interfaces d'entrée $In1$ et $In2$ démarrent en même temps. En ce qui concerne l'interface de sortie $Out1$, elle commence à s'exécuter jusqu'à la fin de l'exécution du sous-graphe. Dans le mode d'exécution relâchée (figure A.7b), le sous-graphe se comporte différemment. En fait, les interfaces d'entrée commencent à un moment différent et l'interface de sortie commence avant la fin de l'exécution du sous-graphe. Ce comportement des interfaces du sous-graphe est ce qui rend en fait l'exécution plus rapide que l'exécution hiérarchique, car il comprime l'exécution de l'ensemble du graphe IBSDF. Mais, il n'est plus possible d'abstraire l'exécution du sous-graphe avec un acteur régulier comme nous l'avons fait dans la figure A.5 pour l'exécution hiérarchique du graphe IBSDF de la figure A.3.

(a) *Exécution hiérarchique du sous-graphe.* **(b)** *Exécution relâchée du sous-graphe.*

Figure A.7 – *Comparaison entre le comportement du sous-graphe dans chaque mode d'exécution du graphe IBSDF de la figure A.3.*

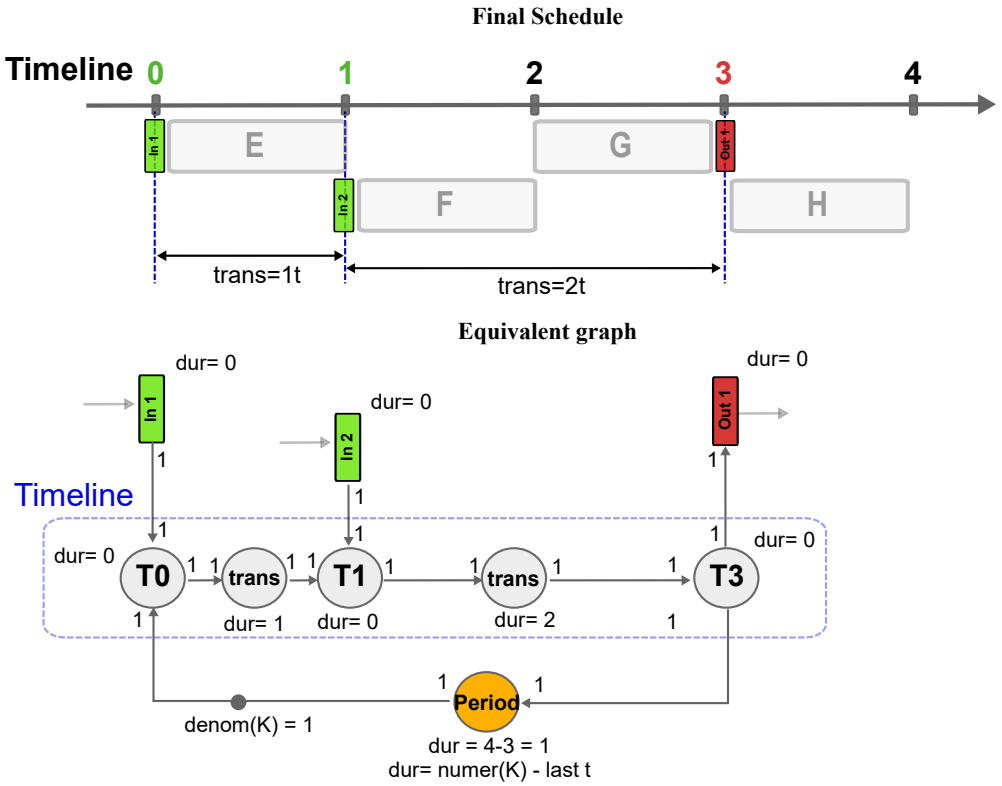


Figure A.8 – *Le sous-graphe équivalent à l'exécution relâchée du sous-graphe EFGH du graphe IBSDF de la figure A.3.*

Figure A.8 montre le sous-graphe équivalent à l'exécution relâchée du sous-graphe $EFGH$ du graphe IBSDF de la figure A.3. Ce nouveau sous-graph qui a été construit en utilisant l'algorithm decrit dans [], sera utilisé pour remplacer l'acteur hiérarchique C.

A.5 Evaluation de la latence minimal

la latence multi-cœur d'une application est la durée minimale possible d'une itération de son graphe de flux de données équivalent lorsqu'elle est exécutée sur une architecture multi-cœur aux ressources illimitées, de sorte que tous le parallélisme de l'application est actif. Le calcul de la latence multi-cœur d'un graphe de flux de données est essentiel pour évaluer l'accélération théorique de l'application lorsqu'elle est exécutée à sa performance maximale comparée à une exécution mono-cœur.

Dans cette section nous présentons les différentes méthodes pour évaluer la latence multi-cœur d'un graph IBSDF. En premier, la méthode classique qui repose sur l'aplatissement de la hiérarchie du graph. Ensuite, les nouvelles techniques qui permettent d'évaluer de très grand graphes IBSDF d'une manière modulaire sans aplatir leur hiérarchie.

