Extension and Analysis of Dataflow Models of Computation for Embedded Runtimes

Florian Arrestier

DOCTORAT BRETAGNE LOIRE / MATHSTIC

INSA RENNES

THÈSE DE DOCTORAT DE

L'INSTITUT NATIONAL DES SCIENCES APPLIQUEES DE RENNES

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

Par

Florian ARRESTIER

Extension and Analysis of Dataflow Models of Computation for Embedded Runtimes

Thèse présentée et soutenue à Rennes, le 10 novembre 2020
Unité de recherche : IETR
Thèse N° :20ISAR 17 / D20 - 17

Rapporteurs avant soutenance :

Tanguy RISSET — Professeur des Universités, INSA Lyon
Renaud PACALET — Directeur d'études, Télécom Paris

Composition du Jury :

Président :	Alix MUNIER-KORDON	Professeure des Universités, Sorbonne Université - LIP6
Examinateurs :	Tanguy RISSET	Professeur des Universités, INSA Lyon
	Renaud PACALET	Directeur d'études, Télécom Paris
	Jocelyn SÉROT	Professeur des Universités, Institut Pascal
Dir. de thèse :	Daniel Ménard	Professeur des Universités, INSA Rennes
Encadrant :	Karol Desnos	Maitre de conférences, INSA Rennes
Encadrant :	Eduardo Juarez	Assistant Professor, Universidad Politécnica de Madrid

Pour toi, Maman.

Table of Contents

Acknowledgements

Pour commencer, j'aimerais remercier mes encadrants de thèse, à savoir le docteur Karol Desnos et les professeurs Daniel Ménard et Eduardo Juarez. Merci de m'avoir accordé votre confiance, que ce soit en me confiant ce sujet ou tout du long de la thèse par l'autonomie que vous m'avez donné. L'encadrement d'une thèse est, pour ma part et au delà des qualités du candidat, l'un des facteurs déterminant de la réussite de celle-ci et je ne pense pas que j'aurais pu en avoir un meilleur. A Karol, je te remercie pour toutes ces discussions autour du café, souvent sans rapport direct, et ça fait du bien, avec la thèse. Je te remercie aussi d'avoir su me redonner la confiance nécessaire pour aller au bout lors de mes quelques moments de doute. A Eduardo, je te remercie pour l'accueil chaleureux que toi et ton équipe, au CITSEM, m'avez réservé lors de mes visites à Madrid. Enfin, à Daniel, que serait un bateau sans son capitaine pour le diriger ? Je ne pense pas que j'aurais signé pour ces 3 ans avec un autre directeur que toi, merci pour la confiance que tu m'as accordé de mon premier stage d'études avec Alexandre à la thèse.

Je tiens également à remercier le professeur Tanguy Risset et monsieur Renaud Pacalet d'avoir fait partie de mon jury de thèse et surtout d'avoir pris le temps, pendant leurs vacances d'été, de relire mon manuscrit. Merci pour vos remarques et retours constructifs. Je tiens également à remercier la professeur Alix Munier-Kordon d'avoir présidée mon jury de thèse et le professeur Jocelyn Sérot pour en avoir fait partie.

Je voudrais aussi remercier l'ensemble de l'équipe de VAADER, pour son accueil, sa bonne ambiance et ses multiples repas et barbecues d'équipe. Évidemment, je remercie tous mes collègues doctorants, Alexandre Honorat, Guillaume Gautier, Ketty Favre, Nico-

las Sourbier, Leonardo Suriano, Claudio Rubattu, Florian Lemarchand, Thomas Amestoy, et enfin, mais pas des moindres, Alexandre Tissier. Vous avez su me donner envie de venir au travail tous les jours, non pas par devoir mais par envie et surtout avec plaisir ! Pour toutes ses pauses, parfois longues il est vrai, et durant lesquelles de nombreux débats, souvent d'un niveau douteux avouons le, ont eut lieu, merci. Sans vous, ça n'aurait pas été la même expérience, c'est certain.

Que seraient ces remerciements si je n'évoquais pas la personne qui m'a convaincu de faire de la recherche ? Je parle bien entendu du docteur Alexandre Mercat. Merci à toi Alex, sincèrement, sans toi, je n'aurais sans doute jamais eu l'idée de réellement faire une thèse. Mais au-delà du cadre de la recherche, merci pour tout ce que tu m'as apporté sur le plan personnel.

Enfin, je voudrais remercier ma famille pour tout le soutient qu'ils m'ont apportés durant ces 3 ans. Que ce soient mes grands parents, mon grand frère, Johann, ou mon papa, Jean-Pierre, je sais que vous avez toujours cru en moi et que vous êtes fier de moi et de mon parcours. Evidemment, je pense à toi, maman. Même si tu n'étais plus avec nous durant ma thèse je sais que tu aurais été extrèmement fier de moi. Tu nous manques tous les jours.

PS: J'aimerais aussi remercier l'ensemble de l'équipe du VandB pour m'avoir permis de me détendre durant ces 3 années de thèse.

PS2: Je voudrais aussi remercier l'Europe et la France, et en particulier les projets CERBERO et ARTEFACT d'avoir financés ma thèse ainsi que, pour au moins une partie, le VandB longschamps.

CHAPTER 1

Introduction

1.1 Context

Our modern society is surrounded by a myriad of embedded systems: from the simple thermostats in our homes, the connected pedometers, to more complex systems such as our phones, our cars and to the highly complex and critical automated navigation systems on airplanes. The number of systems entering our daily life is expected to keep rising in the next few years, with an expected growth of 6.1% from 2020 to 2025 [**marketsandmarkets_embedded_2020**].

Embedded Systems Constraints

We define an embedded system as an integrated electronic and computing system designed for a specific purpose. The combination of specialized hardware inside embedded systems increases the complexity needed for programming them. Indeed, it is not uncommon to have dedicated hardware for encoding and decoding video, or for performing Artificial Intelligence (AI) computations, for managing the Inputs/Outputs (IOs) of the system or for connecting the systems to the internet through modern 4G and 5G networks. All of these dedicated hardware accelerators are generally connected to a Central Processing Unit (CPU) which will need to communicate with them in an efficient manner.

Hence, in the past decades, there have been many developments toward facilitating the exploitation of the computational resources of these embedded systems.

Designing and programming embedded systems require taking into account functional constraints and non-functional constraints. In [**desnos_memory_2014**], the author classify the constraints applied to embedded systems into three categories, namely the *application constrains*, the *cost constraints* and the *external constraints*. In our proposed classification, we split the *application constrains* into *functional constraints* and *design constraints*.

- **Functional constraints** refer to the constraints that an embedded system must satisfy to serve its intended purpose. These functional constraints include, but not exhaustively, performance requirements, reliability of the system or the intrinsic purposes of the system (e.g. encoding and decoding a video).
- **Cost constraints** refer to the constraints that will affect the final cost of the embedded system such as development cost, production cost, recycling cost, etc.
- **External constraints** refer to external requirements that an embedded system must satisfy and that are non essential to fulfill the functional constraints of the system, such as standards compliance (e.g. MPEG standard) or environment constraints (e.g temperature, pressure, etc.).
- **Design constraints** refer to all non-functional constraints (e.g. memory usage, energy consumption, etc.) composing the design space of possible solutions once components satisfying all of the above constraints have been selected. These constraints are orthogonal to the functional constraints, which means that finding a solution that satisfies a design constraint should not question the fulfillment of functional constraints. Indeed, design constraints are constraints applied on top of satisfied functional constraints and thus, must be compatible with them. For instance, finding the most energy efficient solution can not affect the intrinsic functional aspect of the system as the search space is restricted to compatible solutions.

All of these constraints might result in orthogonal solutions. For instance, finding a energy optimal solution might result in sub-optimal memory consumption constraint or in an higher cost of the system. Hence, the design of an embedded system most often boils down to finding and accepting the most appropriate, or acceptable, trade-off across all of the possible solutions. The process of finding this trade-off is often refereed to as

the Design Space Exploration (DSE) of a system and has been an active field of research with the development of dedicated tools.

Reducing the Software Productivity Gap

The famous Moore's law [**moore_cramming_1998**] that predicts that the number of transistor inside of an integrated circuit would double every 18 months is reaching its end. For the past two decades, a new law called the "More than Moore" has appeared [**zhang_more_2009**; **waldrop_chips_2016**] in order to keep the Moore's law going by leveraging the heterogeneity of modern System on Chips (SoCs).

In [**leiserson_theres_2020**], authors show that even though hardware is not evolving as fast as it was, there is now a lot of room for improving current software. Indeed, as mentioned in [**zhang_more_2009**] the software productivity, i.e the complexity of the program that are written, of programmers has not been keeping up with the evolution rate of hardware (2.5 times slower in 2009). This differential in productivity in regard to the evolution of hardware is refereed to as the **software productivity gap**. One of the reason behind the software productivity gap is that one of the most popular programming language for embedded systems still remains the C programming language, a low level and imperative language not adapted to the modern massively parallel and heterogeneous systems. Hence, recent years have seen the development of high level programming languages and rapid prototyping techniques to bridge the software productivity gap. One of the programming paradigm intended for higher software productivity of programmers is the dataflow programming paradigm.

Dataflow Programming

Dataflow Models of Computation (MoCs) is a programming paradigm that was developed in order to naturally capture the parallelism of applications. In dataflow MoCs, applications are described as graphs where vertices are computational entities of different level of granularity, from the addition instruction level of granularity to complex processing kernel level of granularity; and edges are data communication channels.

Many dataflow MoCs have been proposed since the original work of Kahn in 1974 [**kahn_semantics_1974**]. Every proposed dataflow MoC comes with its own semantics and execution rules that are designed for expressiveness, compactness or efficiency of the model. For instance, static models such as the Synchronous DataFlow

(SDF) [**lee_synchronous_1987**] and its derivatives are tailored for modeling static applications with strong execution guarantees such as memory boundness or the absence of deadlock. On the other end of the spectrum, dynamic dataflow MoCs have the same expressiveness as Turing machines [**buck_scheduling_1993**; **lee_dataflow_1995**]. Consequently these models can not statically derive schedules, or check for memory boundness for the infinite execution of an application and need to have these analysis deferred at runtime. The popularity of the dataflow MoCs comes from their abstraction of the underlying implementation of computational kernels and, thus, their compatibility with existing code.

Moreover, outside of the academic formalism of the dataflow MoCs, dataflow based tools have also emerged due to the recent explosion of the deep learning research area such as the Tensorflow tools [**abadi_tensorflow:_2016**]. Similarly, a new dataflow based standard, called OpenVX [**kronos_group_openvx_2013**], has emerged for modeling computer vision application. More generally, there is a trend in the academic and the industry to go toward tools with higher level of abstraction (often dataflow based), increasing the productivity of the programmer [**elliott_national_2007**; **augonnet_starpu:_2009**; **pelcat_preesm:_2014**; **heulot_runtime_2015**]. These tools abstract the complexity of modern heterogeneous Multi-Processor System on Chips (MPSoCs) to the programmer with the aim of high performance generated software code or hardware layout.

1.2 Objectives and Contributions of this Thesis

There is a variety of dataflow MoCs available in the literature, each tailored for specific use cases. Using dataflow graphs as an high level programming paradigm and as en execution model offers benefits such as compile time analysis, predictability, abstraction from hardware, etc. However, dynamic dataflow MoCs require a portion of the analysis to be done at runtime. For instance, deadlock analysis or scheduling are only possible at runtime in fully dynamic MoCs. Therefore, in order not to invalidate satisfied design constraints of embedded systems, *the overhead of runtime analysis should not interfere with the application functionality.*

Moreover, with the growing complexity of modern embedded systems architecture (e.g. many core platforms, heterogeneous MPSoCs, etc.), there is a need for efficient rapid prototyping tools to explore the DSE of these systems [**augonnet_starpu:_2009**;

gautier_xkaapi:_2013; **pelcat_preesm:_2014**; **heulot_runtime_2015**; **dauphin_odyn_2**
Finally, these tools should also be flexible enough to allow for developing, comparing and testing new algorithms easily.

This thesis studies dataflow MoCs from the model aspect to the implementation of a dedicated embedded runtime. The main objectives of this thesis are 3 folds:

1. Propose an extension to existing dataflow MoCs that improve their expressiveness without adding complexity to their analysis.
2. Develop a novel intermediate representation for the management of dynamic dataflow based applications in an embedded context.
3. Provide a research framework for experimenting, and integrating analysis algorithms for dynamic dataflow applications onto heterogeneous platforms and testing them on real world applications (e.g. signal or image processing applications, machine learning applications, etc.).

To address these objectives, the following contributions are made:

- **Dataflow MoCs Extension**: Chapter 4 proposes a new meta-model called State-Aware dataflow Meta-Model (SaMM) extending the expressiveness and flexibility of a given MoC. SaMM introduces new semantics for dynamically initializing delays using dataflow actors and extends the notion of states of hierarchical dataflow graphs with explicit state persistence and state forwarding. The work presented in this chapter has been accepted in the 2018 SAMOS international conference and published in the ACM International Conference Proceeding Series (ICPS) [**arrestier_delays_2018**]
- **Dataflow MoCs Analysis**: Chapter 5 presents a new lightweight intermediate representation for fast resources allocation of an application graph. The proposed Intermediate Representation (IR) has been implemented into the Spider tool [**heulot_runtime_2015**] as a proof of concept for the resource allocation of image processing and machine learning applications. The work presented in this chapter has been accepted in the 2019 EMSOFT international conference and published in the ACM Transactions on Embedded Computing Systems (TECS) journal [**arrestier_numerical_2019**].
- **Dataflow MoCs Implementation**: Chapter 6 presents an improved implementation of the existing Spider runtime [**heulot_runtime_2015**] called Spider 2.0. Spider 2.0 is an embedded runtime for experimenting with dataflow analysis algorithms for reconfigurable applications. The Spider 2.0 runtime uses the Parame-

terized and Interfaced Synchronous DataFlow (πSDF) MoC [**desnos_pimm:_2013**] extended with the SaMM delay semantics from Chapter 4.

All of this work was developed within the European project CERBERO and the French ANR project ARTEFACT. Additionally, this thesis was co-supervised by Eduardo Juarez from the Universidad Politechnica de Madrid (UPM), which was also part of the CERBERO consortium.

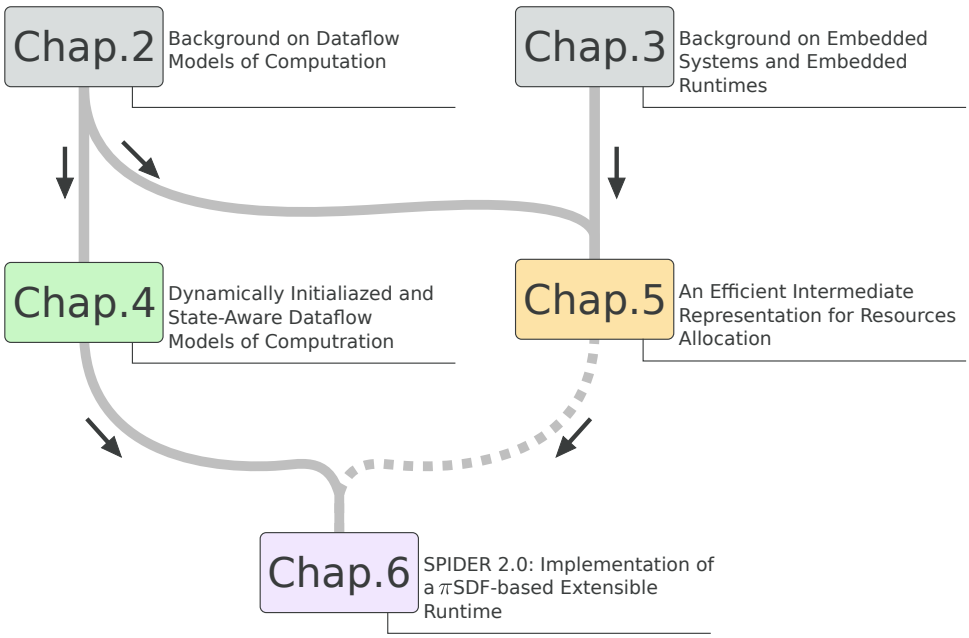


Figure 1.1 – Summary of the organization and contributions of this thesis. Colored chapters correspond to contributions chapters and gray chapters correspond to state-of-the-art chapters. Filled lines correspond to direct connections between chapters and dotted lines correspond to possible connection that was explored during this thesis.

Figure 1.1 illustrates the organization of this document and emphasizes the relationship between the different contribution of this thesis. State-of-the-art chapters, Chapters 2 and 3, are colored in gray and the colored chapters correspond to the contribution chapters (Chapters 4 to 6). The direct relationship between chapters is marked by a gray line. The dotted line between Chapter 5 and Chapter 6 indicates a potential connection that was not explored during this thesis due to a lack of time.

Outline

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

In Part I, Chapter 2 presents the concepts of dataflow MoCs and emphases on the specific MoCs used in this work. Then, Chapter 3 presents the challenges of designing an embedded runtime for heterogeneous MPSoCs along with existing solutions.

In Part II, limitations of current dataflow MoCs are presented with the proposition of a new meta-modeling technique called SaMM in Chapter 4 that aims at removing some of these limitations. Then, Chapter 5 introduces a new IR for dataflow graphs tailored for on-the-fly resources allocation, which involve mapping, scheduling and memory allocation, in an embedded runtime context. Chapter 6 presents an embedded runtime for MPSoCs developed during this thesis and that takes on the legacy of the existing runtime Spider [**heulot_runtime_2015**] with improved performances. Finally, Chapter 7 concludes this work and lean into the future of the studied themes of this thesis.

Part I

Background

CHAPTER 2

Dataflow Model of Computations

2.1 Introduction

The computational complexity of modern applications has been rising exponentially for the past 50 years. In parallel, to keep up with this increase in computational power demand, the complexity architecture of embedded systems has also been increasing exponentially. As computers become more and more complex, it is necessary to increase the level of abstraction for programming them. The first abstraction layer when programming computers is the programming language. When the C language replaced assembler [**kernighan_c_2011**], it revolutionized computer programming due its much higher abstraction, while maintaining good performances.

While in the recent years Multi-Processor System on Chips (MPSoCs) have become more and more parallel with an increasing number of cores and dedicated hardware accelerators, the C programming language is inherently sequential and was not designed for parallel computing. For instance the AMD EPYC 7742 processor [1] possess 64 processing cores and 128 threads. Programming such highly parallel processor requires a change in the programming paradigm.

Graphical based programming languages offers an high level of abstraction and are a popular alternative to classical text based programming languages such as the C language.

1. **https://www.amd.com/fr/products/cpu/amd-epyc-7742** .

Labview [**johnson_labview_2006**] and Matlab Simulink [**klee_simulation_2018**] are among the most popular diagram-based programming languages used in academic and industrial environments.

This thesis focuses on the study of diagram-based MoCs called dataflow MoCs. Dataflow MoCs are commonly used to model stream processing applications in a wide range of domains such as video and audio processing [**park_extended_1999**; **pelcat_preesm:_2014**; **theelen_scenario-aware_2006**], telecommunications [**dardaillon_new_20**] computer vision [**kronos_group_openvx_2013**], and machine learning [**abadi_et_al._tensorflow:_**] Dataflow MoCs and related languages are subject to an increasing popularity due to their advanced analyzability and their natural expressiveness of parallelism.

This chapter is organized as follows. Section 2.2 formally presents the concepts and properties of dataflow MoCs. Then, Section 2.3 present static MoCs with the Synchronous DataFlow (SDF) MoC and one of its many extensions. Section 2.4 present the concept of compositionality in dataflow and two examples of hierarchical dataflow MoC. Dynamic and Reconfigurable MoCs are presented in Section 2.5. Finally, Section 2.6 summarizes and concludes this chapter.

2.2 Dataflow MoCs: overview

In computer science, a Model of Computation (MoC) describes the set of elementary operations and rules that define how a computer program is executed. Similarly to the laws of physics that govern how our universe *works* and how objects interact with each other within it, a MoC govern how a program works and what are the interactions between the MoC objects. It is important, however, to differentiate between MoCs and programming languages as a programming language can be used to implement different MoCs. For instance, it is possible to implement a Turing machine in C, but it is also possible to implement any dataflow MoC with it.

There exists a variety of MoCs such as sequential MoCs (e.g. Turing machines [**turing_computable_1937**], Finite-State Machines (FSMs)), functional MoCs (e.g. lambda calculus [**church_set_1932**; **barendregt_lambda_2012**]), or concurrent MoCs like the Kahn Process Networks (KPNs) [**kahn_semantics_1974**] and dataflow MoCs [**davis_data_1982**].

2.2.1 Dataflow Process Network

The Kahn Process Network (KPN) [**kahn_semantics_1974**] is probably the first known dataflow MoC. Kahn defined a KPN as a network of concurrent tasks connected by directed unbounded First-In First-Out Queues (FIFOs) channels transmitting data tokens. In a KPN, data tokens are indivisible, produced only once and consumed only once, and they cannot be shared by tasks.

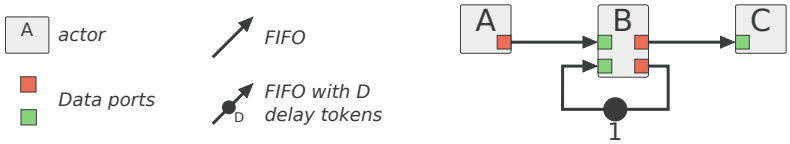


Figure 2.1 – DPN graph example.

In [**lee_dataflow_1995**], Lee and Parks introduced a specialization of KPNs, called the Dataflow Process Network (DPN), and specified the semantics of dataflow networks. Formally a Dataflow Process Network is defined as follows:

Definition 2.2.1 (DPN definition)
A Dataflow Process Network (DPN) is a directed graph $G = \langle A, F \rangle$ where:

- ***A*** *is the set of vertices called actors of G representing computational tasks. Each actor $a \in A$ defined as a tuple $a = \langle D^{in}, D^{out}, \mathbb{F}, R \rangle$ embeds:*
  - *D^{in}, D^{out} are the data input and output ports of the actor, respectively.*
  - *$\mathbb{F} = \{F_1, F_2, ..., F_n\}$ are the firing rules of the actor. A firing rule $F_i \in \mathbb{F}$ is condition that when satisfied, can start an execution (or firing) of the associated actor.*
  - *R: A set of data rates. A rate is the number of data tokens consumed at a given data input port $d_i^{in} \in D^{in}$ or produced on a data output port $d_i^{out} \in D^{out}$ corresponding to a specific firing rule.*
- ***F*** *is the set of edges of the graph G, representing unbounded FIFO data queues. These FIFOs are the medium through which data tokens are exchanged between actors of the graph. Each FIFO $f \in F$ is associated with a produced and a consumer actor, producing and consuming data tokens on the FIFO, respectively. A FIFO is also associated to a delay value which correspond to the number of initial data tokens present in the FIFO at the start of the application.*

The semantics of the DPN MoC only specifies the execution behavior of the graph, but does not say anything about the internal behavior of actors. The description of the internal

behavior of any actor of a DPN graph can be implemented using various programming languages such as C, Python, Java or VHDL. Specific dataflow programming languages also exist, such as CAPH [**serot_implementing_2011**] or CAL [**eker_cal_2003**]. These languages describe both the graph structure and the internal behavior of the actors.

2.2.2 Dataflow MoC: Properties

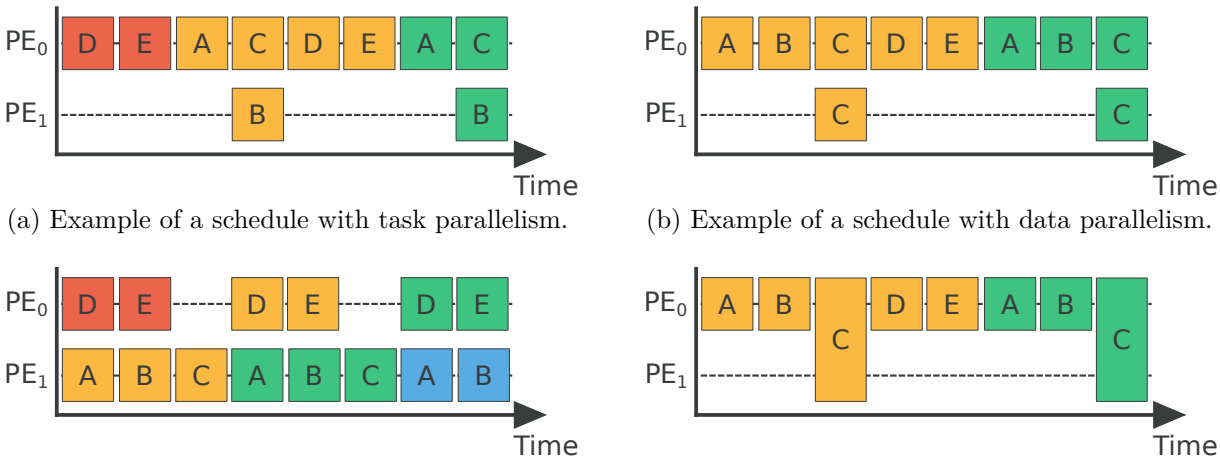
Since the original dataflow MoCs proposed in the early 80's [**goos_first_1974**; **davis_data_1982**], numerous dataflow MoCs have been introduced. New dataflow MoCs are often introduced to fill in the gaps of pre-existing models. For instance, some MoCs extends the semantics of existing models in order to capture a wider range of applications specificities [**park_extended_1999**], or to encompass analysis information directly into the model semantics [**damavandpeyma_schedule-extended_2013**]. Other models introduce new concepts to, or generalize, an existing MoC [**piat_interface-based_2009**; **bilsen_cycle-static_1996**]. Finally, some models are introduced to enforce the analyzability of existing models with strong mathematical formalism [**theelen_scenario-aware_2006**].

2.2.2.1 Parallelism in Dataflow MoCs

In [**zhou_scheduling_2016**], authors identify 4 types of parallelism that are commonly found in dataflow based applications, namely the *task parallelism*, *data parallelism*, the *pipeline parallelism* (or time parallelism), and the *actor level parallelism*. Figures 2.2 illustrate these four types of parallelism with four corresponding schedules. These schedules does not reflect the behavior of any precise dataflow graph. In the different schedules of Figures 2.2, it is assumed that there exists a notion of iterations to the presented schedules, distinguished by their color.

The four types of parallelism are defined as follows.

- **Task Parallelism:** This form of parallelism is illustrated in Figure 2.2a and corresponds to distinct tasks being executed concurrently. For instance, in Figure 2.2a, tasks B and C are executed concurrently.
- **Data Parallelism:** Data parallelism corresponds to the possibility of a task to have multiple instances executed concurrently. This is only possible in dataflow MoCs where actors have no internal states possible. In Figure 2.2b, task C exploits data parallelism as multiple instances are executed at the same time.



(a) Example of a schedule with task parallelism.

(b) Example of a schedule with data parallelism.

(c) Example of a schedule with time parallelism.

(d) Example of a schedule with actor level parallelism.

Figure 2.2 – Illustration of the different typ of parallelism in dataflow graphs.

- **Time Parallelism:** Also refereed as *pipelining*, time parallelism refers to the overlapping of multiple iterations of a given application graph. Figure 2.2c illustrates this property with up to 2 iterations overlapping.
- **Actor level Parallelism:** This final form of parallelism, illustrated in Figure 2.2d, is related to the internal behavior of tasks, with tasks being executed on multiple Processing Elements (PEs) at the same time. For instance, in a hierarchical dataflow MoC, the internal execution of a hierarchical actor onto multiple PEs could be seen as actor level of parallelism [**bhattacharya_parameterized_2001**; **neuendorffer_hierarchical_2004**; **piat_interface-based_2009**; **desnos_pimm:_2013**].
  An other source for actor level parallelism is to use a programming language that allows parallel computation for describing the internal behavior of the actor. The use of OpenMP [**chapman_using_2008**] inside the specification of an actor for the parallelization of for-loops is a common practical example.

2.2.2.2 Dataflow MoCs Properties

In [**desnos_memory_2014**], Desnos proposed a classification of properties of dataflow MoC. The list of properties proposed is used for comparing dataflow MoCs and is split into two distinct set of properties, namely the properties objectively measurable and the properties that can not be objectively measured. In the following definitions,

the *decidability*, *determinism*, *compositionality* and *reconfigurability* properties are objectively measurable properties of dataflow MoCs. On the other hand, the *predictability* and *expressiveness* properties are not objectively measurable but provide nonetheless useful information when comparing different MoCs.

- **Decidability:** A dataflow MoC is decidable if it is possible to prove statically, i.e. at compile time, that an application described with this MoC will execute within bounded memory, and that there exists at least one sequence of firings, called a schedule, for which the application will execute without deadlock.
- **Determinism:** The determinism property of a dataflow MoC is for applications modeled with this MoC to solely depends on the data flowing in the application and not on external factors [**lee_dataflow_1995**].
- **Compositionality:** This is the property of a dataflow MoC to be analyzable independently from the internal specification of the actors composing an application graph modeled with this MoC [**tripakis_compositionality_2013**]. In other words, in a compositional MoC, modifying the internal behavior of any actor does not affect the analyzable properties (consistency, liveness, etc.) of a given graph.
- **Reconfigurability:** A reconfigurable dataflow MoC is a MoC in which the firing rules of actors can change during the execution of an application. In order to maintain some predictability or partial determinism, reconfiguration should only happen at quiescent points [**neuendorffer_hierarchical_2004**].
- **Predictability:** The predictability property of a dataflow MoC encompass all the behaviors of a MoC that are related to the change of its firing rules. For instance, in [**neuendorffer_hierarchical_2004**], the predictability is defined in respect to how often the firing rules of a MoC changes, but in [**heulot_runtime_2015**] predictability is defined as a function of how the firing rules of a MoC are changed.
- **Expressiveness:** The expressiveness, or expressive power, of a dataflow MoC represent the range of possible applications that can be representated with this MoC. For instance, the ability to model conditional structures, for-loops, or infinite loops is what defines the expressiveness of a MoC. The more expressive a MoC, the broader the range of applications that can be modeled with it. Static dataflow MoCs have a limited expressiveness as they can not model applications with dynamic behavior, hence dynamic MoCs are more *expressive* than static MoCs.

2.3 Static Dataflow MoCs

There are two main categories among the existing dataflow MoCs, namely the *dynamic* and *static* MoCs. In this section, static dataflow MoCs are presented, starting with the SDF MoC, one of the most studied and derived dataflow MoC, and originally presented in [**lee_synchronous_1987**]. Static dataflow MoCs are a restrictive subset of the DPN MoC [**lee_dataflow_1995**] where the firing rules are fixed and that a complete execution of a graph, if it exists, can be analyzed at compile time. This property restrict the expressiveness of static MoCs, i.e it restricts the class of applications that can be modeled with a static MoC.

This section is not an exhaustive list of existing static dataflow MoCs but introduces the reader to different models in order to understand the changes in semantics that justify the existence of different MoCs. For a more in depth dive into static and dynamic dataflow MoCs, the interested reader will find in [**bhattacharyya_decidable_2013**] and in [**bhattacharyya_dynamic_2019**; **bouakaz_survey_2017**] more complete surveys on static and dynamic dataflow MoCs, respectively.

2.3.1 Synchronous DataFlow

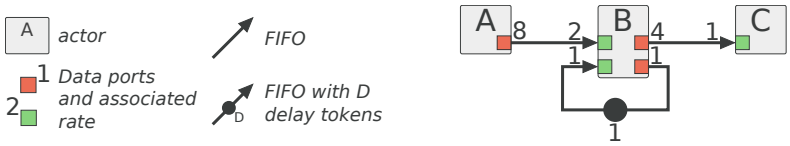


Figure 2.3 – SDF graphical semantics and a graph example.

The Synchronous DataFlow (SDF) MoC [**lee_synchronous_1987**] is the most popular static specialization of the DPN MoC [**lee_dataflow_1995**]. Firing rules of the SDF MoC define data token production and consumption rates of actors as fixed scalars, meaning that rates are set at design time and are fixed for the entire execution of the application. The graphical semantics of the SDF MoC and an example of SDF graph are presented in Figure 2.3.

Formally, an SDF graph $G = (\mathbb{A}, \mathbb{F})$ contains a set of actors $\mathbb{A}$ that are interconnected through a set of Fifos $\mathbb{F}$. An actor $a \in \mathbb{A}$ reads data tokens from its input ports and produces data tokens on its output ports. The execution of an actor is called a firing and for an actor to fire, enough data tokens need to be available on all of its input ports. In

the graph of Figure 2.3, actor B can only fire when 2 data tokens are present on the FIFO ($\vec{AB}$) and 1 data token is present on its self-loop. The initial data tokens of a FIFO $f \in \mathbb{F}$ are called delays. The value n of the delay is the number of initial data tokens of f.

The popularity of the SDF MoC comes from its great analyzability. Indeed, using static analyses, the consistency and liveness properties of an SDF graph can be verified. When an SDF graph is schedulable, i.e it is consistent and live, a minimal sequence of firings of the actors exists to achieve an infinite execution with bounded memory. Such minimal sequence is called a *graph iteration* and the number of firings of each actor is given by the coefficients of the Repetition Vector (RV) of the graph. Figure 2.3 presents an SDF graph that is consistent and live. For each graph iteration, actor A is executed 1 time, actor B 4 times, and actor C 16 times.

SDF: Consistency and Liveness

There are two important properties of SDFs Graphs (SDFGs) that guarantee the execution of an SDFG in bounded memory and that the execution will not deadlock, namely the *consistency* and the *liveness* properties.

- **Consistency** is the property of an SDFG means that no data token will indefinitely accumulate in any FIFO of the graph. Consistency is checked through the analysis of the topology matrix Γ associated with an SDF graph [**lee_synchronous_1987**]. Formally, $\Gamma(i, j)$ is the number of data tokens produced or consumed by actor i on FIFO j. $\Gamma(i, j)$ is a positive number if the actor i produces data tokens on the FIFO j and a negative number if the actor consumes data tokens. The graph is consistent if and only if $rank(\Gamma) = |A| - 1$, with $|A|$ the number of connected actors in the graph. The Repetition Vector (RV), noted q, is defined as the smallest non-zero integer vector verifying $\Gamma * q = 0$.
  Finding the vector q, corresponding to the nullspace of the topology matrix Γ, for large SDFGs is computationally expensive. An efficient algorithm for computing the repetition vector of an SDFG is given in [**bhattacharyya_software_1996**]. This algorithm computes the repetition vector by computing the least common multiplier of the rationals of the different edges of an SDFG.
- **Liveness** is the property of a graph to run indefinitely without any deadlocks. A live graph has sufficient initial data tokens to execute a full iteration and each graph iteration starts with the same number of initial data tokens. Different approaches to the problem of verifying liveness of a graph exist

in the literature. The most common approach is called the Symbolic Execution (SE) [**bilsen_cycle-static_1996**]. SE consists of performing a symbolic execution of a graph to check if the graph can go through a full iteration and returns to the same initial state. SE is not suited for large graph as it takes a long time to perform the symbolic execution. Mathematical approaches have been researched to make this analysis faster. In [**ghamarian_liveness_2006**], Ghamarian et al. give a necessary and sufficient condition based on the analysis of the strongly connected components of an SDF graph.

Static extensions to the SDF MoC have been proposed to enforce its expressiveness and conciseness while maintaining the same level of analyzability and predictability. The Cyclo-Static Dataflow (CSDF) MoC [**bilsen_cycle-static_1996**] has the same expressiveness as the SDF MoC but is more concise. In CSDF, data rates change according to static cycles defined at the creation of the graph. The Interfaced Based Synchronous Dataflow (IBSDF) MoC [**piat_interface-based_2009**] enforces the compositionality and expressiveness of the SDF MoC by adding explicit and well-defined levels of hierarchy. In an IBSDF graph, actors can be defined by the mean of other IBSDF graphs. The Parameterized DataFlow (PDF) [**bhattacharya_parameterized_2001**], Schedulable Parametric Dataflow (SPDF) [**fradet_spdf:_2012**], and Parameterized and Interfaced Synchronous DataFlow (πSDF) [**desnos_pimm:_2013**] are reconfigurable extensions of the SDF that enforce dynamic reconfiguration of dataflow graphs.

2.3.2 Cyclo-Static Dataflow and Affine DataFlow

The CSDF MoC is an extension and a generalization of the SDF MoC introduced in [**bilsen_cycle-static_1996**].

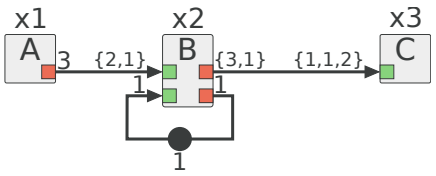


Figure 2.4 – A CSDF graph example.

In a CSDF graph, the data rates of the Fifos of the graph are defined in term of finite and cyclic sequences of fixed scalar values. For instance, in the graph of Figure 2.4, actor B has an input data rate sequence of 2 and 1 data tokens, corresponding to its first and

second firing, respectively. Similarly, actor B produces 3 and 1 data tokens on its output port which are consumed by the 3 firings of actor C following the consumption sequence of 1, 1 and 2 data tokens, respectively.

Although, the CSDF MoC is a generalization of the SDF MoC, it does not increase its expressiveness. Indeed, in [**parks_comparison_1996**], authors show that every CSDF graph can be transformed into an equivalent SDF graph. Moreover, due the generalization of the firing rules of the CSDF MoC over the SDF MoC, the scheduling of a CSDF graph may be unnecessarily complicate in some situations. However, CSDF graphs are often more compact and have simpler precedence constraints over SDF graphs. For instance, in a CSDF graph, cyclic path may be live without requiring to the need of delayed data tokens as in the SDF MoC. Finally, several years after its introduction, it has been shown in the Section 5.2 of [**thies_empirical_2010**] that the CSDF MoC revealed itself to be impractical and confusing for the user.

The Affine DataFlow (ADF) MoC is a generalization of the CSDF MoC introduced in [**bouakaz_affine_2012**]. The ADF MoC extends the CSDF MoC with a finite sequence of initialization rates. In other words, in addition to the infinite sequence of rates of the CSDF MoC, the ADF MoC proposes some initialization period. The combination of both sequences is called the ultimately periodic sequence. It is worth noting, however, that similarly to the CSDF MoC, the ADF MoC does not increase the expressiveness of the SDF MoC as there exists a transformation from an ADF graph to an SDF graph.

2.4 Hierarchical Dataflow MoCs

«*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*»- Hunt and Thomas [**hunt_pragmatic_2000**]

From building structures out of basic Lego blocks to building complex applications out of basic processing blocks, we are used to the compositionality principle. Compositionality is a feature of programming languages that makes building higher order functions out of simpler one possible. Specifically, a definition of compositionality in dataflow MoCs is given in Section 2.2.2.2.

Compositional dataflow MoCs improve the design, optimization and reusability of dataflow applications. For instance, primitive building blocks such as Fast-Fourier Transform (FFT), convolution filter, Finite Impulse Response (FIR) filter, can be designed, optimized and reused in a multitude of applications.

In dataflow MoC, composing application graphs out of other application graphs is not something new and was originally introduced in [**davis_data_1982**]. However, in the work of Davis et al., *macro functions* representing dataflow subgraphs are just an abstraction and are compositional in the sens of the definition above. Similarly, in the SDF MoC [**lee_synchronous_1987**], composing actors are discussed, in the term of *hierarchical* actors, but the SDF MoC does not offer any compositional mechanism. Several extensions to the SDF MoC were proposed later on, to add compositionality to the MoC [**piat_interface-based_2009**; **tripakis_compositionality_2013**].

2.4.1 Naïve Hierarchical SDF

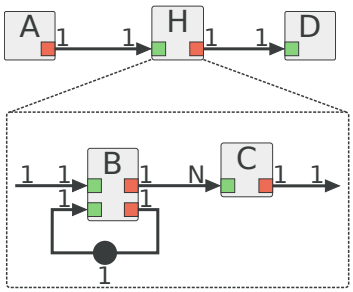


Figure 2.5 – Graph example of a hierarchical SDFG.

The first approach to hierarchical modeling for SDFGs is introduced in [**lee_synchronous_1987**] and consists in defining the behavior of a hierarchical actor using an SDFG as its refinement. This replacement can be seen as *syntactic sugar* as it does not reflect any compositional behavior of the SDF MoC. Indeed, the behavior of an SDF subgraph directly affects the behavior of its parent graph.

Taking Figure 2.5 as an example of a hierarchical SDFG. The top level graph is composed of the three actors A, H and D. Actors A and D are *regular* SDF actors, whereas actor H is a hierarchical actor with its internal subgraph being composed of actors B and C. In the subgraph of H, the number of repetition of the actors directly influence the repetition of the top level graph. Indeed, if the graphs are analyzed independently, the corresponding repetition vectors q_{top} and q_H are given by Equations (2.1) and (2.2) for the top level graph and the subgraph of actor H, respectively. In this scenario, actors A,

H and D have a repetition value of 1, and actors B and C have a repetition value of N and 1, respectively.

$$q_{top} = \begin{matrix} A \\ H \\ D \end{matrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{2.1}$$

$$q_H = \begin{matrix} B \\ C \end{matrix} \begin{bmatrix} N \\ 1 \end{bmatrix} \tag{2.2}$$

However, in a hierarchical SDFG, the inner subgraph of a hierarchical actor is not *encapsulated* and abstracted from its *parent* graph. Thus, in the example of Figure 2.5, the repetition value of actor A is directly influenced by the one of actor B and its repetition value is then N and not 1. Moreover, the presence a delay on the self-loop of actor B is hidden from the top level graph and may cause a potential deadlock. In [**pino_hierarchical_1995**], Pinot et al. provides an hierarchical scheduling framework for SDFGs that relies on the clusterization of SDFGs and define four distinct criteria to check for the validity of the obtained clusters. Authors in [**tripakis_compositionality_2013**] propose a solution to the problem of compositionality of the SDF MoC with the form of a compositional abstraction called Deterministic SDF with Shared Fifos (DSSF) profiles.

2.4.2 Interfaced Based Synchronous Dataflow: Adding Compositionality to SDF

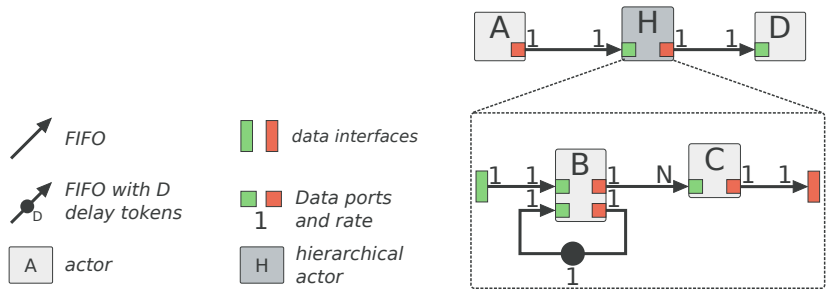


Figure 2.6 – IBSDF graphical semantic and a graph example.

The IBSDF MoC is a generalization of the SDF MoC introduced in [**piat_interface-based_2009**]. Similarly to the SDF MoC, in the IBSDF MoC it is possible to define the behavior of dataflow actors using hierarchical subgraphs. However, the IBSDF MoC introduces the compositionality property to the SDF MoC through the use of data interfaces and specific execution rules for subgraph.

Formally, an IBSDF graph $G = (\mathbb{A}, \mathbb{F}, \mathbb{I})$ contains in addition to a set of actors $\mathbb{A}$ a set of FIFOs $\mathbb{F}$, a set of interfaces $\mathbb{I}$. Interfaces $\mathbb{I} = (I^{in}, I^{out})$ decorrelate the inner definition of a subgraph G from the apparent behavior of the associated hierarchical actor H_G. A *source* interface $i^{in} \in I^{in}$ is a vertex transmitting to the subgraph the data tokens received on the corresponding data input port of the corresponding hierarchical actor. If more data tokens are consumed by the subgraph during an iteration than made available by the interface, the source interface behaves as a circular buffer and produces the same data tokens as many times as needed. Symmetrically, a *sink* interface $i^{out} \in I^{out}$ only transmits the last data tokens produced during a subgraph iteration, and discards any excess of data tokens relatively to the production rate of the parent actor.

The behavior of the interfaces of the IBSDF MoC is one of the key features that makes the MoC compositional. An other important feature is the property of the interfaces of a subgraph to be *write locked* and *read locked* during the entire execution of the subgraph for the source interfaces and the sink interfaces, respectively. Finally, delays have no side effects in hierarchical IBSDF graphs [**piat_interface-based_2009**]. Combining all of these features makes the IBSDF MoC a compositional MoC.

Figure 2.6 presents an example of an IBSDF graph along with the associated semantics. The example graph of Figure 2.6 is the same example as in Figure 2.5 but expressed in the IBSDF MoC. In Figure 2.6, actors A, B, C and D are non-hierarchical actors, and H is a hierarchical actor. In the subgraph of actor H, data input and output interfaces are added for every input and output data ports of actor H, respectively. In Figure 2.6, actor B is executed N times within the subgraph of the hierarchical actor H. Using the semantics of the IBSDF MoC, the data input interface connected to actor B will repeat its data token N times. Hence, as opposed to the non compositional example of Figure 2.5 presented in Section 2.4.1, the repetition vectors given by Equations (2.1) and (2.2) are valid for the graph of Figure 2.6 regardless of the value of N.

Using the compositional property of a dataflow MoC, it is possible to perform hierarchical analysis for specific properties such as the latency or the throughput of IBSDF graphs. In [**deroui_relaxed_2017**], Deroui et al. show that using the hierarchy and the

compositional property of the IBSDF MoC, it is possible to perform faster throughput analysis compared to state-of-the-art approaches.

2.5 Dynamic and Reconfigurable Dataflow MoCs

Static dataflow MoCs offer great analyzability and predictability thanks to fully static firing rules and data rates. However, static MoCs trade all of these analyzability tools for a limited expressiveness. Additionally, many applications are dynamic per nature and can not be fully predictable at compile time. For instance, a computer vision application that reads signalization signs for autonomous vehicles is unpredictable per nature. Indeed, if there are no signalization sign, then no processing needs to be done. Symmetrically, if there are 1, 2 or any number of signalization signs then the application will read each of these signs, hence the processing load can not be known at compile time.

To overcome this limitation, many generalizations of the SDF MoC have been introduced over the years to model dynamic and reconfigurable applications. One of the main challenge of these dynamic dataflow MoCs is to increase the expressiveness of the SDF MoC while maintaining predictability and analyzability as much as possible. Next sections present two reconfigurable dataflow MoCs: the πSDF and the SPDF MoCs.

2.5.1 Parameterized and Interfaced Synchronous DataFlow

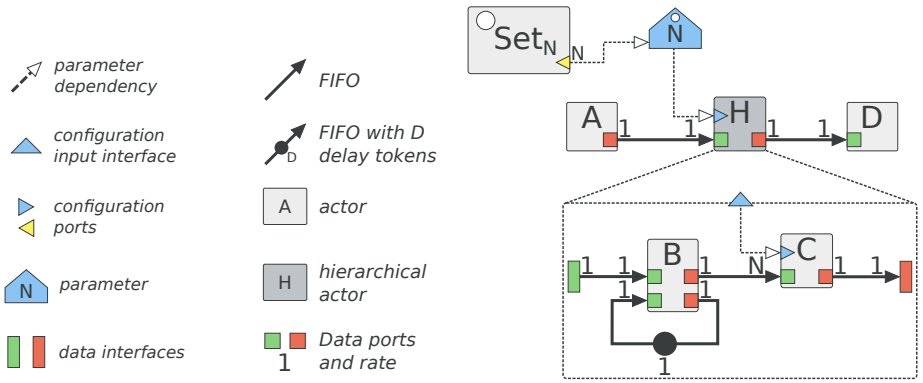


Figure 2.7 – πSDF graphical semantic and a graph example.

The πSDF MoC [**desnos_pimm:_2013**] is a hierarchical and dynamically reconfigurable extension of the SDF and IBSDF MoCs. Similarly as in the IBSDF MoC, in a πSDF graph, a hierarchical actor is an actor whose internal behavior is defined by a πSDF

graph. Figure 2.7 presents an example of a πSDF graph with the associated graphical semantics. Actor H is a hierarchical actor defined by the subgraph formed by actors B and C.

Formally, a πSDF graph $G = (\mathbb{A}, \mathbb{F}, I, \Pi, \Delta)$ contains in addition to a set of actors $\mathbb{A}$ and a set of FIFOs $\mathbb{F}$, a set of hierarchical interfaces I, a set of parameters Π, and a set of parameter dependencies Δ. The hierarchical interfaces of the πSDF MoC [**desnos_pimm:_2013**] are directly inherited from the IBSDF MoC [**piat_interface-based_2009**]. This direct inheritance of the interfaces make the πSDF MoC a compositional MoC which means that the internal specification of the actors composing a graph do not influence its analyzability. In Figure 2.7, the definition of the subgraph formed by actors B and C does not impact the analysis performed on the top-level graph.

Parameters $\pi \in \Pi$ are associated with parameter values $v \in \mathbb{N}$. Parameter values can either be statically defined, derived from other parameters, or dynamically set by configure actors at runtime. A dynamically set parameter is called a *configurable* parameter and its value is set by a configure actor at runtime. The value of a configure parameter is only set **once** per graph iteration to which it belongs. For instance, in Figure 2.7, parameter N at the top-level graph is a configurable parameter set by the configure actor Set_N.

Dependencies $\delta \in \Delta$ are directed edges of the graph that propagate the values of parameters to, and from, configuration input and output ports of actors and parameters. For hierarchical actors, configuration ports are also called configuration interfaces and are considered as static parameters inside the associated subgraph. The combination of parameters Π and dependencies Δ form a so-called parameter dependency tree $T = (\Pi, \Delta)$. Parameter dependencies do not follow the same synchronization and precedence rules as the FIFOs of a dataflow graph. Indeed, in πSDF, parameters are made available, virtually, instantly to every connected data dependency as soon as their values is fixed.

Reconfigurability of the πSDF MoC directly comes from parameters whose values are used to influence different properties, namely the computation of an actor, the rates of the data ports of an actor, the value of another parameter and the number of delays in a FIFO. Importantly, if an actor A has all of its input and output data rate set to 0, then A **will not be executed.** Combining the parameterized data rates and the property of non-execution of actors gives the possibility to change the topology of a πSDF graph dynamically. For instance, a data path can be completely disabled, and therefore non taken into account by any of the consistency and liveness analysis.

In Figure 2.7, parameter N is set the by the actor Set_N and controls the number of firings of actor B inside the hierarchical actor H but, due to the compositional nature of the πSDF MoC, does not affect the analysis of the top-level graph.

Configure Actors

Configurable parameters are set by actors which possess configuration output ports. These actors are called configure actors and have a distinct runtime behavior. Configure actors are the source of reconfigurability in πSDF and thus their execution can only happen at quiescent points [**neuendorffer_hierarchical_2004**; **desnos_pimm:_2013**]. For any configure actor a_{cfg} of a graph G, the following restriction are applied.

- a_{cfg} must be fired **exactly once** per graph iteration of G. The firing of every configure actors must happen before any firing of non-configure actors.
- a_{cfg} must consume data tokens only from data input interfaces and must consume all available tokens during its unique firing.
- Data input and output rates of a_{cfg} must only depend on locally static parameters values of G. Consequently, it is not possible to use a configure actor to set the value of a parameter that is used in the data rates of an other configure actor.
- Data output ports of a_{cfg} are seen as data input interfaces by other actors. Therefore, the same rules of consumption are applied to them

Runtime Operational Semantics

The execution rules for any πSDF graph are given in [**desnos_pimm:_2013**] and reminded here. Let $G = (A, F, \Pi, \Delta)$ be a πSDF graph associated to a hierarchical actor H_G. For every firing of the actor H_G in its containing graph, G starts a new iteration which follows the following steps.

1. Wait for all input parameters of the graph to be available. This step is called a partial configuration of the graph G.
2. Compute the data rates of all input and output data interface using the values of the parameters obtained during the partial configuration of G.
3. Wait for H_G to be fired in its containing graph.
4. Fire every configuration actors, if any, of G. By setting the values of all of its configurable parameters, G reaches a complete configuration.

5. Compute the RV of G and check its consistency and liveness and, if possible, compute a schedule for G.
6. Fire all non-configure actors of G following the derived schedule from the previous step.s
7. Produce the output data tokens and output parameters computed by the actors of the graph on the corresponding data output interfaces and configuration output interfaces, respectively.
8. If necessary, go back to step 3 and start a new firing of the graph.

The runtime operational semantics of the πSDF is similar to the one of the Parameterized Synchronous DataFlow (PSDF) MoC [**bhattacharya_parameterized_2001**]. Steps 1 and 2 correspond to the init phase of a PSDF graph, steps 3 to 5 correspond to the subinit phase and finally steps 6 to 8 correspond to the execution of the body subgraph of a PSDF graph.

Design Patterns in πSDF

With the possibility to selectively enable or disable actors in the πSDF MoC it is possible to build conditional structures in πSDF graphs.

Switch Pattern

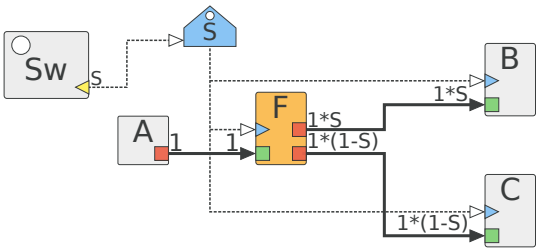


Figure 2.8 – Example of a switch pattern modeled in πSDF.

Figure 2.8 shows an example of a *switch pattern* modeled using the πSDF MoC. In Figure 2.8, the configure actor *Sw* sets the value of the configurable parameter S which will in return sets the value of the data rates of actors F, B and C. Actor F is a special actor called a *fork* actor which splits the input tokens it receives on its unique data input port to all of its data output ports. Importantly, the sum of all the data rates of the data output ports of a fork actor is strictly equal to the data rate of its data input port.

In the graph of Figure 2.8, the values taken by S are supposed to either be 0 or 1. In the case of S being equal to 1, the first data output port of actor F will have a data rate equal to 1 and the second data output port a data rate equal to 0. Similarly, the data input rate of actors B and C will be equal to 1 and 0, respectively. Following the semantics of the πSDF MoC, since all of its data rates are equal to 0, actor C will not be executed. Symmetrically, when S has a value of 0, actor B will not be executed and C will be executed. To summarize, the behavior of the graph of Figure 2.8 is as follows.

- $S = 0$**:** The data token produced by actor A is forwarded to actor C using the fork actor F. Actor B is not executed.
- $S = 1$**:** The data token produced by actor A is forwarded to actor B using the fork actor F. Actor C is not executed.

Select Pattern

Figure 2.9 shows an example of a *select pattern* modeled using the πSDF MoC. The select pattern is the symmetric of the switch pattern presented above. In the graph of Figure 2.9, the configure actor *Sel* sets the value of the configurable parameter S which will in return sets the value of the data rates of actors J, A and B. Similarly to actor F in the graph of Figure 2.8, J is a special actor called a *join* actor which merges all the input tokens it receives to its unique output data port. Importantly, the sum of all the data rates of the data input ports of a join actor is strictly equal to the data rate of its data output port.

Finally, using the same reasoning as the one used for the switch pattern, the behavior of the graph of Figure 2.9 is as follows.

- $S = 0$**:** The data token produced by actor A is forwarded to actor C using the join actor J. Actor B is not executed.
- $S = 1$**:** The data token produced by actor B is forwarded to actor C using the join actor J. Actor A is not executed.

2.5.2 Schedulable Parametric Dataflow

The Schedulable Parametric Dataflow (SPDF) MoC [**fradet_spdf:_2012**] is a reconfigurable and parameterized extension of the SDF MoC.

Reconfiguration in SPDF comes from the possibility to dynamically, i.e. at runtime, change the values of the parameters of the graph. Importantly, parameters in

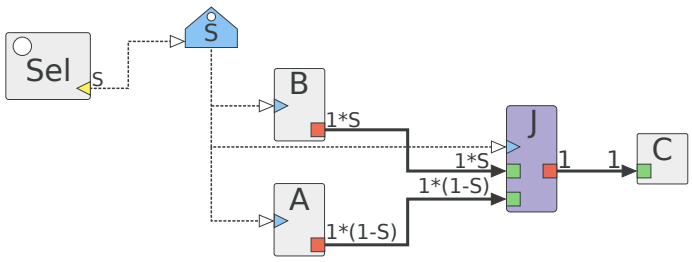


Figure 2.9 – Example of a select pattern modeled in πSDF.

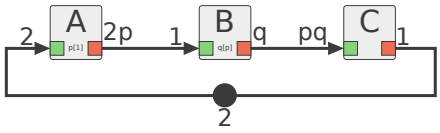


Figure 2.10 – Example of an SPDF graph.

SPDF can not have a null value, thus making the SPDF a dynamic MoC for data rates but not for the topology of a graph. However, the Boolean Parametric Dataflow (BPDF) MoC [**bebelis_bpdf:_2013**], which is an extension of the SPDF MoC, supports dynamic topology but still does not support null data rates.

In SPDF, the value of a parameter is set by an actor called the *modifier* of the parameter. Each parameter has only **one** modifier. Moreover, a parameter is not allowed to change its value arbitrarily; the changes can only occur at precise quiescent points called the period of the parameter which is denoted by $p[\alpha]$, where p is the name of the parameter and α its associated period. In SPDF, the period of a parameter can also be defined as an expression of other parameters.

For instance, in the graph of Figure 2.10, actors A and B are both modifiers of parameters p and q, respectively. The periods of parameters p and q are 1 and p, respectively. In other words, the value of the parameter p changes every 1 firing of actor A and the value of the parameter q changes every p firings of actor B.

One of the key feature of the SPDF MoC is its static analyzability for consistency and liveness of an SPDF graph. Indeed, in SPDF, if the graph is consistent, a symbolic solution to the balance equations of the graph is found, otherwise the graph is not consistent. This symbolic solution is valid for any value of the parameters of the graph. Moreover, the periods of parameters are checked to ensure that they do not introduce any deadlock or inconsistency. This check is performed by the determination of so-called *influence regions* of parameters which correspond to the set of edges which are influenced by a

given parameter. For instance, in Figure 2.10, the influence region of the parameter p is the set of edges $\{\overrightarrow{AB}, \overrightarrow{BC}\}$.

This concept of region of influence and bound change rate of parameter is similar to the concept of hierarchy of the πSDF MoC. In πSDF, the parameters can not change their values during the execution of the graph in which they are used which is enforced by the hierarchical semantics and execution rules of the πSDF.

2.6 Conclusion

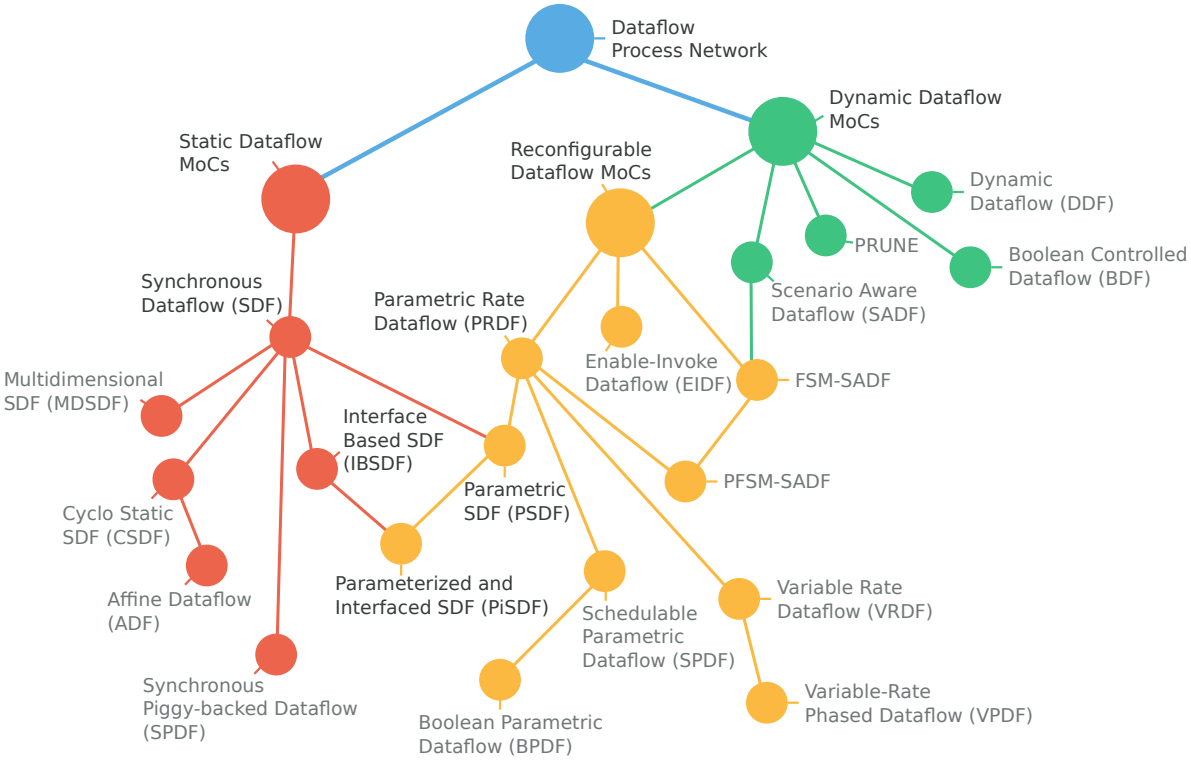


Figure 2.11 – An non exhaustive illustration of the dataflow MoCs landscape.

Dataflow Models of Computation (MoCs) are well suited for describing parallelism in data oriented applications. The high-level diagram based representation of dataflow MoC offer a natural representation of these applications. Moreover, dataflow MoCs have been extensively studied in the literature with numerous models available, each with specific semantics, advantages and restrictions. Figure 2.11 summarizes the existing landscape of dataflow MoCs in a non exhaustive manner. In Figure 2.11, the different MoCs are classified in three main categories, the *static*, *dynamic* and *reconfigurable* MoCs. We define

the difference between dynamic and reconfigurable MoCs as the possibility for a given MoC to being able to model a Turing machine or not. Additionally, Figure 2.11 shows the relationship between different MoCs, with dataflow MoCs inheriting from other MoCs being toward the bottom of the figure.

The πSDF MoC is the MoC that will serve as a basis to the work of this thesis and is only one of the many existing MoCs. This model combines properties of expressivity, compositionality and predictability in such a way that a wide range of applications may be represented and exploited efficiently. Reconfiguration using parameters allows good predictability of the application execution that immediately follows, permitting informed scheduling decisions.

In this thesis, we will focus on the extension of dataflow MoCs delay semantics by the mean of a new meta-model called State-Aware dataflow Meta-Model (SaMM). The capabilities of SaMM will be demonstrated with its application to the πSDF MoC. We will then study the dependencies modeling of the hierarchical semantics of the πSDF MoC to accelerate resource allocation algorithm used during the runtime management of πSDF applications.

CHAPTER 3

Embedded Runtimes for Multi-Processor System on Chips

3.1 Introduction

As previously defined, an embedded system is an integrated electronic and computing system designed to address a specific purpose. The Apollo Guidance Computer system used during the NASA Apollo missions is often considered as being the first modern embedded system. This system was embedded in the command of the lunar modules of the Apollo program. It was used for guidance, navigation and control of modules. This embedded computer was designed during the 1960's by Charles Stark Draper from the MIT Instrumentation Laboratory.

Embedded systems have largely evolve since then, and modern embedded systems are more complex than ever. Due to physical constraints, increasing the speed of General Purpose Processors (GPPs) inside computers is no longer sufficient to keep up with the ever growing demand for processing power. In the last two decades, embedded systems have changed to become more heterogeneous than ever, featuring multiple processor types on a same chip. These systems, called Multi-Processor System on Chips (MPSoCs) are highly complex to program due to the different types of PEs and memory architectures they embed. Hence, a lot of research has been done in order to provide tools to programmers that abstracts the complexity of these systems while trying to harness their computational power as efficiently as possible.

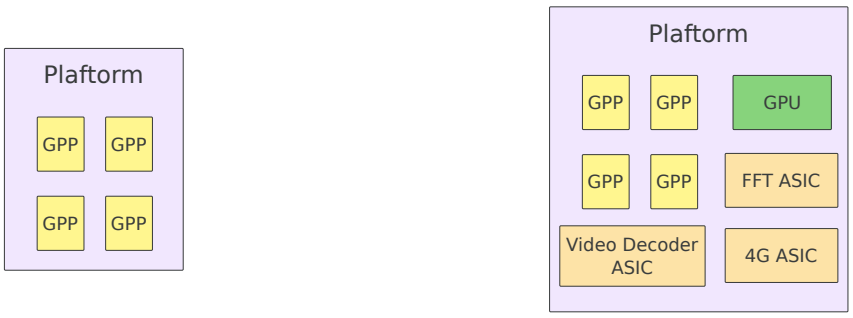
In this chapter, the overview of embedded systems will be described in Section 3.2. Multicore scheduling techniques will be subsequently presented in Section 3.3. Then, Section 3.4 presents a list of existing model-based runtimes.

3.2 Embedded Systems

3.2.1 Heterogeneity in Embedded Systems

One of the main challenges when programming heterogeneous embedded systems is to take into account the different types of PEs composing it, and the different types of memory architectures.

We define a PE type as being a compute element capable of doing one or multiple type of computation. There are two larger types of PEs: the *programmable PEs* and the dedicated co-processors. We reefer as programmable PEs as PEs having a dedicated instruction set, which list the available instructions for programming the PE (e.g. GPPs, Graphics Processing Units (GPUs), etc.). On the other side of the spectrum, dedicated co-processors are highly specialized PEs designed for a single task (e.g. FFT, video decoding, channel demodulation, etc.).



(a) Example of an homogeneous platform.

(b) Example of an heterogeneous platform.

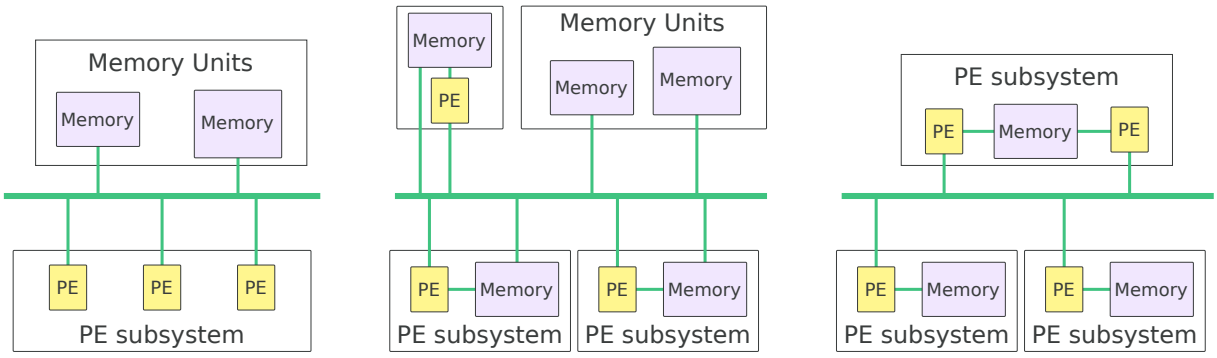
Figure 3.1 – Examples of homogeneous and heterogeneous platforms.

Embedded MPSoCs will often contains different PE types. For instance, the Texas Instrument Keystone II [**texas_instrument_keystone_2013**] is an heterogeneous platform that contains 4 ARM cores and 8 Digital Signal Processing (DSP) cores, along with dedicated FFT, network, and other co-processors. The NVIDIA Jetson TX-2 is a rela-

tively *less complex* heterogeneous architecture which features 4 ARM A57 cores, 2 Denver ARM cores and an NVIDIA Pascal GPU.

3.2.2 Memory Architectures in Embedded Systems

In addition to the PEs heterogeneity of embedded MPSoC platforms, the memory architecture of these platform is not unique, and there exists multiple type of memory architecture layout and hierarchy.



(a) Uniform Memory Access

(b) Non-Uniform Memory Access

(c) NO Remote Memory Access

Figure 3.2 – Examples of memory architectures.

Among the different type of memory architectures, shared memory systems are the simplest one. In a shared memory system, all PEs have access to all available memory of the system. This is the type of memory architecture used in most desktop computers for example. However, other schemes of memory architecture exist. In the literature, shared and distributed memory systems are commonly separated into three memory models, namely the *Uniform Memory Access (UMA)*, the *Non-Uniform Memory Access (NUMA)*, and the *NO Remote Memory Access (NORMA)* memory architectures as illustrated in Figure 3.2.

Figure 3.2a illustrates the shared memory system of a UMA machine with multiple memory units. Each unit can be accessed by every PEs. Both reliability and access speed are identical for each PE. UMA systems are often implemented using a memory bus (PEs share connection to each memory) or a crossbar network (each PE has its private connection to each memory). With increasing number of PEs, the UMA generally experiences bottlenecks (for the bus implementation) or requires complex memory subsystems (for

the crossbar implementation). UMA systems are thus not suitable for massively parallel platforms.

A NUMA machine provides multiple memory units where each unit can also be accessed by every PEs, as shown in Figure 3.2b. However, the access speed to each memory unit is dictated by the accessing PEs. This memory architecture is used to provide a dedicated access at a memory unit to a specific PE. This type of memory architecture does not necessarily scale well and as the number of PEs increases, this memory architecture restricts the memory subsystem growth.

To improve the scalability of systems with a high number of PEs, distributed memory systems are used. They are called NORMA machines. As is indicated in Figure 3.2c, PEs can only access local memory units. Communication for data exchange with other PEs is possible through dedicated hardware. Interconnections between cores can be implemented using a Network On Chip (NoC) which improves communication performance and scalability but also increases programming complexity. NORMA memory systems have very good scalability properties but their complexity brings new challenges in MPSoC programming. The MPPA architecture is a distributed NORMA systems with a total of 256 PEs and 16 memory units exchanging data over a NoC [**hascoe_distributed_2018**].

3.3 Multicore Scheduling

Multicore, or multi PEs, platforms are challenging on multiple levels, including programming them and performing mapping and scheduling of applications onto them. Scheduling applications onto multicore platforms is often decomposed into four distinct steps, namely the *extraction*, *mapping*, *ordering* and *timing* steps. Figure 3.3 illustrates these four steps. Note that the *mapping* and the *ordering* are often done at once.

- **Extraction:** The extraction step is the first step of the multicore scheduling process and consists of extracting the different parallel and sequential executable tasks of a given application. As seen in Section 2.2.2.1, in dataflow applications, parallelism may come from different sources such as parallel data paths, or auto concurrent executions, etc.
- **Mapping:** After all executable tasks have been extracted, the mapping step will assign a given PE to each of them. The assignment of a task to a PE is often done using heuristics such as the communication costs, or the energy efficiency of

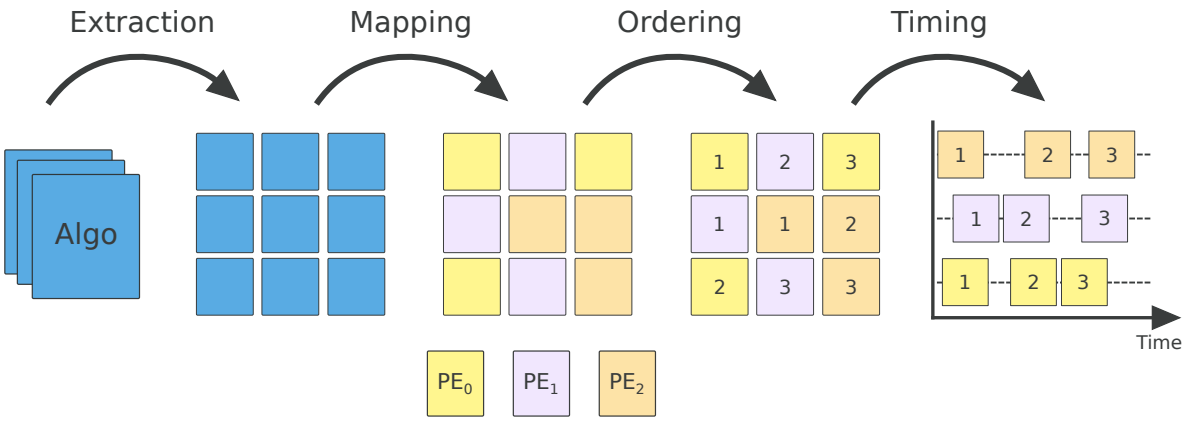


Figure 3.3 – Multicore scheduling flow decomposed for an application onto a 3 PEs (yellow, orange and purple) platforms. The extraction consists of extracting executable tasks from a application, then the mapping assigns a PE to each of the tasks which are then ordered and finally, the timing phase attributes a given start time to each of the tasks.

every PE capable of running the task for instance. The communication costs can be computed using the the predecessors and successors of the task being mapped.

- **Ordering:** Once all the tasks have been mapped, the ordering phase consists of assigning an execution order to the different tasks of the application based on their mutual dependencies. On dataflow applications, data dependencies will impose a strict ordering of tasks in a same data path.
- **Timing:** This last phase consists of assigning time point at which each task will start. This step is often left to the runtime manager of the application or the underlying host operating system. The value of each start point is based on multiple factors such as the current level of processing load of the platform, synchronization points such as data availability in the case of dataflow applications, etc.

Table 3.1 summarizes the different scheduling strategies obtained when the four steps are either done at compile time or at runtime, as defined in [**lee_scheduling_1989**]. A scheduling strategy is the result of a specific combination of compile-time and runtime steps.

For instance, the *fully dynamic* strategy consists of letting the runtime performs every steps on-the-fly, which will induce an significant overhead to the application execution time. On the other hand, a *fully static* strategy will consist of applying each of the steps at compile-time, thus resulting in lower overhead. The different scheduling strategies have their advantages and disadvantages. Indeed, deferring a given step to the runtime will add

Table 3.1 – Multicore scheduling strategies.

Strategy	Mapping	Ordering	Timing
Fully dynamic	runtime	runtime	runtime
Static assignment	compile-time	runtime	runtime
Self timed	compile-time	compile-time	runtime
Fully static	compile-time	compile-time	compile-time

to the complexity of the application execution and add some overhead; but it will also offer more flexibility that a compile-time strategy can not achieve.

Taking a telecommunication application such as the 5G network as example, depending on the number of connected devices, the processing load will be completely different and the topology of the application itself may change. Since the number of user is not constant and will vary during the runtime of the application, a dynamic strategy will be able to adapt to this varying number of users.

3.3.1 Scheduling approaches

In order to exploit efficiently the complexity of modern MPSoCs architectures, there exists multiple approaches to the scheduling of parallel applications. In [**park_multiprocessor_2009**], Parks et al. classify these approaches into 3 main categories, namely the *compiler based*, the *language extension* and the *model-based* approaches, illustrated in Figure 3.4.

- **Compiler based:** This approach is the most straightforward of the three approaches. Indeed, in this approach, the input is the sequential source code of the application written in a language such as C, C++, Fortran, etc. Then the compiler analyzes the source code in order to extract sensible information, such as parallelism using data dependencies analysis, control flow analysis, etc. Polyhedral analysis is one of the possible transformation that can be applied to sequential source code in order to unveil parallelism opportunities of the application. From the extracted information, the compiler will then generate multi threaded code in order to exploit as much as possible the available parallelism.
- **Language extension:** Similarly to the compiler based approach, this approach takes as input a sequential source code of the application. However,

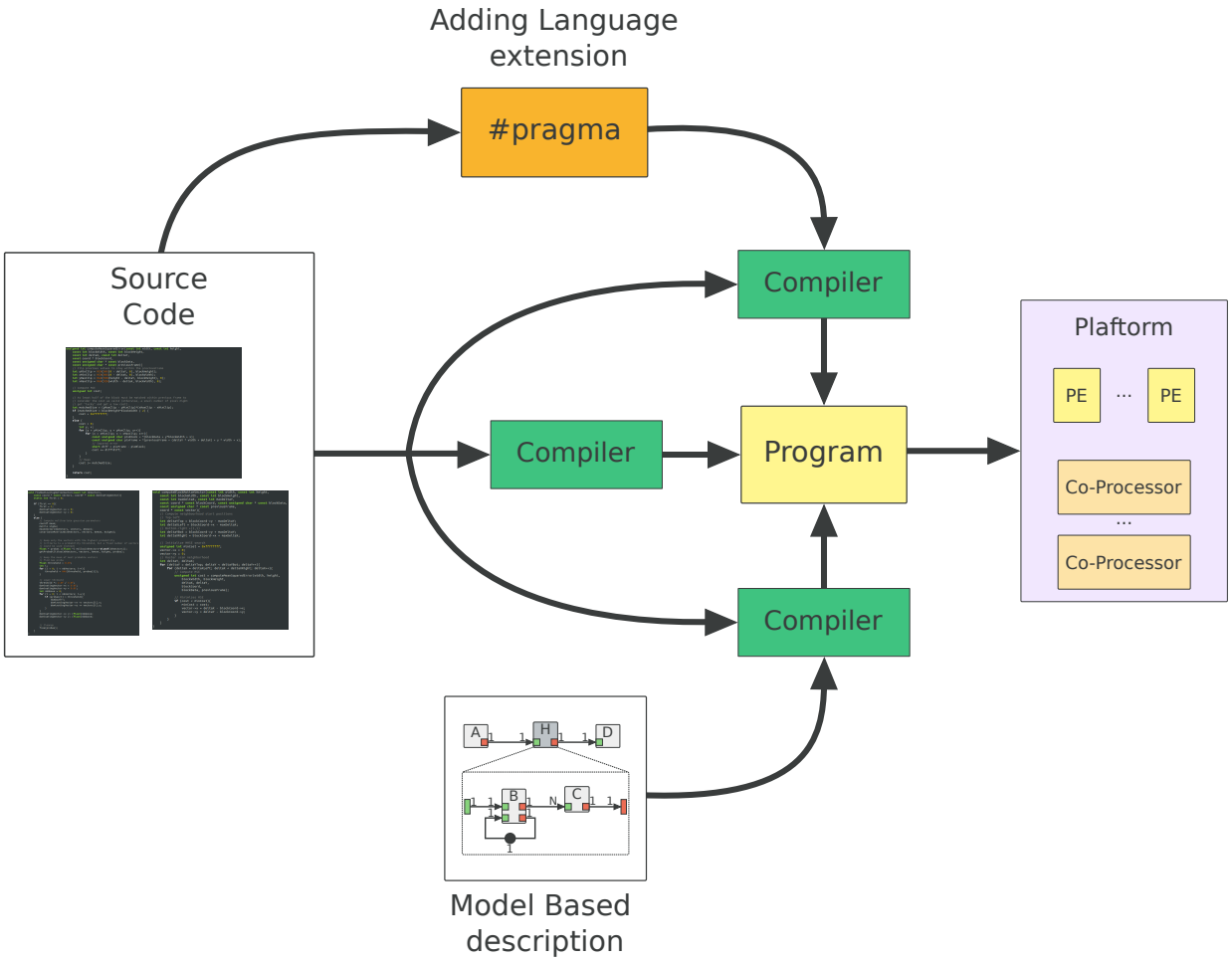


Figure 3.4 – Illustration of the three main approaches to parallel scheduling of applications onto MPSoCs

to ease the extraction of parallelism information, user provided indications are added. These indications are used to highlight parallel regions of the code. The OpenMP [**chandra_parallel_2007**] Application Programming Interface (API) is a well known example of this approach. OpenMP supports shared memory architecture and generally targets homogeneous platforms. Based on C, C++ or Fortran code, this API uses compiler directives such as pragmas to define parallel regions and to provide a multicore communication API. The software developer must manually identify and specify parallel regions in his code. OpenMP is now widely supported on embedded MPSoC platforms, such as TI's keystone architecture or the MPPA platform [**tagliavini_unleashing_2018**]. Other tools or APIs based on language extensions are available, such as Open-

MPI [**graham_open_2006**], OpenCL [**stone_opencl_2010**] or the dedicated NVIDIA CUDA API [**nvidia_nvidia_nodate**].

- **Model-based:** The model-based approach uses a high level model-based description of the application conjointly to the source code of the application. This approach has gained popularity in the recent years due to its high level of abstraction from both the underlying low level implementation of the application and the complexity of the MPSoC architecture to which the application needs to be mapped. In a model-based approach, the application is described using one or multiple MoCs. Dataflow based MoCs are a popular choice when design signal processing applications to heterogeneous architecture. In a dataflow MoC, the extraction of parallelism information becomes less complex compared to the compiler based approach due the natural representation of task and data parallelism in dataflow graphs. Some dataflow models also enforce, by construction, the deadlock freeness of the application. Dataflow models have become widely adopted now for hardware and software co-design. For instance, the CAPH language [**serot_implementing_2011**] is a dataflow based programming language that, given a dataflow based specification, produces VHDL code for hardware synthesis. There exists many tools based on dataflow MoCs such as the Ptomely tool [**eker_taming_2003**], PREESM [**pelcat_preesm:_2014**], Orcc [**yviquel_orcc_2013**], or StreamIt [**goos_streamit_2002**].

The next section presents existing dataflow or task based runtime managers as one of the contribution of this thesis is the creation of a new dataflow based runtime manager for heterogeneous architectures.

3.4 Model-Based Embedded Runtimes

Modern dataflow applications such as state-of-the-art video encoding, deep learning applications, or 5G telecommunication network require high computational power. Recently, these applications have been deployed onto heterogeneous platforms embedding GPPs, GPUs and dedicated co-processors to improve their performance and energy efficiency [**abadi_et_al._tensorflow:_2016**; **dardaillon_new_2016**; **pelcat_preesm:_2014**].

Historically, the main approach for the scheduling and the mapping of dataflow applications onto heterogeneous platforms have been to use static methods [**sriram_embedded_2009**].

Since static methods are computed at compile time, these methods can leverage high computational power to search for solutions in a larger design space that dynamic, i.e. done at runtime, methods. Moreover, static methods are often associated with automatic code generation or provide sensible information that a programmer will need to apply to its application, hence they offer very low overhead on the applications runtime.

However, the main advantage of static methods is also their biggest downside. Since all the analysis is performed at compile-time, they lack the flexibility required by modern applications. While pre-computed mapping and scheduling offer a great predictability on the behavior of an application, it is not possible to adapt to dynamic changes at runtime such as changes in the application topology, or in the available processing resources (e.g. the underlying operating system allocating resources to other application).

Applications runtime managers are an alternative to static approaches that add flexibility by taking into account the runtime constraints of applications. In this thesis, we introduce SPIDER 2.0 a novel dataflow based runtime manager, that is based upon the SPIDER [**heulot_runtime_2015**] runtime, for image and signal processing applications on heterogeneous platform. In this section we present different dataflow based runtime managers, in order to situate SPIDER 2.0 with respect to existing solutions.

Table 3.2 summarizes the different features of the presented runtimes. The deadlock prevention is based on the definition given in [**dauphin_odyn_2019**] and consider the capability of a runtime to ensure that no deadlock will occur during the runtime. PRUNE, HTGS Model-Based Engine (HMBE) and the proposed SPIDER runtimes fail to propose any deadlock prevention mechanism. However each of these runtimes provide a way for detecting if a deadlock occur but in a reactive and not a pro-active manner. As shown by Table 3.2, SPIDER supports heterogeneous architecture, with NUMA, and with a fully dynamic mapping and scheduling strategy.

- **Odyn:** Odyn [**dauphin_odyn_2019**] is a dataflow model-based runtime that offers an hybrid approach with both static analysis at compile time and runtime analysis, and supports heterogeneous platforms composed of GPPs, GPUs and co-processors and with NUMA.
  Odyn uses an input graph modeled with the SDF MoC annotated with real-time constraints for each actors such as deadlines, Worst Case Execution Times (WCETs), and mapping information containing the actors to PEs assignment, the allocation of buffers to PEs memory, and the size of each memory units of the architecture . In Odyn, the dynamic scheduling and memory management decisions are

Table 3.2 – A comparison of existing dataflow-based and tasks-based runtimes.

Feature	Odyn	StarPU	XKaapi	PRUNE	HMBE	SPIDER
Heterogeneity	CPU/DSP/ FPGA	CPU/GPU	CPU/GPU	CPU/GPU	no (CPU)	CPU/DSP
Dynamic mapping	No	Yes	Yes	Yes	No	Yes
Dynamic scheduling	Yes	Yes	Yes	Yes	Yes	Yes
Deadlock prevention	Yes	Yes	No	No	No	No
NUMA support	Yes	Yes	Yes	undef.	Yes	Yes
Reconfigurable Applications	No	Yes	Yes	Yes	No	Yes
Targeted System	real-time	HPC	HPC	General	real-time	real-time

taken at runtime, independently for each PEs. The input SDF graph is transformed into an Acyclic Homogeneous SDF (AHSDF) graph which is then partitioned, and each PE of the platform manages its own partition.
Odyn uses a variant of the Memory Exclusion Graphs (MEGs) introduced in [**desnos_memory_2012**] to perform memory deadlock analysis, which is done at compile time due to its high computational cost.

- **StarPU:** StarPU [**augonnet_starpu:_2009**] is a runtime that supports heterogeneous platforms composed of multi-Central Processing Units (CPUs) and multi GPUs. StarPU dynamically schedules and maps Directed Acyclic Graph (DAG) based application and is optimized for High-Performance Computing (HPC). The scheduler of StarPU uses a variant of the Heterogeneous Earliest-Finish-Time (HEFT) [**goos_scope_1995**] algorithm based on cost models for data transfer and task execution time [**hutchison_automatic_2010**].
- **XKaapi:** Similarly to StarPU, XKaapi [**gautier_xkaapi:_2013**] is task based dataflow runtime oriented toward HPC. XKaapi is an evolution of the original Kaapi runtime [**gautier_kaapi:_2007**] and supports heterogeneous platforms composed of multi-CPUs and multi GPUs. The main difference between StarPU and XKaapi lies in the scheduling policy and the communication with the GPUs. In StarPU, a synchronization is enforced when tasks are sent to the GPUs to ensure that all kernels are properly finished, whereas XKaapi uses an asynchronous mechanism. Moreover, the HEFT scheduling policy of StarPU does not react well with variations in the system load or in the tasks execution times. XKaapi, on the other hand, uses a dynamic job stealing scheduling policy which

was showed to be as efficient as the StarPU scheduler in its default configuration [**lima_exploiting_2012**].