A.5.1 Méthode classique pour l'évaluation de la latence

Une façon simple d'évaluer la latence multi-cœur d'un graphe IBSDF est d'aplatir sa hiérarchie en un graphe plat et de l'évaluer comme si c'était un grand graphe SDF. Cette approche classique utilisée précédemment pour le problème d'évaluation du débit s'est avérée facile à mettre en œuvre et à intégrer dans les outils de prototypage existants. Dans la pratique, l'approche classique consiste à aplatir d'abord la hiérarchie du graphe

IBSDF en un graphe plat équivalent puis évaluer sa latence en utilisant l'une des méthodes d'évaluation de modèle SDF. Dans la littérature, la latence multi-core d'un graphique SDF, souvent appelée latence minimale réalisable, est calculée à l'aide de deux méthodes :

- méthode du chemin critique (flat-CPM) : la méthode du chemin critique est utilisée pour trouver le chemin critique des applications de traitement du signal qui sont modélisées à l'aide d'un graphe de flux de données. Le chemin critique d'une application est le chemin le plus long des acteurs dépendants dans le graphe acyclique orienté équivalent, où la longueur du chemin est calculée comme la somme de la durée de ses acteurs. Du point de vue de l'exécution, le chemin critique représente la séquence d'exécution la plus longue des acteurs qui ne peuvent pas être exécutés en parallèle en raison de leur dépendance de données. Par conséquent, la longueur du chemin critique représente la durée minimale possible d'une itération graphique malgré le nombre infini de PE disponibles, c'est-à-dire la latence multicœur minimale possible de l'application.

- méthode de l'exécution symbolique (flat-SE) : La méthode basée sur l'exécution symbolique (SE) reste une technique courante pour évaluer les propriétés et les métriques des applications de traitement du signal basée sur la simulation de leur comportement. Pour l'évaluation de la latence des graphes SDF, la méthode consiste à simuler une itération complète du graphe et à mesurer sa durée totale. Dans le contexte de la latence multicœur, un nombre infini de PE disponibles est pris en compte lors de la simulation afin que chaque acteur soit exécuté dès qu'il est prêt. Par conséquent, en adoptant un ordonnancement ASAP, la méthode est capable de simuler l'exécution la plus rapide possible d'un graphe de flux de données. Ainsi, la durée résultante du graphe représente la latence multi-cœur minimale réalisable de l'application.

Le calcul de la latence multi-core d'un graphe IBSDF avec l'approche classique nécessite d'aplatir l'ensemble de la hiérarchie en un graphe plat équivalent. Une telle transformation se traduit souvent par un grand graphe avec un nombre exponentiel d'acteurs et d'arêtes. En conséquence, la complexité temporelle et spatiale des méthodes d'évaluation du modèle SDF augmente de façon exponentielle. Pour certains graphes IBSDF, il est même impossible d'aplatir complètement leur hiérarchie avec une quantité raisonnable de mémoire, ce qui rend l'approche classique inutile dans ces cas. De plus, dans le contexte du prototypage rapide, le développeur doit être en mesure d'évaluer si possible en temps réel la latence multicœur de l'application au fur et à mesure que le processus de conception avance. A cette fin, l'approche classique est à nouveau définie comme inadaptée à l'évaluation rapide des performances des graphes IBSDF.

A.5.2 Nouvelle Méthode : Hierarchical-Symbolic-Execution (H-SE)

La méthode H-SE (Hierarchical-Symbolic-Execution) adapte la method d'exécution symbolique (SE) du modèle SDF pour le modèle IBSDF. La méthode H-SE évalue la latence multi-core du graphe IBSDF dans le mode d'exécution hiérarchique. Elle consiste à simuler hiérarchiquement l'exécution la plus rapide du graphe IBSDF pour mesurer la durée minimale de son itération complète. Ainsi, au lieu de simuler l'itération complète du graphe plat equivalent comme le fait l'approche classique flat-SE, la méthode H-SE commence par simuler l'exécution multi-coeurs de chaque sous-graphe pour définir la durée minimale de son acteur hiérachique père. Ainsi, dans une approche ascendante, la durée de l'ensemble de la hiérarchie est abstraite par la durée minimale des acteurs hiérarchiques du

topgraphe. Par conséquent, simuler une seule itération du topgraphe dans lequel les acteurs hiérarchiques sont considérés comme des acteurs réguliers avec une durée minimale, permet de mesurer la durée de l'ensemble du graph hiérarchique. Ainsi, la méthode H-SE est capable d'évaluer la latence multi-cœur du graphe IBSDF dans le mode d'exécution hiérarchique sans aplatir sa hiérarchie.