- **PRUNE:** PRUNE [**boutellier_prune_2018**] is an hybrid runtime using both static analysis and runtime analysis. PRUNE uses a dynamic extension of the SDF MoC and applies deadlock freedom and bounded memory usage analysis at compile time. At runtime the user is free to use a static or a dynamic mapping. Finally, each actor is assigned to a different thread and the scheduling is left to the host operating system (e.g. Linux or Windows). PRUNE supports heterogeneous architectures composed of CPUs and GPUs.
- **HMBE:** HMBE [**wu_model-based_2017**] is dataflow model-based runtime for heterogeneous platforms composed of CPUs and GPU. HMBE is a model-based abstraction of the Hybrid Task Graph Scheduler (HTGS) [**blattner_hybrid_2017**] API. Applications are modeled using the Windowed Synchronous DataFlow (WSDF) MoC [**keinert_windowed_2005**] and statically transformed into an Acyclic Precedence Expansion Graph (APEG). During runtime, each actor in the APEG of the input graph is assigned to a thread, meaning that there are as many threads as there are actors in the APEG. To avoid contention due to the high number of threads in regard to the number of available CPU cores, the HMBE scheduler uses an heuristic to put threads in *dormant* state and uses a ready list type of scheduling policy to choose among the pool of ready tasks which one to wake up. To the best of our knowledge, although HTGS supported CPUs + GPUs heterogeneous architecture, it does not seem to be the case of HMBE.
- **Spider:** SPIDER [**heulot_runtime_2015**] is a dataflow model-based runtime that supports heterogeneous platforms. SPIDER relies on a πSDF modeling of an application associated with runtime information composed of WCETs for each actors and for each PEs of the platform, and actors to PEs mapping constraints. The support of heterogeneous platforms in SPIDER is done by re-implementing a low level API corresponding to the communication primitives used in the platform. For instance, on shared memory architectures, SPIDER uses semaphores to synchronize multiple threads, whereas on the Keystone II platform, SPIDER employs the dedicated communication queues of the platform to perform low level synchronization.

3.5 Conclusion

Embedded systems are heterogeneous systems, featuring multiple processor types on a same chip, called Multi-Processor System on Chips (MPSoCs). Due to the complexity of programming Multi-Processor System on Chips (MPSoCs), multiple approaches have been investigated. First, compiler based approaches try to automatically extract sensible information from classical sequential code (e.g. C, C++ or fortran code) in order to parallelize tasks as much as possible on modern platforms. These approaches require really complex algorithms as the information needs to be extracted from code not meant for parallel MPSoCs but offer more productivity to programmers as they do not need to change anything in their code.

A secondary approach, which is complementary to the first one, is based on language extensions and APIs such as OpenMP [**chandra_parallel_2007**], or OpenCL [**stone_opencl_2010**]. In this approach, the programmer needs to investigate which portion of the code is the bottleneck and use the API to indicate the compiler which area of the code need to be parallelized and how.

Finally, the last approach seen in this chapter is the model-based approach. In a model-based approach, the programmer needs to describe his application using one or multiple MoCs. For instance, for signal processing applications, using a dataflow MoC to model the application will ease the job of the compiler due to the natural definition of parallelism in dataflow MoCs.

In order to improve the efficiency and adaptivity to external events, numerous runtime managers are proposed in the literature. In this chapter, we presented a list of these runtimes that are similar to the one we introduce in Chapter 6.

One of the major contribution of this thesis is the creation of a dataflow based runtime called Spider 2.0. Spider 2.0 is designed to support heterogeneous architectures composed of GPPs, GPUs and hardware accelerators, including NUMA support. Spider 2.0 takes as input a πSDF modeling of an application and a logical hardware model of the platform and performs the mapping, scheduling and memory management of the application onto the physical platform. Spider 2.0 is introduced in Chapter 6.

Part II

Contributions

CHAPTER 4

Dynamically Initialized and State-Aware Dataflow MoCs

4.1 Introduction

In this chapter, a new meta-model called State-Aware dataflow Meta-Model (SaMM) is proposed. SaMM extends the semantics of a targeted dataflow MoC with explicit and dataflow based semantics for the initialization of delays. Additionally, SaMM provides the extended dataflow MoC with unambiguous and controllable persistence scope for the delayed data tokens. SaMM is applicable to any dataflow MoC with a well-defined concept of graph iteration. For example, MoCs deriving from the Synchronous DataFlow (SDF) MoC such as the Parameterized Synchronous DataFlow (PSDF), Parameterized and Interfaced Synchronous DataFlow (πSDF), Cyclo-Static Dataflow (CSDF) or the WSDF naturally satisfy this condition as it is possible to derive a periodic schedule for all of them, which is a natural expression of the concept of iteration.

Given a delay between two actors in a dataflow graph, SaMM extends the semantics of a targeted dataflow MoC by allowing direct data connections to and from the delay with other actors from the graph. These additional connections give the possibility to dynamically *initialize* the data tokens of the delay at the beginning of a graph iteration and to *retrieve* the data tokens from the delay at the end of a graph iteration. Providing such a mechanism to delays results in concise and expressive representation of complex applications containing nested loops. In addition, SaMM comes with a well-defined notion

of local and global state scope. In SaMM, deep hierarchy graphs may have a local persistent state at low-levels hierarchy that do not impact the data parallelism of upper level actors. Thus, SaMM provides programmers with flexibility in the design of complex applications by enforcing the compositionality of hierarchical graphs. In this chapter, the capabilities of the State-Aware dataflow Meta-Model will be demonstrated by applying it to the πSDF MoC. The choice of the πSDF MoC for the application of SaMM is mainly motivated by the fact that this is the MoC used by the targeted tools of this thesis which are the PREESM [1] and SPIDER [2] tools. In the following, the notation State-Aware Dataflow (*SAD*) graph refers to any kind of dataflow graph using the semantics of SaMM. This notation is used in order to maintain good readability. The work presented in this chapter has been accepted in the 2018 SAMOS international conference and published in the ACM International Conference Proceeding Series (ICPS).

This chapter is organized as follows, the novel initialization semantics of SaMM is introduced in Section 4.3 along with the intermediate transformations that are necessary for deriving properties such as consistency and liveness of *SAD* graphs. Section 4.4 presents the persistence concept and the associated semantics of SaMM with an application of the meta-model to the πSDF MoC, resulting in the State-Aware Parameterized and Interfaced Synchronous DataFlow (SA-πSDF) MoC. Finally, Section 4.5 presents an example of a reinforcement learning application modeled with SaMM.

4.2 Motivation: The Limitations of Delays in Dataflow MoCs

In the SDF MoC [**lee_synchronous_1987**], delays are defined as being the initial state of a given First-In First-Out Queue (FIFO). The initial state of a FIFO corresponds to the initial number of data tokens present in the FIFO at the beginning of the graph iteration. Importantly, tokens present in FIFOs at the end of a graph iteration are used for the initial state of the next graph iteration. Such mechanism allows creating cyclic data path and temporal pipelining, which would be impossible otherwise. All the other dataflow MoCs that derive from the SDF MoC have the same definition of delays. Figure 4.1 illustrate the 2 main usages of delays in the SDF MoC: pipelining actor firings, and ensuring liveness of cyclic data-paths.

1. https://github.com/preesm/preesm
2. https://github.com/preesm/spider

(a) Example of a pipeline in an SDF graph.

(b) Example of an SDF graph with a cyclic path.

Figure 4.1 – Examples of delay usage in dataflow graphs. Delays are represented graphically by a filled circle on a FIFO.

4.2.1 Initialization of Delays

In the SDF MoC, having a *pipelining* delay on a FIFO f means that an offset exists between the iteration in which data tokens are produced on f, and the iteration in which these data tokens are consumed. In Figure 4.1a, the token produced by actor A at iteration n is consumed by actor B at iteration $n + 1$. Consequently, the delay of Figure 4.1a also removes the direct data dependency between actor A and actor B, allowing the execution of actor B to occur in parallel of the execution of the actor A using delayed data tokens.

Initial tokens inside FIFOs also prevent deadlocks in dataflow graphs that contain cyclic data paths. For example, during an iteration of the graph of Figure 4.1b, actor B fires only when actor A produces 1 data token, and actor A needs a data token from actor B to fire. Hence, a sufficient number of delays are needed in this cyclic data path to initiate the first actor firing, and prevent a deadlock.

Although the concept of delays exists in most dataflow MoCs, two main aspects of their properties are often left unspecified: the initial value property and the persistence property. The initial values given to the corresponding data tokens of a delay is hardly mentioned, let alone specified, in the literature. When specified, initial values are usually set to 0 [**lee_synchronous_1987**; **sriram_embedded_2009**]. In [**davis_data_1982**], the initial value of the initial state seems to be defined by the user prior the execution of the first graph iteration. The lack of specification on the initial values of delays leads to inconsistent behaviors across different programming tools. With the explicit new semantic of delay proposed in SaMM, the initialization of delays is explicit and can no longer change from one implementation to the other.

4.2.2 Persistence of Delays

The persistence of data tokens across multiple graph iterations is an important property that means that data tokens produced at a given graph iteration n are not necessarily consumed during iteration n but instead are propagated to further graph iterations. This property is particularly useful in streaming signal processing application or signal filtering where delayed samples are often needed. In dataflow MoCs, the persistence property of data tokens is conveyed using delays. The persistence property of the data tokens of delays raises multiple questions, especially regarding the notion of global state of a dataflow graph. In a hierarchical MoC, the persistence of delays also influence the internal and local state of subgraphs. This notion of persistence is nearly never mentioned in any dataflow MoC. In a flat model, that is without hierarchy, like the SDF MoC [**lee_synchronous_1987**], the last data tokens produced during an iteration n are used for the initial state of iteration $n + 1$. This definition induce a global state to an SDF graph saved inside the delays of the FIFOs of the graph. This persistence across multiple iterations is what allows pattern such as time pipelining as illustrated in Figure 4.1a. When clustering an SDF graph, this notion of delays has to be taken into account as shown in [**pino_hierarchical_1995**]. Indeed, when clustering graph containing delays information such as precedence relationship between actor firings may be lost or cyclic data path may be broken.

However, in hierarchical MoCs, the internal behavior of actors can be defined by dataflow graphs. Thus, what happens to the persistence of the delays contained in a hierarchical actor? Additionally, what does it mean for the internal state of this actor? In the Interfaced Based Synchronous Dataflow (IBSDF) MoC [**piat_interface-based_2009**], and in the πSDF MoC [**desnos_pimm:_2013**], levels of hierarchy are clearly delimited with the use of data input interfaces and data output interfaces. Therefore, in the IBSDF MoC, a hierarchical actor is considered in the same manner as an atomic actor. This means that if a hierarchical actor is fired several times per iteration of its parent graph with no feedback loop, the data parallelism property makes it possible for these firings to occur in parallel. In this context, if the subgraph of a hierarchical actor contains a delay, the corresponding data tokens may not persist across firings of this hierarchical actor, as it would force the sequential execution of the hierarchical actor and, by extension, of the parent graph. In other words, the delay is only persistent within the scope of the subgraph. Going further, this means that in the IBSDF MoC, a hierarchical actor is

not permitted to have an internal local state that will persist for multiple iteration of the actor.

In the Dataflow Process Network (DPN) MoC [**lee_dataflow_1995**], *Lee et al.* state that delays in hierarchical actors may result in non-deterministic behaviors even with a consistent and live subgraph. *Lee et al.* propose to make delays persistent across all levels of hierarchy with implicit feedback loops around a hierarchical actor in order to maintain the precedence relationship between multiple successive firings of the actor. However, serializing the execution of an actor induces a loss of data parallelism. Losing data parallelism significantly impacts performance in graphs with deep nested hierarchy levels as it forces the serialization of the execution of the entire hierarchy tree. Additionally, persistent delays in hierarchical graphs makes the DPN MoC a non compositional MoC as the behavior of hierarchical actors directly impact the analysis of the graph they belong.

In recent dataflow-based domain-specific programming language, the semantics of delays is also problematic. In OpenVX [**kronos_group_openvx_2013**], a computer vision dataflow-based programming language, delays object are considered persistent, thus limiting hierarchical graph composition due to possible hidden internal states. In TensorFlow [**abadi_et_al._tensorflow:_2016**], there is no explicit notion of delay. Tensors, the basic data type in TensorFlow, are considered to be globally persistent during the lifetime of the application and are stored in the global shared state of the application.

In the rest of this chapter, we introduce the State-Aware dataflow Meta-Model (SaMM). SaMM can be used similarly to the Parameterized and Interfaced Meta-Model (PiMM) [**desnos_pimm:_2013**] or the *Parameterized DataFlow* [**bhattacharya_parameterized**] to extend the semantics of any dataflow MoC implementing a well-defined notion of graph iteration. SaMM adds both explicit initialization of delays and hierarchical state awareness through the use of a explicit and tunable persistence scope of delays to the extended MoC. The next section presents the new initialization semantics of delays of SaMM and demonstrates its efficiency to model simple algorithm structures with a for-like structure example.

4.3 Dynamic Initialization of Dataflow Graphs

4.3.1 Extending Delay Initialization Semantic

The SaMM semantics extends the definition of delays of Section **??** and is applicable to any dataflow MoC with a well-defined concept of graph iteration. In the proposed semantics, a delay $d = (f, n, c_{in}, c_{out})$ contains in addition to a Fifo f and a number of initial tokens n, two optional data connections c_{in} and c_{out}. The input data connection c_{in} of the delay associates a *Setter* actor responsible for initializing the n data tokens of the delay. The output data connection c_{out} of the delay associates a *Getter* actor receiving the last n data tokens held by the delay. The dataflow rates of c_{in} and c_{out} are such as $rate(c_{in}) = rate(c_{out}) = n$. However, the production rate of the *Setter* actor and the consumption rate of the *Getter* actor are not required to be equal to n.

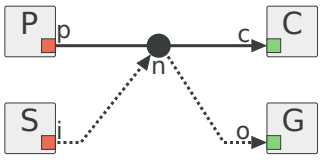


Figure 4.2 – Proposed initialization semantics of delay.

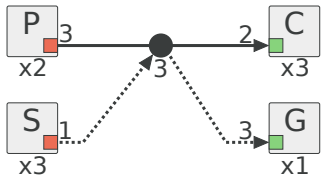
Delays are usually represented by a filled circle positioned on a Fifo as displayed in Figure 4.1. Figure 4.2 introduces a new graphical representation of delays with the additional data connections. Actors P and C are the production and the consumption actors of the Fifo f of the delay, respectively; and actors S and G are the setter and getter actors of the delay, respectively. The Fifo between the setter actor S and the delay, and the delay and the getter actor G, are drawn with a dashed line to avoid any confusion with the Fifo to which the delay is attached.

The new data connections induce the following precedence rules in the firing sequence of actors during each graph iteration.

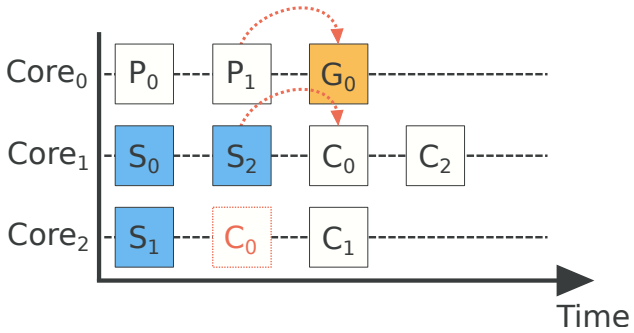
R1. All firings of the *Setter* actor of a delay must occur prior the first firing of the *Consumption* actor of this delay.

R2. All firings of the *Getter* actor of a delay must occur after the last firing of the *Production* actor of this delay.

Figures 4.3a and 4.3b illustrate the firing rules **R1** and **R2**. In the graph of Figure 4.3a, actors S and G are the setter and getter actors of the delay and actors P and C are the



(a) Graph example of the firing rules of SaMM



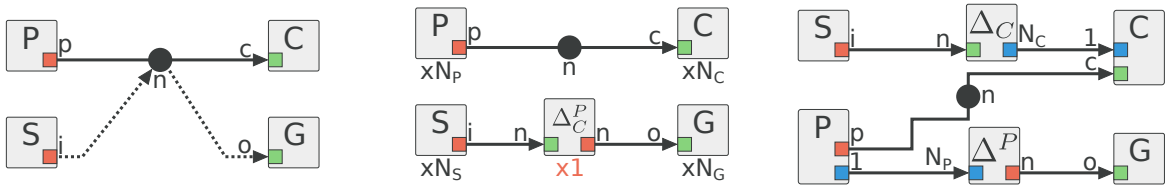
(b) Possible schedule of the graph of Figure 4.3a.

Figure 4.3 – Illustration of the firing rules of SaMM.

production and consumption actors of the delay, respectively. The repetition value of the different actors are noted below them. Figure 4.3b gives a possible schedule of the graph of Figure 4.3a. The red dashed arrows illustrate the implication of the rules **R1** and **R2**. Indeed, the first firing of the actor C is only possible after the last firing of actor S due to the rule **R1** and, similarly, due to rule **R2**, the firing of actor G only occurs after the last firing of actor P. The red dashed box C_0 of Figure 4.3b illustrate a potential placement of the firing 0 of actor C satisfying its consumption requirement of its input data port but not satisfying the rule **R1**. The aforementioned rules enforce the expected initialization behavior of delays and ensure that no data tokens a delay is consumed prior the end of its initialization and prior the end of the pipeline, or cycle, to which the delay is attached.

On the initialization side, **the data tokens of a delay must be explicitly initialized** for the delay to be fully specified. The default initialization of the proposed semantics is to set all data tokens of a delay to zero. Explicitly initializing the delays means that new initialization tokens are produced on each graph iteration. Thus, if no actor is connected to the c_{out} data connection of a delay, the produced data tokens have to be discarded to ensure bounded memory execution. Importantly, making the initialization of delays explicit for each graph iteration unambiguously removes memory persistence across graph iterations. Indeed, each graph iteration starts with initial data tokens independent from previous computations. Therefore, dataflow initialized delays are no longer allowed to transfer data tokens from iteration n to the iteration $n+1$. Section 4.4 introduces new unambiguous semantics for modeling this persistence of data tokens across graph iterations. The next section details how the delay initialization semantics of SaMM impacts the consistency, liveness, and scheduling analyses of a dataflow graph.

4.3.2 Consistency and Liveness Analysis



(a) *SAD* graph example.

(b) Consistency Equivalent Graph of Figure 4.4a.

(c) Consistency and Liveness Equivalent Graph of Figure 4.4a.

Figure 4.4 – *SAD* graph example and equivalent graphs used for consistency and liveness analyses.

A key feature of SaMM is its compatibility with state-of-the-art methods for analyzing the consistency and liveness of a graph. The proposed method for verifying the consistency and liveness of a graph consists of 4 steps.

Step 1. The analyzed dataflow graph is transformed into a Consistency Equivalent Graph (CEG). The CEG is an intermediate representation used for the consistency analysis of State-Aware Dataflow (*SAD*) graphs. To build a CEG, every delay with a *Setter* actor or a *Getter* actor is replaced with a delay with no c_{in} nor c_{out} connection. The setter and getter actors of every delay are now connected to *virtual delay* actors. The virtual delay actor is noted Δ_C^P, with P and C being the names of the *Production* and *Consumption* actors connected to the delay, respectively. The virtual delay actor has a unique input data port and a unique output port. The rates of the data ports of the virtual delay actor are equal to the value n of the delay it replaces. Figure 4.4b illustrates the CEG transformation of the graph of Figure 4.4a. Actors S and G are now connected to the virtual delay actor Δ_C^P instead of the delay. The rates of the input and output data ports of Δ_C^P are equal to the value n of the delay.

Step 2. The consistency of a CEG is verified by analyzing the topology matrix Γ of the CEG using the same method as for SDF graphs [**lee_synchronous_1987**]. The transformation into the CEG may result in disconnected graphs as illustrated by Figure 4.4b. Thus, the consistency of every graphs in the CEG has to be verified for the CEG to be consistent. A necessary but not sufficient condition for the liveness of a *SAD* graph is that every *virtual delay* actor must have a Repetition Vector (RV) value of 1, using the RV of the CEG. In Figure 4.4b, the RV values are noted below each actor of the CEG.

Step 3. The CEG is transformed into a Consistency and Liveness Equivalent Graph (CLEG) using the RV values computed during step 2 to verify the liveness of the original graph. The CLEG enforces and models the precedence rules **R1** and **R2** of Section 4.3.1. The CLEG transformation splits virtual delay actors in two virtual actors and adds *virtual data ports* and FIFOs to every production and consumption actors connected to delays.

The virtual actors are illustrated in Figure 4.4c, which shows the CLEG of the graph of Figure 4.4a. Virtual actors Δ_C and Δ_P replace the actor Δ_C^P of Figure 4.4b and enforce the rules **R1** and **R2**, respectively. N_P and N_C are the RV values of actors P and C in the CEG of Figure 4.4b and n is the number of delays. Actor Δ_C has a consumption rate of n on its input port and a production rate of N_C on its output port. Symmetrically, actor Δ^P has a consumption rate of N_P on its input port and a production rate of n on its output port. The virtual data ports of actors P and C, represented in blue, have a production and consumption rate equal to 1, respectively. The CLEG in Figure 4.4c exposes both the precedence relationships and the explicit data productions and consumptions of Figure 4.4a.

Step 4. The liveness of the CLEG is verified using methods of the state-of-the-art such as the Symbolic Execution method [**lee_synchronous_1987**] or the mathematical analysis in [**marchetti_sufficient_2009**].

The scheduling of a graph using the proposed delays is compatible with the scheduling techniques used by current dataflow MoCs. Indeed, the CLEG gives all the dependencies between firings of actors and can be used to derive a schedule for the original graph. Note that the *virtual ports*, *actors*, and *FIFOs* are used for analysis and scheduling purposes only. The virtual FIFOs do not convey actual data tokens, and the *virtual actors* have a null execution time. Another important note is that the CLEG can not be used directly to allocate memory as the virtual FIFO do not convey the notion of shared FIFO between the *Setter* actor, the *Production* actor, the *Consumption* actor and the *Getter* actors. For memory allocation, the original *SAD* graph is used. With the semantics of SaMM, FIFOs connecting *setter* and *getter* actors to delay are directly identified and can be physically allocated to the same FIFO f of the *production* and consumption *actors* of the delay.

An example of the analysis workflow of a *SAD* graph is given in Figure 4.5. The input *SAD* graph is used to derive both the CEG and the CLEG. The CEG and the CLEG are then used directly for consistency and liveness analysis, respectively. The CLEG is also used directly for the mapping and scheduling task. Then, the memory allocation task uses both the output information of the mapping and scheduling task and the original

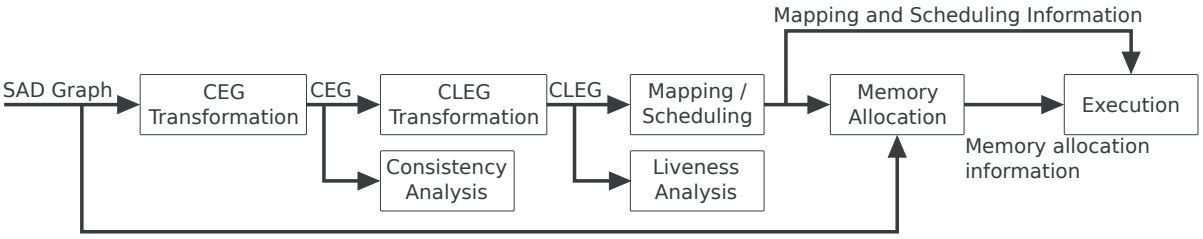
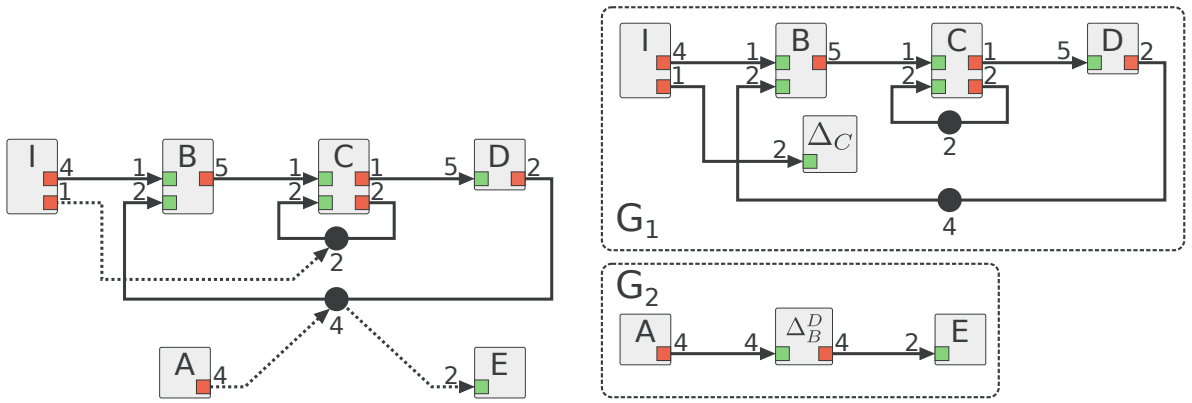


Figure 4.5 – Example of an analysis workflow of a *SAD* graph.

input *SAD* graph to allocate the different FIFOs of the graph. Finally, the graph can be executed using both the mapping and scheduling information along with the memory allocation information.

The next section presents a synthetic graph example with nested initialization of delays for using the newly introduced semantics. Then Section 4.3.3 presents some forbidden graph construction patterns in SaMM.

4.3.2.1 Synthetic Example



(a) An SDF graph with the proposed semantics of delays.

(b) CEG of the graph of Figure 4.6a.

Figure 4.6 – CEG transformation of a synthetic *SAD* graph.

Figure 4.6a presents an example of a complex synthetic SDF graph with the newly introduced semantics for delays. The graph of Figure 4.6a is used as an illustration of the analyses steps presented in Section 4.3.2. In this graph, the delay on FIFO *(D,B)* is used to avoid a deadlock and the delay on the self-loop of actor C is used to specify explicitly the transmission of a state between successive firings of actor C. The two delays of Figure 4.6a

are initialized by actors A, for the delay on FIFO *(D,B)*, and by I for the delay on the self-loop of actor C. Finally, after the last iteration of the cycle composed of actors B, C and D, the two firings of actor E use the data tokens held by the delay of FIFO *(D,B)*. Figure 4.6b gives the CEG transformation of the original graph of Figure 4.6a. Actors A and E are now connected to the virtual delay actor Δ_B^D and actor I is connected to the virtual actor Δ_C. The original graph of Figure 4.6a is now split into two unconnected graphs in the CEG, namely G_1 and G_2. Thus, checking the consistency of the original graph is equivalent to checking the consistency of both graphs in the CEG. Equations 4.1 and 4.2 gives the topology matrices Γ_1 and Γ_2 of the graphs G_1 and G_2, respectively.

$$\Gamma_1 = \begin{array}{c} \\ I\Delta_C \\ IB \\ BC \\ CD \\ DB \end{array} \begin{array}{c} \begin{array}{ccccc} I & \Delta_C & B & C & D \end{array} \\ \begin{bmatrix} 1 & -2 & 0 & 0 & 0 \\ 4 & 0 & -1 & 0 & 0 \\ 0 & 0 & 5 & -1 & 0 \\ 0 & 0 & 0 & 1 & -5 \\ 0 & 0 & -2 & 0 & 2 \end{bmatrix} \end{array}, q_1 = \begin{array}{c} \\ I \\ \Delta_C \\ B \\ C \\ D \end{array} \begin{bmatrix} 2 \\ \mathbf{1} \\ 8 \\ 40 \\ 8 \end{bmatrix} \tag{4.1}$$

$$\Gamma_2 = \begin{array}{c} \\ A\Delta_B^D \\ \Delta_B^D E \end{array} \begin{array}{c} \begin{array}{ccc} A & \Delta_B^D & E \end{array} \\ \begin{bmatrix} 4 & -4 & 0 \\ 0 & 4 & -2 \end{bmatrix} \end{array}, q_2 = \begin{array}{c} A \\ \Delta_B^D \\ E \end{array} \begin{bmatrix} 1 \\ \mathbf{1} \\ 2 \end{bmatrix} \tag{4.2}$$

On top of Γ_1 and Γ_2 are noted the names of the actors to which the columns of the matrices refer. On the left of Γ_1 and Γ_2 are noted the names of the FIFOs to which the lines of the matrices refer. $rank(\Gamma_1) = 4$ and $rank(\Gamma_2) = 2$, thus the graphs G_1 and G_2 are consistent and so is the graph of Figure 4.6a. The RVs q_1 and q_2 of respectively G_1 and G_2 give a repetition factor of 1 for both Δ_C and Δ_B^D actors. Figure 4.7 shows the CLEG of the graph of Figure 4.6a. Actor Δ_B is now connected to actor B through a virtual port with a production rate of $q_1(B) = 8$ and a consumption rate of 1 as specified by Section 4.3.2. Similarly actor Δ_C is connected to actor C with a production rate of $q_1(C) = 40$. Finally, actor Δ^D is connected to actor E and actor E has a consumption rate of 8 on this FIFO which is equal to $q_1(D)$. The CLEG of Figure 4.7 is live and thus the graph of Figure 4.6a is both consistent and live.

4.3.3 Corner Cases: Recursive and Multiple Initialization

Figures 4.8a and 4.8b illustrate two potential scenarios raising from the possibility of dynamically initialized delays. Figures 4.8a shows a recursive initialization pattern for the

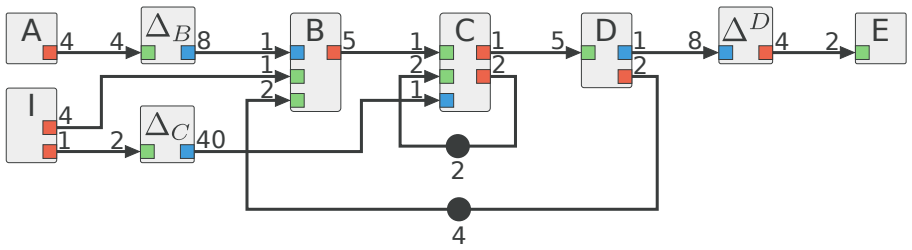
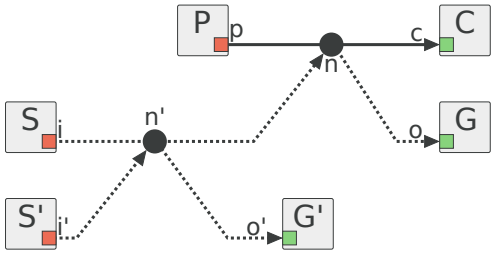
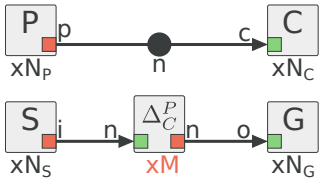


Figure 4.7 – CLEG of the graph of Figure 4.6a.



(a) Example of recursive initialization of delay.



(b) CEG of Figure 4.4a with multiple initialization of the delay.

Figure 4.8 – Illustration of recursive and multiple initialization of delays.

delay attached to the edge $\vec{PC}$ and Figure 4.8b uses the CEG transformation to show a multiple initialization pattern of the delay of Figure 4.4a. The recursive initialization pattern emerges from the fact that if an edge can exists from, or to, a dataflow actor and to, or from, a delay; then there can be a delay on this edge. This property means that initial data tokens of a delay could be set by multiple stages of *setter* and *getter* actors (see Figure 4.8a) and, symmetrically, there could be multiple stages on the edge connecting a delay to a *getter* actors which could themselves be initialized by delays. This pattern has been judged to add too more complexity into the implementation of tools supporting SaMM semantics for no clear functional benefits. Thus, in SaMM, there can be **no delay on an edge going from a *setter* actor to a delay or from a delay to a *getter* actor.**

The multiple initialization pattern shown in Figure 4.8b emerges from the fact that data rates on edges connecting a *setter* actor to delay or a delay to a *getter* actor are not required to be equal to the delay value (see Section 4.3.1). Thus, using the CEG transformation and then computing the topology matrix on the obtained graph, *virtual delay* actor could have a RV greater to 1. Although this does not seem to be a problem at first glance, multiple repetitions of the *virtual delay* actor raises multiple question as

to how to interpret such behavior. Indeed, multiple repetition of the *virtual delay* actor imply that there are either more data tokens produced on the delay by the *setter* actor than the delay value or that there are more data tokens consumed by the *getter* actor than the delay value which leads to two possible scenarios:

S1. Delays should behave as roundbuffers in respect to *setter* actors and as repeat-buffers in respect to *getter* actors when more tokens are produced onto the delay or consumed from the delay, respectively.

S2. The entire path to which the delay is attached should be repeated as many times as the *virtual delay* actor is repeated. For instance, in the graph of Figure 4.8b, actors P and C are repeated N_P and N_C times, respectively, and the *virtual delay* actor Δ_C^P is repeated M times. In this scenario, the entire path composed of N_P execution of actor P and N_C should be executed M times.

Scenario **S1** adds a functional behavior to the delays, meaning that delays are no longer considered as *just* initial data tokens of a given FIFO but as functional entities with potential runtime behavior. However, the purpose of the proposed extended semantics of delays is to enforce the behavior of the existing semantics while adding the possibility of dynamic initialization of delays, hence this scenario is discarded in SaMM. Scenario **S2** adds complexity to the analysis of *SAD* graphs due to the fact that the entire path affected by a given delay need to be determined in order to apply the scaling factor which may not be trivial in graphs with complex cycle and pipeline structures. For this reason, the scenario **S2** is also discarded in SaMM, meaning that *virtual delay* actors are **required to have a repetition value strictly equal to 1** as mentioned in the **Step 3** of Section 4.3.2.

The next section demonstrates the advantages of the proposed semantics in term of conciseness, readability and memory usage over standard SDF based dataflow MoCs. Then an application example is given in Section 4.3.4.2 with the use of the SaMM semantics to model the computation of a matrix multiplication.

4.3.4 Modeling of an Iterative Structure in Dataflow

4.3.4.1 Generic Iterative Process Example

In this section, we model a simple for-loop algorithm structure with the SDF MoC to demonstrate the lack of proper semantics to expose efficiently fine grained parallelism. Then the same structure is modeled with a *SAD* graph to show how the proposed delay

semantics improves the expressiveness of the SDF MoC. In many applications, iterative computations similar to the one shown in Algorithm 1 are used. Algorithm 1 is decomposed in 3 distinct phases, namely the *prologue* (line 1), the *loop kernel* (line 2-4) and the *epilogue* (line 5).

Algorithm 1: Iterative Process Example

Input: Number of iterations N;
Parameter file *paramFile*;

1 $dataBuffer = execute(P,\ inputs_P)$; // actor P
2 **for** $i \in [0 : N]$ **do** // actor I
3 $\quad parameters_i = readFile(paramFile)$; // actor R
4 $\quad dataBuffer = execute(B,\ dataBuffer, parameters_i)$; // actor B
5 $execute(E,\ dataBuffer)$; // actor E

The *prologue* phase is the phase initializing *dataBuffer*. The *loop kernel* is the phase corresponding to the iterative computations of the loop. Finally, the *epilogue* is the processing done after the final iteration of the loop. The three phases of Algorithm 1 are sequential due to the data dependency of *dataBuffer*. This data dependency is enforced at line 4 of the loop kernel phase, where results from the previous loop iterations are used. In other words, the line 4 computation of iteration $n + 1$ and iteration n are not executable in parallel. Nevertheless, parallelism can still be exploited inside each of the 3 phases.

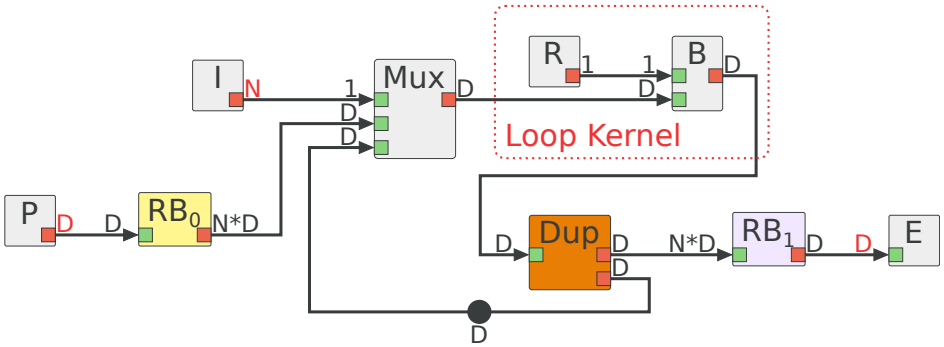


Figure 4.9 – Equivalent SDF graph of Algorithm 1.

Figure 4.9 shows how Algorithm 1 is expressed in the strict SDF MoC. Note that the inner processing of the loop kernel phase is fully exposed in the strict SDF MoC. Actors P, $\{R, B\}$ and E represent the prologue, the loop kernel and the epilogue phases

of Algorithm 1, respectively. The size of *dataBuffer* is noted D and corresponds to the consumption and production rates of actor B. Actor I is used here to set the number N of iterations of the *for loop*. Actors RB_0, RB_1 and *Dup* are special actors used to manage the loop context of Algorithm 1. RB_0 and RB_1 are used to guarantee unique execution of the prologue and epilogue phases. RB_0 duplicates N times the tokens received on its input port to its output port and symmetrically, RB_1 forwards only the last D tokens received on its input port to its output port. The *Mux* actor is a multiplexer used to select which tokens are forwarded to actor B. Since actors are stateless in the SDF MoC, the *Mux* actor distinguishes the first iteration from the rest of the loop based on the values produced by actor I. On the first firing of actor B, tokens produced by actor P are used. For every other firings, actor B uses the tokens produced by its previous firing through a feedback FIFO. The *Dup* actor is a duplicate actor and is used to forward the data tokens produced by actor B to both *Mux* and RB_1.

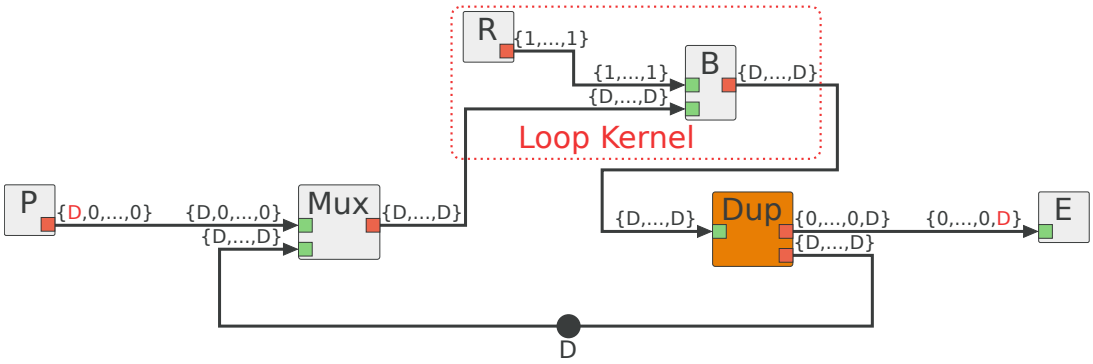


Figure 4.10 – Equivalent CSDF graph of Algorithm 1.

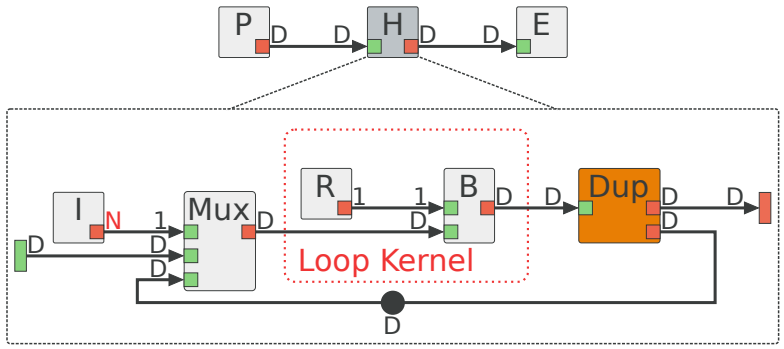


Figure 4.11 – Equivalent πSDF graph of Algorithm 1.

Figures 4.10 and 4.11 show how Algorithm 1 can be expressed using the CSDF [**bilsen_cycle-static_1996**] and πSDF MoCs, respectively. In Figure 4.10, the

cyclic rates remove the need for the RB_0 and RB_1 actors of Figure 4.9. Indeed, the *Mux* actor consumes D data tokens on its first input for its first iteration and then consume 0 data tokens for the $N-1$ other iterations. Symetrically, the *Dup* actor produces 0 data tokens on its first output for the first $N-1$ iterations and then produces D data tokens on the last iteration. The modeling of the Algorithm 1 with the πSDF MoC in Figure 4.11 is very similar to the one with the SDF MoC in Figure 4.9. The main difference is the use of the hierarchy semantics of the πSDF MoC to benefit from the implicit circular buffers of the data input and output interfaces which have the same role as the RB_0 and RB_1 actors of Figure 4.9.

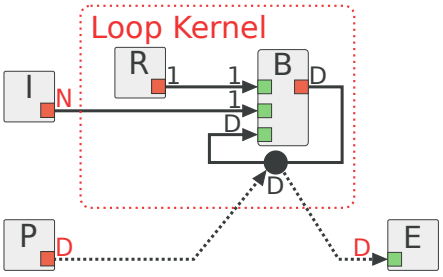


Figure 4.12 – Equivalent *SAD* graph of Algorithm 1.

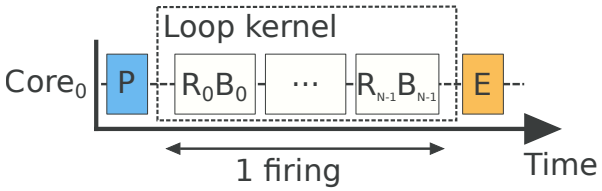
Figure 4.12 shows the representation of Algorithm 1 using the SaMM delay initialization semantics presented in Section 4.3.2. The initialization of the delay is used as the prologue phase of Algorithm 1, then actor B is fired sequentially N times. Finally, the last data tokens produced by B are automatically forwarded to actor E through the delay. Figure 4.12 demonstrates the conciseness improvement offered by the *SAD* semantics of delay.

It would be possible to simplify the graph of Figure 4.9 by encapsulating all iterations of the loop kernel as a single firing of a unique actor. The interest of exposing multiple iteration of the loop kernel in dataflow is demonstrated by analyzing the resulting schedules of both approaches. When the iterations of the loop kernel are not exposed in the dataflow graph, parallelism opportunities such as time pipelining are lost as showed in the resulting schedule of Figure 4.13a. In the schedule of Figure 4.13a, P and E correspond respectively to the prologue and epilogue phases of Algorithm 1. R_i and B_i are the computations of lines 3 and 4 of the i^{th} iteration of the loop. The entire loop kernel is done in 1 firing of the unique actor in which R and B are encapsulated. In the schedule

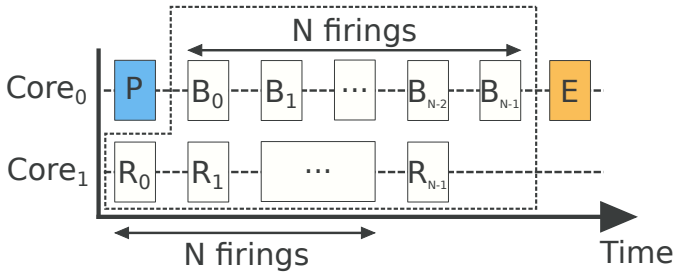
of Figure 4.13a, the total execution time of Algorithm 1 is defined by

$$T1_{critical} = T_P + N * (T_R + T_B) + T_E \tag{4.3}$$

where T_x is the execution time of the corresponding actor x and N the number of iterations of the loop.