A.5.3 Nouvelle Méthode : Hierarchical-Critical-Path Method (H-CPM)

La méthode H-CPM (Hierarchical-Critical-Path Method) adapte la méthode CPM (Critical-Path Method) du modèle SDF pour le modèle IBSDF. La méthode H-CPM consiste à évaluer la latence multi-cœur du graphe IBSDF en mode d'exécution relâchée, qui représente la latence minimale réalisable de l'application quelque soit les ressources illimitées dont l'architecture peut disposer. Comparé à l'approche classique flat-CPM qui calcule le chemin critique de l'ensemble du graphe plat équivalent, la méthode H-CPM construit hiérarchiquement le chemin critique du graphe IBSDF sans aplatir complètement sa hiérarchie. Pour ce faire, nous avons d'abord démontré que le chemin critique d'un graphe IBSDF est une composition de plusieurs sous-chemins critiques qui proviennent des sous-graphes de l'IBSDF. Ensuite, nous avons introduit un nouveau modèle de graphe appelé LPEG (Longest-Path Equivalent Graph) qui extrait le chemin le plus long entre chaque interface d'entrée et de sortie d'un sous-graphe IBSDF par des arêtes pondérées. En utilisant ce modèle d'abstraction dans une approche ascendante, la H-CPM fait abstraction, avec un ensemble de LPEG équivalents, de tous les sous-chemins possibles dans la hiérarchie qui peuvent composer le chemin critique du graphe IBSDF. Par conséquent, calculer uniquement le chemin critique du topgraphe dans lequel les acteurs hiérarchiques sont remplacés par le LPEG équivalent de leur sous-graphe, permet d'évaluer le chemin critique de l'ensemble du graphique IBSDF. Ainsi, la méthode H-CPM est capable d'évaluer la latence multi-cœur du graphe IBSDF en mode d'exécution relâchée sans aplatir sa hiérarchie ni explorer l'intégralité du graphe plat équivalent.

A.6 Experimentations numériques

A.6.1 Déroulement des tests

Dans cette section, nous confirmons notre étude théorique en comparant la performance des approches classiques avec celle des nouvelles approches développées durant cette thèse. L'expérimentation numérique consiste à mesurer le temps d'exécution de chaque méthode pour calculer les métriques de performance de chaque graphe IBSDF du jeu de données. Le temps d'exécution mesuré inclut le temps pour le processus d'aplatissement des graphes. Le jeu de données utilisé dans ces expériences numériques est composé de deux catégories de graphes IBSDF. La première catégorie est l'ensemble des applications DSP réelles modélisées sous forme de graphes IBSDF pour lesquels un code source est disponible dans [Pre]. La deuxième catégorie est l'ensemble des graphes IBSDF synthétiques qui sont générés de manière aléatoire à l'aide d'un générateur de graphes IBSDF basé sur l'outil Turbine [BLDMK14]. La deuxième catégorie est principalement utilisée pour compléter le jeu d'applications réelles par de grands graphes IBSDF afin de fournir une comparaison étendue des performances. En fait, jusqu'à aujourd'hui, le nombre de niveaux hiérarchiques dans les applications réelles reste faible en raison de la difficulté de compiler de grands graphes en utilisant les approches classiques. Le tableau A.1 présente les caractéristiques des deux catégories du jeu de données : les applications réelles (Crypto, Large FFT, LTE [PAPN13], et Stereo Matching [Heu15]) et les graphes synthétiques (Graphe 1 à 6). A

partir de la taille des graphes plats équivalents, nous pouvons clairement voir que le processus d'aplatissement se traduit par une croissance exponentielle du nombre d'acteurs et d'arêtes. En fait, il n'a pas été possible de convertir les deux derniers graphes IBSDF synthétiques Graph 5 et Graph 6 avec la RAM disponible de 8Go. En conséquence, les méthodes classiques n'ont pas réussi à calculer ni le débit ni la latence de ces deux graphes.

Table A.1 – *Description du jeu de données utilisé pour la comparaison de perfomrances entre les méthodes classiques et les nouvelles méthodes.*

Graphes IBSDF				Graphes plat equivalents	
Nom	Niv. Hierar.	Acteurs	Acteurs Hier.	Acteurs	Arêtes
Crypto	2	10	1	34	85
Large FFT	2	10	1	267	1300
LTE	4	18	4	250	641
Stereo Matching	2	41	3	1604	5829
Graphe 1	3	15	2	503	1654
Graphe 2	5	20	4	17727	80976
Graphe 3	6	24	5	84440	338391
Graphe 4	5	150	4	653289	3253811
Graphe 5	8	240	7	39 E10	-
Graphe 6	10	100	9	31 E15	-

A.6.2 Résultats des tests pour l'évaluation du débit

Comme le montre le tableau des résultats A.2, les méthodes classiques n'ont pas réussi à calculer le débit des deux derniers graphes, puisqu'elles reposent sur le processus d'aplatissement. Cependant, les deux nouvelles méthodes SR et ESR ont été capables de calculer avec succès le débit de tous les graphes IBSDF en moins de 2 secondes. Pour les petits graphes IBSDF, l'approche classique est plus rapide que les nouvelles techniques. Mais, le temps d'exécution des deux méthodes classiques Flat-SSE et Flat-Periodic augmente

Table A.2 – *Comparaison des performances entre les nouvelles techniques SR et ESR, et les méthodes classiques Flat-SSE et Flat-Periodic.*

Graphes IBSDF		Temps d'exécution total			
Name	Levels	Flat-SEE	Flat-Periodic	SR	ESR
Crypto	2	4 ms	8 ms	38 ms	45 ms
Large FFT	2	44 ms	48 ms	36 ms	74 ms
LTE	4	152 ms	32 ms	32 ms	58 ms
Stereo	2	4320 ms	151 ms	37 ms	130 ms
Graphe 1	3	11984 ms	67 ms	34 ms	59 ms
Graphe 2	5	>5 min	3060 ms	34 ms	70 ms
Graphe 3	6	>5 min	14600 ms	34 ms	90 ms
Graphe 4	5	>5 min	234000 ms	61 ms	306 ms
Graphe 5	8	-	-	61 ms	560 ms
Graphe 6	10	-	-	72 ms	1930 ms

exponentiellement à chaque fois que le nombre de niveaux et que la taille du graphe plat équivalent augmentent. Les résultats confirment que les nouvelles techniques SR et ESR conviennent mieux aux graphes IBSDF plus larges que les méthodes basées sur l'aplatissement.

A.6.3 Résultats des tests pour l'évaluation de la latence

De même que pour l'évaluation du débit, les approches classiques n'ont pas été en mesure de calculer la latence multi-core de tous les graphes A.3. Cependant, les nouvelles techniques H-SE et H-CPM ont permis d'évaluer la latence multi-core de tous les graphiques IBSDF grâce à leur approche modulaire qui évite d'aplatir la hiérarchie du graphe IBSDF. Par exemple, les nouvelles techniques H-SE et H-CPM ont évalué la latence multi-core du graphique 4 en quelques millisecondes, alors que les approches classiques ont pris 6 secondes pour obtenir un résultat.

Table A.3 – *Comparaison des performances entre les nouvelles techniques H-SE et H-CPM, et les méthodes classiques Flat-CPM et Flat-SE.*

IBSDF graph	Temps d'exécution total			
	Flat-CPM	Flat-SE	H-SE	H-CPM
Crypto	0.65 ms	1.15 ms	0.50 ms	0.50 ms
Large FFT	4.66 ms	6.11 ms	0.50 ms	3.73 ms
LTE	2.52 ms	2.85 ms	0.50 ms	1.49 ms
Stereo Matching	18.20 ms	23.56 ms	0.87 ms	15.00 ms
Graphe 1	2.40 ms	3.32 ms	0.93 ms	1.83 ms
Graphe 2	108.16 ms	134.00 ms	0.99 ms	3.91 ms
Graphe 3	441.40 ms	415.55 ms	1.00 ms	5.11 ms
Graphe 4	6.25 s	6.64 s	1.20 ms	48.14 ms
Graphe 5	-	-	4.74 ms	83.32 ms
Graphe 6	-	-	12.81 ms	308.60 ms

A.7 Conclusion

Dans cette thèse, nous avons étudié le processus de développement d'applications de traitement du signal modélisées avec des graphes IBSDF dans le contexte du prototypage rapide. Précisément, nous avons développé de nouvelles méthodes pour l'évaluation du débit maximum et de la latence minimum qui représentent deux paramètres importants à évaluer le plus tôt possible durant le processus de développement . En fait, les méthodes actuelles, que nous appelons dans cette thèse les méthodes classiques, consistent à aplatir le graphe IBSDF en un graphe plat équivalent avec un nombre exponentiel d'acteurs et d'arêtes. En conséquence, les méthodes classiques ne sont pas en mesure d'évaluer les métriques de performance des larges graphes IBSDF. Nous avons donc développé les nouvelles techniques sur la base d'une approche modulaire qui consiste à évaluer chaque sous-graphe indépendamment sans aplatir l'ensemble de la hiérarchie du graphe IBSDF. Ainsi, les nouvelles techniques permettent d'évaluer en quelques millisecondes la performance de larges graphes IBSDF, ce qui a été confirmé par une série d'expériences numériques. Un travail futur consiste à développer de nouvelles méthodes modulaires pour le placement et l'ordonnancement de grands graphes IBSDF sur des architectures multi-cœurs.