(a) Sequential schedule of Algorithm 1.



(b) Pipelined schedule of Algorithm 1.

Figure 4.13 – Potential schedules of Algorithm 1 depending on the level of parallelism of the loop kernel exposed.

In Algorithm 1, the *readFile* function calls (line 3) are independent from the loop iteration. This independence of the executions of the actor R is naturally represented in the equivalent dataflow graphs (Figures 4.9 , 4.10 , 4.11 and 4.12) where R has no incoming dependency and no feedback loop. Thus, it is possible to execute the N firings of actor R in parallel, without interleaving them with the N firings of actor B. Figure 4.13b shows the schedule of the latter scenario. Executions of actor R starts before the start of the loop and allow actor B to immediately starts its computation after the end of the prologue. The resulting total execution time of Figure 4.13b is defined by

$$T2_{critical} = \max(T_B + N * T_R,\ N * T_B + T_P) + T_E \tag{4.4}$$

Given $T1_{critical}$ and $T2_{critical}$ definitions, exposing the inner-loop parallelism gives a significantly shorter execution time. In addition, exposing such loop structures in dataflow is also relevant in the context of Field Programmable Gate Array (FPGA) implemen-

tation of nested loop kernels [**milford_constructive_2016**]. Thus, in the following of this section, we only consider the exposed dataflow representation of the inner-loop.
In Figure 4.9, actors *Mux*, RB_0 and RB_1 and *Dup* added to manage the loop context have a non negligible impact on memory. In the graph of Figure 4.9, *dataBuffer* values are stored simultaneously in 4 different FIFOs for each iteration of the loop: *(Br, Sw)*, *(Sw, B)*, *(B, Br)* and (Br, RB_1). The main issue is that the values of *dataBuffer* are only useful for actor B but get copied 3 times. The remaining FIFOs (RB_0, Sw), (P, RB_0) and (RB_1, E) lead to a total memory allocation of the graph for *dataBuffer* defined by

$$M = D * (2 * N + 5) \tag{4.5}$$

with D the size of *dataBuffer* and N the number of loop iterations.
In Figure 4.12, using the delay to manage the loop means that only 1 FIFO is needed for the entire loop structure. In the graph of Figure 4.9 the size of the allocated memory is dependent on the number N of iterations of the loop (Equation 4.5) whereas with the proposed semantics the allocated memory size is always constant and equal to the size of *dataBuffer*. In a dynamic context where the number of iterations of the loop is resolved at runtime, dynamic allocations of all the buffers combined to the memory transfer operations can have a great overhead on the performance of an application.

4.3.4.2 Matrix Multiplication Example

In this section, the new delay initialization semantic is used for modeling a matrix multiplication computation. The matrix computation is given in Algorithm 2. Algorithm 2 takes two matrices as input, namely A and B, of dimensions $m \times n$ and $n \times p$, respectively, and produces as output a matrix C of dimension $m \times p$.

Algorithm 2: Matrix Multiplication

Input : Matrices $A^{m \times n}$, $B^{n \times p}$;
Output: Matrix $C^{m \times p}$;

```
1 for i ∈ [1 : m] do
2     for j ∈ [1 : p] do
3         C[i][j] = 0;
4         for k ∈ [1 : n] do
5             C[i][j] += A[i][k] × B[k][j];
```

Figure 4.14 shows a possible model of Algorithm 2 using the SaMM semantics. The actors *MatA* and *MatB* produces the two input matrices A and B of the matrix multiplication. Actor $*$ correspond to the Hadamard product of two 1-D vectors (entrywise product) and actor Σ performs the sum of all its input tokens. The combination of actors $*$ and Σ result in the dot product of one row of the matrix A and one column of the matrix B. The *Trans* actor compute the transpose of its input matrix, and is used here to reshape the data of the matrix B in order to be able to use the dot product actor. The *Rep* and *Dup* actors are two special actors that perform specific memory operations. The *Rep* actor is called a *repeat* actor and repeats its input data tokens a given number of times k on its output data port, with $k \in \mathbb{N}^*$. The *Dup* actor is called a *duplicate* actor and duplicates its input data tokens on all of its output data port.

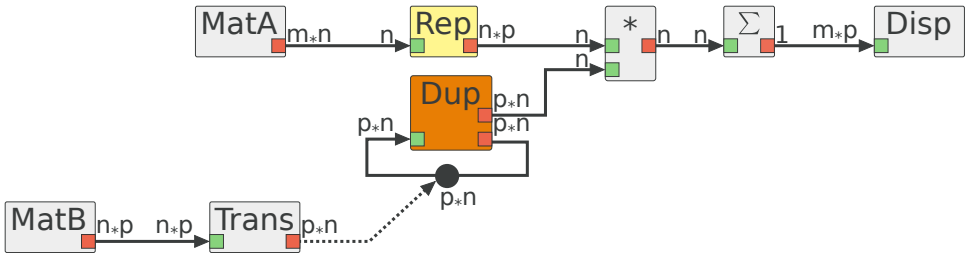


Figure 4.14 – Example of a matrix multiplication modeled with a *SAD* graph.

A firing of actors $*$ and Σ corresponds to the computation of one output value in the output matrix C. Hence, computing 1 row of the output matrix C correspond to p firings of the $*$ and Σ actors, with p the number of columns of matrix B. Data wise, computing one output row requires 1 row of the matrix A and the entire transposed matrix B. To compute the entire output matrix C, it is then necessary to execute $*$ and Σ actors $m * p$ times, with m the number of rows of the matrix A. In Figure 4.14, actor *Dup* is used to repeat the entire transposed matrix B^T m times, with the delay of the self-loop around the actor *Dup* being initialized by the output of the actor *Trans*. The rational behind using this rather odd pattern with the self-loop around the duplicate actor is based on the fact that duplicate actors are often optimized and end up resulting in no-op operations with all of the output data tokens on all output data ports pointing directly to the input data tokens. The duplicate actor is then mainly used here to enforce the need of repetition of the matrix. For example, on an FPGA, this pattern would be entirely optimized away with the $*$ actor fetching data directly from the output of the *Trans* actor m times.

Figure 4.15 shows the CEG of the graph in Figure 4.14 with the different repetition values of the actors noted below them. Analysis of the CEG of Figure 4.15 using the same

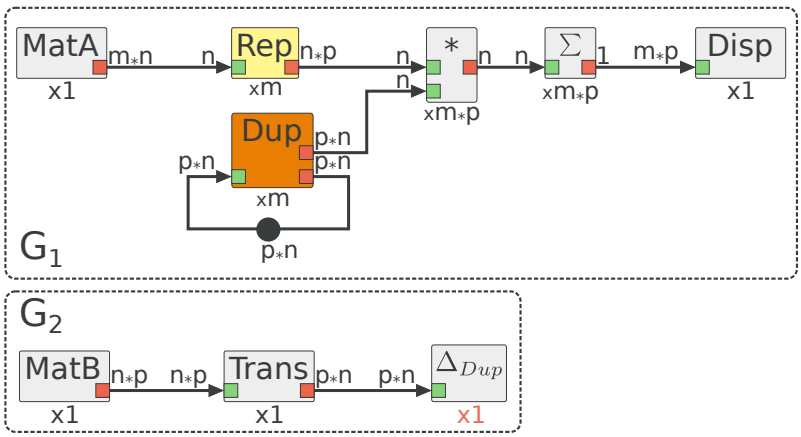


Figure 4.15 – CEG of the graph of Figure 4.14.

approach as in Section 4.3.2.1 shows that the graph in Figure 4.14 is consistent. Finally, the CLEG in Figure 4.16 shows that the matrix multiplication graph is also live.

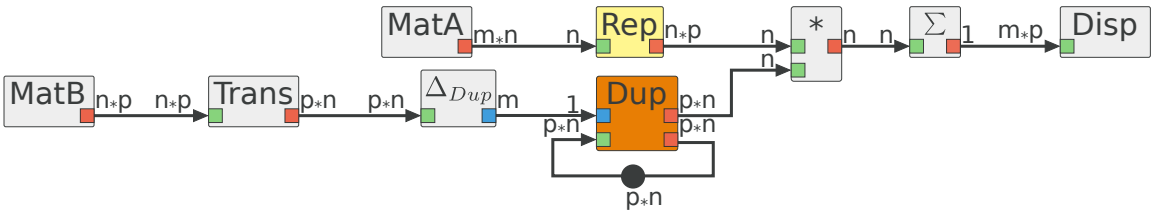


Figure 4.16 – CLEG of the graph of Figure 4.14.

This more concrete example showed how the newly introduced semantics allows to model fined grained parallelism applications in a concise and expressive manner. Modeling such applications is interesting particularly in FPGAs [**milford_constructive_2016**] where each atomic operations, such as multiplications and additions, are mapped to discrete logic blocks leveraging more data parallelism.

4.4 Initial Tokens Values: A Matter of State

Following the concept of graph iteration, a delay in a graph G associated with an explicit initialization such as the one introduced in Section 4.3.1 is initialized once per iteration of G. Therefore, *Consumption* actors of delays always have new data tokens for their first firing of each graph iteration. Having the fixed initial conditions at every iteration of a *SAD* graph G means that pipelined behavior, such as the one depicted in Figure 4.1a, are no longer modelable. Indeed, pipelining multiple executions of a graph

is only possible due to the forwarding of data tokens produced by a given iteration n to iteration $n+1$; which is not possible with the initialization and retrieval semantics presented in previous section. To reconcile pipeline functionality with the new semantics of delay introduced in Section 4.3.1, it is necessary to define unambiguous persistence scopes for delays. The persistence of delays defines whether data tokens inside a delayed FIFO should be discarded or preserved for the next graph iteration. In other words, are graphs allowed to have a persistent state or not?

In this section, the new semantics for persistence of delay of SaMM is presented along with the application of SaMM onto the πSDF MoC, resulting in the State-Aware Parameterized and Interfaced Synchronous DataFlow (SA-πSDF) MoC.

4.4.1 Persistence Scope of Delays

4.4.1.1 Definition

As explained in Section 4.2, having a delay in a hierarchical subgraph G_H of an actor H induces an internal state for H. Such a state can either be discarded at the end of the firing of H or preserved for the next firing. In order to preserve the state of H, it is necessary to expand explicitly the persistence scope of the delay outside of the subgraph G_H. Preserving the state of H has the consequence of inducing a precedence relationship between its successive firings. An actor with a persistent state across graph iterations also constitutes a state for its parent actor, which in turns is considered as having a state, and must have serialized firings. Hence, expanding the persistence scope of a delay to all levels of hierarchy, as proposed in [**lee_dataflow_1995**], induces a precedence relationship between every parent graphs of H. Having such strong constraint on the firing sequences of actors becomes problematic in complex applications with a deep hierarchy as it will strongly undermine the data parallelism of the application. To control the persistence scope of a state in a hierarchical graph, SaMM introduces 3 different types of delays: the *Local Delays*, the *Locally Persistent Delays* and the *Globally Persistent Delays*. The different types of delays are defined here after:

Definition 4.4.1 (Local Delays)
Local Delays (LDs) use the semantics presented in Section 4.3.1. Thus, an LD can be initialized dynamically by dataflow actors. The data tokens contained in the FIFO of an LD are preserved within the scope of a unique graph iteration.

Definition 4.4.2 (Locally Persistent Delays)
Locally Persistent Delays (LPDs) are delays whose data tokens persist outside of the scope of the graph to which the LPD belongs. An LPD specifies the persistence of a delay for one level of hierarchy and establishes a precedence relationship for successive firing of the parent actor H of the subgraph G_H to which the LPD belongs. Any subgraph containing an LPD is guaranteed to have the state associated to this delay preserved from one firing to another inside the graph to which it belongs. Additionally, the persistence scope of an LPD can be extended to any given number N of level of hierarchy independently from the instance of the subgraph to which it belongs. In other words, multiple hierarchical actors sharing the same internal subgraph are allowed to have different levels of persistence. Finally, LPDs can not be initialized using dataflow actors due to the fact that delays need to be initialized prior the first use of the data tokens they held. Therefore, it would imply that setter and getter actors of an LPD would be fired in a different level of hierarchy of the one of the delay they are connected to. However, to be connected to an LPD, setter and getter actors need to be in the same subgraph as the LPD meaning that they will be fired once per subgraph iteration which will initialize the LPD once per subgraph iteration, thus nullifying the persistent property of the LPD.

Definition 4.4.3 (Globally Persistent Delays)
*Globally Persistent Delays (GPDs) are delays that persist across all levels of hierarchy up to the top-level graph. GPDs are initialized only once in the lifetime of an application, prior to the first firing of the top-level graph. Since dataflow actors are fired once per graph iteration, they cannot be used to initialize a GPD once in the application lifetime. Therefore, a GPD is initialized either with a function or a constant value directly associated with the delay. GPDs are equivalent to the delays described in [***lee_dataflow_1995***]. **By definition, any LPD in the top-level of hierarchy is a GPD.***

Figure 4.17 illustrates the different levels of delay persistence definitions. The LDs persist within exactly 1 level of hierarchy, the LPDs persist for N levels of hierarchy and finally the GPDs persist for all the levels of hierarchy. The unambiguous persistence semantics of SaMM enforces the compositionality of hierarchical actors and their reusability. Listing 4.1 shows an equivalence of the SaMM persistence scopes using the C language. In this example, the 3 types of delays of SaMM are illustrated. Local Delays, at line 3, are equivalent to locally declared arrays with a life scope ending at the end of the scope within which they are declared. Locally Persistent Delays, at line 8, are equivalent

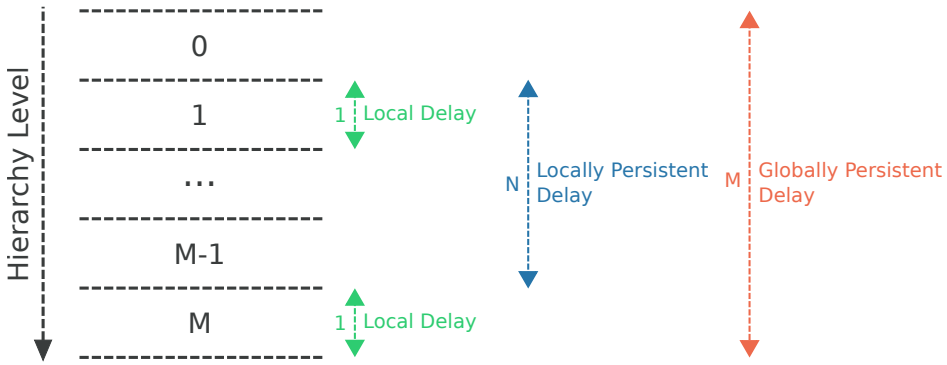


Figure 4.17 – Illustration of the different levels of persistence for delays in SaMM. LDs persist within only 1 level of hierarchy, LPDs persist for N levels of hierarchy and GPDs persist for all the levels of hierarchy.

to arrays declared in a scope greater than the one of the function call they are used in. Finally, Globally Persistent Delays, at line 2, are equivalent to statically declared arrays.

```
1  void subgraph(int *lpd) {
2      static int GPD[4] = { 0 };  // Globally Persistent Delay
3      int LD[4] = { 1, 3, 3, 7 }; // Local Delay
4      // ...
5  }
6  int main(int argc, char *argv[]) {
7      {
8          int LPD[4] = { 0 }; // Locally Persistent Delay
9          for (int i = 0; i < 10; ++i) {
10             subgraph(LPD);
11         }
12     }
13     return 0;
14 }
```

Listing 4.1 – Equivalence of Persistence Scopes in C Language.

4.4.1.2 Hierarchical Implications: Serial vs Parallel Execution

In this section, the hierarchical implications of the persistence of delays are discussed. In other words, how does the level of persistence of a delay influences the execution of a hierarchical graph?

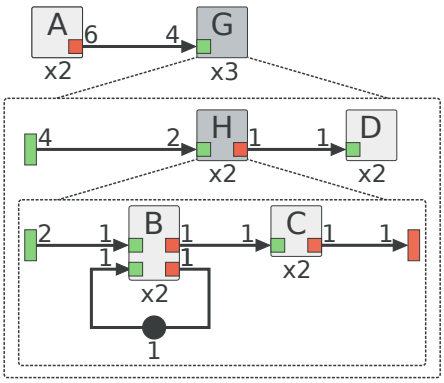


Figure 4.18 – πSDF graph example with an internal delay.

Figure 4.18 shows an example of a πSDF graph with multiple levels of hierarchy. All repetitions values of the actors are noted below them and are relative to their respective graph. For instance, actors B and C have a repetition value of 2 within the subgraph H but a total repetition count of 12 given that actor H is repeated 2 times within the subgraph G, which is itself repeated 3 times. The different levels of hierarchy of the graph of Figure 4.18 are composed of the hierarchical actor G in the top-level graph and the hierarchical actor H in the subgraph of actor G. In the subgraph of actor H, there is a self-loop FIFO around the actor B with a delay attached to it, which leads to the question of how long should the tokens inside the FIFO persist?

The πSDF MoC is a fully compositional MoC [**desnos_pimm:_2013**] which means that each level of hierarchy can be analyzed independently and every hierarchical actors are considered as atomic actors for all analysis tools. Importantly, the hierarchical actors are considered to share the same properties as other atomic SDF actors. One of the most interesting property of SDF actors is the stateless property [**lee_synchronous_1987**]. In other words, SDF actors do not hide internal state and so do πSDF hierarchical actors which means that actors can be executed in an *auto concurrent* context, i.e multiple firings of a same actor can be executed in parallel.

Going back to the Figure 4.18, the property of composition thus defines that all possible firings of a hierarchical actor can be executed in parallel as the internal delay does not create any link between successive firing of a hierarchical actor. Hence, Figure 4.19 shows a possible schedule of the graph of Figure 4.18. Firings of the internal actors of the subgraph of actor H are not showed on the schedule as they are not relevant to the current demonstration. In the schedule of Figure 4.19, all firings of both levels of hierarchy occur in parallel due to the fact that there is no internal state that need to be passed from one firing

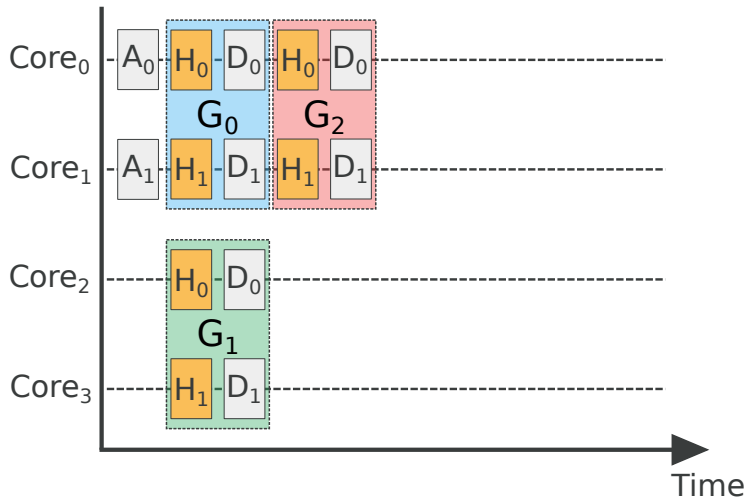


Figure 4.19 – Possible schedule of the graph of Figure 4.18 with no internal state.

of actor H to another. Hence, using the persistence scope definitions of Section 4.4.1.1, all delays inside hierarchical actors of the πSDF MoC, and the delay of Figure 4.18, are defined as Local Delays (LDs). Importantly, in the πSDF MoC, delays located at the top-level graph have the same persistence scope as the delays of the SDF MoC and are considered as Globally Persistent Delays (GPDs).

Figures 4.20 and 4.21 show potential schedules of the graph of Figure 4.18 using an LPD and a GPD for the delay of the subgraph of actor H, respectively. In Figure 4.20, the LPD imposes a serialization of actor H due to the persistence of the delay. Importantly, in this scenario, the LPD only specify the persistence of the delay for one level of hierarchy which has the consequence of creating a local state for actor H for each firing of actor G. Consequently, in this scenario, all firings of the hierarchical actor G can still occur in parallel. However, in Figure 4.21, following the Definition 4.4.3 of Section 4.4.1.1, the persistence of the delay is extended to the top-level graph. This extended persistence scope induces that firings of actor G also need to be serialized.

In conclusion, the graph example of Figure 4.18 associated with the different schedules in Figures 4.19 , 4.20 and 4.21 illustrate the design flexibility of the explicit persistence semantics for delays of SaMM. Indeed, the different persistence scopes definitions of delays offered by SaMM lead to controlled data parallelism in hierarchical graphs which can be taken into account during the analysis and scheduling of the graphs. The next section present the application of SaMM to the πSDF MoC.

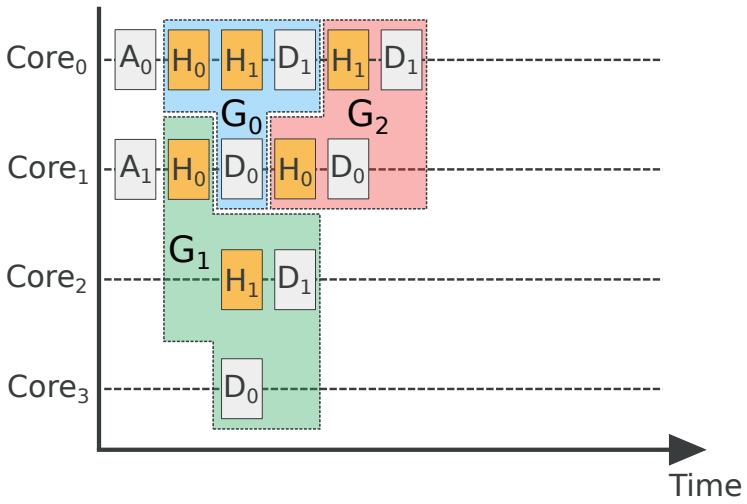


Figure 4.20 – Possible schedule of the graph of Figure 4.18 with a local internal state.

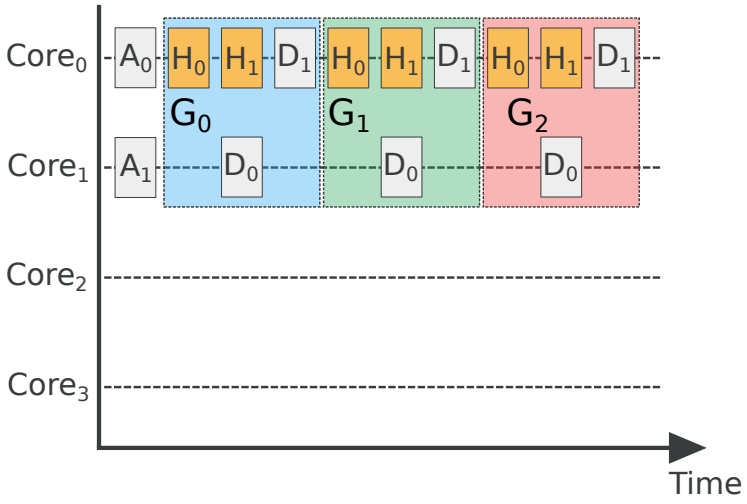


Figure 4.21 – Possible schedule of the graph of Figure 4.18 with a global internal state.

4.4.2 Application to the πSDF MoC

4.4.2.1 SA-πSDF Semantics

The application of SaMM to the πSDF MoC result in the State-Aware Parameterized and Interfaced Synchronous DataFlow (SA-πSDF) MoC. The additional graphical semantics of the SA-πSDF MoC over the πSDF MoC are presented in Figure 4.22. In Figure 4.22, the delay inside the hierarchical actor H is an LPD and its persistence scope extends up to the top-level graph. This extended persistence scope is made explicit in Figure 4.22 with a dashed self-loop around the actor H. The explicit representation of the

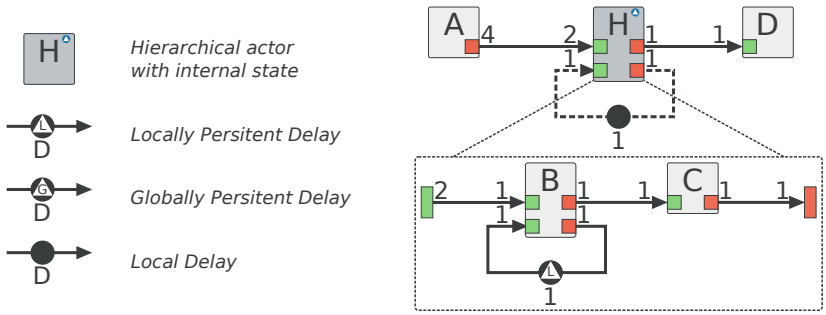


Figure 4.22 – SA-πSDF graph example and associated graphical semantics.

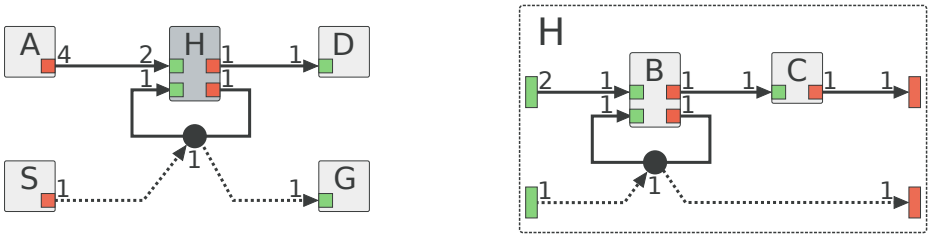


Figure 4.23 – Example of emulating the LPD of the graph of Figure 4.22 using LDs.

persistence scope of the delay is only represented in Figure 4.22 to help the reader and is not required in practice when designing SA-πSDF graphs. Definition 4.4.2 states that LPDs can not be initialized dynamically using dataflow actors. This property holds in the SA-πSDF MoC. However, it is possible to emulate the behavior of an LPD and have the dynamic initialization with dataflow actors using LDs and the inherited πSDF data interfaces. Figure 4.23 shows the equivalent graph of Figure 4.22 using LDs and *setter* and *getter* actors. Using this emulated persistence representation, it is the responsibility of the application designer to make sure that the inner state of the graph is correctly propagated with a correct number of delays to upper levels of hierarchy.

4.4.2.2 SA-πSDF Runtime Operational Semantics

The execution of a SA-πSDF graph G associated to a hierarchical actor a_G follows the same steps as a πSDF graph [**desnos_pimm:_2013**]:

1. Wait for partial configuration of G.
2. Compute the production and consumption rates of the data interfaces based on the partial configuration.
3. Wait until a_G is fired by its parent graph.

4. Fire the configuration actors that will set the configurable parameters of G: a complete configuration is reached.
5. Check the local synchrony of G with the rates and delays resulting from the complete configuration and compute a schedule (if possible).
6. Fire the non-configuration actors: i.e complete an iteration of G.
7. Produce on the output ports of a_G the data tokens and parameter values written by the actors of G.
8. Go back to step 3 to start a new iteration of G, i.e. a new firing of a_G.

In reconfigurable and hierarchical dataflow MoCs such as the πSDF MoC, a graph can dynamically configure parameters of hierarchical subgraphs. Specifically, a graph can change the number of delays of a subgraph at runtime [**desnos_pimm:_2013**]. Changing the number of delays of a graph affects the liveness and the consistency of the graph as mentioned in Section 4.2. Thus, the number of delays must be known when the liveness and consistency analysis of a graph are verified.

With the SaMM semantics, persistent delays can impact different levels of hierarchy depending on their persistence scope. Thus, since a persistent delay inside a subgraph s_G is forwarded at least to the parent graph p_G, any change to the number of delays has to be known when firing p_G (step 3) in order to be able to check the local synchrony of p_G (step 5). The dependency between p_G and s_G means that the value of a **persistent delay can not be set by a reconfigurable actor within** s_G. This property extends to the highest level of persistence of a delay, that is if a Locally Persistent Delay is set to be persistent across 2 levels of hierarchy then its value can neither be set by a reconfigurable actor in its original subgraph but neither in the second level in which it is set to persists. In other terms, **the number of data tokens of a persistent delay can not depend on a parameter located in a level of hierarchy below the highest level of hierarchy in which the delay persists.** The SA-πSDF MoC inherits the semantics, the compositionality, and the schedulability properties of the πSDF MoC [**desnos_pimm:_2013**]. Thus, in the SA-πSDF MoC, the static parameter tree inherited from the πSDF MoC naturally enforces this property.

Having parameterized value of delays means that a delay can vary in size during the lifetime of an application. For Local Delays, it does not affect their behavior since they are only allowed to change their size during steps 1 to 2 of the firing of a graph. However, it is necessary to explicit the behavior of persistent delays when changes in size occur. If a

persistent delay increases in size, the additional tokens are set to 0. If the delay decreases in size, only the last tokens of the FIFO are kept. For instance, if a delay is of size D and decreases to size $D2$, only the $D2$ last tokens of the original FIFO are kept.

4.5 SA-πSDF Application Example

4.5.1 Application Description

In this section we use the Continuous Actor Critic Learning Automaton (CACLA) algorithm [**van_hasselt_reinforcement_2007**] as an application example to demonstrate the conciseness and memory efficiency of the SA-πSDF MoC. CACLA is part of the reinforcement learning branch of machine learning. Reinforcement learning consists of learning the model of an environment E and taking actions accordingly without prior knowledge of E. The reinforcement learning algorithm learns the model of E based on an abstract representation of E called the state (S) of the environment. For instance, in the case of the control of a robotic arm, the environment E is the arm, the state S would be the position and velocity of each motor, and the actions would be the commands of the motors of the arm. Due to a lack of space, the whole application is not detailed here, but it is available in the PREESM tool [**pelcat_preesm:_2014**] open-source repository [3].

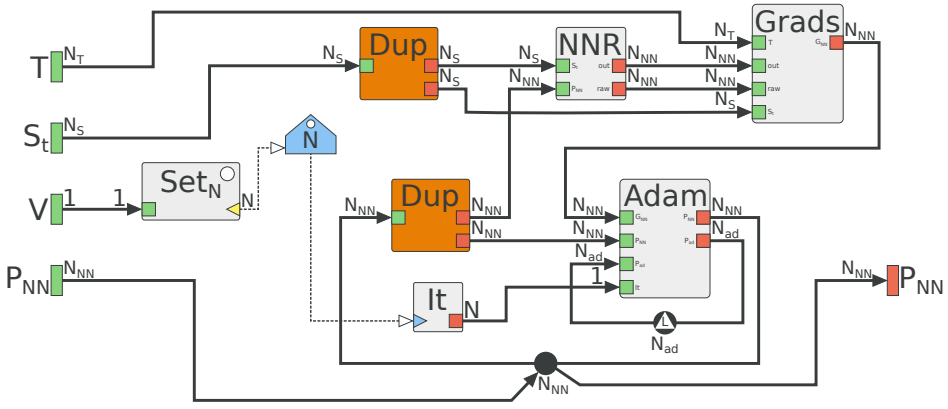


Figure 4.24 – SA-πSDF graph of the CACLA actor update algorithm. N_x is the size of the associated x parameter.

CACLA uses multiple neural networks to make predictions and the coefficients of the neural networks are updated at each iteration of the main *loop* of the algorithm. Figure 4.24 illustrates the subgraph responsible for the update of the coefficients of the

3. https://github.com/preesm/preesm-apps

neural network that predicts the actions to apply to E. The graph takes as inputs the coefficients of the neural network W_{NN}, the target T toward which the network is updated, the state S_t of the environment and the variance V of the temporal difference error [**van_hasselt_reinforcement_2007**]. The graph produces as output the updated coefficients of the neural network P_{NN}. The *NNR*, *Adam* and *Grads* actors are hierarchical actors whose internal behavior are not detailed here.

In the graph of Figure 4.24, the coefficients of the neural network P_{NN} are updated iteratively N times, N being set by the configuration actor Set_N. The *NNR* actor is a Multi Layer Perceptron (MLP) neural network that takes the state of the environment S_t as input and predicts the action based on current coefficients of the network. The *Grads* actor computes the gradients of each coefficient of the neural network based on a target T and the outputs of *NNR*. Finally, the *Adam* actor apply the gradients on the parameters using the Adam optimizer algorithm [**kingma_adam:_2014**]. The graph of Figure 4.24 uses both semantics introduced by the State-Aware dataflow Meta-Model (SaMM). The Locally Persistent Delay is used to store the hyper parameters and coefficients needed by the Adam algorithm [**kingma_adam:_2014**]. The Local Delay is used for the iterative update of the coefficients of the neural network.

4.5.2 Results

Table 4.1 – Comparison of memory usage of SA-πSDF and πSDF implementations of the Cacla algorithm.

	πSDF	SA-πSDF	Gain
Update of neural network (in bytes)	9716	**4430**	56, 53%
Full application (in bytes)	27492	**17720**	35, 4%

Table 4.1 shows the difference in memory usage between the SA-πSDF and a strict πSDF implementation of the Cacla algorithm [**van_hasselt_reinforcement_2007**]. The memory usage is defined as being the amount of allocated memory needed to run the application. Table 4.1 presents the difference in memory usage for both the full Cacla application and the subgraph of the update of the neural network of Figure 4.24. The memory comparison is based on a neural networks with 3 layers and a total of 101 parameters per network, where each parameter is encoded on 4 bytes. A gain of 35.4% in total memory usage is observed for the whole application with a gain of 56.53% of memory

usage for the update graph alone. The update of the neural network is the part of CACLA benefiting the most from the new semantics due to the use of a persistent delay and due to the iterative loop needed for the update. Indeed, in the strict πSDF MoC, making a delay persistent across levels of hierarchy induce the need of extra actors similarly to Figure 4.9. The results of Table 4.1 show that for applications with iterative computations and with locally persistent data, the new semantics of *SAD* can drastically reduce the memory footprint of the application. Thus, *SAD* is particularly well suited for modeling applications on embedded platforms with sparse ressources.

4.6 Conclusion

Delays are an important part of existing dataflow MoCs and yet,they lack of a proper definition and flexibility. Indeed, in current dataflow MoCs, delays are seen as unspecified initial conditions of an application and are used to endure the state of the application by passing their data tokens from one iteration of the application to the next. However, these properties of delays are limited and does not permit the modeling of patterns such as iterative structures efficiently. The problem is of greater importance with hierarchical dataflow MoCs where the persistence property, i.e the property of how long data tokens inside a delay should persist, is either ambiguous or limited leading to potentially inefficient scheduling decisions.

In this chapter, we have proposed a new dataflow meta-model called the State-Aware dataflow Meta-Model (SaMM) that extends the semantics of delays of a given dataflow MoC. SaMM can be applied to a wide range of dataflow MoCs to extend their expressivity and conciseness, while preserving the analysis tools of the extended MoC. We have shown that SaMM is well suited to expose the fine-grain parallelism of nested loops with less memory overhead compared to state-of-the-art dataflow MoCs. SaMM brings functional aspect of an application into the model space by expliciting initial conditions and persistence of hierarchical graphs at any given point in time, thus ensuring independence of the model from its implementation. The well-defined notion of local and global state scope provided by SaMM leads to higher design flexibility of complex hierarchical application compared to state-of-the-art MoCs. Finally, we have demonstrated the conciseness and memory efficiency of SA-πSDF through a reinforcement learning example application.

CHAPTER 5

An Efficient Intermediate Representation for Resources Allocation

5.1 Introduction

Stream processing applications running on Heterogeneous Multi-Processor Systems on Chips (HMPSoCs) require efficient resource allocation and management, both at compile-time and at runtime. To cope with modern adaptive applications whose behavior can not be exhaustively predicted at compile-time, runtime managers must be able to take resource allocation decisions on-the-fly, with a minimum overhead on application performance.

Resource allocation algorithms often rely on an internal modeling of an application. Directed Acyclic Graphs (DAGs) are the most commonly used models for capturing control and data dependencies between tasks. DAGs are notably used as an intermediate representation for deploying applications modeled with a dataflow MoC on HMPSoCs. For multirate and reconfigurable dataflow MoCs, building such an intermediate representation at runtime for massively parallel applications is costly both in terms of computation and memory overhead. Indeed, in multirate MoCs, one data parallel node in the original model graph can transform into potentially hundreds of nodes in the final DAG. Additionally, cyclic paths are necessarily unrolled in the resulting DAG, adding to the number of nodes.

In this chapter, an Intermediate Representation (IR) of DAGs for resource allocation is presented. The proposed IR is composed of the original dataflow graph and an ad-hoc numerical model encompassing all the data dependencies that would otherwise be included in the DAG representation. This new representation shows improved performance for runtime analysis of dataflow graphs with less overhead in both computation time and memory footprint. The performances of the proposed representation are evaluated on a set of computer vision and machine learning applications.

In an embedded context, taking fast and efficient decisions requires an efficient intermediate representation of the application. Using compact and expressive dataflow MoCs, such as the CSDF [**bilsen_cycle-static_1996**], the Schedulable Parametric Dataflow (SPDF) [**fradet_spdf:_2012**] or the IBSDF [**piat_interface-based_2009**] allows for a high-level description of an application. However, the more compact and expressive the representation, the more costly it can be to extract information. For instance, extracting fine-grain dependencies information from a DAG is a straightforward operation. However, it is first necessary to compute a model transformation on a CSDF-based application to do so. The more expensive stages of expressive model analysis have led to the more frequent use of DAG-based models in programming frameworks. Frameworks such as StarPU [**augonnet_starpu:_2009**], XKaapi [**gautier_xkaapi:_2013**], OpenVX [**kronos_group_openvx_2013**] or TensorFlow [**abadi_et_al._tensorflow:_2016**] rely on DAGs dataflow MoCs. DAGs efficiently model directed workflows with task-level parallelism. However, complex structures such as loops are cumbersome to model with DAGs due to the fact that the entire loops have to be unrolled, making it impractical on large loops.

There is a paradox between developing more expressive and more compact dataflow MoCs, and the fact that analysis methods often depend on the need of expanding expressive graphs into DAGs. Some works, however, try to take advantage of the expressiveness of the original MoC [**deroui_relaxed_2017**] or to limit the expansion of graphs and accelerate analysis [**zaki_partial_2012**].

Construction of the intermediate DAG representation at runtime is a costly step that needs to be repeated multiple times in the context of reconfigurable applications. In this chapter, we propose a numerical modeling of the expanded DAG representation of the SDF-based MoC and some of its extensions which avoids having to build the intermediate DAG completely, thus improving significantly the performance of embedded runtimes. Our representation allows using DAG oriented analysis methods while maintaining the

compactness and the expressiveness of the targeted dataflow MoC. The proposed numerical modeling of DAGs was implemented in the Spider tool [**heulot_spider:_2014**] on three different platforms ranging from a medium laptop to a low power embedded platform. Our experiments show a significant reduction of the overhead of the Spider embedded runtime both in terms of execution time and memory footprint of the runtime. The work presented in this chapter has been accepted in the 2019 EMSOFT international conference and published in the ACM Transactions on Embedded Computing Systems (TECS) journal.

The challenges of dynamic scheduling of dataflow applications are presented in Section 5.2. The Single-Rate Directed Acyclic Graph (SR-DAG) transformation is presented in Section 5.2.1, followed by a presentation of existing runtimes that use DAG representation and methods that aim at avoiding the full expansion of DAG in Sections 5.2.3 and 5.2.4. Then, our numerical representation of the DAG is presented in Section 5.3. Section 5.4 presents experimental results of the implementation of our contribution into the Spider tool [**heulot_spider:_2014**] on signal processing applications. Finally, Section 5.5 concludes this chapter.

5.2 Dynamic Scheduling of Dataflow Applications Challenges

In this section, the SR-DAG specialization of SDF [**lee_dataflow_1995**] and the related transformation between an SDF Graph (SDFG) and an SR-DAG are first presented. Then, the different challenges the proposed contribution addresses are detailed in Section 5.2.2 followed by a presentation of existing techniques and frameworks where these challenges are partially addressed.

5.2.1 The Single-Rate Directed Acyclic Graph Transformation

A Single-Rate Directed Acyclic Graph (SR-DAG), also called Acyclic Precedence Expansion Graph (APEG) in the literature [**lee_synchronous_1987**], is a specialization of an SDFG. An SR-DAG does not contain any cycle and all the data rates on the edges composing the graph are unitary which means that for every edge, the production and consumption rates are equal. Figure 5.1 shows the transformation of a πSDF graph, in the upper part of the figure, to the equivalent SR-DAG, in the lower part of the figure.

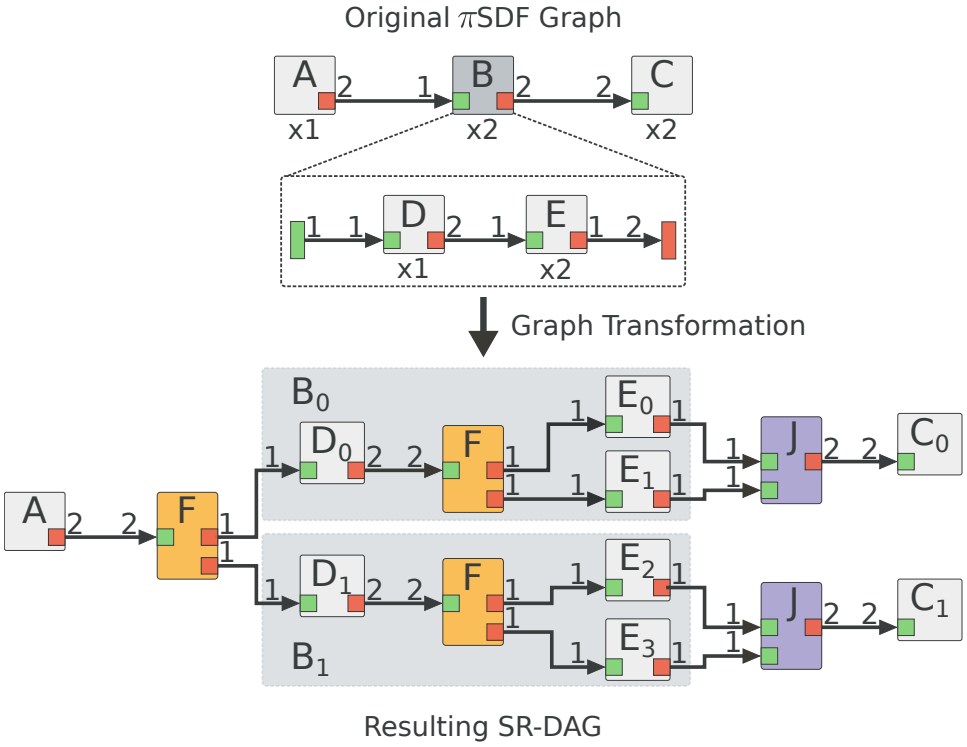


Figure 5.1 – A πSDF to SR-DAG transformation example.