List of Figures

List of Tables

Acronyms

πSDF Parameterized and Interfaced SDF. 8, 16, 18, 19, 21, 23, 36, 90, 109

AAA Algorithm-Architecture Adequation. 22

AI Artificial Intelligence. 4

ALAP As Late As Possible. 57, 110

ASAP As Soon As Possible. 21, 31, 32, 40, 42, 43, 45, 47–49, 51, 52, 57, 73, 74, 76, 93, 100, 101, 109, 110

BDF Boolean DataFlow. 7, 12–14, 19, 109

CMOS Complementary Metal–Oxide–Semiconductor. 4

CP Critical-Path. 27, 71

CPM Critical-Path Method. 71–74, 77, 78, 80, 83–85, 87, 88, 90, 92, 111, 113

CPU Central Processing Unit. 33

CSDF Cyclo-Static Dataflow. 7, 11, 12, 16, 37, 38, 46, 109

DAG Directed Acyclic Graph. 11, 21, 26–30, 38, 57, 66, 67, 71–74, 77, 78, 80–84, 86–88, 110, 113

DPN Dataflow Process Network. 7–10, 109

DSE Design Space Exploration. 5, 39, 90

DSP Digital Signal Processor. 33, 34, 84, 91, 105

ESEM Equivalent Subgraph Execution Model. 64

ESR Evaluate-Schedule-Replace. iii, 5, 40, 51, 54–56, 59–64, 92, 96, 102, 110, 113

Fifo First-In First-Out. 8, 10–13, 15, 18, 24, 26–32, 37, 38, 40, 46, 47, 57, 58, 71, 73, 74, 84, 97

FSM Finite-State Machine. 13

GRT Global RunTime. 36, 37

H-CPM Hierarchical-Critical-Path Method. iii, 6, 75, 80, 81, 83–88, 90, 92, 93, 96, 105, 111, 113

H-SE Hierarchical-Symbolic-Execution. iii, 6, 75–77, 83–90, 92, 104, 105, 111, 113

Hierarchical SDF Hierarchical SDF. ii, 8, 14–16, 21, 25, 26, 31, 109

HSDF Homogeneous SDF. ii, 7, 11, 21, 26, 27, 29, 30, 41–43, 49, 55, 57, 109

IBSDF Interface-Based SDF. ii, iii, 5, 6, 8, 15, 16, 18, 19, 21, 23–26, 28, 29, 31, 38–40, 46–72, 74–86, 88–94, 96–107, 109–111, 113

IoT Internet of Things. 4

KPN Kahn Process Network. 7, 8

LCG Linear Constraint Graph. 42, 49

LPEG Longest-Path Equivalent Graph. 80–83, 88, 90, 110

LRT Local RunTime. 36, 37

MCM Maximum Cycle Mean. 41

MCR Maximum Cost-to-Time Ratio. 41, 42, 44, 45, 49, 57–59

MoC Model of Computation. 5–19, 23, 25, 34, 35, 65, 91, 92, 109

MPSoC Multiprocessor System-on-Chip. 5, 7, 24, 29, 31, 33, 36, 38, 39, 91, 109

NoC Network on Chip. 33, 39

OS Operating System. 3, 4

Preesm Parallel and Real-time Embedded Executives Scheduling Method. 21, 23, 36, 92

PCG Phased Computation Graph. 38

PE Processing Element. 7, 9, 23, 27, 32–34, 36, 37, 39, 48, 51, 65, 67, 69–73, 91, 93, 99, 101, 111

PERT Project Evaluation and Review Technique. 71

PiMM Parameterized and Interfaced dataflow Meta-Model. 18, 19, 109

PN Petri Net. 32

PSDF Parameterized SDF. 8, 16–19, 109

RTOS Real-Time Operating System. 34

RV Repetition Vector. 24–28, 30, 36, 40, 41, 109

Spider Synchronous Parameterized Interfaced Dataflow Embedded Runtime. 21, 36, 37, 109

SADF Scenario-Aware Dataflow. 7, 12, 13, 19, 109

SDF Synchronous Dataflow. ii, iii, 5–8, 10–19, 21, 24–32, 37–46, 49–52, 54, 61, 63–69, 71–75, 80, 89–91, 97, 98, 100–104, 109, 111, 115, 116

SDF^3 Synchronous Dataflow For Free. 37, 38

SE Symbolic-Execution. 71, 73, 74, 83–85, 87–89, 111, 113

SoC System-on-Chip. 4

SR Schedule-Replace. iii, 5, 40, 51–54, 60–64, 68, 75, 76, 80, 92, 93, 96, 101, 102, 110, 113

srSDF Single-Rate SDF. 7, 11, 21, 27–29, 38, 47–55, 57–64, 72, 74, 109, 110, 113

SSE State-Space Exploration. 42, 49, 60–62, 73, 74

WCET Worst-Case Execution Time. 31, 35, 39, 65, 84

WEG Weighted Event Graph. 32

Personal Publications

[DDNKM+17a] H. Deroui, K. Desnos, J.-F. Nezan, and A. Munier-Kordon Throughput evaluation of DSP applications based on hierarchical dataflow models. In Proceedings of the 50th International Symposium on Circuits and Systems. ISCAS, 2017.

[DDNKM+17b] H. Deroui, K. Desnos, J.-F. Nezan, and A. Munier-Kordon. Relaxed subgraph execution model for the throughput evaluation of IBSDF graphs. In Proceedings of the 17th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation. SAMOS, 2017.

Bibliography

[BB01] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001. 16, 17

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *, 1995 International Conference on Acoustics, Speech, and Signal Processing, 1995. ICASSP-95*, volume 5, pages 3255–3258 vol.5, May 1995. 11

[BHR09] A. Benoit, M. Hakem, and Y. Robert. Optimizing the Latency of Streaming Applications under Throughput and Reliability Constraints. In *2009 International Conference on Parallel Processing*, pages 325–332, September 2009. 65

[BL93] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *IEEE International Conference on Acoustics Speech and Signal Processing*, pages 429–432 vol.1, Minneapolis, MN, USA, 1993. IEEE. 13

[BLDMK14] Bruno Bodin, Youen Lesparre, Jean-Marc Delosme, and Alix Munier-Kordon. Fast and efficient dataflow graph generation. pages 40–49. ACM Press, 2014. 30, 37, 60, 84

[BLM96a] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. 24, 27, 67

[BLM96b] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. 74

[BMKdD16] Bruno Bodin, Alix Munier-Kordon, and Benoît Dupont de Dinechin. Optimal and fast throughput evaluation of CSDF. pages 1–6. ACM Press, 2016. 46

[BNHMMK12] A. Benabid-Najjar, C. Hanen, O. Marchetti, and A. Munier-Kordon. Periodic Schedules for Bounded Timed Weighted Event Graphs. *IEEE Trans-*

actions on Automatic Control, 57(5):1222–1232, May 2012. 32, 51, 54, 57, 61

[BS12] Mohamed A. Bamakhrama and Todor Stefanov. Managing Latency in Embedded Streaming Applications Under Hard-real-time Scheduling. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 83–92, New York, NY, USA, 2012. ACM. 65

[CH89] J. E. Cooling and T. S. Hughes. The emergence of rapid prototyping as a real-time software development tool. In *Second International Conference on Software Engineering for Real Time Systems, 1989.*, pages 60–64, September 1989. 22

[CH17] Junchul Choi and Soonhoi Ha. Worst-Case Response Time Analysis of a Synchronous Dataflow Graph in a Multiprocessor System with Real-Time Tasks. *ACM Trans. Des. Autom. Electron. Syst.*, 22(2):36:1–36:26, January 2017. 65

[CK88] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on software engineering*, 14(2):141–154, 1988. 34, 35

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, editors. *Introduction to algorithms*. MIT Press, Cambridge, Mass., 3. ed edition, 2009. OCLC: 698955316. 72, 83

[Das04] Ali Dasdan. Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):385–418, October 2004. 41, 65

[DDNMK17a] H. Deroui, K. Desnos, J. F. Nezan, and A. Munier-Kordon. Throughput evaluation of DSP applications based on hierarchical dataflow models. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017. 68, 71, 75

[DDNMK17b] Hamza Deroui, Karol Desnos, Jean-François Nezan, and Alix Munier-Kordon. Relaxed Subgraph Execution Model for the Throughput Evaluation of IBSDF Graphs. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, SAMOS, Greece, July 2017. 65

[Des14] Karol Desnos. *Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs*. phdthesis, INSA de Rennes, September 2014. 4, 18, 19, 22, 35, 109

[dGKBS12] Robert de Groote, Jan Kuper, Hajo Broersma, and Gerard J.M. Smit. Max-Plus Algebraic Throughput Analysis of Synchronous Dataflow Graphs. pages 29–38. IEEE, September 2012. 42

[DIG99] A. Dasdan, S.S. Irani, and R.K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 37–42, 1999. 41

[DLC15] X. K. Do, S. Louise, and A. Cohen. Managing the Latency of Data-Dependent Tasks in Embedded Streaming Applications. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 9–16, September 2015. 65

[DP18] Karol Desnos and Francesca Palumbo. Dataflow Modeling for Reconfigurable Signal Processing Systems. In *Handbook of Signal Processing Systems, 3rd Edition*. October 2018. 16

[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 *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, pages 41–48, July 2013. 18, 34, 90

[FS14] Nikolina Frid and Vlado Sruk. Design space exploration in Multi- Processor System-on-Chip platforms. 2014. 90

[GGS+06] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M.J.G. Bekooij, B.D. Theelen, and M.R. Mousavi. Throughput Analysis of Synchronous Data Flow Graphs. In *Sixth International Conference on Application of Concurrency to System Design, 2006. ACSD 2006*, pages 25–36, June 2006. 32, 42, 51, 54, 61, 73

[Gha08] Amir Hossein Ghamarian. *Timing analysis of synchronous data flow graphs*. PhD Thesis, PhD thesis, Eindhoven University of Technology, 2008. 73

[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 *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings.*, pages 123–132, June 2003. 22