Under each πSDF actor of Figure 5.1 are noted their repetition value relative to their containing graph. Actors D and E have repetition values of 1 and 2, respectively, within 1 iteration of actor B but a global repetition value of 2 and 4, respectively. In the SR-DAG, all actors have a repetition value strictly of 1.

In our work, SR-DAG is considered to respect the SDF MoCsemantics. Particularly, each data port can be connected to a unique edge. Thus, in order to respect this constraint, special actors are introduced to distribute data tokens from a single produced to multiple consumers, and symmetrically to merge the data tokens from multiple producers to a single consumer. Fork actors split a given edge into multiple edges such as $\sum_{j=0}^{n-1}(p_j) = P_F$, where p_j is the production rate of the split edge j of the Fork actor and P_F is the production rate of the original edge. In Figure 5.1, three Fork actors are added for the edges $\vec{AB}$ and $\vec{DE}$ during the SR-DAG transformation. Symmetrically, Join actors merge multiple edges into one edge, with $\sum_{j=0}^{n-1}(c_j) = C_J$, where c_j is the rate of merged edge j and C_J is the consumption rate of the obtained merged edge. In Figure 5.1, two Join actors are added for the edge $\vec{BC}$ of the original πSDF graph, which becomes an edge $\vec{JC}$ after the SR-DAG transformation.

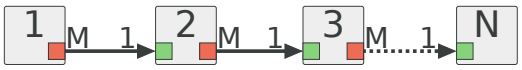


Figure 5.2 – An SDF graph resulting in $O(M^N)$ SR-DAG actors.

Building the SR-DAG of an SDFG is a way of explicitly exposing dependencies across all actor firings of the original SDFG. The SR-DAG exposes all information a scheduler needs to take decisions. However, once the SR-DAG is built, the scheduler no longer benefits from the compact and expressive representation of the original MoC used to describe the application. For instance, using the πSDF representation of the graph in Figure 5.1, a scheduler could easily perform hierarchical scheduling of actor B, whereas using the SR-DAG representation this information is lost. Having a tunable intermediate representation where information is already pre-processed helps to make simpler and faster scheduling algorithms. Finally, the complexity of building the SR-DAG on graphs with a high degree of parallelism grows exponentially with the repetition values of the actors and so does the complexity of the scheduling algorithm. An example of a graph with such exponential growth is given in Figure 5.2 where each actor is executed M times relatively to its predecessor. Indeed, in Figure 5.2 the actor 2 is fired M times and the actor 3 is also fired M times for each firing of the actor 2 which results in a total of M^2 firings of the actor 3. Following this pattern up to the last actor of the graph leads to M^{N-1} firing of the actor N. Building the SR-DAG representation of an SDFG is therefore not well-suited for embedded runtimes where scheduling needs to be done on-the-fly.

5.2.2 RunTime Challenges

In the context of this work, reconfigurable dataflow MoCs such as the πSDF [**desnos_pimm:_20**
or the SPDF [**fradet_spdf:_2012**] are considered. Here, reconfigurable means that application graphs may evolve at runtime with changes in data rates or in the graph topology itself. Reconfigurable dataflow MoC imply that full static analysis of an application is not always possible at compile-time and needs to be handled at runtime. SPDF and πSDF MoCs allow for a quasi-static schedule to be derived at compile-time, removing a part of the runtime overhead. However, this work only consider the case where quasi-static schedules are not derived at compile-time as it is the worst case scenario for these models.

When dealing with dynamic behavior such as graph reconfiguration, the first challenge is to perform graph analysis and scheduling of the application with an overhead as low

as possible relative to the application execution time. Ideally, the time allocated for these analyses should always be negligible compared to computation time of the application itself.

A second concern of matter should be the memory footprint of the runtime manager. Some analysis techniques require storing additional information that is only used for analysis purposes. For instance, in the Kalray Massively Parallel Processor Array (MPPA), memory is a great concern. The MPPA architecture features 16 clusters composed of 16 VLIW processing core each. Each of the cluster has a local memory of 2MB and, although it has access to a larger shared-memory, reading and writing to this memory is expensive and should be avoided as much as possible. In such a context, storing additional information only for analysis purposes can result in more frequent accesses to the shared-memory and thus in a downgrade of overall performances.

5.2.3 Existing Runtimes

HMBE Integrated HTGS (HI-HTGS) [**wu_design_2018**] is a design tool that aims at automating analysis and optimizations of WSDF [**keinert_modeling_2006**] graphs. HI-HTGS provides a lock-free and race-condition-free scheduler that dynamically adapts to changes in actor execution times and cope with non-deterministic characteristics of thread-based execution. HI-HTGS works in two distinct phases: a compile-time phase and a runtime phase. During the compile-time phase, HI-HTGS builds the SR-DAG representation of the WSDF user graph and performs various analyses that will be used during the runtime phase. At runtime, HI-HTGS uses the built SR-DAG and additional information of the compile-time phase to perform dynamic scheduling on multi-core processors. Due to the compile-time construction of the SR-DAG, HI-HTGS only handle static applications.

Spider [**heulot_spider:_2014**] is a runtime manager designed for the execution of reconfigurable πSDF [**desnos_pimm:_2013**] applications on HMPSoCs platforms. Spider takes a high-level πSDF graph description of an application as input. Due to the reconfigurable nature of the πSDF MoC, Spider derives an SR-DAG and performs graph optimizations, mapping and scheduling of the application at runtime, as opposed to HI-HTGS [**wu_design_2018**]. The transformation to SR-DAG may take a non-negligible time on reconfigurable applications with a high-degree of task and data parallelism and with low complexity computation kernels, hence the need for a more compact representation of the SR-DAG.

The OpenVX [**kronos_group_openvx_2013**] standard is a graph-based Application Programming Interface (API) proposed by the Khronos group for developing and deploying computer vision applications on embedded platforms. The MoC used by OpenVX is an SR-DAG specialization of the SDF MoC [**lee_synchronous_1987**]. As seen in Section 5.2.1, SR-DAGs are less-expressive and more restrictive than SDFGs but allow for global high-level optimization. However, they limit the data-parallelism opportunities due to the fact that in OpenVX each node is supposed to be an atomic computer vision, or deep-learning, computation kernel. In SDFGs, non-unitary data rates between actors favor data parallelism allowing for each computation kernel to be further parallelized. Hence, OpenVX standard relies mostly on task-parallelism.

Other runtimes such as StarPU [**augonnet_starpu:_2009**] or XKaapi [**gautier_xkaapi:_201**] are task-graph based runtimes. Similarly to OpenVX, StarPU and XKaapi use a DAG dataflow model to schedule the different tasks. However, StarPU and XKaapi mainly focus on High-Performance Computing (HPC) on heterogeneous architectures composed of multi-core CPUs and GPUs whereas OpenVX main focus are computer vision applications on embedded platforms. It is important to note that contrary to OpenVX, StarPU schedules the application graph at the same time it is constructed, thus limiting its vision of the full application for resource allocation decisions but allowing for dynamic reconfiguration of the application.

5.2.4 Avoiding Excessive Graph Expansion

Building the SR-DAG of an SDF application does not guarantee the best performance. The resulting graph often contains more parallelism than what can actually be exploited by the targeted architecture. Moreover, the exponential growth of the SR-DAG with respect to the original SDFG increases the complexity of scheduling algorithms for HMPSoCs platforms.

To limit the explosion of nodes in the SR-DAG transformation, the clustering of the original SDFG is proposed in [**pino_hierarchical_1995**], where four clustering criteria are identified. These clustering criteria provide sufficient condition for checking the introduction of deadlocks in resulting clustered graphs. Pino et al. then propose a hierarchical scheduling algorithm and show that clustered SDFGs result in faster scheduling with very low impact on the obtained makespan compared to scheduling the full SR-DAGs. Using a MoC that is hierarchical and compositional by nature, as in the IBSDF [**piat_interface-based_2009**] or the πSDF [**desnos_pimm:_2013**] MoCs,

removes the need for the clustering step and the hierarchical scheduling algorithm may be used directly.

Another approach to avoid the full-expansion of an SR-DAG is called the vectorization of SDFGs [**ritz_optimum_1993**]. In [**ritz_optimum_1993**], the optimal vectorization of an SDFG is achieved by multiplying the rates of the original graph by integers resulting in less invocation of the actors of the SDFG. Partial Expansion Graphs (PEGs) [**zaki_implementation_2017**] formulation provides a framework in which the vectorization of actors is integrated efficiently for multiprocessor scheduling context. Zaki et al. use Particle Swarm Optimization (PSO) to find and adjust the amount of expansion, or vectorization, of the actors of the graph.

Schedule-Extended SDFGs [**damavandpeyma_schedule-extended_2013**] are another class of SDFGs that aims at providing a more compact representation for throughput analysis and buffer sizing than SR-DAGs. Damavandpeyma et al. show that encompassing scheduling information directly into the original SDFG significantly reduces time for iterative throughput and buffer sizing analysis. Additionally, authors show that SR-DAG representation may lead to overestimated required buffer sizes compared to applying the same buffer sizing technique on schedule-extended SDFGs. The authors also mention that the construction time of the SR-DAG is very low compared to the analysis time. Although this is true in the context of static analysis at compile-time, the same assumption can not be made when the construction of SR-DAG is performed at runtime. Experiments in Section 5.4 show that in the Spider tool [**heulot_spider:_2014**], for all applications and platforms, the overhead induced by the construction time of the SR-DAG alone is significantly higher than the scheduling time of the SR-DAG.

Most of the existing work presented in this section show that using an SR-DAG transformation for scheduling and analysis of dataflow graphs is the most classical approach. SR-DAG offers a complete exposure of task and data parallelism available in the application. However, most of the presented work use static dataflow MoCs and SR-DAG computation time is neglected, as it can be computed at compile-time. In the context of a reconfigurable MoC such as the πSDF MoC [**desnos_pimm:_2013**], embedded runtimes need to compute SR-DAG on-the-fly which may have a significant overhead on application performance, especially in the context of embedded platforms.

In the rest of this chapter, we show that it is possible to use SR-DAG information without having to pay the actual cost of building and storing it. In Section 5.4, the results

of the implementation of our contribution in the SPIDER [**heulot_spider:_2014**] tool show significant gain both in term of memory footprint and computation time overhead.

5.3 Numerical Modeling of Dataflow MoCs

In this section, the proposed IR which consists of a dataflow graph augmented with an ad-hoc numerical model of the dependencies of the graph is presented. In Section 5.3.1, we show how it is possible to numerically model an SR-DAG by the equations of dependencies that define it. Then, we show that it is possible to further tune these equations in order to encompass the hierarchy semantics of the πSDF MoC.

5.3.1 Modeling a Flat Dataflow MoC: The SDF case

In this section, a numerical representation of the dependencies of an SDFG is presented. First, the use of an SR-DAG is illustrated with an example, then the numerical model of dependencies is developed. In Sections 5.3.2 and 5.3.3, this model is extended to take into account the specificity of the πSDF MoC.

In the following, we refer to the *firing* a_i of actor a as being the i^{th} invocation of actor a during 1 iteration of the graph containing it. The last firing of actor a is a_{q_a-1}, with q_a being the repetition value of actor a. In the original work of Lee et al. [**lee_synchronous_1987**], SR-DAG is depicted as a step needed for scheduling an SDFG.

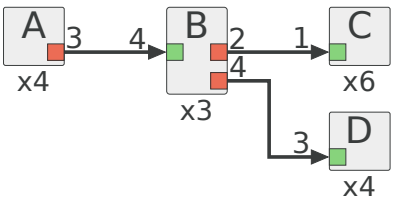


Figure 5.3 – SDF graph with overlapping dependencies.

The SR-DAG removes all the cycles and exposes the precedence relationship between the different firings of all actors within one iteration of a given SDFG. Figure 5.3 shows an example of a simple SDFG in which there is some overlapping in dependencies for the execution. We refer to overlapping dependencies as the fact that multiple firings of a same actor depend on the same firings of another actor. For example, in the graph of Figure 5.4 which is the SR-DAG of the graph of Figure 5.3, firings D_0 and D_1 both

depend on firing B_0. The SR-DAG of Figure 5.4 unravel all the dependencies of the graph of Figure 5.3 both for scheduling the execution of the graph and for the memory allocation of the different Fifos. For instance, D_0 depends only on B_0 but D_1 depends on both B_0 and B_1. On the other hand, every two firings of actor C depend on only one firing of actor B meaning that a scheduler minimizing memory allocation could schedule two successive firings of C between two firings of B so that the allocated buffer of the Fifo $\vec{BC}$ is reused. Importantly, the added Fork and Join actors are necessary in the SR-DAG transformation to explicit the shared dependencies but they are not necessary to model those dependencies and thus will not appear in the proposed numerical representation.

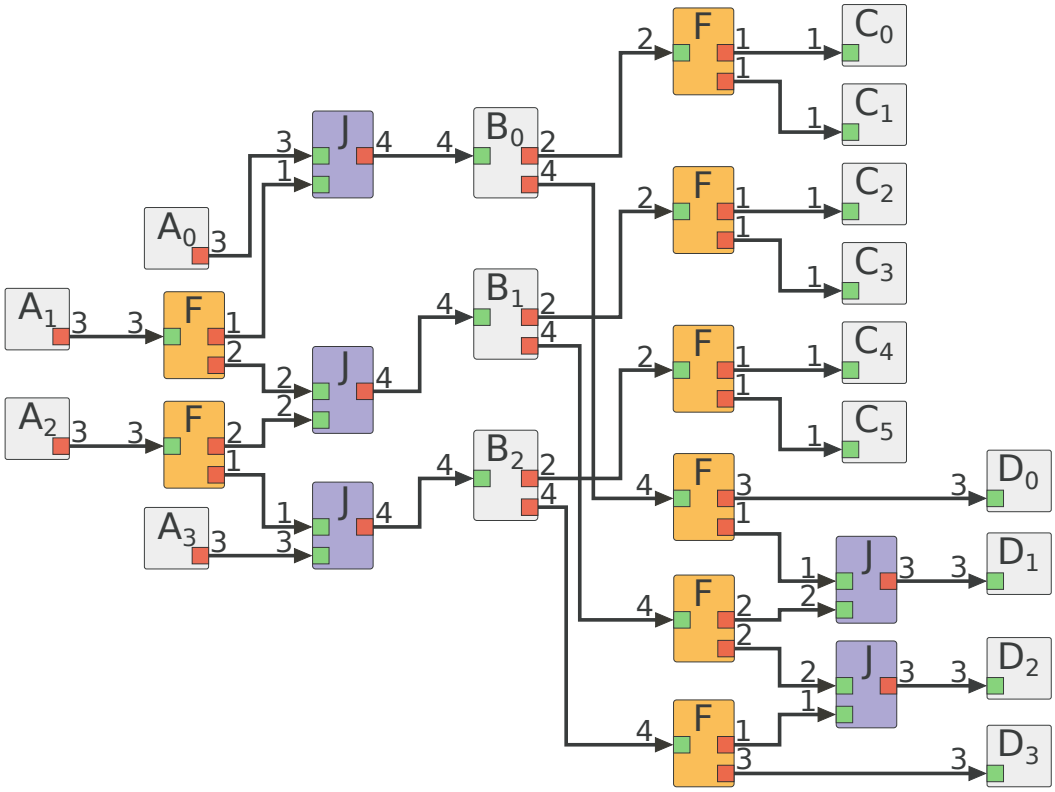


Figure 5.4 – SR-DAG of πSDF graph of Figure 5.3.

Building the SR-DAG, however, is not necessary to have the information of the dependencies. All dependencies between firings of actors can be derived numerically by analyzing the production and consumption rates of the different edges of the graph. Let Δ_a be the dependency matrix of an actor a in Equation (5.1a). Dimensions of Δ_a are $N_{in} \times q_a$, with N_{in} the number of input edges of actor a and q_a the repetition value of a. There is one row for each input edge e_j of actor a and one column per firing k of a. Each

value of Δ_a, noted $\delta_{j,k}$ (Equation (5.1b)), is a sub-matrix of size 1×2 that corresponds to an interval of dependency for edge e_j and firing k of actor a.

The first value of $\delta_{j,k}$, noted $\delta^0_{j,k}$, corresponds to the first firing of $src(e_j)$ needed for the firing of a_k, with $src(e_j)$ being the actor producing data tokens on e_j. The second value of $\delta_{j,k}$, noted $\delta^1_{j,k}$, correspond to the last firing of $src(e_j)$ needed for the firing of a_k. In other words, $\delta_{j,k}$ represent the interval of firings of $src(e_j)$ on which firing k of actor a depends to execute. Since dependencies are necessary in increasing order, the first and the last firing number of $src(e_j)$ are sufficient to define completely the dependency interval.

$$\Delta_a = \underset{\text{edges}}{\downarrow} \overset{\text{firings of } a}{\longrightarrow} \begin{bmatrix} \delta_{0,0} & \delta_{0,1} & \cdots & \delta_{0,q_a-1} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{N_{in}-1,0} & \delta_{N_{in}-1,1} & \cdots & \delta_{N_{in}-1,q_a-1} \end{bmatrix} \tag{5.1a}$$

$$\delta_{j,k} = \begin{bmatrix} \delta^0_{j,k} & \delta^1_{j,k} \end{bmatrix} \tag{5.1b}$$

Taking the example of Figure 5.3, Equation (5.2) gives the corresponding dependency matrix of actor D. Firing 0 of actor D, D_0, depends on the firings 0 to 0 of actor B, i.e D_0 can be fired as soon as B_0 is finished. Similarly, D_1 depends on firings 0 to 1 of actor B. Hence, D_1 can be fired if and only if B_0 and B_1 have finished their execution.

$$\Delta_D = \overset{\vec{BD}}{} \begin{matrix} \begin{matrix} D_0 & D_1 & D_2 & D_3 \end{matrix} \\ \begin{bmatrix} [0\ 0] & [0\ 1] & [1\ 2] & [2\ 2] \end{bmatrix} \end{matrix} \tag{5.2}$$

Theorem 5.3.1

Let G be a consistent and live SDF Graph, and $\mathbb{A}$ be the associated set of actors. If and only if G is consistent, there exists a repetition vector q of size $|\mathbb{A}|$. For any firing k of an actor $a \in \mathbb{A}$, and for any input edge $e_j \in a$, there exists a dependency interval $\delta_{j,k} = \begin{bmatrix} \delta^0_{j,k} & \delta^1_{j,k} \end{bmatrix}$ with $\delta^0_{j,k}$, $\delta^1_{j,k}$ the first and last dependencies of e_j, respectively.

It comes:

$$\delta^0_{j,k} = \left\lfloor \frac{k * c_j - d_j}{p_j} \right\rfloor \tag{5.3a}$$

$$\delta^1_{j,k} = \left\lfloor \frac{c_j * (k+1) - d_j - 1}{p_j} \right\rfloor \tag{5.3b}$$

where c_j, p_j and d_j are the consumption rate, the production rate and the number of initial delays on the edge e_j, respectively.

Proof of Equation (5.3a). Let $b \in \mathbb{A}$ be the actor producing data tokens on input edge e_j of actor a and q_b and q_a be the repetition values of b and a, respectively. If and only if G is consistent, then the sum of all data tokens produced by actor b is equal to the sum of all data tokens consumed by actor a. Equation (5.4) formalizes this property.

$$\sum_{l=0}^{q_a-1}(c_j) = \sum_{i=0}^{q_b-1}(p_j) \tag{5.4}$$

To execute any firing k of actor a, the sum of all the data tokens consumed by firings of actor a up to k, k excluded, must be strictly less to the sum of all the data tokens produced by actor b plus the initial delays of the edge e_j. Formally, for any a_k, $k \in [0; q_a[$, there exists a given positive integer $n \in [0; q_b[$ verifying Equation (5.5).

$$\sum_{l=0}^{k-1}(c_j) < \sum_{i=0}^{n}(p_j) + d_j \tag{5.5}$$

We search the minimal value n_0 of n such that Equation (5.5) holds. In other words, we search the minimal value n_0 for which the sum of all the data tokens produced by actor b and the initial delays is strictly greater than the sum of data tokens consumed by actor a up to, but not and including, its k^{th} firing. Consequently, this means that for $n_0 - 1$, the sum of all data tokens produced by actor b and the initial delays is less or equal to the sum of the data tokens consumed by actor a up to firing k which translates in Equation (5.6).

$$\sum_{l=0}^{k-1}(c_j) \geq \sum_{i=0}^{n_0-1}(p_j) + d_j \tag{5.6}$$

By developing the sums in Equation (5.5) with $n = n_0$ comes:

$$k * c_j < (n_0 + 1) * p_j + d_j \tag{5.7a}$$

$$\frac{k * c_j - d_j}{p_j} < n_0 + 1 \tag{5.7b}$$

Developing Equation (5.6):

$$k * c_j \geq n_0 * p_j + d_j \tag{5.8a}$$

$$\frac{k * c_j - d_j}{p_j} \geq n_0 \tag{5.8b}$$

And using the fact that $\lfloor x \rfloor = m,\ m \in \mathbb{N}$ if and only if $m + 1 > x \geq m$:

$$n_0 = \left\lfloor \frac{k * c_j - d_j}{p_j} \right\rfloor \tag{5.9}$$

■

Proof of Equation (5.3b). Let $b \in \mathbb{A}$ be the actor producing data tokens on input edge e_j of actor a and q_b and q_a be the repetition values of b and a, respectively.

To execute any firing k of actor a, the sum of all the data tokens consumed by firings of actor a up to k, k included, must be less or equal to the sum of all the data tokens produced by actor b and the initial delays of the edge e_j. Formally, for any $a_k,\ k \in [0; q_a[$, there exists a positive integer $m \in [0; q_b[$ verifying Equation (5.10).

$$\sum_{l=0}^{k}(c_j) \leq \sum_{i=0}^{m}(p_j) + d_j \tag{5.10}$$

We search the minimal value m_0 of m such that Equation (5.10) holds. In other words, we search the minimal value m_0 for which the sum of all the data tokens produced by actor b and the initial delays is greater or equal to the sum of data tokens consumed by actor a up to, and including, its k^{th} firing. Consequently, this means that for $m_0 - 1$, the sum of all data tokens produced by actor b and the initial delays is strictly inferior to the sum of the data tokens consumed by actor a up to firing k which translates in Equation (5.11).

$$\sum_{l=0}^{k}(c_j) > \sum_{i=0}^{m_0-1}(p_j) + d_j \tag{5.11}$$

By developing the sums in Equation (5.10) with $m = m_0$ comes:

$$(k+1) * c_j \leq (m_0 + 1) * p_j + d_j \tag{5.12a}$$

$$\frac{(k+1) * c_j - d_j}{p_j} \leq m_0 + 1 \tag{5.12b}$$

Developing Equation (5.11):

$$(k+1) * c_j > m_0 * p_j + d_j \tag{5.13a}$$

$$\frac{(k+1) * c_j - d_j}{p_j} > m_0 \tag{5.13b}$$

And using the fact that $\lceil x \rceil = n,\ n \in \mathbb{N}$ if and only if $n \geq x > n-1$:

$$m_0 + 1 = \left\lceil \frac{(k+1) * c_j - d_j}{p_j} \right\rceil \tag{5.14a}$$

$$m_0 = \left\lceil \frac{(k+1) * c_j - d_j}{p_j} \right\rceil - 1 \tag{5.14b}$$

Finally, since k, c_j and p_j are positive integers, comes:

$$m_0 = \left\lfloor \frac{(k+1) * c_j - d_j - 1}{p_j} \right\rfloor \tag{5.15}$$

■

Having delays on a FIFO may result in negative values for $\delta^0_{j,k}$ and $\delta^1_{j,k}$. If $\delta^0_{j,k}$ or $\delta^1_{j,k}$ is negative, this means that firing k of actor a depends on initialization tokens coming either from previous graph iteration or from a setter actor setting those initial tokens [**arrestier_delays_2018**].

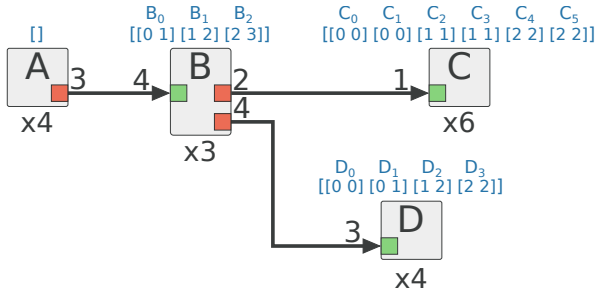


Figure 5.5 – Example of the proposed IR of the graph of Figure 5.3. Each actor in the graph is annotated with its corresponding dependency matrix.

Figure 5.5 shows the proposed IR, which is the results of the computation of all the dependency matrices and the original graph, applied to the graph of Figure 5.3. For each actor a in Figure 5.5, its corresponding dependency matrix Δ_a is annotated above it.

This new augmented graph is sufficient for deriving a complete schedule of the graph of Figure 5.3 an does not require expanding it into the corresponding SR-DAG.

5.3.2 Modeling a Hierarchical and Compositional Dataflow MoC: The πSDF case

Equations (5.3a) and (5.3b) hold in the general case of SDF graphs. However, to take into account the hierarchical specificity of the πSDF and IBSDF MoCs, it is necessary to define additional equations for the behavior of interfaces. In this section, only the interfaces are discussed as the other actors inside a subgraph behave the same way as in a SDFG, meaning that Equations (5.3a) and (5.3b) apply to them. As defined in [**piat_interface-based_2009**; **desnos_pimm:_2013**], input and output interfaces act as a "frontier" between a hierarchical actor and its inner subgraph definition. All data tokens of an input interface must be consumed at least once during an iteration of a subgraph. If more data tokens are consumed, due to repetition values, then the interface behaves like a circular buffer producing the same data tokens as many times as needed. Symmetrically, an output interface only outputs the last data tokens produced by the actor connected to it and discards the rest. Importantly, interfaces have a repetition value of 1.

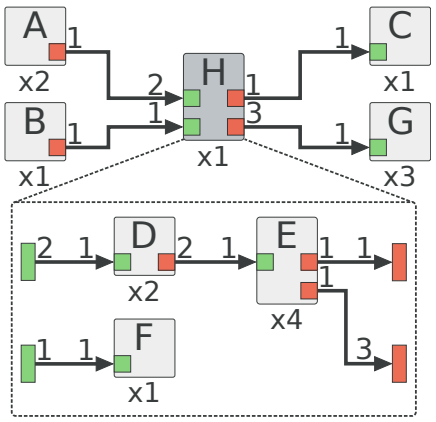


Figure 5.6 – Hierarchical πSDF graph example.

Figure 5.7 illustrates the behavior of the output interface connecting the subgraph H to the actor G from Figure 5.6. In Figure 5.6, the actor E is executed 4 times within the subgraph H, producing 4 data tokens on its output data port connected to the output interface, itself connected to the actor G in the upper-graph. The output interface only

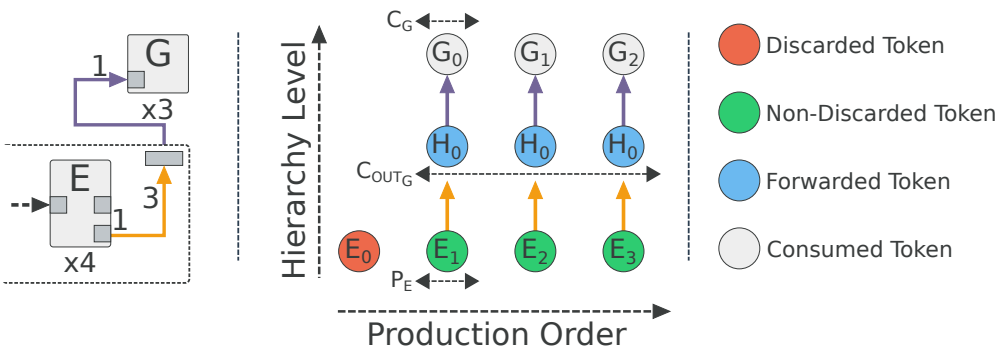


Figure 5.7 – Behavior of the output interface connecting the subgraph H to actor G in Figure 5.6. Tokens are named after the corresponding firing of the actor producing them.

consumes 3 data tokens, meaning that only the last three executions of the actor E are used for this interface, as showed in Figure 5.7 with the first data token produced by actor E being discarded.

Equations (5.16) give the dependency interval definition for an output interface o_{if} of a given subgraph.

$$\delta^0_{o_{if}} = q_P - \left\lceil \frac{c_{o_{if}}}{p_{o_{if}}} \right\rceil \tag{5.16a}$$

$$\delta^1_{o_{if}} = q_P - 1 \tag{5.16b}$$

where q_P is the repetition value of the actor producing data tokens on o_{if}, $c_{o_{if}}$ the consumption rate of the interface o_{if} and $p_{o_{if}}$ the production rate on the interface o_{if}. In Figure 5.7, $p_{o_{if}}$ corresponds to $P_E = 1$, $c_{o_{if}}$ corresponds to $C_{out_G} = 3$ and q_P corresponds to $q_E = 4$. Applying Equations (5.16a) and (5.16b) to Figure 5.7 gives the first dependency on E equal to $\delta^0_{out_G} = 1$ and the last dependency on E equal to $\delta^1_{out_G} = 3$. Note that delays on FIFOs connected to output interfaces do not impact Equations (5.16a) and (5.16b) due to the behavior of the interface to only output the last data tokens produced on it. Dependencies on output interfaces also give to the scheduling algorithm the earliest time at which a hierarchical actor can be considered to have finished its internal execution.

Equation (5.16b) comes directly from the definition of the output interfaces [**piat_interface-based_20**
If the interface only outputs the last data tokens produced on it, then the last dependency of the interface is necessarily the last firing of the actor producing data tokens on it. Equation (5.16a) is derived using a similar development to the one of Equation (5.3a). The aim is to find the minimum number of firings N of the actor producing data tokens

on output interface o_{if} such as:

$$\sum_{i=1}^{N} p_{o_{if}} \geq c_{o_{if}} \tag{5.17a}$$

$$\sum_{i=1}^{N-1} p_{o_{if}} < c_{o_{if}} \tag{5.17b}$$

Using the developments of Equation (5.3a), it comes:

$$N = \left\lceil \frac{c_{o_{if}}}{p_{o_{if}}} \right\rceil \tag{5.18}$$

The first dependency of the output interface is then defined by:

$$\delta^0_{o_{if}} = q_P - N \tag{5.19a}$$

$$\delta^0_{o_{if}} = q_P - \left\lceil \frac{c_{o_{if}}}{p_{o_{if}}} \right\rceil \tag{5.19b}$$

which corresponds to Equation (5.16a).

Input interfaces inherit the dependencies of the hierarchical actors to which they belong. This comes directly from the definition of input interfaces that state that input interfaces can start executing as soon as the hierarchical actor is ready to fire in its parent graph. Therefore, actors connected to input interfaces can start their execution as soon as the subgraph starts and the only dependency to check is related to the presence of delays.

5.3.3 Relaxed execution model for πSDF

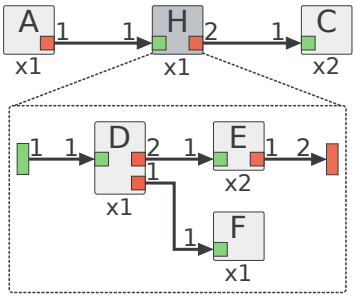


Figure 5.8 – A hierarchical graph example used for illustrating the relaxed execution model of the πSDF MoC.

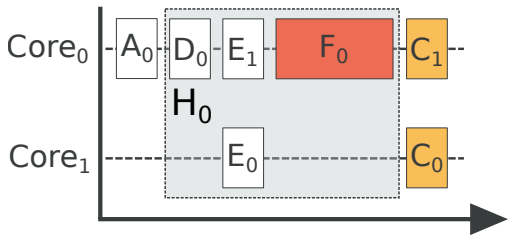


Figure 5.9 – A possible schedule of the graph of Figure 5.8 following the strict execution rules of the πSDF MoC. Firings of actor C can only start after the end of the complete firing of the hierarchical actor H.

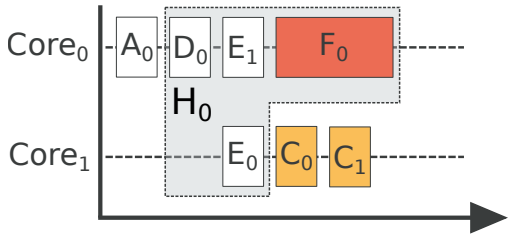


Figure 5.10 – A possible schedule of the graph of Figure 5.8 with relaxed execution rules of the πSDF MoC. Firings of actor C can start in parallel of the firing of the hierarchical actor H, as the data dependencies are satisfied.

In [**deroui_relaxed_2017**], a relaxed model of execution is used on the IBSDF MoC to maximize the throughput of an application containing multiple levels of hierarchy. The relaxed execution model allows for actors contained inside an IBSDF subgraph to start their execution without having to wait for data tokens on all interfaces of their containing hierarchical actor. The same relaxed execution model is applied to the πSDF MoC. For example, Figures 5.9 and 5.10 illustrates two potential schedule of the graph of Figure 5.8 following the strict execution rules of the πSDF MoC and with relaxed execution rules, respectively. In the graph of Figure 5.8, actor C is connected to the output interface of the hierarchical actor H which itself is connected to the actor E. In the πSDF MoC, hierarchical actors are seen as atomic, hence their firing rules follow the one of non hierarchical actors. In other words, a hierarchical actor fires only when sufficient data tokens are available on all of its input data ports and, importantly, its execution is considered as atomic from the perspective of the graph in which it belongs.

Therefore, in the graph of Figure 5.8, actor C is not aware of the internal specification of the actor H and only fires when the complete execution of the subgraph of H is finished, which results in the possible schedule of Figure 5.9. However, as noted in [**deroui_relaxed_2017**], the synchronization enforced by the data input and out-

put interfaces of hierarchical actors can be relaxed as long as the data dependencies are respected. In the context of the graph of Figure 5.8, actor C actually depends on the actor E inside the subgraph of actor H and thus, it is possible to relax the firing rule of C to depend on actor E instead of actor H, which results in the possible schedule of Figure 5.10.

Taking into account the relaxed constraint in the numerical model of the SR-DAG adds some complexity to the previously proposed equations. We will first investigate the case of the output interfaces. Relaxing the execution model of the πSDF leads to extend the dependency resolution problem of an actor depending on a hierarchical actor from its level of hierarchy to the subgraph level. For example, in the graph of Figure 5.6 dependencies of actor C are now 2-dimensional. Indeed, actor C depends on executions of actor H and for each firing of actor H, depends on executions of actor E. The objective is thus to combine Equations (5.3a) and (5.3b) to Equations (5.16a) and (5.16b), respectively.

Let $\delta^N_{a|j,k}$ be a sub-matrix of size 1×2, with N the total number of levels of hierarchy the firing k of an actor a depends on for its input edge e_j. $\delta^N_{a|j,k}$ is a generalization of the matrix $\delta_{j,k}$ introduced in Section 5.3.1. Equation (5.20) gives the general definition of $\delta^N_{a|j,k}$.

$$\delta^N_{a|j,k} = \left[\left[\delta^{0,0}_{a|j,k} \quad \cdots \quad \delta^{0,N-1}_{a|j,k}\right] \quad \left[\delta^{1,0}_{a|j,k} \quad \cdots \quad \delta^{1,N-1}_{a|j,k}\right]\right] \tag{5.20}$$

Similarly to the definition of $\delta_{j,k}$ given in Section 5.3.1, $\delta^N_{a|j,k}$ represents the interval of dependencies of firing k of actor a. The main difference is that $\delta^0_{j,k}$ and $\delta^1_{j,k}$ are now defined as sub-matrices of size $1 \times N$. $\delta^{0,n}_{j,k}$ is the first dependency of the k^{th} firing of actor a at level n of hierarchy. Similarly, $\delta^{1,n}_{j,k}$ is the last dependency of a_k at the level n of hierarchy. The generalized definitions of $\delta^{0,n}_{j,k}$ and $\delta^{1,n}_{j,k}$ are given in Equations (5.21a) and (5.21b), respectively.

$$\delta^{0,n}_{a|j,k} = \begin{cases} \delta^0_{a|j,k}, & n = 0, \text{ see Equation (5.3a)} \\ q_{p_n} - \left\lceil \frac{C^{0,n}_{a|j,k}}{P_n} \right\rceil, & n \in [1; N[\end{cases} \tag{5.21a}$$

$$\delta^{1,n}_{a|j,k} = \begin{cases} \delta^1_{a|j,k}, & n = 0, \text{ see Equation (5.3b)} \\ q_{p_n} - \left\lceil \frac{C^{1,n}_{a|j,k}}{P_n} \right\rceil, & n \in [1; N[\end{cases} \tag{5.21b}$$

where:

— q_{p_n}, the repetition value of the actor producing data tokens on the output interface at level n of hierarchy.

— P_n, the production rate on the output interface at level n of hierarchy.
— $C^{0,n}_{a|j,k}$, the updated consumption rate of the output interface at level n of hierarchy for the first dependency.
— $C^{1,n}_{a|j,k}$, the updated consumption rate of the output interface at level n of hierarchy for the last dependency.

$C^{0,n}_{a|j,k}$ and $C^{1,n}_{a|j,k}$ correspond to the updated consumption rates of the output interface at the level n of hierarchy for the first dependency and the last dependency, respectively. For each level n of hierarchy, the updated consumption rate of the corresponding output interface depends on the one of the level $n-1$, up to the consumption rate of the edge e_j of actor a at the top level of hierarchy. The definitions of $C^{0,n}_{a|j,k}$ and $C^{1,n}_{a|j,k}$ are given by Equations (5.22a) and (5.22b), respectively.

$$C^{0,n}_{a|j,k} = \begin{cases} (\delta^{0}_{a|j,k} + 1) * p_j + d_j - k * c_j, & n = 1 \\ C^{0,n-1}_{a|j,k} - (q_{p_{n-1}} - (\delta^{0,n-1}_{a|j,k} + 1)) * P_{n-1}, & n \in [2; N[\end{cases} \tag{5.22a}$$

$$C^{1,n}_{a|j,k} = \begin{cases} (\delta^{1}_{a|j,k} + 1) * p_j + d_j - (k+1) * c_j + 1, & n = 1 \\ C^{1,n-1}_{a|j,k} - (q_{p_{n-1}} - (\delta^{1,n-1}_{a|j,k} + 1)) * P_{n-1}, & n \in [2; N[\end{cases} \tag{5.22b}$$

where:

— c_j, the consumption rate of edge e_j of actor a.
— p_j, the production rate of edge e_j.
— d_j, the initial delay of edge e_j.
— k, the firing of actor a for which dependencies are computed.

We only provide the proof for Equations 5.22a, as the Equations 5.22b is derived in a similar way.

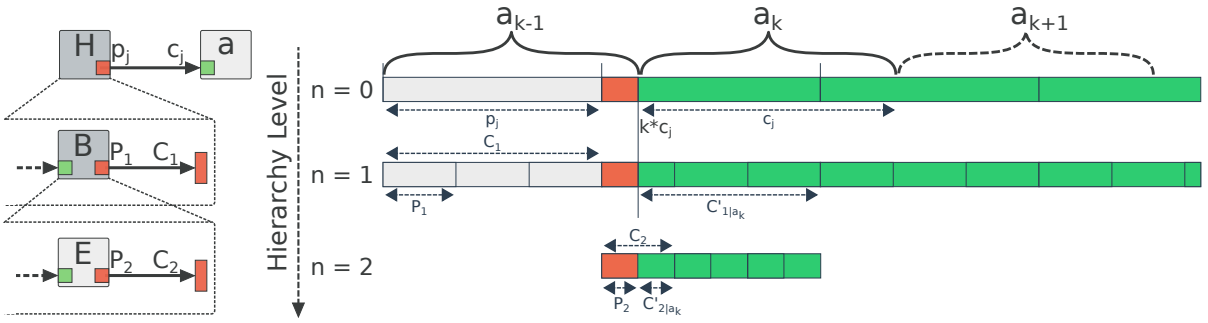


Figure 5.11 – Illustration of $C'_{1|a_k}$, delays are omitted.

Proof of Equation (5.22a). Let a be an actor connected to a hierarchical actor H with N levels of hierarchy through an edge e_j in a graph G. The edge e_j connecting H to a has a production rate p_j, a consumption rate c_j and an initial delay d_j. We will prove by induction that, for any firing k of actor a and for all $n \in \mathbb{Z}$,

$$\delta^{0,n}_{a|j,k} = q_{p_n} - \left\lceil \frac{C^{0,n}_{a|j,k}}{P_n} \right\rceil$$

where:

$$C^{0,n}_{a|j,k} = C^{0,n-1}_{a|j,k} - (q_{p_{n-1}} - (\delta^{0,n-1}_{a|j,k} + 1)) * P_{n-1}$$

and with:

$$C^{0,1}_{a|j,k} = (\delta^{0}_{a|j,k} + 1) * p_j + d_j - k * c_j,\ \delta^{0}_{a|j,k} = \left\lfloor \frac{k * c_j - d_j}{p_j} \right\rfloor$$

Base case: For $n = 1$, Equation (5.21a) gives:

$$\delta^{0,1}_{a|j,k} = q_{p_1} - \left\lceil \frac{(\delta^{0,0}_{a|j,k} + 1) * p_j + d_j - k * c_j}{P_1} \right\rceil \tag{5.23}$$

And from Equation 5.16a we know that:

$$\delta^{0}_{o_{if}} = q_{p_1} - \left\lceil \frac{C_1}{P_1} \right\rceil \tag{5.24}$$

with C_1 the consumption rate of the output interface of H corresponding to the edge e_j, q_{p1} the repetition value of the actor producing data tokens on the interface and P_1 the production rate onto the interface. Let $C'_{1|a_k}$ be the expression of C_1 as a function of k, c_j, d_j and p_j. In other words, $C'_{1|a_k}$ is the transposition of the consumption rate of a_k on H into the inner subgraph of H for the first level of hierarchy. The first dependency of a_k onto H for the edge e_j is given by Equation (5.3a):

$$\delta^{0}_{a|j,k} = \left\lfloor \frac{k * c_j - d_j}{p_j} \right\rfloor$$

$C'_{1|a_k}$ is the sum of data tokens produced by H up to its firing $\delta^{0}_{a|j,k}$ and the initial tokens d_j of the FIFO $\vec{Ha}$ to which the sum of data tokens consumed by actor a up to its k^{th} firing is subtracted. Figure 5.11 illustrate the transposition mechanism and Equations 5.25 formalize its definition.

$$C'_{1,a_k} = \sum_{i=0}^{\delta^0_{a|j,k}} p_j + d_j - k * c_j \tag{5.25a}$$

$$C'_{1,a_k} = (\delta^0_{a|j,k} + 1) * p_j + d_j - k * c_j \tag{5.25b}$$

Inserting $C'_{1|a_k}$ into Equations 5.24 gives Equation 5.26:

$$\delta^0_{o_{if}|a_k} = q_{p_1} - \left\lceil \frac{C'_{1|a_k}}{P_1} \right\rceil \tag{5.26a}$$

$$\delta^0_{o_{if}|a_k} = q_{p_1} - \left\lceil \frac{(\delta^0_{a|j,k} + 1) * p_j + d_j - k * c_j}{P_1} \right\rceil \tag{5.26b}$$

Equation (5.26) is equal to Equation (5.23), thus Equation (5.21a) is true for $n = 1$

Induction step: For $n = 2$, the dependency definition of the interface is given by:

$$\delta^{0,2}_{o_{if}} = q_{p_2} - \left\lceil \frac{C_2}{P_2} \right\rceil$$

with $\delta^{0,2}_{o_{if}}$ the dependency of the output interface in level 2 of hierarchy using Equation (5.16a). Then following the same substitution scheme as in the base case but with the transposition of $C'_{2|a_k}$ into C_2:

$$\begin{aligned} C'_{2,a_k} &= \sum_{i=0}^{\delta^0_{a|j,k}} p_j + d_j - k * c_j - ((q_{p_1} - 1) - \delta^{0,1}_{a|j,k}) * P_1 \\ C'_{2,a_k} &= (\delta^0_{a|j,k} + 1) * p_j + d_j - k * c_j - ((q_{p_1} - 1) - \delta^{0,1}_{a|j,k}) * P_1 \\ C'_{2,a_k} &= C'_{1,a_k} - ((q_{p_1} - 1) - \delta^{0,1}_{a|j,k}) * P_1 \end{aligned}$$

which gives:

$$\delta^{0,2}_{a|j,k} = q_{p_2} - \left\lceil \frac{C'_{2,a_k}}{P_2} \right\rceil$$

Generalizing to n:

$$\delta^{0,n}_{o_{if}} = q_{p_n} - \left\lceil \frac{C_n}{P_n} \right\rceil$$

with $\delta_{oif}^{0,n}$ the dependency of the output interface in level n of hierarchy using Equation (5.16a). Then following the same substitution scheme as before:

$$C'_{n,a_k} = C'_{n-1,a_k} - ((q_{p_{n-1}} - 1) - \delta_{a|j,k}^{0,n-1}) * P_{n-1}$$

Therefore, for $n \in \mathbb{Z}$:

$$\delta_{oif}^{0,n} = q_{p_n} - \left\lceil \frac{C_{a|j,k}^{0,n}}{P_n} \right\rceil$$

with:

$$C_{a|j,k}^{0,n} = C'_{n-1,a_k} - ((q_{p_{n-1}} - 1) - \delta_{a|j,k}^{0,n-1}) * P_{n-1}$$

and:

$$C_{a|j,k}^{0,1} = (\delta_{a|j,k}^{0} + 1) * p_j + d_j - k * c_j$$

■

The rest of this section provides the concepts used to derive the Equations 5.22a and 5.22b. A multi-level hierarchical πSDF graph is presented in Figure 5.12, and Figure 5.13 shows the corresponding data tokens dependency analysis. Figure 5.13 shows the direct data dependencies across the different levels of hierarchy.

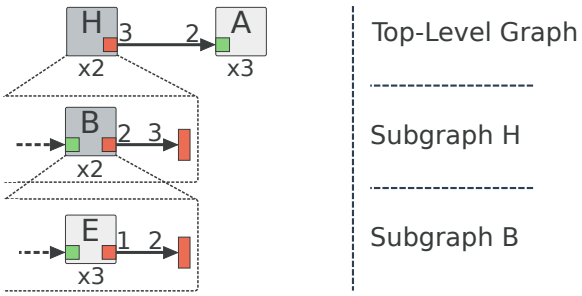


Figure 5.12 – Multi-Level Hierarchical πSDF graph example.

In Figure 5.13, the first data token consumed by A_1 is produced by E_2 during the second firing of the subgraph B (B_1), in the first firing of the subgraph H (H_0). Examples of relaxed and non-relaxed dependencies are given in Figure 5.13 for A_0. With non-relaxed dependencies, A_0 depends only on H_0, then the output interface of H depends on B_1 and finally the output interface of B depends on E_2. In other words, with non-relax dependencies, A_0 has to wait for the complete execution of the second firing of the subgraph B and the corresponding firings of actor E before it can be fired. With relaxed dependencies, A_0

depends directly on E_2, from B_0 and H_0, and the dependencies due to the interfaces of the different levels of hierarchy are omitted.

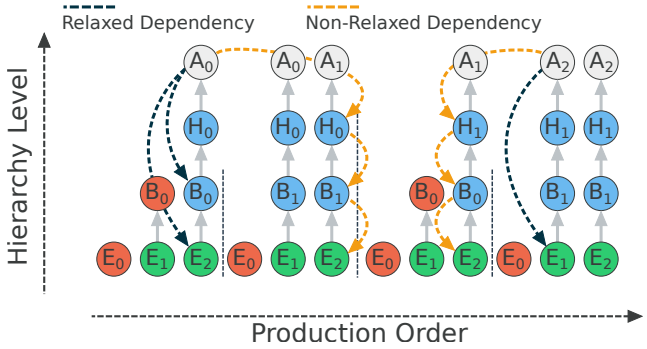


Figure 5.13 – Dependency analysis of the graph of Figure 5.12. The graphical formalism is the same as in Figure 5.7

By analyzing the distribution of the different data tokens and how the consumption rate of the output interfaces is influenced across the different levels of hierarchy, it comes a direct relationship linking the level n to the level $n-1$ that is expressed in the Equations (5.22a) and (5.22b) with the terms $C_{a|j,k}^{0,n-1}$ and $C_{a|j,k}^{1,n-1}$, respectively. The subtraction term of the Equations (5.22a) and (5.22b) corresponds to the offset that should be applied in order to have the actual consumption rate of the output interface of the next level of hierarchy. This subtraction comes from the inverse behavior of the output interfaces. For instance, in Figure 5.13, the real consumption rate of A_0 on the output interface of B_0 is equal to 1 and not 3. Similarly, the consumption rate of A_2 on the output interface of H_1 is 2 instead of 3 which will make A_2 dependent on B_1 and not B_0.

In Section 5.3.1,the definition of an interval was introduced to be sufficient to define the full dependencies of an actor for a given input edge due to the fact that there can be no discontinuity in the dependencies. In other words, if an actor A depends on an actor B with the following interval $[B_0\ B_2]$, then actor A must also depends on B_1. This property is also applicable to the hierarchy case. This means that if an actor A depends on the following dependency interval $[[G_0\ H_0]\ [G_1\ H_1]]$, it must depend on all firings of G and H that fall in between except for the discarded firing of actors due to the behavior of the interfaces. Using Equations (5.20) , (5.21a) and (5.21b) on the example graph of

Figure 5.12, the following dependency intervals for actors A is derived.

$$\delta^0_{A|0,1} = [0\ 1] \tag{5.28a}$$
$$\delta^1_{A|0,1} = [[0\ 1]\ [1\ 0]] \tag{5.28b}$$
$$\delta^2_{A|0,1} = [[0\ 1\ 2]\ [1\ 0\ 2]] \tag{5.28c}$$

Equation (5.28a) corresponds to the dependency interval of A_1 at the top level of hierarchy, Equation (5.28b) corresponds to the dependency interval of A_1 in the subgraph H and Equation (5.28c) corresponds to the fully relaxed dependency interval of A_1. Equation (5.28a) shows that A_1 depends on H_0 to H_1 and Equation (5.28b) shows that A_1 depends on B_1 from H_0 to B_0 from H_1. Finally, Equation (5.28c) shows that A_1 depends on E_2 from $[H_0\ B_1]$ to E_2 from $[H_1\ B_0]$. It is possible to individually tune the number of hierarchy levels for which the execution of an actor is relaxed which gives flexibility to the scheduling algorithm. In our work, only the cases of no-relaxation and full-relaxation are considered.

It is important to note that for dynamic applications with parameter changes in a hierarchical actor H, it is necessary to store the different values of the parameters of each instance of H as it may influence the repetition vector in a particular instance of H and change the dependencies for actors depending on H. This constraint is not necessary for the non-relaxed execution model as the subgraph is hidden from any actor depending on H.

For the case of input interfaces, as stated in Section 5.3.2 no special equations have to be derived. Actors depending on interfaces directly inherit dependencies of the corresponding input edge of the containing hierarchical actor. This inheritance goes up to the top level of hierarchy. However one particular case has to be considered, the case of an actor consuming more data tokens on an input interface that the interface produces. Since, interfaces have a repetition value strictly equal to 1, a special actor, called a *duplicate* actor, is introduced. A duplicate actor has one input port and one output port and duplicates the tokens received on its input port as many times as needed to respect the consistency property. Duplicate actors are automatically inserted during graph analysis.

5.4 Exploitation: Scheduling and Memory Allocation

The proposed numerical approach is compatible with dataflow MoCs derived from the SDF MoC and can be used to derive a schedule the same way a DAG would. Indeed, it is possible to build an API that emulates accesses to an SR-DAG using the proposed numerical model. From the user point of view, the emulated SR-DAG behaves as a standard SR-DAG, the only difference being that dependencies are computed on-the-fly instead of having a pre-built graph. Therefore, any resource allocation algorithm that uses an SR-DAG can be based on the proposed numerical model instead.

The main advantage of our proposed method is to remove the costly step of building and storing the SR-DAG. However, using our method may result in an increase of the complexity of the original resource allocation algorithm, compared to using the SR-DAG, due to the computation of the dependencies done on-the-fly.

In the experiments of the following sections, to demonstrate the capacity of our model to be used in a real resource allocation algorithm and evaluate the performance gain over the SR-DAG representation, a naive *greedy scheduling* algorithm is used. The performance of the greedy algorithm is not the focus of this contribution. The chosen greedy algorithm is described in Algorithm 3.

Algorithm 3: Greedy Scheduling Algorithm

Input : Dataflow graph: G
Processing Elements (PEs) list: *pe_list*

```
1 list = create_list(G); // Create a list with all actors of the graph.
2 while list is not empty do
3     for a ∈ list do // Find an actor a in the list that can be scheduled.
4         if is_schedulable(a) then
5             if is_mappable_onto_pe(a, pe_list) then
6                 map_onto_pe(a, pe_list); // Map actor a onto an available PE.
7                 remove_from_list(a, list); // Remove actor a from the list.
8             else
9                 display_error();
```

The main difference between the SR-DAG-based greedy scheduler and the numerical one comes from the input graph representation used. Using the SR-DAG, the SR-DAG

itself is used and the greedy scheduler directly goes through the actor list of the SR-DAG to find the first actor that can be scheduled according to the DAG dependencies. Using the numerical model, the original πSDF representation of the application is used and the greedy scheduler goes through the πSDF actor list, then for each actor it computes the dependencies of the actor on-the-fly and for the current firing of the actor checks if it can be scheduled.

5.4.1 Experimental Setup

Table 5.1 – Experimental platform characteristics

Platform	Processor	Cores	RAM	GCC
Laptop	Intel Core i7-7820HQ	4	32GB DDR4	7.3.0
Jetson TX2	ARM Cortex-A57 + NVIDIA Denver 2	4 + 2	8GB LPDDR4	5.4.0
ODROID-XU3	Samsung Exynos 5422	4 + 4	2GB LPDDR3	4.9.2

The different experiments are conducted on 3 different platforms ranging from an x86 laptop with medium processor to a very low power ODROID-XU3 platform. The characteristics of these platforms are summarized in Table 5.1. The Spider library was compiled with `O3` level of optimizations on all platforms. Four applications from the official repository [1] of the Preesm tool [**pelcat_preesm:_2014**] have been used to conduct the experiments. These applications are state-of-the-art AI, and computer vision applications. The SqueezeNet application is an implementation of a pre-trained convolution neural network based on the SqueezeNet architecture [**iandola_squeezenet_2016**]. The Reinforcement Learning application corresponds to the implementation of the CACLA reinforcement learning algorithm [**van_hasselt_reinforcement_2007**] applied to the case of lifting-up and stabilizing an inverse pendulum. The Stabilization application is a video stabilization application which applies motion compensation on an input video and produces a stabilized output video. Finally, the Sobel-Morpho application is an edge detection application which applies the Sobel operator followed by the morphological dilation and erosion operations on the luminance channel of an input video. The four applications feature different levels of hierarchy and task and data parallelism, as summarized in Table 5.2 where $|G_{\pi SDF}|$ corresponds to the number of actors in the πSDF representation,

1. https://github.com/preesm/preesm-apps

N_{levels} corresponds to the number of hierarchical levels and $|G_{SR\text{-}DAG}|$ corresponds to the number of actors in the SR-DAG representation of the application. The number of edges for both MoCs is noted in the N_{Edges} column of the corresponding MoC.

In the applications used for our experimentation, all parameter values are changing at each graph iteration, thus triggering a complete rescheduling of the application. Although unrealistic, this behavior was forced, even in case of static parameter values, in order to emphasize the most dynamic, and thus the most complex scenario for the runtime allocation of resources. In the case of a more static behavior, both the DAG-Based and numerical model-based solutions can benefit from optimizations to retain information between successive graph iterations, which is out of the scope of this work.

Table 5.2 – Applications description

Application	πSDF			SR-DAG	
	$\|G_{\pi SDF}\|$	N_{Edges}	N_{levels}	$\|G_{SR\text{-}DAG}\|$	N_{Edges}
SqueezeNet	108	272	2	5436	17248
Reinforcement Learning	188	459	3	417	1114
Stabilization	20	41	2	101	325
Sobel-Morpho	6	7	0	65	85

5.4.2 Results

In this section, the different experimental results obtained for our implementation of the presented numerical model into the Spider tool are presented and compared to the reference implementation that uses an SR-DAG model. Two configurations of the proposed numerical model are compared to the reference implementation. The first configuration is referred to the *relaxed* configuration and corresponds to the use of the relaxed execution model presented in Section 5.3.3, and the second configuration is referred to the *standard* configuration and corresponds to the non-relaxed execution model of the πSDF MoC. In these experiments, the metrics used for comparing the different configurations are the computation time and the memory footprint of the runtime manager performing the scheduling and mapping of a πSDF application onto multi-cores processor platforms. The scheduling algorithm used in these experiments is the greedy scheduling algorithm described in Algorithm 3. Despite being a rather simple algorithm, this schedul-

ing algorithm allows us to rapidly demonstrate the feasibility of our proposed models. Nevertheless, the results show that using the direct numerical model gives overall great improvements both in terms of computational complexity and memory footprint. The measured application performance may be further optimized with a smarter scheduling algorithm [**kwok_high-performance_1997**], which would reduce the scheduling time of all experiments, but would not change the memory nor the construction time overhead of the SR-DAG based runtime.

5.4.2.1 Memory footprint

In this section, the memory footprint of the different representations for the scheduling and mapping is presented. No differentiation is made between the *relaxed* and *standard* configurations of the numerical model as both configurations share the exact same memory footprint.

Table 5.3 shows the total memory footprint of Spider during the scheduling and mapping of the applications. The gains expressed in Table 5.3 represent, as a percentage, the amount of memory saved with the numerical model compared to the reference SR-DAG implementation. Results show significant memory reduction with up to 98.63% of memory reduction for the reinforcement-learning application and an average memory reduction of 97.33%. This high memory reduction is due to the implementation of the proposed IR. Indeed, in the proposed implementation, only two values per πSDF actor are stored: the repetition value of the actor and the current scheduled firing of the actor. All dependencies needed by the mapping and scheduling algorithm, which correspond to the Line 4 of Algorithm 3, are computed on-the-fly when needed.

Table 5.3 – Memory footprint of the representations

Application	Reference (SR-DAG)	Numerical Model	Gain (%)
SqueezeNet	8405.9 KB	**515.3 KB**	93, 87
Reinforcement Learning	5183.7 KB	**70.9 KB**	98.63
Stabilization	782.8 KB	**11.8 KB**	98.49
Sobel-Morpho	404.5 KB	**6.8 KB**	98.32
Average	3694,2 KB	151.2 KB	97.33

In addition to the memory needed for the different representations, there is memory used to store information about the schedule execution. The memory used for the schedule execution is similar in both the SR-DAG and the numerical model and is comprised in the values of Table 5.3. In other words, there exists an upper bound to the potential memory footprint reduction that depends on the memory used for the schedule execution. Figure 5.14 shows the relative memory footprint of the numerical model and the SR-DAG representation over the total memory footprints of Table 5.3, hence highlighting the relative memory footprint of the schedule execution information. Values of Figure 5.14 show that actual memory used by the numerical models to perform scheduling and mapping only account for 0.93% to 11.15% of the total memory footprint of Spider whereas in the case of the SR-DAG representation, actual memory used for the scheduling and mapping is greater than 92% of the total memory footprint. Hence, Figure 5.14 emphasizes the low memory footprint overhead of the proposed approach on the runtime over the reference SR-DAG representation.

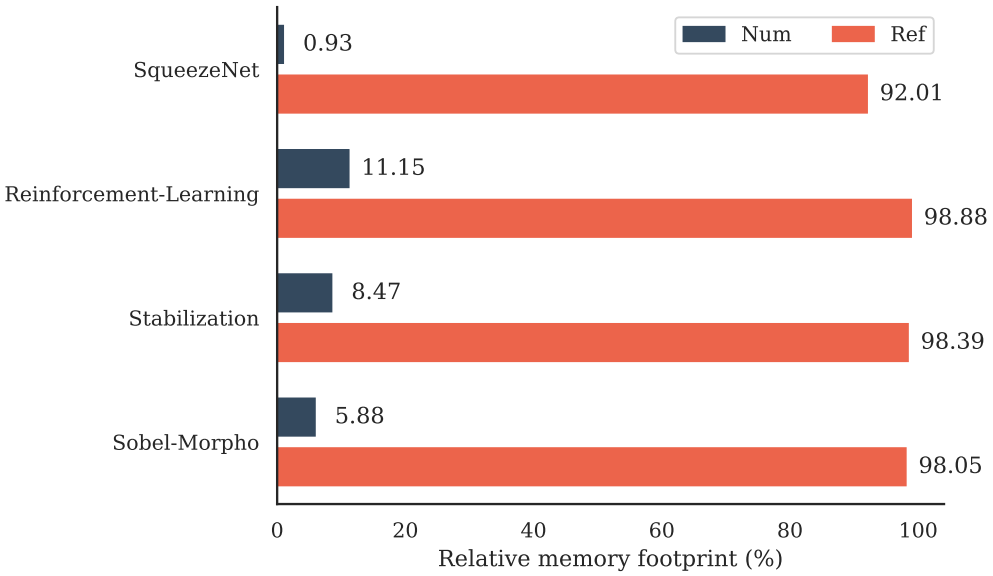


Figure 5.14 – Relative memory footprint of representations over total memory footprint (lower is better). The Num configuration corresponds to the proposed IR and the Ref configuration corresponds to the SR-DAG IR. A low value indicates a low memory footprint of the given IR over the runtime memory usage. The SR-DAG IR is responsible for almost the entire runtime memory footprint (with up to 98.8%) as opposed to the light weight proposed representation.

5.4.2.2 Execution time

In this section, the execution times of the different configurations of the numerical model (*relaxed* and *standard*) are compared to the reference implementation based on the SR-DAG representation. Then, a comparison of the schedule latency, i.e execution time for one graph iteration, for the two configurations of the numerical model is performed, highlighting a potential trade-off between execution time and schedule latency.

Table 5.4 – Intermediate Representation building time in ms for the three tested platforms (lower is better). The values correspond to the time needed by the runtime to build the complete intermediate representation needed for the scheduling and mapping operations.

	Laptop			Jetson TX2			ODROID-XU3		
Application	Ref	Num	Gain(%)	Ref	Num	Gain(%)	Ref	Num	Gain(%)
SqueezeNet	7.105	**0.221**	96.89	39.43	**0.664**	98.32	79.77	**1.90**	97.62
Reinforcement Learning	0.868	**0.180**	79.26	6.03	**0.551**	90.86	12.41	**1.71**	86.22
Stabilization	0.138	**0.017**	87.68	0.67	**0.059**	91.19	1.70	**0.19**	88.82
Sobel-Morpho	0.061	**0.005**	91.80	0.23	**0.017**	92.61	0.69	**0.06**	91.30

Table 5.4 presents the execution times taken by the construction phase of the intermediate representations. In the case of the SR-DAG (Ref column), this time corresponds to the construction of the SR-DAG and the initialization of the schedule execution information. In the case of the proposed numerical model (Num column), the value of Table 5.4 corresponds to the initialization of the schedule execution information and the allocation of the arrays used to store firing information during the scheduling and mapping phase. Note that the construction phase is shared for both *relaxed* and *standard* configurations, thus no difference is made between them in Table 5.4. The Gain column corresponds to the relative reduction time of the proposed approach over the SR-DAG approach. On all three platforms, building the numerical model is significantly faster than building the SR-DAG representation, with a maximum speedup factor of 59.52 for the SqueezeNet application on the Jetson TX2.

Table 5.5 shows the resource allocation execution times for the three compared configurations. In Table 5.5, *Num-R* and *Num-S* refer to the *relaxed* and the *standard* configurations of the numerical model, respectively. The results show significantly lower scheduling times for the *standard* configuration over the two others. This is explained by the hierarchical nature of the *standard* execution model and the greedy scheduling algorithm used.

Table 5.5 – Resource allocation execution time in ms of the different configurations. The values in the table corresponds to the time needed by the runtime scheduler to perform the scheduling and mapping of an application onto a given platform. There is a significant gain using the Num-S configuration on all platform. On the other hand, the Num-R representation is most of the time the slowest configuration.

	Laptop			Jetson TX2			ODROID-XU3		
Application	Ref	Num-R	Num-S	Ref	Num-R	Num-S	Ref	Num-R	Num-S
SqueezeNet	2.491	4.856	**1.043**	22.51	14.50	**2.76**	23.10	23.19	**4.49**
Reinforcement Learning	**0.105**	0.327	0.120	0.50	0.91	**0.35**	0.81	1.47	**0.71**
Stabilization	0.020	0.055	**0.019**	0.08	0.13	**0.06**	0.18	0.24	**0.12**
Sobel-Morpho	0.012	0.010	**0.009**	0.05	0.03	**0.03**	0.11	0.06	**0.05**

Indeed, the greedy algorithm iterates over the actors of a graph until it finds an actor that can be scheduled. In the numerical model configurations, the algorithm is thus much faster, as it iterates over the πSDF graph which contains fewer actors than the SR-DAG one (see Table 5.2). Moreover, in the *standard* configuration, actors located in nested levels of hierarchy are not tested until the corresponding hierarchical actor can be scheduled reducing furthermore the number of tested actors per iteration of the greedy algorithm.

Interestingly, Table 5.5 shows that the *relaxed* configuration has overall higher resource allocation times than the reference configuration. Contrary to the *standard* configuration, in the case of relaxed execution, every actor of the πSDF is tested per iteration of the greedy algorithm. Moreover, the complexity of fetching the dependencies of an actor located in a deep level of hierarchy is significantly higher than when dealing with same level of hierarchy dependencies. This effect is particularly visible with the SqueezeNet application which possesses a high number of dependencies between actors belonging to separate subgraphs. However, the case of the relaxed execution could be improved in future implementations by storing hierarchical dependencies, thus avoiding their re-computation at the cost of an increased memory footprint. Another way of improving the relaxed execution model would be to perform graph analysis before the first graph iteration to simplify the πSDF hierarchy whenever it is possible.

Finally, Table 5.6 gives the relative difference of the obtained schedule latency when scheduling with the numerical models compared to the reference implementation. A value of 0% means that the obtained schedule latency is equal to the one of the reference. Small relative differences in latency (inferior to 5%) are explained by two factors. Firstly, in the SR-DAG representation, Fork and Join actors are explicitly scheduled due to the fact that

Table 5.6 – Relative change in schedule latency (%) for the different configurations. Values $> 0\%$ are increase in the schedule latency and values $< 0\%$ are decrease in the schedule latency.

Application	Laptop		Jetson TX2		ODROID-XU3	
	Num-R	Num-S	Num-R	Num-S	Num-R	Num-S
SqueezeNet	**0.11**	0.22	**-0.05**	0.22	**0.07**	4.30
Reinforcement Learning	**1.36**	8.19	**3.10**	22.25	**1.61**	25.71
Stabilization	5.45	5.45	4.55	4.55	9.20	9.20
Sobel-Morpho	−2.23	−2.23	4.86	4.86	0.00	0.00
Average	**1.17**	2.91	**3.12**	7.97	**2.72**	9.80

they are part of the resulting graph whereas they are not in the numerical representation. Secondly, Spider performs several passes of optimizations on the SR-DAG to reduce the number of special actors (Fork, Join, Broadcast and Roundbuffer actors) introduced during the transformation. However, optimizations do not necessarily remove all special actors introduced during the transformation. Importantly, optimizations passes may also remove special actors that are part of the original πSDF graph which can further improve the obtained schedule latency which is not the case for the numerical representation where no optimizations are performed on the πSDF graph.

Table 5.6 shows no clear improvement of the schedule latency of the relaxed execution model over the standard one on 3 out of the 4 tested applications. For the Sobel-Morpho application, this is explained by the absence of hierarchy, thus there is no need for relaxation. In the case of the Stabilization application, the obtained latency is limited by the topology of the graph itself with synchronization points that can not be reduced. However, in the case of the Reinforcement-Learning application, a significant gain with a difference up to 24.1 percentage points can be achieved using the relaxed execution model at the cost of higher scheduling time.

Figure 5.15 shows the relative total execution time for the three configurations and for the three different platforms. The total execution time is the sum of the intermediate representation building time (Table 5.4) and the scheduling time (Table 5.5). The relative total execution time is the relative difference of the total execution time of the numerical representations with the total execution time of the reference. Figure 5.15 shows that even with higher scheduling time for the relaxed configuration, a minimum reduction of 47.11%

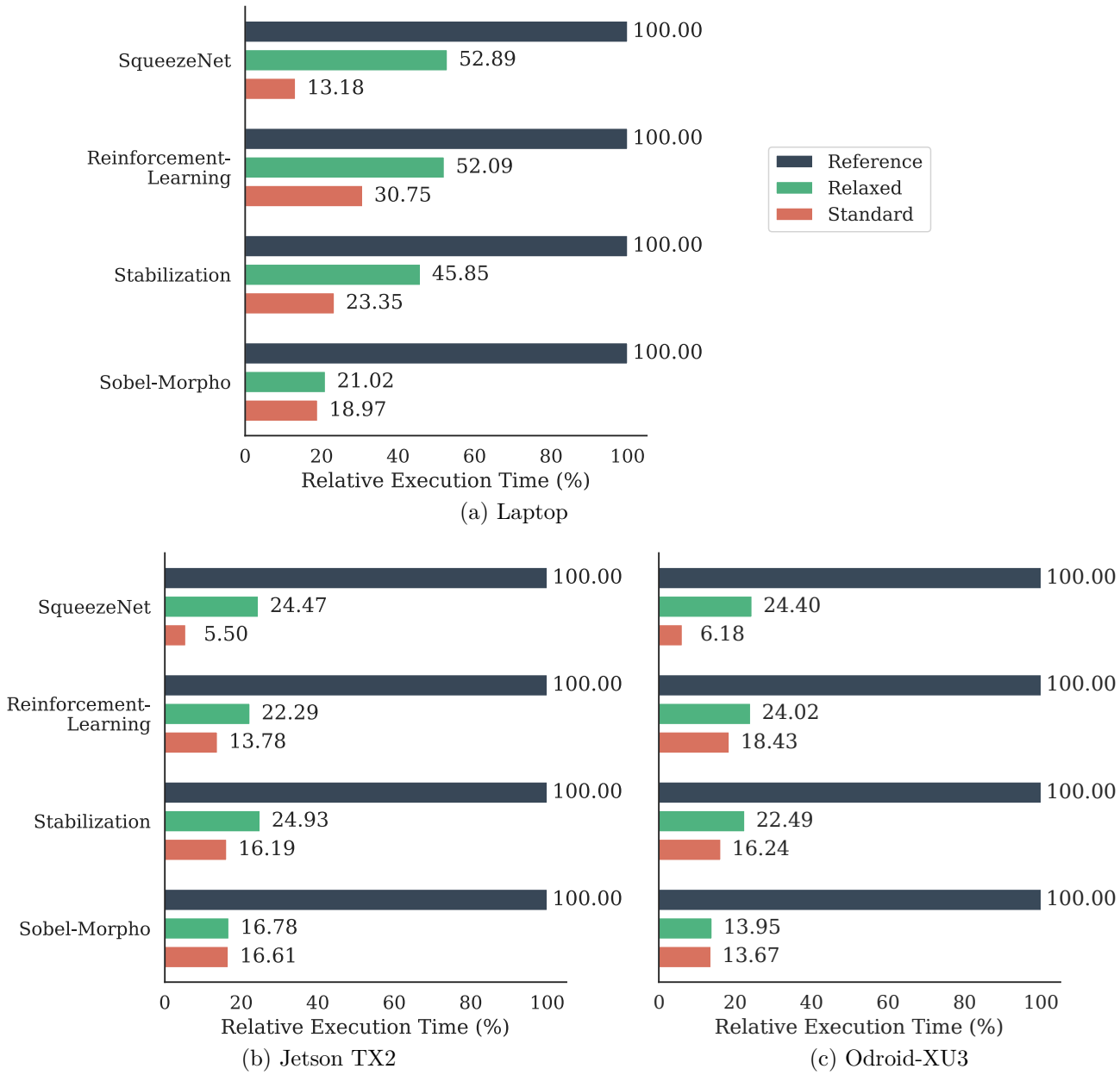


Figure 5.15 – Relative total execution time, intermediate representation building time + scheduling time, for the 3 platforms.

of total execution time is achieved when considering the total execution time spent in the resource allocation phase of SPIDER.

For the SqueezeNet application, a reduction of up to 94.5% of the total execution time is achieved on the Jetson platform with the standard execution model with 0.22% of increase on the obtained schedule latency (Table 5.6). By comparison, the relaxed con-

figuration reduces the execution time of 75.53% on the Jetson platform with a negligible (and positive) impact on the obtained schedule latency (−0.05%). On the other hand, for the reinforcement learning application, there is a non-negligible difference in the obtained schedule latency for the Jetson and Odroid platforms (19.15 and 24.1 percentage points, respectively) with a difference inferior to 10 percentage points of execution time between the relaxed and the standard execution models. Therefore, depending on the application graph topology and the targeted platform, there is a trade-off between better scheduling performance and execution time. Finally, it is important to note that the execution time of the relaxed configuration could be improved with additional optimizations of the implementation in SPIDER, which would reduce the gap with the standard configuration in terms of raw execution time performance.

5.5 Conclusion

Resource allocation algorithms often rely on an internal modeling of an application. DAGs are the most commonly used models for capturing control and data dependencies between tasks. DAGs are notably used as an intermediate representation for deploying applications modeled with a dataflow MoC on HMPSoCs.

In this chapter, we proposed a new Intermediate Representation (IR) for modeling the dependencies relationship between actors first for the SDF MoC and then extended it to the πSDF MoC. This new IR is composed of the original dataflow MoC augmented with an ad-hoc numerical representation of the data dependencies in the application graph.

We showed that our proposed IR implemented into the SPIDER tool is better suited for fast resources allocation of application than DAG-based methods due to the cost of building and storing DAGs. Experiments on various computer vision and machine learning applications showed significant gains compared to DAG-based methods both in scheduling time and memory overhead. On average, the proposed IR reduces the memory overhead of the runtime manager of 97.33% while being at least twice as fast as the DAG-based method implemented into SPIDER, and with a maximum speedup of 18.18.

CHAPTER 6

SPIDER 2.0: Implementation of a πSDF-based Extensible Runtime

6.1 Introduction

In the scientific literature, when conceiving a dataflow Model of Computation (MoC), the practical implementation is too often left aside to future developments. For instance, in [**theelen_scenario-aware_2006**] introduces the Scenario-Aware DataFlow (SADF) MoC but a more practical implementation of the model is only provided later on in [**stuijk_scenario-aware_2011**] with the FSM-SADF. The more features a given MoC supports, the more complex the analysis tools and the implementation will be. For instance, the CSDF MoC [**bilsen_cycle-static_1996**] is a generalization of the SDF MoC which offers more flexibility in the design of dataflow applications but which revealed itself to be impractical to use as stated in Section 5.2 of [**thies_empirical_2010**].

For static MoCs, it is often possible to perform all of the analysis, including mapping and scheduling of tasks, and model checking at compile time allowing for no runtime overhead [**pelcat_preesm:_2014**]. However, steps such as the mapping or the scheduling of tasks are sometimes left to the runtime, using self-timing methods [**ma_communication-aware_2018**] or using performance indicators such as exe-

cution time or memory usage to drive optimization decisions [**dauphin_odyn_2019**; **boutellier_prune_2018**; **wu_model-based_2017**]. The latter approach often requires a dedicated runtime with as low overhead as possible.

For dynamic MoCs, using a dedicated runtime is necessary in most cases. For instance, in the πSDF MoC, features such as dynamic parameterized expression of data token rates or quiescent re-configuration points induced by configure actors require a dedicated runtime. Additionally, due to the dynamic nature of the πSDF MoC, intermediate transformations such as the SR-DAG transformation, or the numerical representation introduced in the previous chapter, have to be computed at runtime. Therefore, these transformations need to be efficiently implemented to reduce as much as possible the overhead of the runtime manager over the actual processing of the application. Other graph transformations such as graph optimizations [**heulot_runtime_2015**] are also applied on-the-fly when required and also need special care in the implementation. Finally, dedicated runtimes are often provided to the end user as shared libraries which require to be loaded into the application memory. On embedded architectures with little memory available, the size of the library should also be taken into consideration.

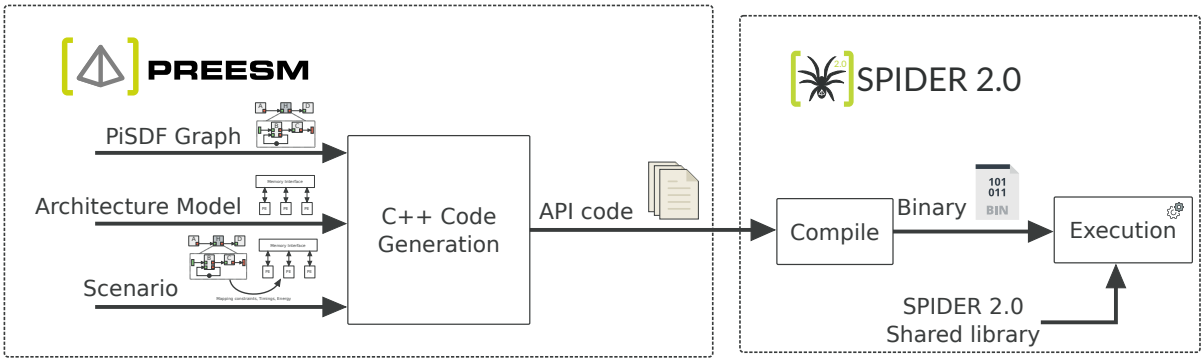


Figure 6.1 – Preesm and Spider 2.0 development framework.

In this chapter, we introduce the Synchronous Parameterized and Interfaced Dataflow Embedded Runtime 2.0 (Spider 2.0), an extensible and open-source runtime for dynamic πSDF based dataflow applications. Spider 2.0 is a new, i.e. written from scratch, runtime which is built upon the legacy of the Spider [**heulot_runtime_2015**] runtime and improves on it. For instance, one of the improvement of Spider 2.0 over Spider is the handling of static applications. In Spider, static applications where treated similarly to dynamic applications with a recompute of the mapping and scheduling of the application graph at each graph iteration. Spider 2.0, on the contrary computes the

mapping and schedule once and then reuse it for the next iterations offering better performances. Spider 2.0 is a cross-plateform (Linux and Windows) shared library fully written in C++11. Spider 2.0 takes as input a πSDF graph, with a full support of the SaMM semantics presented in Chapter 4, and an architecture model description of the targeted platform. Spider 2.0 handles the dynamic execution of the application along with its memory management. Spider 2.0 can be used directly from the provided C++ API or through the Preesm [**pelcat_preesm:_2014**] framework as shown in Figure 6.1. Using the Preesm framework to automatically generate the C++ code for Spider 2.0 requires three input files: the πSDF description file, the System-Level Architecture Model (S-LAM) architecture model [**pelcat_system-level_2009**] and the scenario file. This design approach which consists in splitting the constraints, the architecture, and the application and its implementation is often refereed to as a Y-shaped Design Space Exploration (DSE) [**suriano_damhse_2019**]. Preesm provides the user with a graphical interface to ease the creation and edition of πSDF graphs.

Using the interface of Preesm, the user designs the application graph and associates with each actor of the graph a so-called *refinement* corresponding to the implementation code of the behavior of the actor in C/C++, Cuda, etc.. The S-LAM architecture model describe the physical target platform. Finally, the scenario file embeds all of the runtime constraints of the application which correspond to the mapping constraints of the actors over the different PEs of the target architecture, the execution timings of the actors and the size of the different data types and data structures used in the application. All these information are translated into the corresponding Spider 2.0 API calls by the Preesm C++ automatic code generation for Spider 2.0. Then, the user need to compile the generated code and run the application.

Additionally, a dedicated runtime provided as a shared library should be as light in memory usage as possible. On that matter, the Spider 2.0 runtime is a lightweight runtime with a size of approximately 500KB in release mode. Although this is not currently supported in the official build tools of Spider 2.0, the library is built in such a manner that it is possible to further reduce the size of the library by removing components such as the graph and schedule exporters that are mainly used for debug purposes. Spider 2.0 ships with different mapping and scheduling algorithms, it would also be possible to only compile the necessary ones in order to reduce the library size to reduce as much as possible the memory overhead of the runtime.

The rest of this chapter is organized as follows, Section 6.2 introduces the core components of Spider 2.0 with the architecture model used to describe physical platform and the global runtime structure. Then, Section 6.3 present the different implementation challenges addressed by Spider 2.0. Finally, Section 6.4 concludes this chapter.

6.2 Spider 2.0: Runtime Structure and Design Choices

In this section, the general structure of Spider 2.0 is described along with the motivations that led to the implemented design choices. The general runtime structure is directly inherited from the Spider [**heulot_runtime_2015**] runtime. Spider 2.0 build on the legacy of Spider while optimizing some aspect of the internals and extending the global user experience with a more comprehensive API. As Spider, Spider 2.0 has been thought to be application and platform independent.

6.2.1 Runtime Structure

In this section, the runtime structure of Spider 2.0 is presented. Figure 6.2 shows the centralized runtime structure of Spider 2.0 which is directly inherited from Spider.

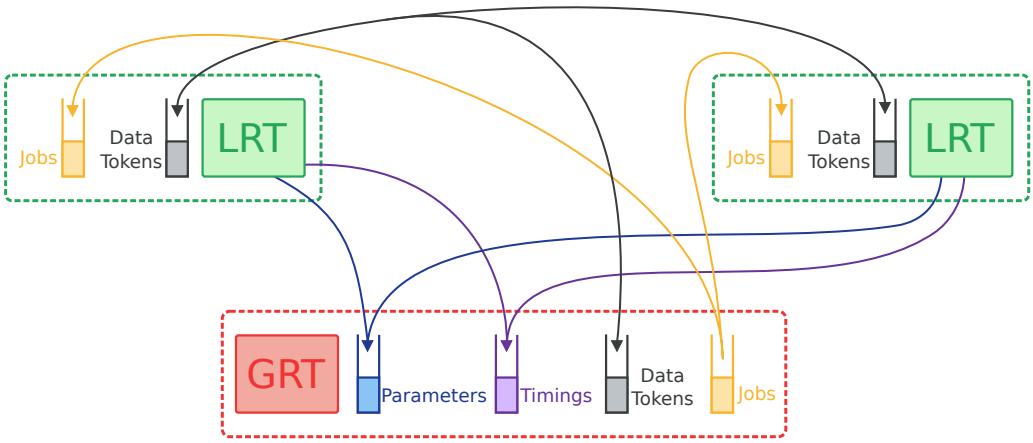


Figure 6.2 – Spider 2.0 runtime structure.

The runtime structure is divided into two main entities, namely the Global RunTime (GRT) and the Local RunTimes (LRTs) and follows a master / slave structure. The GRT is the main manager of Spider 2.0 and centralizes every decisions that need to be taken as opposed to the LRTs which only execute received jobs. There is only one GRT and as many LRTs as there are PEs handled by Spider 2.0. The GRT is in charge of the

intermediate transformations of the input graph, and of the mapping and scheduling of the actors onto the different PEs of the platform. The GRT and the LRTs communicate through different queues which are handled as follows.

— The *Parameters* queue is a centralized queue that is used by the different LRTs to send resolved values of dynamic parameters back to the GRT.
— The *Traces* queue is a centralized queue that is used to send execution traces about the jobs execution back to the GRT.
— The *Jobs* queues are distributed queues in which the GRT distributes the different jobs that need to be executed.
— The *Data Tokens* queues are distributed queues containing the data tokens exchanged by the actors of the application graph.

The *Parameters*, *Traces* and *Jobs* queues are handled directly by the SPIDER 2.0 library and rely on a shared memory address space. Indeed, the current implementation of LRTs relies exclusively on the C++ standard *thread* library. The *Data Tokens* queues are not directly handled by the library itself. The runtime uses user provided function to handle the memory allocation and communication of the data tokens of the application between different PEs. For homogeneous platform, a default implementation based on the *malloc* and *free* C functions is provided.

This design choice was motivated by the fact that most heterogeneous platform follows a Central Processing Unit (CPU) + accelerator(s) pattern. In other words, it is not uncommon to have heterogeneous platforms with multiple general purpose CPUs and specialized accelerators. CPUs will most likely be in a shared memory space whereas accelerators will have their own local memory. SPIDER 2.0 has been developed to be compatible with different kind of heterogeneous platform while maintaining its core internal code as generic as possible. Hence the choice of delegating the definition of platform specific data management to the programmer via dedicated API function calls.

For instance a 2 CPUs + GPU platform could be modeled as having the GRT and 1 LRT on the CPUs but with one either the GRT or the LRT sending the jobs to the GPU. Thus, all parameters, traces and jobs can be communicated in the shared memory space of the CPUs but the data need to be sent to, and received from, the GPU via dedicated function calls. From the point of view of the application, all of the data management such as allocation, copy, move are transparent operations managed by the runtime.

6.2.2 Hardware Model and API

In this section, the specific logical hardware model of Spider 2.0 is presented. The logical term comes from the fact that this hardware model does not reflect precisely the real physical platform but offers an abstraction to the Spider 2.0 runtime.

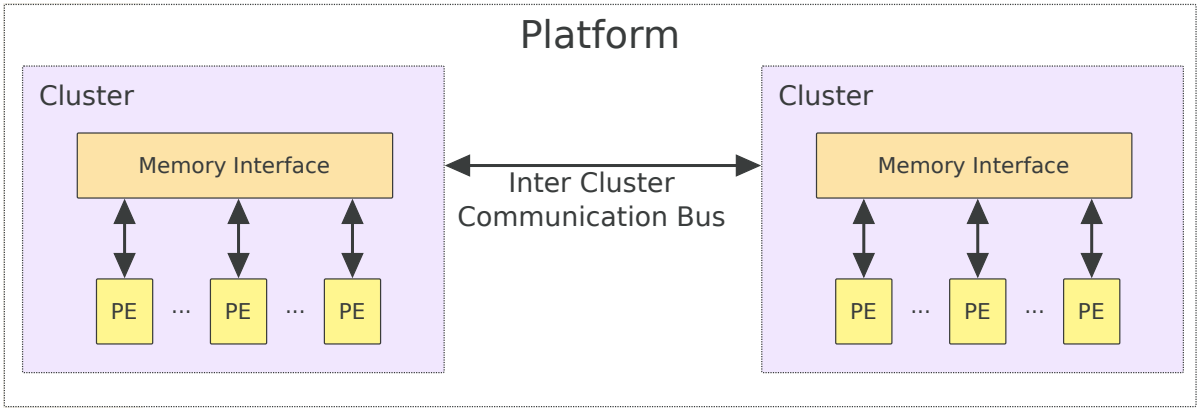


Figure 6.3 – Spider 2.0 hardware logical model.