[GSB+07] A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen, and B.D. Theelen. Latency Minimization for Synchronous Data Flow Graphs. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, 2007. DSD 2007*, pages 189–196, August 2007. 65, 71, 73

[gur] Gurobi Optimization. 61

[Has18] Julien Hascoët. Contributions to Software Runtime for Clustered Manycores Applied to Embedded and High-Performance Applications. December 2018. 18

[Heu15] Julien Heulot. *Runtime multicore scheduling techniques for dispatching parameterized signal and vision dataflow applications on heterogeneous MPSoCs*. PhD thesis, INSA de Rennes, 2015. 18, 84

[HPD+14] J. Heulot, M. Pelcat, K. Desnos, J. Nezan, and S. Aridhi. Spider: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for multicore DSPS. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pages 167–171, September 2014. 34, 36

[KA97] Yu-Kwong Kwok and I. Ahmad. A parallel algorithm for compile-time scheduling of parallel programs on multiprocessors. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 90–101, November 1997. 35

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999. 34, 35

[Kah62] A. B. Kahn. Topological Sorting of Large Networks. *Commun. ACM*, 5(11):558–562, November 1962. 71

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. page 6, 1974. 5, 8

[KAL11] Torsten Kempf, Gerd Ascheid, and Rainer Leupers. *Multiprocessor Systems on Chip: Design Space Exploration*. Springer-Verlag, New York, 2011. 90

[Kel61] James E. Kelley. Critical-Path Planning and Scheduling: Mathematical Basis. *Oper. Res.*, 9(3):296–320, June 1961. 71

[Ker] Harold Kerzner. Project Management: A Systems Approach to Planning, Scheduling, and Controlling, 12th Edition. 71

[KKB17] Guus Kuiper, Philip S. Kurtin, and Marco J.G. Bekooij. Hybrid Latency Minimization Approach using Model Checking and Dataflow Analysis. pages 41–50. ACM Press, 2017. 65

[KM66] Richard M. Karp and Rayamond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966. 8

[Law01] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Courier Corporation, January 2001. Google-Books-ID: m4MvtFenVjEC. 72

[Les17] Youen Lesparre. Efficient evaluation of mappings of dataflow applications onto distributed memory architectures. page 149, 2017. 28, 65

[LH89] Edward A. Lee and Soonhoi Ha. Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'(GLOBECOM), 1989. IEEE*, pages 1279–1283. IEEE, 1989. 34, 35

[LM87a] E. Lee and D.G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987. 24

[LM87b] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. 10, 24, 97

[LP95] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. 8, 10

[MK09] Olivier Marchetti and A. Munier Kordon. Cyclic Scheduling for the Synthesis of Embedded Systems. *Introduction to scheduling*, pages 135–164, 2009. 32

[MMK09] Olivier Marchetti and Alix Munier-Kordon. A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research*, 197(2):532–540, September 2009. 28, 30, 31, 32

[OS01] Timothy W. O'Neil and Edwin Hsing-Mean Sha. Retiming synchronous data-flow graphs to reduce execution time. *IEEE Trans. Signal Processing*, 49:2397–2407, 2001. 65

[PAPN13] Maxime Pelcat, Slaheddine Aridhi, Jonathan Piat, and Jean-François Nezan. *Physical Layer Multi-Core Prototyping*, volume 171 of *Lecture Notes in Electrical Engineering*. Springer London, London, 2013. 35, 84

[PBR09] J. Piat, S.S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *IEEE Workshop on Signal Processing Systems, 2009. SiPS 2009*, pages 145–150, October 2009. 15, 18, 25, 46, 98

[PDH+14] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean François Nezan, and Slaheddine Aridhi. PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming. In *EDERC*, page 36, Italy, September 2014. 23, 61, 71, 84, 98

[PL95] José Luis Pino and Edward A. Lee. Hierarchical static scheduling of dataflow graphs onto multiple processors. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 4, pages 2643–2646. IEEE, 1995. 14

[Pre] Preesm. 60, 84

[RA01] S. Ranaweera and D. P. Agrawal. Scheduling of periodic time critical applications for pipelined execution on heterogeneous systems. In *International Conference on Parallel Processing, 2001*, pages 131–138, September 2001. 65

[RV10] Yves Robert and Frédéric Vivien, editors. *Introduction to scheduling*. Chapman & Hall/CRC computational science series. CRC Press, Boca Raton, Fla., 2010. 43

[SB09a] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded multiprocessors: scheduling and synchronization*. Signal processing and communications. Taylor & Francis, Boca Raton, 2nd ed edition, 2009. 35

[SB09b] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2009. 41

[SGB06] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Sixth International Conference on Application of Concurrency to System Design, 2006. ACSD 2006*, pages 276–278, June 2006. 37, 61

[SGC16] F. Siyoum, M. Geilen, and H. Corporaal. End-to-End Latency Analysis of Dataflow Scenarios Mapped Onto Shared Heterogeneous Resources. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(4):535–548, April 2016. 65

[SGTB11] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Modeling and Simulation 2011 International Conference on Embedded Computer Systems: Architectures*, pages 404–411, July 2011. 13

[SSKH13] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013. 34, 35

[Tar76] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, June 1976. 71

[TGB+06] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, pages 185–194, Napa, CA, USA, 2006. IEEE. 12, 13

[ZDP+13] Z. Zhou, K. Desnos, M. Pelcat, J. F. Nezan, W. Plishker, and S. S. Bhattacharyya. Scheduling of parallelized synchronous dataflow actors. In *2013 International Symposium on System on Chip (SoC)*, pages 1–10, October 2013. 9

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Etude et implantation d'algorithmes pour ~~le placement et~~ l'ordonnancement d'applications Dataflow

Nom Prénom de l'auteur : DEROUI HAMZA

Membres du jury :
- Madame SEROT Jocelyn
- Monsieur JEANNOT Emmanuel
- Monsieur MOY Matthieu
- Madame MUNIER-KORDON Alix
- Monsieur NEZAN Jean-François
- Monsieur DESNOS Karol

Président du jury : M. SEROT Jocelyn

Date de la soutenance : 06 Décembre 2019

Reproduction de la these soutenue

☒ Thèse pouvant être reproduite en l'état (avec titre modifié)
☐ Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 06 Décembre 2019

Signature du président de jury

Le Directeur,

M'hamed DRISSI

INSTITUT NATIONAL des SCIENCES APPLIQUÉES - RENNES

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

INSA RENNES

Titre : Étude et implantation d'algorithmes pour l'ordonnancement d'applications Dataflow

Mots clés : Architectures multi-cœur, Modèle de flot de données, Évaluation de performances

Résumé : La complexité des architectures MPSoC (Multiprocessor System-on-Chip) augmente de manière exponentielle pour répondre à la demande croissante en puissance de calcul des applications DSP (Digital Signal Processor). Les architectures MPSoC modernes, telles que les architectures à cœurs multiple, incorporent déjà des centaines d'éléments de traitement (PE) dans une seule puce et prévoient d'intégrer jusqu'à un millier de PE dans un avenir proche. Conséquemment, la programmation des architectures MPSoC modernes avec les langages de programmation traditionnels basés sur des threads est devenue de plus en plus complexe. Dans ce contexte, les modèles de calcul de flux de données (MoC) sont devenus des paradigmes de programmation populaires offrant à la fois une grande analysabilité et une expression intuitive du parallélisme d'une application DSP basée sur le modèle de graphe de tâches.

Dans cette thèse, nous proposons de nouvelles techniques pour l'évaluation du débit maximal et de la latence minimale du modèle IBSDF (Interface-Based Synchronous Dataflow), ciblant les architectures MPSoC avec des ressources illimitées. Le modèle IBSDF est un modèle hiérarchique à comportement statique qui permet d'évaluer certaines métriques de performance au moment de la conception. Cependant, les méthodes d'évaluation classiques consistent à aplatir la hiérarchie du modèle IBSDF en un graphe de flux de données non hiérarchique comportant un nombre exponentiel de tâches rendant son évaluation difficile, voire impossible. Les nouvelles techniques que nous proposons évaluent les performances des graphes IBSDF de manière modulaire sans aplatir leur hiérarchie. Ainsi, nous avons pu évaluer de très grand graphes IBSDF en quelques secondes, contrairement à l'approche classique qui ne permet pas d'obtenir un résultat.

Title: Study and implementation of algorithms for scheduling Dataflow applications

Keywords: MPSoC architectures, Dataflow models, Performance analysis, Throughput, Latency

Abstract: The complexity of Multiprocessor System-on-Chip (MPSoC) architectures is increasing exponentially to meet the rising computation power demand of Digital Signal Processor (DSP) applications. Modern MPSoC architectures like the many-cores architectures, already embed hundreds of Processing Elements (PEs) in one single chip, and plan to integrate up to thousand PEs in the near future. As consequences, programming modern MPSoC architectures with the traditional thread-based programming languages have become more and more complex. In this context, Dataflow Models of Computation (MoCs) have emerged as popular programming paradigms which offer both, a great analyzability and an intuitive expression of the parallelism of a DSP application based on a task graph pattern.

In this thesis, we propose new techniques for the evaluation of the maximum throughput and the minimum latency of the Interface-Based Synchronous Dataflow (IBSDF) MoC, targeting MPSoC architectures with unlimited resources. The IBSDF MoC is a hierarchical model with a static behavior which enables the evaluation of some performance metrics at design-time. However, classical evaluation methods consist on flattening the hierarchy of the IBSDF model into a flat dataflow graph with an exponential number of tasks that makes it hard even impossible to evaluate. Our new techniques evaluate the performance of IBSDF graphs in a modular way without flattening their hierarchy. Thus, we have been able to evaluate very large IBSDF graph in few seconds, compared to the classical approach which fails to return a result.