Figure 6.3 shows an illustration of the logical hardware model. The model is defined using the following concepts.

- **Processing Elements (PEs)** are the basic compute units of the model. In Spider 2.0, a PE can either be an accelerator such as a GPU, or an FPGA or general purpose CPU. Any CPU PE can either be an LRT, that will manage jobs in addition to running them, or a pure compute unit, thus seen as an accelerator, that will need to be attached to an LRT. An accelerator PE is always necessarily attached to an LRT which means that there must be a way for sending work to the accelerator from one of the available CPU.
- **Clusters** are defined as groups of PEs sharing a memory node. For instance, on a classical 4 cores CPU, there would be 1 cluster in the sense of the Spider 2.0 model. An other example is the MPPA architecture from Kalray which features 16 physical clusters of 16 PEs and two 4 cores CPU, each with their own physical local memory. In the Spider 2.0 model, this will effectively result in 18 clusters. An other example would be the Exynos big.LITTLE architecture which feature 4 ARM 15 cores and 4 ARM A7 cores. The eight cores of the architecture share the same memory unit, hence there would be only 1 cluster of 2 different PE types.
- **Memory Interfaces** represent the physical memory units of the platform. Each memory interface provides the primitives for allocating and deallocating memory

on the physical associated physical memory to the runtime. There are exactly as many memory interfaces as there are clusters.

- **Inter Cluster Memory Buses** are the basic communication links between clusters of the SPIDER 2.0 model and are composed of two unidirectional buses: one from cluster A to cluster B, and one from cluster B to cluster A. The Inter Cluster Memory Buses provide the *send* and *receive* primitives that are needed by the runtime to manage data movements between different physical memory units.

The mapping and scheduling algorithms implemented into SPIDER 2.0 rely completely on this logical model. Importantly, when supporting a new platform, the user should not have to change any code inside the runtime but instead should use the provided API to describe its specific platform. For instance, a CPUs + GPU architecture can be seen as two cluster of PEs with one being composed of the CPUs and one composed of the GPU. For NVIDIA GPUs, the allocation and deallocation of the memory interface, and the send and receive primitives of the communication buses will then be defined using the NVIDIA CUDA library function calls.

```
1  #include <spider.h>
2
3  constexpr size_t CLUSTER_COUNT = 1;
4  constexpr size_t PE_COUNT = 2;
5
6  void createPlatform() {
7      using namespace spider;
8      /* == Creates the main platform == */
9      api::createPlatform(CLUSTER_COUNT, PE_COUNT);
10     /* == Creates the intra MemoryInterface of the cluster == */
11     auto *x86MemoryInterface =  api::createMemoryInterface
           (1073741824);
12     /* == Creates the actual Cluster == */
13     auto *x86Cluster = api::createCluster(2, x86MemoryInterface);
14     /* == Creates the processing element(s) of the cluster == */
15     auto *x86Core0 = api::createProcessingElement(TYPE_X86,
           PE_X86_CORE0, x86Cluster, "Core0", PEType::LRT, 0);
16     api::createProcessingElement(TYPE_X86, PE_X86_CORE1,
           x86Cluster, "Core1", PEType::LRT, 1);
17     /* == Set the GRT == */
18     api::setSpiderGRTPE(x86Core0);
19 }
```

Listing 6.1 – Example of the creation of an homogeneous platform in Spider 2.0.

Listing 6.1 shows a example code snippet example of the creation of an x86 platform containing two cores using the Spider 2.0 API. The main platform is created in line 9 with the total number of clusters and PEs of the platform. On line 11, the *memory interface* that will be used by the x86 cluster is created with an affected memory size of 1GB. Then, on line 13, the x86 *cluster* with 2 PEs and the previously created memory interface is declared. On line 15 and 16, the two PEs associated corresponding to the 2 cores of the platform are created. The parameters passed to the PEs are the following ones.

- PE hardware type: unique identifier associated with a given PE type (e.g. x86, ARM, GPU, etc.).
- PE identifier: unique identifier associated with a given PE.

- Cluster: cluster to which the PE belongs.
- PE name: name of the PE (used for human readable traces).
- SPIDER 2.0 runtime PE type: indicate whether the PE is an LRT or an accelerator.
- Thread affinity: optional parameter indicating the thread affinity of the PE, if left unspecified, the thread affinity management is left to the host operating system.

Finally, line 18 indicates to SPIDER 2.0 which PE will be the GRT. A complete snippet of the SPIDER 2.0 API for creating platform is available in Listing B.1 in Appendix B.

6.2.3 Heterogeneous Dynamic Mapping and Scheduling

In SPIDER 2.0, actors mapping to PEs and actors scheduling are done on-the-fly during runtime analysis. Indeed, due to the reconfigurable nature of the πSDF MoC used in SPIDER 2.0, a good mapping obtained for a given iteration of the graph will not necessary yield good performance, or even be usable due to topology changes in the graph, at the next iteration. At compile time, the user gives a set of mapping constraints for every actors of the application graph, corresponding of a list of execution constraints and Worst Case Execution Times (WCETs) for each PEs type actors can be executed on. These mapping constraints will serve SPIDER 2.0 for the mapping and scheduling of tasks at runtime.

The scheduling analysis of SPIDER 2.0 is based on a list of hypothesis that are similar to the ones of the Odyn runtime [**dauphin_odyn_2019**] and are defined here-after.

- Application:
  - H_1 An application is modeled using the πSDF MoC, with the addition of the SaMM semantics, and all actors are stateless.
- Architecture:
  - H_2 Each PE of the platform belongs to exactly one cluster.
  - H_3 There are exactly as many clusters as there are different memory nodes.
  - H_4 All PEs connected to the same memory node belong to the same cluster.
  - H_5 PEs can be dynamically enabled or disabled during runtime but all possible PEs are known at compile time.
- Mapping:
  - H_6 Buffers are dynamically allocated into the memory node attached to the PE where their consumer or producer actors have been mapped and are not allowed to migrate at runtime.
  - H_7 Explicit communication primitives must be inserted for migrating memory from one cluster to another.

  - H_8 Data transfer time between clusters are computed based on the transferred data sizes and user-defined cost functions.
  - H_9 Task migration between PE is not permitted once a mapping has been derived.
- Scheduling:
  - H_{10} There is no task preemption which means that a given PE can only execute one task at a time.
  - H_{11} There is no risk of deadlock on memory and there are no data loss during transfers.
  - H_{12} Data transfers between PEs can not be interrupted.
  - H_{13} User defined primitive for data memory allocation should respect H_{10} and H_{11}.

These different hypothesis ensure correctness of the execution relatively to the computed mapping and scheduling (H_9, H_{10} and H_{11}). Some of the scheduling hypothesis of Spider 2.0 also open the door for optimizations of the mapping process such as H_4 which states that all PEs connected to a given memory node necessarily belong to the same cluster. Thus, if an actor is not executable on a given PE type, and if the cluster is homogeneous in PE types, or can not use a specific memory node, then the entire cluster can be discarded at once reducing the DSE. Similarly, dynamically disabled PEs (H_5) are ignored during the mapping and scheduling phases. Hypothesis H_7 and H_8 enforce the logical hardware model into the mapping and scheduling analysis. Indeed, based on these hypothesis the mapping and scheduling algorithm can systematically map and schedule communications in an automatic and generic manner using solely the Fifo data sizes and user-defined transfer cost functions from one cluster to another. In other words, the user does not need to reimplement the mapping and scheduling algorithms when changing the targeted platform of the runtime.

6.2.4 Execution Modes

The Spider 2.0 runtime is designed to accelerate signal processing applications by handling the deployment of the processing on different Processing Elements (PEs). The runtime relies on a πSDF modeling of an application and a logical hardware model, presented in Section 6.2.2, of the platform on which it runs. Using these information, Spider 2.0 automatically performs mapping and scheduling of dataflow applications onto multicore platforms. Spider 2.0 also manages the memory allocation of the application

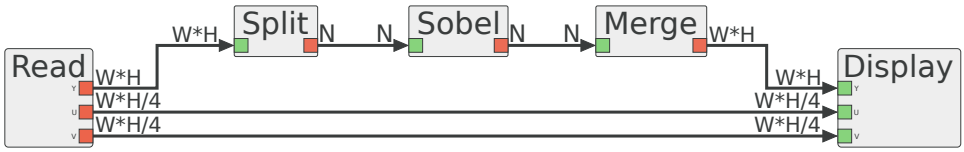
FIFOs and handles the communications and synchronizations between the different PEs. As opposed to the SPIDER runtime, SPIDER 2.0 has two main operating modes: the *accelerator* mode and the *master* mode.

In the *master* mode, similarly to SPIDER, the runtime is in charge of the complete application, which means that SPIDER 2.0 is the main entry point of the application and that the entire application needs to be modeled using the πSDF MoC.

In the *accelerator* mode, SPIDER 2.0 only handles a specific portion of an application and has the possibility to receive, and transmit, data tokens from, and to the outside of the dataflow graph through dedicated data interfaces. The *accelerator* mode is similar to the way OpenMP works. Using the *accelerator* mode, only a specific portion of an application that would benefit from a dataflow modeling needs to be modeled which will ease the transition from imperative language modeling to dataflow modeling for the user.

However in *accelerator* mode, contrary to the *master* mode, SPIDER 2.0 is not in control of the entire application resource management and the application designer must pay attention not to starve the resources assigned to SPIDER 2.0. The *accelerator* mode is particularly interesting in the case of applications having a lot of Input/Output (IO) operations and only a small portion of parallel processing.

Finally, on threaded platforms, such as the Linux operating system, it is possible to explicitly set a thread affinity for each PE and each LRT in order to enforce the logical hardware model onto the actual physical hardware layout. However, this is not mandatory and the host operating system can be let in charge of managing the PEs and LRTs thread placement. In some applications, letting the host operating system manages the load-balancing of the threads might yield better performances compared to fixed thread affinity [**hautala_toward_2018**].



(a) SDF graph of the Sobel application fully modeled in dataflow.

(b) SDF graph of the Sobel application partially modeled in dataflow.

Figure 6.4 – Example of modeling of the Sobel application using *master* mode and *accelerator* mode of SPIDER 2.0, respectively.

Figure 6.4a and 6.4b illustrate the difference of modeling for both execution mode of Spider 2.0 for the Sobel application available on the preesm-apps repository [1]. The Sobel application applies the Sobel operator onto the Y component of a given raw YUV image. Figure 6.4a shows the complete modeling of the Sobel application and Figure 6.4b shows only the processing part that actually needs acceleration. In Figure 6.4a, actors *Read* and *Display* read a YUV image (either from a camera or a video file) of size $W * H$, with U and V components being downsampled, and where W is the width of the image and H its height, and display it to the screen, respectively. Actor *Split* splits the input image into multiple slices that are then processed by *Sobel* actor. For instance if the *Sobel* operator is to be applied onto 4 slices, the *Split* actor divides the input image into 4 slices of size $W * H/4$, and then for each slice adds extra rows at the beginning and at the end of the slice due to the convolution of the Sobel operator that need neighboring pixels. Hence, the final size of the slice depends on the size of the chosen Sobel operator and not strictly equal to $W * H/4$. Symmetrically, actor *Merge* performs the reverse operation of merging the different slices into one image of size $W * H$.

In Figure 6.4b, only the data path corresponding to the Sobel filtering operation is modeled. The data input and output interfaces Y and Y_{out} are used to communicate with the outside of the dataflow graph. In Figure 6.4b, Spider 2.0 will not handle the memory allocation of the input and output data buffers, it is up to the user to provide buffers to the runtime that are valid during the entire execution of the application graph.

6.2.5 Supported Model Features

Spider 2.0 is a model-based embedded runtime implementing the πSDF MoC with the extended semantics of SaMM, presented in Chapter 4. Model-based tools do not necessary support of all the features that a MoC provides. We refeer as the feature of a MoC as being an intrinsic property of a MoC, capturing a particular behavior or functionality. For instance, parametric rates are a feature of the πSDF MoC.

Table 6.1 list the main features of the PiMM and SaMM models and their official support by the Preesm [**pelcat_preesm:_2014**], Spider [**heulot_runtime_2015**] and Spider 2.0 tools. Supported features marked with a star are features added during this thesis and that were not available before. The *parameterized delays* and *parameterized (dynamic) delays* features correspond to the possibility of having parameterized delay

1. **https://github.com/preesm/preesm-apps/tree/master/tutorials/org.ietr.preesm.sobel**.

Table 6.1 – PiMM and SaMM model feature support by the Preesm, Spider, and Spider 2.0 tools.

PiMM / SaMM model feature	Preesm	Spider	Spider 2.0
Static Parameters	Yes	Yes	Yes
Dynamic Parameters	No	Yes	Yes
Parameters Dependency Tree	Yes	Yes	Yes
Parameterized Delays (LDs, LPDs, GPDs)	Yes (all)	Yes* (LDs and GPDs)	Yes (all)
Parameterized (dynamic) Delays (LDs, LPDs)	No (all)	Yes* (LDs)	Yes (LDs)
Hierarchical Graphs	Yes	Yes	Yes
Hierarchical Configure Actors	No	No	No
Data Path Selection	Yes* (static)	Yes	Yes
Dataflow Initialized Delays	Yes*	Yes*	Yes

values. When combined with dynamic parameters, this lead to potentially variable delay sizes during the execution of an application. Chapter 4 details how parametric delays can change their value when using the persistence semantics of SaMM. The *data path selection* feature corresponds to the possibility of disabling completely some data path in a πSDF graph. To do so, it is necessary that all data rate in the data path are null.

The Preesm tool support all of the static features of the PiMM and SaMM models whereas the Spider and Spider 2.0 tools support all of the static features and the reconfigurability of PiMM and SaMM, with the exception of reconfigurable LPDs. Interestingly, although it is possible to create hierarchical configure actor in Preesm, this feature is usable neither by Preesm itself nor by the Spider and Spider 2.0 runtimes.

In the next section, some of the implementation details of Spider 2.0 are presented and discussed.

6.3 Implementation Details for an Efficient Model-Based Runtime

Due to the embedded nature of the Spider 2.0 runtime, internal features of the runtime need to be implemented efficiently. In this section, details about the implementation of the dynamic expression evaluator, intermediate transformation and the synchronization mechanism used inside the runtime are discussed.

Section 6.3.1 presents the implementation of a fast and efficient math expression parser used for the dynamic evaluation of the πSDF parameterized expressions. Then Section 6.3.2 introduces a new intermediate transformation that enforces the execution rules of the πSDF MoC and leads to a simplification in analysis algorithms used in the runtime. Finally, Section 6.3.3 presents the asynchronous notification system of communications between the LRTs and the GRT.

6.3.1 Efficient Parameterized Expression Parser

In the πSDF MoC, parameter values can either be constants or set dynamically at runtime by the use of *configure actors* [**desnos_pimm:_2013**]. Parameters can also be the result of math expressions depending on other parameters that are either static or dynamic. In the latter case, the math expression of a parameter can only be evaluated at runtime after the values of the dynamic parameters on which it depends have been set. In the πSDF MoC, parameterized expressions are used for fixing the value of data rates on the Fifos of a graph which means they will be evaluated many times per graph iteration.

Additionally, some applications may require complex parameter expressions involving trigonometric functions, exponential and logarithm functions, power and square root functions, etc. For instance, the SIFT application from the preesm-apps repository [2] contains many complex parameter expressions including conditional statements, square roots, floor and ceil functions, and geometric sums. For those type of dynamic expressions, an efficient and fast expression evaluation mechanism is needed. Such a parser is presented in this section and performances are evaluated against a state-of-the-art math parser.

6.3.1.1 Just-In-Time Compiled Expression Parser

Spider 2.0 has been designed to have as small runtime overhead as possible and also to have a low memory impact. This means that Spider 2.0 should be fast enough so that its execution time is negligible relatively to the execution time of the application the runtime is managing, and the size of the generated shared library should be as small as possible not to clutter the application memory.

Three main optimization criteria are derived for the expression parser:

C1 The parser should not introduce any runtime latency.

C2 The parser should not increase the binary size of the Spider 2.0 library.

2. **https://github.com/preesm/preesm-apps/tree/master/SIFT/**.

C3 The parser should depend on as few external libraries as possible.

Algorithm 4: Expression creation algorithm

Input : Literal infix math expression : E
Symbol list: $S = \{s_i\}$

Output: Output expression object: E_{out}

```
1  if checkExpressionSyntax(E) == ERROR then
2      exit("syntax error"); // Exit if the syntax of the expression is ill-formed.
3  infixStack = tokenize(E); // Tokenize input expression.
4  if isExpressionStatic(E) then
5      postFixStack = convertToRPN(infixStack); // Convert the infix stack into a
         postfix stack using the RPN.
6      E_out = createExpressionFromStaticStack(postFixStack, S); // Statically
         evaluate the expression and create an object with the constant value.
7  else
8      functionFile = printCppFunctionCode(infixStack); // Create a CPP file
         containing a function with the input expression E.
9      libraryFile = systemCallForJITCompilation(functionFile); // Calls a C++
         compiler to compile the generated file into a shared library.
10     loadedLibrary = loadDynamicLibrary(libraryFile); // Load the compiled shared
         library into the runtime.
11     E_out = createExpressionFromLibrary(loadedLibrary, S); // Creates an object
         associated with the dynamically loaded library for runtime evaluation.
12 return E_out; // Return the expression object.
```

The two criteria **C1** and **C2** directly depend on the implementation choices, and the third criterion **C3** is to improve the portability and maintainability of the Spider 2.0 code. An important point to take into consideration is that applications modeled with a dataflow MoC are often stream processing applications with a long sustained execution time. In other word, it is rather uncommon to model a dataflow application with its main top graph being only executed once. This strong hypothesis is at the heart of the proposed solution. Indeed, the proposed parser is a "*compile once, evaluate many times*" kind of parser which means that the compilation of the expressions is supposed to happen only once during the initialization of the application and may take a non negligible time.

However, during the execution phase of the application, expressions may be copied and evaluated very often for close to natively compiled code performances.

Algorithm 4 describes the expression parsing principles in SPIDER 2.0. Algorithm 4 takes as input an infix expression E and a list of symbol $S = \{s_i\}$ and return an expression object E_{out} used for later evaluation. The infix expression notation corresponds to the *classical* way of writing mathematical expressions and is opposed to the postfix notation style. Specifically, the Reverse Polish Notation (RPN) is used in SPIDER 2.0. Equation (6.1) shows an example of a mathematical function using the infix notation and Equation (6.2) shows the same expression written using the RPN. RPN was proposed in 1954 [**burks_analysis_1954**] as a memory efficient approach to functions evaluation for computers. The main advantage of using polish notation or RPN over infix notation is that it removes the need for parenthesis, as there are no ambiguity in the precedence of operators.

$$f(x) = (3 * (x + 1))/(4 * x) \tag{6.1}$$

$$f(x) = 3\ x\ 1\ +\ *\ 4\ x\ *\ / \tag{6.2}$$

The second input of Algorithm 4 is the list of symbol $S = \{s_i\}$, which corresponds to the list of symbolic variables of a dynamic mathematical expression. In other words, each $s_i \in S$ corresponds to a variable of the expression that is dynamically set and that is only known at runtime. For example, in Equation (6.1), $S = \{\ x\ \}$.

Algorithm 4 works as follows. First, the syntax of the input expression E is checked in line 1. This step checks for missing parenthesis and verifies that operators are not missing operands. For example, the + operator requires two operands and the syntax checker will return an error if only one operand is provided to this operator. The second step is to *tokenize* the infix expression (line 3), i.e. to extract all atomic elements making the expression. For example, the tokenized output of Equation (6.1) is $T(E) = \{(, 3, *, (, x, +, 1,),), /, (, 4, *,)\}$. This second step is an important step as it allows for constructing the RPN of the expression and for checking if the expression contains variable symbols.

Thirdly, the static property of E is checked on line 4. The expression is considered to be static if and only if all symbols used in the expression are either constant numeric values or static parameters. In case E is static, E is converted to postfix notation and

statically evaluated (lines 5 and 6). The result of the static evaluation is stored inside the expression object E_{out} as a constant value, resulting in a simple value lookup at runtime. If the expression E is dynamic, i.e. it contains at least one dynamic symbol, then the JIT compilation process starts.

```
1  #include <cmath>
2  #include <functional>
3
4  extern "C" {
5      double expr_0(const double *const args[]) {
6          using namespace std;
7          const auto x = *(args[0]);
8          return ((3*(x+1))/(4*x));
9      }
10 }
```

Listing 6.2 – Example of an automatically generated function call for Equation (6.1).

The first step of the JIT compilation of an expression is to create an equivalent function call. This function call is then printed into an automatically generated Cpp file (line 8). Listing 6.2 shows an example of automatically generated function call for Equation (6.1). Then, the automatically generated Cpp file is compiled into a dynamic shared library using a system call to a compiler (line 9). On Linux-based systems, the default compiler used is GCC[3] and the command of Listing 6.3 is used for compilation, where #filename.cpp correspond to the generated file (line 8). To limit the impact and the number of system call to the compiler, a cache system is used to avoid recompiling a given expression. This cache is based on a computed hash of the expression infix string. Using an hash-based cache raises the potential issue of having collisions and thus, avoiding to compile a new expression resulting in incorrect evaluation at runtime. Indeed, if two different expressions result in the same hash then the cache system will not compile the second expression and directly use the pre-compiled first expression. However, in the studied use case (dataflow applications), there will unlikely be hundred of different dynamic parameter expressions, thus lowering the risk of collision.

3. **https://gcc.gnu.org/onlinedocs/gcc-10.1.0/gcc/**.

```
$ g++ -shared -o #filename.cpp -std=c++11 -O2 -fPIC -lm
```

Listing 6.3 – System call used for the JIT compilation of dynamic expressions in the Linux environment.

Finally, the last step of the JIT process is to load the compiled shared library into the program memory space (line 10). On Linux-based system, this is done using the **dlopen** and **dlsym** functions. Once the JIT-compiled function is loaded, the associated function pointer is stored inside the expression object E_{out} (line 11), which will result in close to natively compiled code performance during runtime evaluations of the expression.

Spider 2.0 is a cross-platform runtime running also under the Windows operating system. However, as of today, the JIT compiled approach presented in this section uses specific UNIX system function call and currently works only under the Linux operating system. Therefore, when running on Windows, Spider 2.0 uses a fallback method using a runtime evaluation similar to the one of the exprtk library. This fallback approach will not be evaluated in the following section.

6.3.1.2 Experimental Results

In this section, the proposed JIT compiled expression parser is evaluated against the current state of the art implementation of a math expression parser in C++. All the experimentation are done on a x86-x64 Laptop equipped with an Intel Core i7-7820HQ processor and 32GB of DDR4 RAM memory. The GCC version used in these experiments is the version 7.3.0 with the O2 level of optimization and the Link Time Optimization (LTO) enabled.

Spider 2.0 expression parser is compared against the **exprtk** [4] header library and natively compiled code. The natively compiled code correspond to the case of the evaluated function being known at compile time and thus benefiting from the compiler optimizations. It correspond to the best case scenario. The exprtk library was chosen due to the fact that it seems to be the fastest open-source library for math expression parsing available to the best of our knowledge. Indeed, when compared to other libraries [5], exprtk is the best overall library.

4. **https://github.com/ArashPartow/exprtk**.
5. **https://github.com/ArashPartow/math-parser-benchmark-project**.

Table 6.2 – Functions used in the benchmark for the expression parsing.

Function	Expression
F_0	$y + x$
F_1	$2 * (y + x)$
F_2	$(2 * y + 2 * x)$
F_3	$((1.23 * x^2)/y) - 123.123$
F_4	$(y + x/y) * (x - y/x)$
F_5	$x/((x + y) + (x - y))/y$
F_6	$1 - ((x * y) + (y/x)) - 3$
F_7	$(5.5 + x) + (2 * x - 2/3 * y) * (x/3 + y/4) + (y + 7.7)$
F_8	$1.1x^1 + 2.2y^2 - 3.3x^3 + 4.4y^15 - 5.5x^23 + 6.6y^55$
F_9	$sin(2 * x) + cos(pi/y)$
F_{10}	$1 - sin(2 * x) + cos(pi/y)$
F_{11}	$sqrt(111.111 - sin(2 * x) + cos(pi/y)/333.333)$
F_{12}	$(x^2/sin(2 * pi/y)) - x/2$
F_{13}	$x + (cos(y - sin(2/x * pi)) - sin(x - cos(2 * y/pi))) - y$
F_{14}	$max(3.33, min(sqrt(1 - sin(2 * x) + cos(pi/y)/3), 1.11))$
F_{15}	$if((y + (x * 2.2)) <= (x + y + 1.1), x - y, x * y) + 2 * pi/x$

Table 6.2 shows the 16 functions that were used in this comparison. All of these functions directly come from the proposed benchmark of the exprtk library [6] and span across a variety of math expressions. Although mathematically equivalent, functions F_0 and F_1 test the capability of the expression parser to optimize patterns such as factorization with a constant value. Each function has been ran for a total of 72072 values for x and y, respectively, spanning from -100 to $+100$ with a delta of $0,002775$. These values directly come the benchmark of the exprtk library. Each run has been averaged over 100 iterations to further reduce measurement noise.

Figure 6.5 shows the average evaluation rate in million of evaluations per seconds (Mevals / s) for each function of Table 6.2 and for each of the method. These values are also reported in Table 6.3 along with the compile time needed by each function for

6. **https://github.com/ArashPartow/exprtk**.

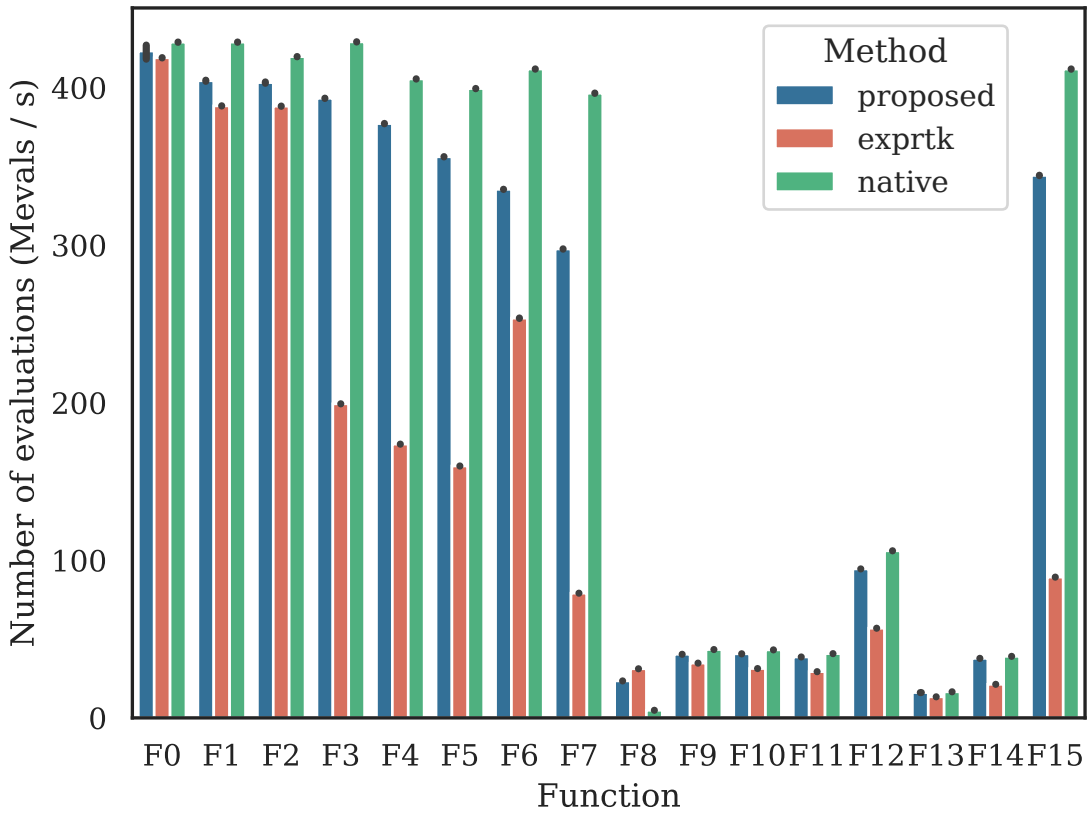


Figure 6.5 – Average evaluation rate in Mevals / s for the functions of Table 6.2 for the three different approaches.

the JIT approach. On Figure 6.5, the standard deviation for each function is represented with a black oval on top of each bar. The first take away message from Figure 6.5 is that natively compiled code is faster than both the proposed and the exprtk approaches for every functions, except for the function F_8. The function F_8 uses the *std::pow* function in the natively compiled code and an optimized version for the other two approaches. Indeed, the *std::pow* function is a generic method using a floating point data type for the exponent whereas the optimized approaches consider both integer and floating point exponents.

In Figure 6.5, there is a clear benefit to the JIT compilation approach over the fully runtime approach of the exprtk library, with the JIT approach being faster for every function, except function F_8. On trivial functions like the F_0, F_1 and F_2 functions, there is no clear advantage for either of the methods. However, with functions using the *division* operator (F_3 to F_7 and F_{12}), the compiler seems to be able to apply drastic optimizations.

Table 6.3 – Average evaluation rate in Mevals / s for every function of Table 6.2 for the three different approaches. The compile time (in ms) of each function for the JIT-Compiled approach is also reported.

	JIT-Compiled		Exprtk	Native
Function	Compile Time(ms)	Mevals / s	Mevals / s	Mevals / s
F_0	167	425.4	418.5	**428.4**
F_1	165	403.8	387.9	**428.4**
F_2	165	402.7	387.8	**419.1**
F_3	163	392.8	198.9	**428.6**
F_4	165	376.6	173.0	**404.9**
F_5	170	355.6	159.0	**398.9**
F_6	169	334.9	253.0	**411.3**
F_7	168	296.9	79.0	**395.9**
F_8	178	23.0	**31.0**	4.0
F_9	166	40.0	34.0	**43.0**
F_{10}	168	40.0	31.0	**43.0**
F_{11}	178	38.0	29.0	**40.0**
F_{12}	166	94.0	56.0	**105.8**
F_{13}	168	16.0	13.0	**16.0**
F_{14}	178	37.0	21.0	**39.0**
F_{15}	166	343.8	89.0	**411.3**

Indeed, there is an average speedup of 2.19, and up to 3.76 for F_7, using the proposed JIT-Compiled method compared to the exprtk library on these functions. Interestingly, functions including trigonometry such as the F_9, F_{10}, F_{11} and F_{13} function have a similar evaluation rate for the three approaches. It seems that in this case, the computation is bounded by the standard library math function and that no particular optimization can benefit from compile time optimizations of these functions.

Table 6.3 also reports the compile time for each function for the proposed JIT-Compiled approach. On the processor used for these experiments, there is an average compile time of 170ms which is non negligible. However, in the context of Spider 2.0, the expressions are only compiled once on startup which will result in a high initialization

time (measured in seconds); but considering that applications that run with SPIDER 2.0 will most likely run for several minutes, this initialization time is not an issue.

6.3.2 Enforcing the πSDF Execution Model

In this section, a new intermediate transformation of a πSDF graph is proposed that enforce both the execution rules of the πSDF MoC [**desnos_pimm:_2013**] and the data dependencies associated to the reconfiguration semantics of πSDF.

The πSDF MoC is a hierarchical, parameterized and reconfigurable dataflow MoC. The reconfigurable property of the πSDF MoC comes from the dynamic value of parameters. Indeed, in a πSDF graph, there are two types of parameters: *static* and *dynamic* parameters. Static parameters have fixed values that can not change during the runtime execution of the application graph. Dynamic parameters on the other hand, have dynamic values that are set by *configure* actor. The value of a dynamic parameter can change between two successive firings of a hierarchical graph but the value is fixed during the firing of the hierarchical graph to which it belongs.

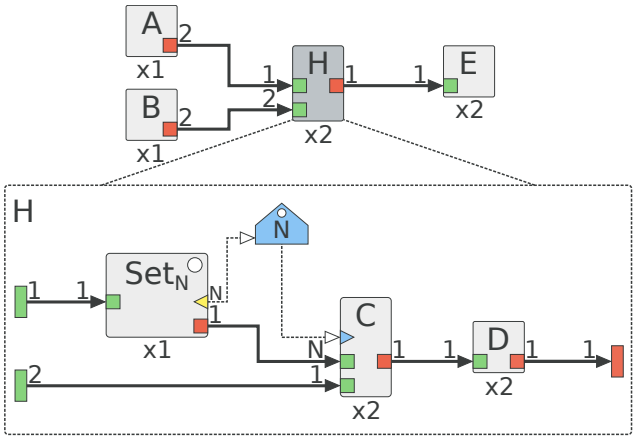


Figure 6.6 – Example of a reconfigurable hierarchical graph.

Figure 6.6 shows an example of a hierarchical and reconfigurable πSDF graph, with the value of the parameter N, inside the hierarchical actor H, being set by the configure actor Set_N. In Figure 6.6, the hierarchical actor H is executed 2 times, which means that the parameter N can change two times, at the beginning of each firing of H.

In [**desnos_pimm:_2013**], the authors define the execution rules of a πSDF graph which state that all configure actor are executed before any other non-configure actors in the graph. Additionally, configure actors have a repetition value strictly equal to 1 and

all data output ports of a configure actor act equivalently to data input interfaces for the actors connected to them. This means that if more data tokens are consumed by an actor connected to a configure actor, then the data tokens will be repeated as many times as necessary.

In Spider [**heulot_runtime_2015**], in order to respect those execution rules, specializations of the SR-DAG transformation and of the scheduling functions were used. Indeed, the SR-DAG transformation has a special partial transformation which will perform the single rate linkage only up to the next configure actors ready to be executed and their direct dependencies; and a general single rate linkage algorithm for the other non configure actors. Similarly, the scheduling algorithms of Spider also possess partial specializations for only scheduling configure actors and their dependencies. Such specializations imply an higher cost on maintainability of the runtime on the long term and require writing additional specific section to otherwise generic algorithms.

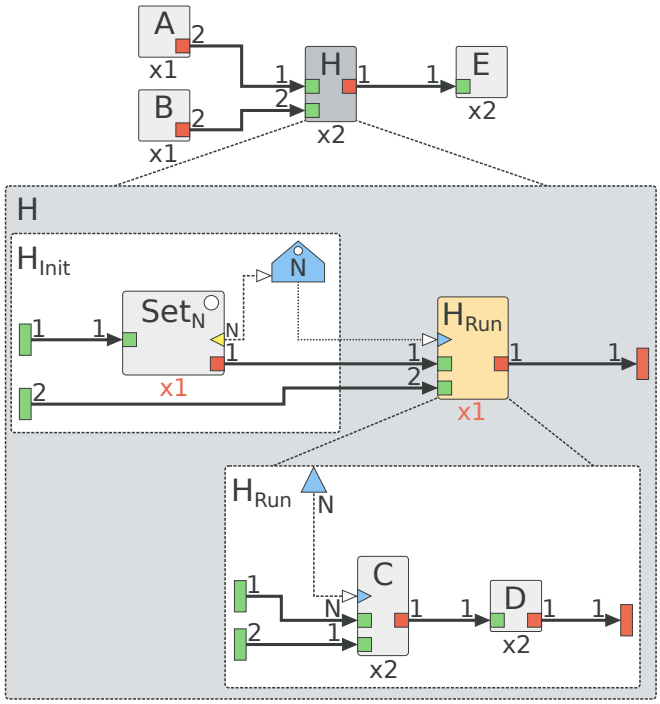


Figure 6.7 – Enforced execution model of the graph of Figure 6.6.

To avoid such issues in Spider 2.0, a new intermediate transformation is applied to every reconfigurable graph of a πSDF application during the initialization phase of the runtime. This transformation enforces the execution rules of the πSDF MoC while maintaining the genericity of algorithms used to manipulate πSDF graphs. The proposed inter-

mediate transformation consists of systematically splitting reconfigurable πSDF graphs into two distinct subgraphs corresponding to their initialization and execution phase, respectively. Given a reconfigurable graph G, the transformation steps are as follows:

1. Create a new subgraph G_{Run} inside G and copy every non-configure actors of G to G_{Run}.
2. For every data input and output interfaces in G connected to a non-configure actor, add a corresponding data interface in G_{Run}.
3. Connect every data interfaces of G_{Run} created during Step 2 to the original data interfaces of G, and to the copied non-configure actors inside G_{Run}.
4. For every edge between a configure actor and a non-configure actor in G, add a data input interface to G_{Run}. Connect the new data interface to the non-configure actor in G_{Run} and connect the configure actor in G to the input interface of G_{Run}.
5. Remove all the non-configure actors and the associated edges in G.
6. For every parameters in G, create a corresponding configure input interface (see [**desnos_pimm:_2013**] for more details) in G_{run}.

Figure 6.7 shows the result of the intermediate transformation applied to the graph of Figure 6.6. All the non-configure actors have been moved inside the new hierarchical actor H_{Run} with the appropriate data interface connections. This new transformation naturally enforces the execution rules of a πSDF by adding a precedence between configure actors and non-configure actors. The transformation also naturally enforces the consumption rules on data output port of configure actors by explicitly adding data input interfaces.

Importantly, the graph of Figure 6.7 requires no specialization of the SR-DAG transformation and scheduling algorithms. Indeed, during the SR-DAG transformation, hierarchical graphs are iteratively processed in a top-down approach. This means that for a given hierarchical graph H, every hierarchical actors contained in H are first treated as atomic actors by the transformation and are then replaced by their inner subgraph at the next iteration of the transformation. With the proposed pre-processing intermediate transformation, it is then only necessary to annotate "Run" graph as *future jobs* to defer their replacement after the resolution of the dynamic parameters they depend on. Similarly the scheduling algorithm does not need to know about the specificity of the πSDF MoC but will instead schedule all available single rate actors of the current iteration of the SR-DAG transformation.

6.3.3 Notification-based Synchronization between LRTs

The Spider 2.0 runtime relies on a master/slave structure with the GRT being the master and the LRTs being the slaves. The synchronization between the LRTs and the GRT and the LRTs is done using an asynchronous notification system.

6.3.3.1 Notification Messages and LRTs State Machine

Synchronization between multi-threaded applications is often performed using synchronous semaphore or mutexes primitives. In [**koster_using_2001**], authors use a message-based system to synchronize multiple thread and applications using asynchronous messages. The rationale behind this idea is that synchronizations done using semaphores or mutexes tend to be difficult to maintain, introduce potential bugs that are complicated to track and are more prone to synchronization errors in complex applications.

In Spider 2.0, the same message based approach is used. Whenever an LRT needs to send an information to another LRT, the first LRT sends a notification at a time t to the second that will handle it at some other time point t_2.

When executing actors, LRTs need to wait for all the input buffers that are required to be written by the predecessors of the actor. In Spider [**heulot_runtime_2015**], a semaphore was attached to each buffer controlling the access to it and the LRTs needed to wait for the semaphores of the buffers to be increased in order to pursue their execution. This means that, if a job was not executable right away, the LRTs would have to wait for all the input buffers to be ready and were not able to process any other signal that could arise from the GRT in the mean time.

In [**miomandre_embedded_2018**], the synchronization pattern used to implement the Spider runtime on a many core architecture was based on the *observer* pattern. Each LRT would embed a job counter for every other LRTs. When an LRT needs to fire an actor that does not have all of its dependencies satisfied, it first *registers* itself upon the LRTs to which these dependencies are mapped by sending them a notification. When the LRTs finish their current execution, they go through all of their pending requests and *notify* the demanding LRTs with the new value of their job counter. Additionally, in the implementation of [**miomandre_embedded_2018**], each LRT also has the possibility to *peek* at the job counter values of other LRTs to limit the synchronization wait.

Algorithm 5: LRT State Machine

```
1  while run == true do
2      while gotNotification() do
3          notifcation = readAndPopNotification();
4          handleNotification(notification);
5      if gotJobsToDo() then
6          job = popNextJob();
7          if isJobExecutable(job) then
8              runJob(job);
9              sendEndOfJobNotification(job);
10         else
11             pushFrontJob(job);
```

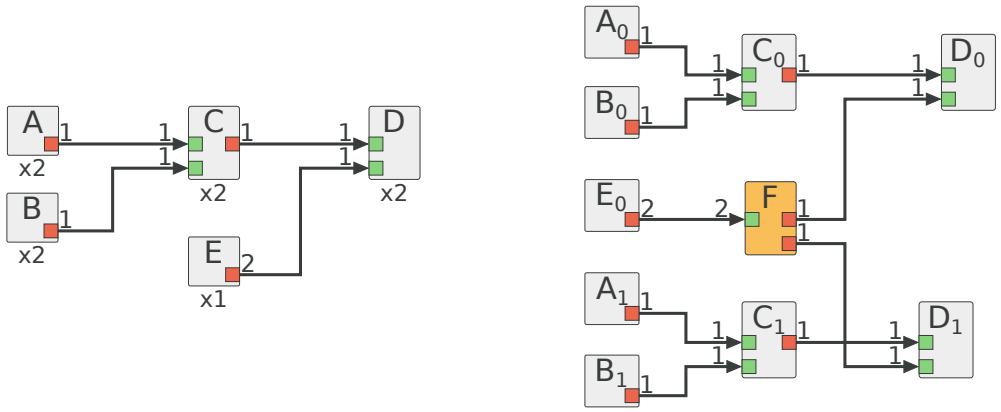
In Spider 2.0, similarly to [**miomandre_embedded_2018**], each LRT possess a local job counter value for all other LRT for job synchronization. However, the synchronization mechanism of Spider 2.0 relies on the scheduling algorithm and instead of the *observer* pattern to limit the synchronization. For each scheduled task, a job message is sent to its mapped LRT. This job message possess a unique job id and a list of job dependencies that must be satisfied before executing the associated task. The list of dependencies of task is a list of pair of integers, where the first integer corresponds to the job id, set by the scheduler, of the dependency and the second integer corresponds to the id of the LRT to which the dependency is mapped. Every job message also contains an array of boolean notification flags. These flags are set by the scheduler and indicate to the LRT receiving the job message which LRTs should be notified upon the job completion. Depending on the scheduling policy, which may vary from one platform to another, there is the possibility to limit the number of exchanged notification without having to change the LRTs state machine and implementation.

Algorithm 5 describes the LRT state machine based on the described asynchronous notification system. An LRT runs in an infinite loop until it receives a notification to stop. In line 2 and 3, the LRT checks and reads from its notification queue until there are no more notifications available. The read to the notification queue can be either blocking or non blocking. The read is blocking in the case of an LRT having no more jobs to execute and thus waiting for commands from the GRT and non blocking else.

Then, after handling all of its input notifications, the LRT checks if it has non executed jobs available (line 5). The LRT pop the first job from its job queue and checks if its dependencies are satisfied (line 7) by comparing the expected job id of the dependencies to the local values of the other LRTs job counters. If the dependencies are satisfied, the LRT executes the job and send a notification to the LRTs required by the job.

6.3.3.2 JIT and Delayed Execution of Tasks

This section details the two implemented mode of jobs execution in Spider 2.0 and their impact on the number of exchanged notification messages is compared in Section 6.3.3.3.



(a) Example of a an SDF graph with multiple dependencies.

(b) Resulting SR-DAG of the graph of Figure 6.8a

Figure 6.8 – Example of an SDF graph containing actors with multiple input data dependencies and its corresponding SR-DAG.

In Spider 2.0, the possibility to implement any kind of mapping and scheduling algorithm gives the user the flexibility to adapt the runtime to specificities of a given platform similarly to what was done in Spider for the MPPA architecture [**miomandre_embedded_2018**]. Moreover, Spider 2.0 support two specific execution policies, namely the JIT execution and the delayed execution policies. The JIT execution policy consists of an execution policy where all jobs are executed during the scheduling of the tasks, which means that jobs are sent to LRTs as soon as they are scheduled. The delayed policy on the other hand first maps and schedules all the available task and then send all the tasks to the LRTs for the execution.

Any scheduling algorithms implemented in SPIDER 2.0 has a dedicated *traits* indicating which execution policy of SPIDER 2.0 it supports. Scheduling algorithms are not required to support both policies but should support at least one of them. The default mapping and scheduling algorithm implemented in SPIDER 2.0 is a LIST algorithm based on the work of [**kwok_high-performance_1997**].

The JIT execution policy sends every job to its mapped LRT as soon as it has been mapped and scheduled. Using this policy, it is not possible to perform any kind of optimizations regarding the synchronization of jobs. In other words, since jobs are executed concurrently to the scheduling algorithm, they have no vision of the future and can not know in advance which LRT should be notified upon their completion. Therefore, using the JIT execution policy, every LRT must *broadcast* their current job counter to every other LRTs after each job completion.

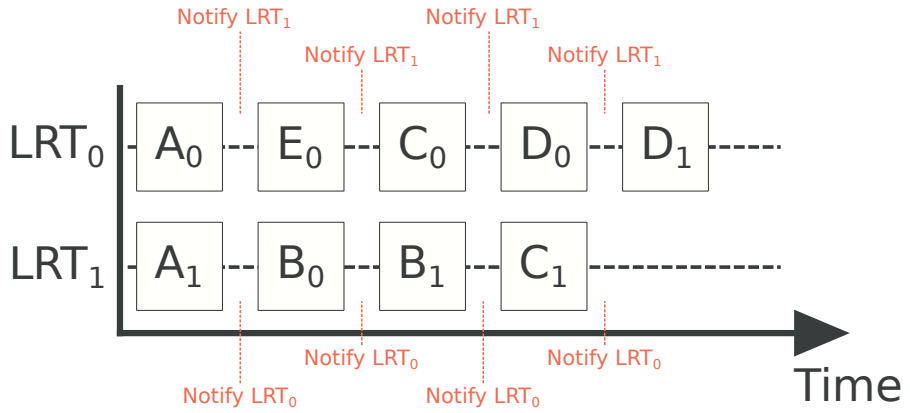


Figure 6.9 – Example of notification synchronization when using a JIT schedule and execute approach.

Figure 6.8a and 6.8b show an example of an SDF graph with its corresponding SR-DAG. In the graph of Figure 6.8a, there are multiple actors with multiple input data dependencies. For instance, actors C and D depends on actors A and B, and C and E, respectively. Figure 6.9 shows a potential schedule of the graph of Figure 6.8a using the JIT execution policy. In this schedule, a notification is sent after the execution of every jobs, respecting the execution policy, resulting in unnecessary synchronizations. For instance, actor A_0 is executed on the LRT_0 and its direct successor, actor C_0, is also executed on LRT_0 making the notification to LRT_1 unnecessary.

In the delayed execution policy, the mapping and scheduling algorithm used by SPIDER 2.0 first perform the mapping and scheduling of all actors that are currently

available for mapping and scheduling before sending any jobs to the LRTs. Hence, only the required notifications are sent.

Using this execution policy, the LRTs spend more time in IDLE (or sleep) than with the JIT execution policy. However, delaying the execution of the actors after the mapping and scheduling offers better optimization opportunities. Indeed, once the complete mapping and schedule is obtained, it is possible to reduce synchronization points using two optimization passes. The first synchronization optimization leverages the fact that synchronizations between actors depend on their data dependencies in the application graph, but that there is a limited number of LRTs. For instance, an actor might have 4 data dependencies but if there are only two LRTs in the platform, then at most, there is only 1 synchronization necessary between both LRTs and not 4.

The second synchronization optimization is related to special actors behavior. It is possible to remove unnecessary fork and duplicate actors, two special actors that are used to either split a given input buffer into multiple output buffers or to duplicate the input buffer on multiple outputs, respectively. In [**heulot_runtime_2015**], the author explains that if the memory allocation of the Fifos of the graph takes into account the specific behavior of these actors, they do not need to be executed at all, thus removing unnecessary synchronization points.

Algorithm 6 details the first synchronization optimization for a list of actors that have been mapped and scheduled. The algorithm first goes through all scheduled actors (line 1) and, for each actor A, goes through all of its dependencies (line 3). Then, for each dependency D of A, it starts by disabling the sending of a notification from the PE onto which D is mapped, to the PE onto which A is mapped (line 4). This is done due to the fact that the dependency D might have been mapped onto the same PE as A, and since the scheduler will necessary have scheduled the dependency D before the actor A, when the PE starts the execution of A, there is not need to wait for the completion of D.

Then, the second step is to check if the PE to which D is mapped is the same as the one where A is mapped (line 6). If it is not the case, meaning that both actors are mapped onto different PEs, then the test of line 8 verifies if there is already a notification coming from the PE to which D is mapped. If not, then a notification should be sent after the completion of actor D to the PE of actor A (line 9). In the case where there already exists a synchronization between the PE of actor D and the one of actor A, then the current actor D_2 sending the notification is fetched (line 11). Finally, the scheduling positions of

both D and D_2 are compared (line 14) and if D is further placed into the schedule, i.e it will be executed after D_2, then the synchronization is switched from D_2 to D.

Algorithm 6: Notification Optimization Algorithm

Input : List of actor : L_A
List of actor dependencies : L_D
Schedule and Mapping information: SM

```
1  for actor A ∈ L_A do
2      mappedPE_A = getMappedPE(A, SM);
3      for actor D ∈ L_D(A) do
4          setNotificationFlag(D, mappedPE_A, false);
5          mappedPE_D = getMappedPE(D, SM);
6          if mappedPE_D == mappedPE_A then
7              continue;
8          if gotNotificationFromPE(mappedPE_D) == false then
9              setNotificationFlag(D, mappedPE_A, true);
10         else
11             D_2 = getJobNotifyingFromPE(mappedPE_D, A);
12             position_D2 = getJobPositionInSchedule(D_2, SM);
13             position_D = getJobPositionInSchedule(D, SM);
14             if position_D2 < position_D then
15                 setNotificationFlag(D_2, mappedPE_A, false);
16                 setNotificationFlag(D, mappedPE_A, true);
```

Figure 6.10 shows the schedule, equivalent to Figure 6.9, of the graph of Figure 6.8a but with Algorithm 6 applied to it before the start of the execution. Compared to the schedule of Figure 6.9, there are significantly less notifications between both LRTs with only two notifications left. Indeed, in this scenario,both actors C_0 and D_1 have one of their dependencies on the same LRT as they are mapped to (A_0 for actor C_0 and E_0 for actor D_1) and the other one on the second LRT. Hence no notifications is required for the first dependency and only the dependency of B_0 to C_0 and C_1 to D_1 are required.

In the next section, the JIT and delayed execution approaches are compared on different image and signal processing, and machine learning applications. Although it seems that the JIT policy will always result in low performances due to its high synchronization

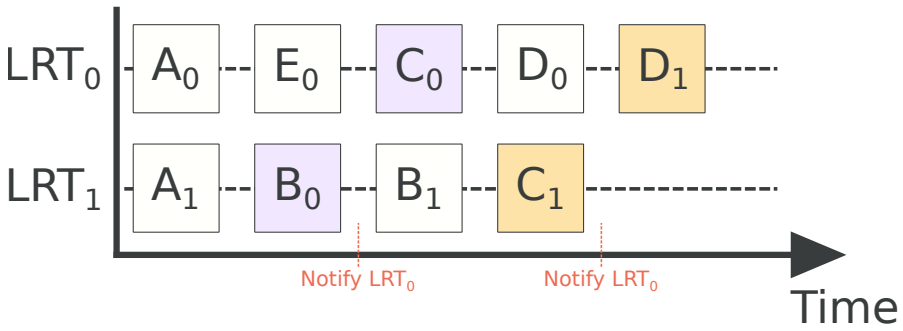


Figure 6.10 – Example of notification synchronization when using a delayed execution after schedule approach. Compared to Figure 6.9, there is less notification *noise* for unnecessary synchronizations.

cost, this not the case as shown in the results of the next section. Moreover, in reconfigurable applications, the JIT policy could yield better performances compared to the delayed policy due to the possibility of having configure actors running sooner and thus, having less wait for the reconfiguration phases.

6.3.3.3 Notification rate Results

In this section, the JIT execution and the delayed execution policies are compared. The average total number of notification messages exchanged along with the average number of Iterations per seconds (IPS) for 5 different image and signal processing applications are measured. The experiments are done on an x86 Intel Core i7-7820HQ Laptop processor with 4 cores and equipped with 32 GB of RAM memory. All applications have been compiled with O2 level of optimization using GCC 9.2.1 version.

The SqueezeNet, Reinforcement Learning and Stabilization applications are the same image processing and machine learning applications as the ones used in the Chapter 5. The 6-Steps Fast-Fourier Transform (FFT) application is a signal processing application used in [**heulot_runtime_2015**], that performs a Discrete Fourier Transform (DFT) based on the algorithm described in [**bailey_ffts_1990**]. Although the original 6-Steps FFT of [**heulot_runtime_2015**] is a reconfigurable application, in these experiments, the value of the parameters were fixed and the FFT used was of size 65536. Finally, the Adaptive Filtering application is a reconfigurable application that works as follows:

1. Read an RGB image from a camera and convert it to the YUV format.
2. Compute the average brightness level of each of the three components.

3. For each of the luminance (Y) and chrominances (UV) components of the image, if the average brightness of the component is below a given threshold go to step 6, else go to step 4.
4. Apply a sobel operator onto the component.
5. Apply a sharpening filter using a median filter of size 3x3 onto the sobelized image.
6. Merge the different filtered, and unfiltered, components into one image.
7. Display the merged image.

In the following experiments, due to the highly content dependent nature of the adaptive filtering application, the decision making of step (3) is replaced by a uniform random decision to ensure that the processing load of the application do not depend on the experimental setup. Additionally, the application is then ran 10 times and the average value of the number of exchanged notification of the different runs is taken, along with the standard deviation of these values.

Table 6.4 – Number of average notification messages exchanged during 100 graph iterations of various image and signal processing application, depending on the execution policy used.

	JIT-Execution		Delayed-Execution	
Application	#N	IPS	#N	IPS
SqueezeNet	1634757	5.8	**18744**	**5.9**
Reinforcement Learning	126253	945	**27830**	**1220**
Stabilization	30449	81	**4401**	**83**
6-Steps FFT	20398	**570**	**10460**	550
Adaptive Filtering	57842 ± 36.44	30.22 ± 0.34	**31399** ± 14.73	30.35 ± 0.15

Table 6.4 shows the obtained results for the different applications using both the JIT and delayed execution approaches. As expected, the number of notifications exchanged during the lifetime of the applications is significantly lower with the delayed execution approach compared to the JIT. Indeed, as seen in Section 6.3.3.2, when using the delayed execution, all synchronizations between actors are optimized depending on which LRTs they are running on, whereas the JIT execution has no vision of the future and thus require a lot more message exchange.

However, this significant difference in synchronization points does not reflect on the actual performances of each application with almost no difference in the number of IPS between both execution policies, except for the Reinforcement Learning application. The difference in IPS for the Reinforcement Learning application is explained by the fact that the majority of the computation kernels of the application are very simple, being at most 30 multiplications and additions, and thus, the cost of synchronization is higher than the computation itself.

For the rest of the applications, this seems to be linked to the fact that x86 processors have a very low overhead for mutexes and condition variables and that the cost of synchronization is lower than the cost of putting a thread to sleep and waking it up. However, these are late results obtained during the writing of this thesis and no further investigation was conducted on this matter.

Consequently, on the x86 platform it seems that both method are suitable for synchronization between LRTs. However, on a many core platform like the MPPA platform used in [**miomandre_embedded_2018**], the delayed execution approach might yield better performance due to less congestion of the Network On Chip (NoC) used for the communication of the different PEs cluster.

6.4 Conclusion

Developing new dataflow MoCs requires developing associated tools for supporting them. These tools often serve as a demonstration of what it is possible to do, or not, with a MoC. However, the implementation of the tools is often left aside in the literature, and it is not uncommon to face abandoned projects that were developed only for supporting the publication of a given MoC. Additionally, when targeting embedded platforms the implementation of a dedicated tool needs to be carefully thought out in order not to introduce any overhead that would invalidate the potential benefits of using a given MoC.

In this chapter, we presented the Spider 2.0 runtime, the spiritual son of the Spider runtime. Spider 2.0 is an open source library that is thought to be used as an experimental platform for the πSDF MoC. Spider 2.0 directly inherits the global runtime structure from Spider but improve on some aspect of its implementation. Spider 2.0 embeds a state-of-the art expression parser that is dedicated to the evaluate the dynamic expression of the πSDF MoC as fast as possible. When compared to the fastest available expression parsing library, we showed that our proposed JIT-compiled expression is 2.19

times faster on average. Spider 2.0 also improves on the maintainability of the library source code with a new intermediate transformation that removes the need for dedicated partial specializations of the SR-DAG transformation and scheduling algorithms.

The Spider 2.0 offers more flexibility to the user than Spider with a more comprehensive API. The Spider 2.0 offers a flexible core library that aim at easing the support of different multicore platforms. Using a logical hardware model, the user can describe a given platform architecture (memory communications, number of PEs, etc.) with a dedicated API independently from the library source code. Spider 2.0 also offers debug capabilities with the possibility to output execution traces on the different PEs of the platform.

Even though, Spider 2.0 has only been tested on x86-x64 and ARM platforms, the authors are confident that supporting new platforms as a future work should not be a problem. Preliminary results on the Jetson TX-2 show the capability of Spider 2.0 to handle ARM + GPU platforms.

Indeed, during the writing of this thesis a preliminary work of supporting a CPU + GPU platform was made. In this preliminary work, the platform has been modeled as a 2 cluster architecture with one cluster of 5 CPUs and 1 cluster of 1 GPU using the hardware model and API of Spider 2.0. The GPU is handled by an LRT running on 1 of the 6 CPU cores of the Jetson platform, using the official NVIDIA CUDA library specific memory primitive to allocate and transfert data from the CPU cluster to the GPU one.

CHAPTER 7

Conclusion

In the recent years, the evolution of embedded systems has resulted in more and more heterogeneous platforms integrating multiple type of processing elements into one MPSoC. At the same time, the rise of new types of computational heavy applications such as deep learning applications have increased the demand for more efficient and specialized systems as well as more efficient and high-level programming techniques of these systems.

For these purposes, the dataflow Models of Computation (MoCs) have emerged as a popular programming technique to unveil and exploit the parallelism of applications as demonstrated by the emergence of programming framework such as the OpenVX [**kronos_group_openvx_2013**] or the tensorflow [**abadi_tensorflow:_2016**] frameworks. Dataflow MoCs have proven efficient means for modeling computational aspects of Cyber-Physical Systems due to their natural expression of parallel processing pipelines and their high level of abstraction which benefit both to software and hardware acceleration. Over the years, diverse Models of Computation (MoCs) have been proposed that offer trade-offs between expressivity, conciseness, predictability, and reconfigurability.

7.1 Research Contributions

In this thesis, contributions are made to increase the expressiveness and compactness of dataflow MoCs along increasing the efficiency of their analysis for resources allocation

in a context of scarce resources available. Finally, an embedded runtime system called SPIDER 2.0 and dedicated to the support the πSDF MoC was developed during this thesis to address the issue of managing complex dataflow based application onto MPSoCs.

7.1.1 Dataflow Model Extension

Chapter 4 introduces a new meta-model called SaMM which brings functional aspects into the model space. SaMM can be applied to a dataflow MoC to increase its expressiveness and its compactness.

SaMM introduces new semantics for dynamically initializing delays using dataflow actors. This new semantics of initialization of delays brings a more compact representation of iterative patterns to dataflow MoCs while offering more memory optimizations opportunities when compared to current state-of-the-art dataflow MoCs. The new representation is also well suited in cases such as high level synthesis for FPGAs with more direct translation of the input graph to the synthesized hardware. Indeed, as shown in [**serot_implementing_2011**] and [**serot_high-level_2014**], dataflow MoCs are efficient for modeling streaming applications on FPGAs both as an high level programming model and as an execution model.

Secondly, SaMM extends the notion of states of hierarchical dataflow graphs with explicit state persistence and state forwarding. Indeed, in SaMM it is possible to fine tune the degree of task parallelism of a given hierarchical graph by selecting the degree of persistence of its inner state. The state of a hierarchical dataflow graph is handled the delay belonging to the graph.

7.1.2 Dataflow Graph Dependencies Analysis

In Chapter 5, a new Intermediate Representation (IR) of SDF graphs for resource allocation is proposed. The intermediate representation is then extended to the πSDF MoC.

The new proposed IR is composed of the original dataflow graph combined with an ad-hoc numerical model of the dependencies of the graph. The proposed IR has been implemented into the SPIDER tool [**heulot_runtime_2015**] as a proof of concept for the resource allocation of image processing and machine learning applications. On the set of tested applications, the results show that our proposed IR reduces the memory overhead of the runtime by 97.33% in average while offering a speed up of the resource allocation algorithm of a factor up to more than 18.

7.1.3 Spider 2.0: A Dataflow Runtime

In Chapter 6, the Spider 2.0 runtime is presented. Spider 2.0 is an efficient and open-source framework for experimenting with dataflow analysis algorithms for reconfigurable applications. The Spider 2.0 runtime uses the πSDF MoC extended with the SaMM delay semantics. The runtime has been thought to be application and platform independent. Indeed, it relies on an internal logical hardware model for abstracting the physical platform on which it runs.

Some optimizations implemented into the runtime are presented in Chapter 6 that reduces its overall overhead such as a very fast and efficient expression parser (and evaluator) for the dynamic parameter expressions of the πSDF MoC. The expression parser method employed in Spider 2.0 uses a JIT compilation approach, which results in close to natively compiled code results in term of speed of evaluation.

Secondly, Spider 2.0 also embeds a pre-process intermediate transformation that enforces the execution rules of the πSDF MoC which simplifies the analysis algorithms used inside the framework by removing any partial specialization of these algorithms compared to the original Spider runtime.

Finally, the asynchronous system of notifications used for the synchronization of the different LRTs was discussed, with a proposed algorithm for optimizing the number of notifications to its minimal possible value. Interestingly, results show that less notifications does not yield significantly better performances on x86 processors. Indeed, it seems that more notifications avoid the different threads of the runtime of getting into IDLE mode too often which outweighs the cost of the synchronizations itself. However, due to a lack of time to investigate this phenomenon further, this remark is solely based on empirical observations.

7.2 Prospects – Future Works

The work presented in this document opens opportunities for future work, especially in the improvement of the runtime capabilities of the proposed Spider 2.0 runtime. This section outlines some the improvements the authors think could be made to the existing runtime.

7.2.1 Just-In-Time Reconfiguration

In Chapter 6, we presented the Spider 2.0 runtime. A fast and lightweight runtime supporting the dataflow πSDF MoC [**desnos_pimm:_2013**]. The πSDF MoC is a reconfigurable dataflow MoC, which means that the graph topology of a given application graph is not completely fixed and can change during the execution. The πSDF reconfigurability directly comes from the dynamic values of application parameters. As of today, even if no parameter changes between two executions of a reconfigurable graph, the runtime still need to recompute the graph transformations, the mapping and the scheduling of the graph.

In a future work, it could be interesting to integrate a system of cache for the values of dynamic parameters. For a given reconfigurable and hierarchical graph, there can be multiple reconfiguration points; the cache system could store the state of each of these reconfiguration points (mapping and scheduling and intermediate transformation of the application graph) with the corresponding parameter values. On the next iteration, the execution would start using the previously cached states until one of the dynamic parameters value changes compared to the cached values. In this scenario, the cache would be invalidated from this reconfiguration point up to the end of the graph iteration.

Such a cache system would probably lead to better performances in application with a low rate of parameters change. However, in applications where parameters change their value at every iteration of the application graph, such a cache would be counter productive by adding an extra overhead. Adding a detection of parameters change rate could, however, automatically enable or disable the cache system depending on the current context of the application (low parameters change rate or high parameters change rate).

7.2.2 Dynamic Memory Compression

In dataflow graphs, synchronization between actors is done based on the exchange of data tokens and in the case of complex heterogeneous architecture, it is often needed to move data from one memory space to another. Moving or copying memory is often one of the most expensive operation within an embedded system. The proposed future work would be to investigate the possibility for the runtime to automatically compress and decompress on-the-fly data in order to accelerate data transmission between LRTs of the platform and to reduce IOs operations.

Online compression is an active research field with popular algorithms such as the LZO [**oberhumer_lz0_nodate**], the LZ4 [**collet_realtime_nodate**] or the ACE method [**krintz_adaptive_2006**]. Recently, with the development of more and more powerful gaming console such as the Playstation 4, or the Xbox One, there have been development focused on algorithms with high decompression speed such as the Kraken algorithm [1]. Although thought to be used directly within software with compression and decompression phases handled by CPUs, these algorithms have also seen specific hardware implementations [**bartik_lz4_2015**].

With dedicated hardware support, or very fast compression and decompression algorithms, the Spider 2.0 runtime could handle applications with higher memory usage (such as ultra HD video input) on architectures with small local memory available. For instance, on the Kalray MPPA architecture where each cluster only possess less than 2MB of local memory, dynamic decompression could lead to less wait for data of the PEs inside the cluster. Indeed, with 16 PEs per cluster of the MPPA, 2MB of memory is not always sufficient for all PEs to work simultaneously.

7.2.3 Debugging Capabilities

Dataflow MoCs naturally express data parallel applications with an high level of abstraction. However, as any parallel type of parallel programming paradigm, debugging a dataflow application can be tedious for the least.

As presented in Chapter 6, the Spider 2.0 runtime is an experimental framework for the prototyping of dataflow applications. Adding dataflow debugging capabilities to such a framework would increase programmer productivity and efficiency. Although, Spider 2.0 already proposes a *verbose* mode displaying extensive execution traces of the internal functioning of the runtime, these traces are not easy to interpret and thus, here after are noted some future lead for improvements.

- **Dataflow Breakpoints**: almost all programming languages possess the notion of execution breakpoints which allows the programmer to freeze the execution of an application under certain conditions.
  For dataflow graph, breakpoints could correspond to freezing the entire application when the graph execution reaches a given actor and for a given firing of the actor, or based on the content of a given Fifo of the graph. The runtime would then

1. **http://www.radgametools.com/oodlekraken.htm**.

automatically handle the freeze of all LRTs and gives the user a command line (or graphical) interface to inspect the current execution state of the dataflow graph and LRTs.

- **Sandboxed Actor Executions**: One of the issue when designing complex dataflow graphs is the possibility to individually test an actor of a graph as it often relies on the data production of its predecessors. Adding the possibility to isolate an actor within a graph and execute it with buffers containing pre-computed, or randomly generated, values without having to generate dedicated graphs would greatly improve programmer productivity.
- **Dynamic Execution Traces**: In the current implementation of the runtime, it is possible to output execution traces but either the execution traces information appear in the console directly or it is necessary to wait the end of a graph iteration to use an external tool. Having the possibility to graphically analyze in real time the state of the runtime, specifically the load of each LRT, the memory usage, the system calls execution traces (memory allocation time for instance) or the current mapping of the application, would add great value to an experimental tool such as Spider 2.0.

APPENDIX A

French Summary

A.1 Introduction

Notre société moderne est entourée d'une myriade de systèmes embarqués : des simples thermostats dans nos maisons, aux podomètres connectés, en passant par des systèmes plus complexes tels que nos téléphones, nos voitures et les systèmes très complexes et critiques de navigation automatisés dans les avions. Le nombre de systèmes entrant dans notre vie quotidienne devrait continuer à augmenter dans les prochaines années, avec une croissance prévue de 6,1 % entre 2020 et 2025 [**marketsandmarkets_embedded_2020**].

A.1.1 Problématiques

La conception et la programmation de systèmes embarqués nécessitent la prise en compte de contraintes fonctionnelles et non fonctionnelles. Dans [**desnos_memory_2014**], l'auteur classe les contraintes appliquées aux systèmes embarqués en trois catégories, à savoir les contraintes d'application, les contraintes de coût et les contraintes externes. Dans la classification que nous proposons, nous divisons les contraintes d'application en contraintes fonctionnelles et en contraintes de conception.

Toutes ces contraintes peuvent déboucher sur des solutions orthogonales. Par exemple, trouver un la solution optimale sur le plan énergétique pourrait entraîner une contrainte de consommation de mémoire sous-optimale ou dans un coût plus élevé du système. Par conséquent, la conception d'un système embarqué le plus souvent se résume à trouver et à accepter le compromis le plus approprié, ou le plus acceptable, entre toutes les solutions possibles. Le processus de recherche de ce compromis est souvent désigné sous le nom de l'Exploration de l'Espace de Conception (EEC) d'un système et a été un domaine de recherche actif avec le développement d'outils dédiés.

Parallèlement, la célèbre loi de Moore [**moore_cramming_1998**] qui prévoit que le nombre de transistors dans un circuit intégré doublerait tous les deux ans arrive à son terme. Au cours des deux dernières décennies, une nouvelle loi appelée "Plus que Moore" est apparue [**zhang_more_2009** ; **waldrop_chips_2016**] afin de maintenir la loi de Moore en vigueur en tirant parti de l'hétérogénéité des systèmes modernes. Afin de tenir compte de cette hétérogénité, ces dernières années ont vu le développement de langages de programmation de haut niveau et de techniques de prototypage rapide pour combler le fossé de la productivité des logiciels. L'un des paradigmes de programmation visant à accroître la productivité des programmeurs est celui de la programmation basée flot de données.

Les Modèles de Calcul (MdCs) flot de données (MoCs) sont un paradigme de programmation qui a été développé afin de capturer naturellement le parallélisme des applications. Dans les MdCs flot de données, les applications sont décrites comme étant des graphes où les sommets sont des entités de calcul de différents niveaux de granularité, du niveau de granularité très fin correspondant aux instructions élémentaires (ex : addition, multiplication, etc.) au niveau de granularité élevé correspondant à des traitements complexe (ex : convolution) ; et les arêtes sont des canaux de communication de données. De nombreux MdCs flot de données ont été proposés depuis le travail initial de Kahn en 1974 [**kahn_semantics_1974**]. Chaque MdC flot de données proposé est accompagné de sa propre sémantique et de ses propres règles d'exécution qui sont conçues pour l'expressivité, la compacité ou l'efficacité du modèle. Par exemple, les modèles statiques tels que le SDF [**lee_synchronous_1987**] et ses dérivés sont conçus pour modéliser des applications statiques avec de fortes garanties d'exécution telles que la limitation de la mémoire ou l'absence de blocage. À l'inverse, les MdCs flot de données dynamiques ont la même expressivité que des machines de Turing [**buck_scheduling_1993** ; **lee_dataflow_1995**]. Par conséquent, ces modèles ne peuvent pas dériver statiquement

des ordonnancements, ni vérifier la consommation mémoire nécessaire pour l'exécution infinie d'une application ; et ont besoin de décaler ces analyses au moment de l'exécution. La popularité des MdCs flot de données vient de leur abstraction de l'implémentation sous-jacente des noyaux de calcul et, par conséquent, de leur compatibilité avec le code existant.

A.1.2 Plan

Cette thèse est organisée en deux parties distinctes : La première partie présente les concept et problématiques de recherche étudiées dans cette thèse, et la deuxième partie présente les différentes contributions de cette thèse.

Dans la partie 1, le Chapitre 2 présente les concepts des Modèles de Calcul (MdC) flot de données et présente spécifiquement le modèle utilisé au cours de cette thèse. Ensuite, le Chapitre 3 présente les différents problèmes liés au design d'un manageur applicatif embarqué pour des plateformes multi processeurs hétérogènes.

Dans la partie 2, certaines limitations des MdCs flot de données existant sont présentées et un méta-modèle appelé SaMM est proposé dans le Chapitre 4 pour répondre à certaines de ces limitations. Ensuite, le Chapitre 5 introduit une nouvelle Représentation Intermédiaire (RI) pour graphes flot de données conçu spécifiquement pour l'allocation de ressources matérielle dynamique dans un contexte embarqué. Le Chapitre 6 présente un manageur applicatif pour plateformes multi processeurs hétérogène développé au cours de cette thèse, et basé sur une version pré-existante du manageur SPIDER [**heulot_runtime_2015**]. Enfin, le Chapitre 7 conclut cette thèse.

A.2 État de l'Art

A.2.1 Modèles de Calcul Flots de Données

En informatique, un Modèle de Calcul (MdC) décrit l'ensemble des opérations et des règles élémentaires qui définissent comment un programme informatique est exécuté. Comme les lois de la physique qui régissent le fonctionnement de notre univers et l'interaction entre les objets qui le composent, un MdC régit le fonctionnement d'un programme et les interactions entre les éléments du MdC. Il est toutefois important de faire la différence entre les MdCs et les langages de programmation. En effet, un langage de programmation peut être utilisé pour mettre en œuvre différents MdCs. Par exemple, il est possible d'im-

plémenter une machine de Turing en langage C, mais il est également possible d'utiliser ce langage pour implémenter n'importe quel MdC flot de données.

Modèle de flot de données synchrone (SDF)

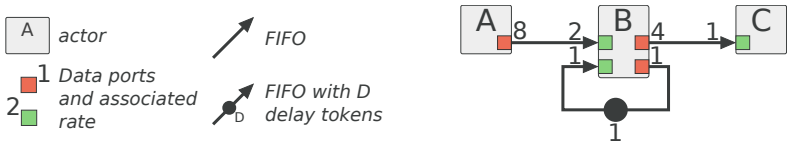


FIGURE A.1 – Modèle de Calcul SDF.

Le MdC flot de données synchrone (SDF) [**lee_synchronous_1987**] est la spécialisation statique la plus populaire du MdC DPN [**lee_dataflow_1995**]. Les règles d'exécution du MdC SDF définissent les taux de production et de consommation des jetons de données des acteurs comme étant des scalaires fixes, ce qui signifie que les taux sont fixés au moment de la conception du graphe et sont fixés pour toute l'exécution de l'application. La sémantique graphique du MdC SDF et un exemple de graphique SDF sont présentés dans la Figure A.1.

Formellement, un graphique SDF $G = (A, F)$ contient un ensemble d'acteurs A qui sont interconnectés par un ensemble de Fifos F. Un acteur $a \in A$ lit des jetons de données sur ses ports d'entrée et produit des jetons de données sur ses ports de sortie. L'exécution d'un acteur s'appelle une *exécution* et pour qu'un acteur puisse s'exécuter, il faut qu'il y ait suffisamment de jetons de données disponibles sur tous ses ports d'entrée. Dans le graphique de la Figure A.1, l'acteur B ne peut s'exécuter que lorsque 2 jetons de données sont présents sur la Fifo et 1 jeton de données est présent sur sa boucle. Les jetons de données initiaux d'une Fifo $f \in F(AB)$ sont appelés des *délais*. La valeur n du délai est le nombre de jetons de données initiaux de f.

La popularité du MdC SDF vient de sa grande capacité d'analyse. En effet, à l'aide d'analyses statiques, les propriétés de consistance et de vivacité d'un graphe SDF peuvent être vérifiées. Lorsqu'un graphe SDF est ordonnançable, c'est-à-dire qu'il est consistant et vivant, une séquence minimale, et répétable à l'infini, d'exécutions des acteurs le composant existe en mémoire fini . Cette séquence minimale est appelée une itération de graphe et le nombre d'exécution de chaque acteur est donné par les coefficients du Vecteur de Répétition (VR) du graphe. La Figure A.1 présente un graphe SDF consistant et vivant. Pour chaque itération du graphe, l'acteur A est exécuté 1 fois, l'acteur B 4 fois, et l'acteur C 16 fois.

Modèle de flot de données Synchrone, Interfacé et Paramétré (πSDF)

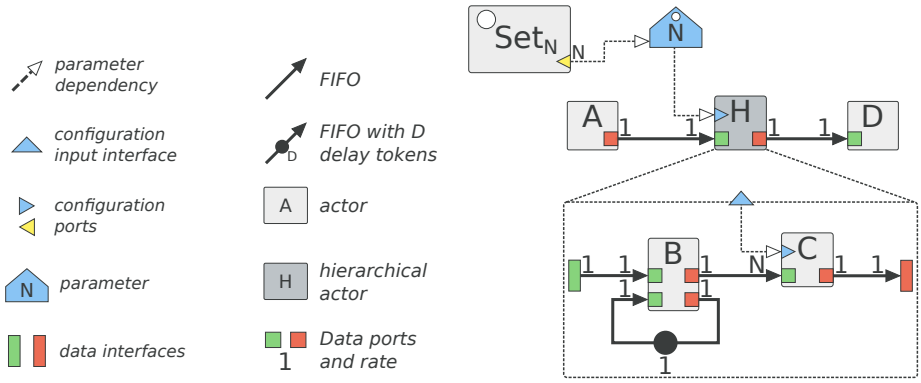


FIGURE A.2 – Modèle de Calcul πSDF.

Le MdC flot de données Synchrone, Interfacé et Paramétré (πSDF) [**desnos_pimm:_2013**] est une extension hiérarchique et reconfigurable dynamiquement des MdCs SDF et IBSDF [**piat_interface-based_2009**]. De même que dans le MdC IBSDF, dans un graphe πSDF, un acteur hiérarchique est un acteur dont le comportement interne est défini par un graphe πSDF. La Figure A.2 présente un exemple de graphe πSDF avec la sémantique graphique associée. L'acteur H est un acteur hiérarchique défini par le sous-graphe formé par les acteurs B et C. Formellement, un graphe πSDF $G = (A, F, I, \Pi, \Delta)$ contient en plus d'un ensemble d'acteurs A et un ensemble de Fifos F, un ensemble d'interfaces hiérarchiques I, un ensemble de paramètres Π, et un ensemble de dépendances de paramètres Δ. Les interfaces hiérarchiques du MdC πSDF sont directement héritées du MdC IBSDF. Cet héritage direct des interfaces fait du MdC πSDF un MdC compositionnel, ce qui signifie que la spécification interne des acteurs composant un graphe n'influence pas son analysabilité. Dans la Figure A.2, la définition du sous-graphe formé par les acteurs B et C n'a pas d'impact sur l'analyse effectuée sur le graphe de niveau supérieur.

Les paramètres $\pi \in \Pi$ sont associés à des valeurs de paramètres $v \in N$. Les valeurs de paramètres peuvent soit être définies statiquement, dérivées d'autres paramètres, ou définies dynamiquement par les acteurs de configuration au moment de l'exécution. Un paramètre défini dynamiquement est appelé paramètre configurable et sa valeur est fixée par un acteur de configuration au moment de l'exécution. La valeur d'un paramètre configurable n'est fixée qu'une seule fois par itération du graphe auquel il appartient.

Par exemple, dans la Figure A.2, le paramètre N au niveau supérieur du graphe est un paramètre configurable défini par l'acteur de configuration Set_N.

Les dépendances $\delta \in \Delta$ sont des liens dirigés du graphe qui propagent les valeurs de paramètres vers et depuis les ports d'entrée et de sortie de configuration des acteurs et des paramètres. Pour les acteurs hiérarchiques, les ports de configuration sont également appelés interfaces de configuration et sont considérés comme des paramètres statiques à l'intérieur du sous-graphe associé. La combinaison des paramètres Π et des dépendances Δ forme ce que l'on appelle un arbre de dépendance des paramètres $T = (\Pi, \Delta)$. Les dépendances de paramètres ne suivent pas les mêmes règles de synchronisation et de priorité que les Fifos d'un graphe flot de données. En effet, dans le MdC πSDF, les paramètres sont mis à disposition, virtuellement, instantanément à chaque dépendance de données connectée dès que leur valeur est fixée.

La reconfigurabilité du MdC πSDF provient directement des paramètres dont les valeurs sont utilisés pour influencer différentes propriétés, à savoir le coeur de calcul d'un acteur, les taux de production ou de consommation des ports de données d'un acteur, la valeur d'un autre paramètre et le nombre de délais dans une Fifo. Il est important de noter que si les taux de production et de consommation de données d'entrée et de sortie d'un acteur A est égal à 0, alors A ne sera pas exécuté. La combinaison des taux de production et de consommation de données paramétrés et de la propriété de non-exécution des acteurs donne la possibilité de modifier la topologie d'un graphe πSDF de manière dynamique. Par exemple, un chemin de données peut être complètement désactivé, et donc non pris en compte par l'analyse de consistance et de vivacité. Dans la Figure A.2, le paramètre N est défini par l'acteur Set_N et contrôle le nombre de répétitions de l'acteur B à l'intérieur de l'acteur hiérarchique H mais, en raison de la nature compositionnelle du MdC πSDF, n'affecte pas l'analyse du graphe de niveau supérieur.

A.3 Extension de Modèles Flot de Données

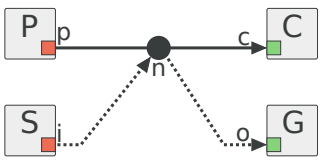


FIGURE A.3 – Nouvelle sémantique d'initialisation des délais.

Le Chapitre 4 présente un nouveau méta-modèle appelé State-Aware dataflow Meta-Model (SaMM) qui introduit des aspects fonctionnels dans l'espace des modèles de calcul. SaMM peut être appliqué à un MdC flot de données pour en augmenter l'expressivité et la compacité. SaMM introduit une nouvelle sémantique pour l'initialisation dynamique des délais en utilisant des acteurs du graphe flot de données directement, présentée en Figure A.3. Cette nouvelle sémantique d'initialisation des délais apporte une représentation plus compacte des structures itératives (par exemple des boucles) aux MdC flot de données tout en offrant plus d'opportunités d'optimisations mémoire. La nouvelle représentation est également bien adaptée dans des cas tels que la synthèse de haut niveau pour les FPGAs avec une traduction plus directe du graphe applicatif d'entrée vers le matériel synthétisé. En effet, comme indiqué dans [**serot_implementing_2011**] et [**serot_high-level_2014**], les MdC flot de données sont efficaces pour modéliser les applications de flux de données sur les FPGAs à la fois comme modèles de programmation de haut niveau et comme modèles d'exécution.

Dans un second temps, SaMM étend la notion d'états des graphes flot de données hiérarchiques avec une persistance d'état et une transmission d'état explicites. En effet, dans SaMM, il est possible d'affiner le degré de parallélisme des tâches d'un graphe hiérarchique donné en sélectionnant le degré de persistance de son état interne. L'état d'un graph de flot de données hiérarchique étant géré uniquement en utilisant les délais appartenant à ce graphe.

A.4 Analyse de Modèles Flot de Données

Dans le Chapitre 5, une nouvelle Représentation Intermédiaire (RI) des graphes SDF pour l'allocation des ressources est proposée. La représentation intermédiaire est ensuite étendue au MdC πSDF. La nouvelle RI proposée est composée du graphe flot de données original combiné à un modèle numérique ad-hoc des dépendances de données. La RI proposée a été mise en œuvre dans l'outil Spider [**heulot_runtime_2015**] comme preuve de concept pour l'allocation des ressources d'applications de traitement d'images et d'apprentissage automatique. Sur l'ensemble des applications testées, les résultats montrent que la RI proposée réduit l'occupation mémoire du manageur d'applications de $97,33\%$ en moyenne tout en offrant une accélération de l'allocation des ressources d'un facteur allant jusqu'à 18.

A.5 Spider 2.0 : Un Manageur de Ressource d'Applications Basées Flot de Données

Dans le Chapitre 6, le manageur applicatif Spider 2.0 est présenté. Spider 2.0 est un environnement open-source permettant d'expérimenter des algorithmes d'analyse sur des graphes flot de données pour des applications reconfigurables. Le manageur applicatif Spider 2.0 utilise le MdC πSDF étendu avec la sémantique de SaMM. Le manageur a été pensé pour être indépendant des applications et des plateformes sur lequel il est exécuté. En effet, il s'appuie sur un modèle interne matériel our abstraire la plateforme physique sur laquelle il fonctionne.

Certaines optimisations mises en œuvre dans le manageur sont présentées au Chapitre 6, qui réduisent sa charge globale, comme un analyseur (et un évaluateur) d'expressions mathématiques très rapide et efficace pour les expressions de paramètres dynamiques du MdC πSDF. La méthode de l'analyseur d'expressions utilisée dans Spider 2.0 utilise une approche de compilation Juste à Temps (JAT), qui donne des résultats proches des résultats du code compilé en natif.

Deuxièmement, Spider 2.0 intègre également une transformation intermédiaire qui applique les règles d'exécution du MoC πSDF, ce qui simplifie les algorithmes d'analyse utilisés en supprimant toute spécialisation partielle de ces algorithmes par rapport à l'ancienne version du manageur.

Enfin, le système asynchrone de notifications utilisé pour la synchronisation des différents Manageurs Applicatif Local (MAL) a été discuté, avec une proposition d'algorithme pour optimiser le nombre de notifications. Il est intéressant de noter que les résultats montrent que moins de notifications ne donne pas de performances significativement meilleures sur les processeurs x86. En effet, il semble qu'un plus grand nombre de notifications évite aux différents fils du temps d'exécution de passer trop souvent en mode repos, ce qui dépasse le coût des synchronisations elles mêmes. Cependant, faute de temps pour approfondir ce phénomène, cette remarque est uniquement basée sur des observations empiriques.

A.6 Conclusion

Dans cette thèse, des contributions sont faites pour améliorer l'expressivité et la compacité des Modèles de Calcul (MdC) basé flot de données tout en augmentant l'efficacité

de leur analyse pour l'allocation des ressources dans un contexte de ressources peu disponibles des systèmes embarqués. Enfin, un manageur de ressources intégré appelé Spider 2.0 et dédié au support du MdC πSDF a été développé au cours de cette thèse afin d'aborder la question de la gestion des applications complexes basées flot de données sur plateformes hétérogènes multi-processeurs embarqués.

ANNEXE B

Platform API

This appendix presents the list of functions available in the platform API of SPIDER 2.0. Using all of these functions, the programmer can describe heterogeneous platforms with specific communication routines between different memory space. It is also possible to define exchange costs between two memory units based on the size of the data exchanged ; these costs will be used at runtime to take mapping and scheduling decisions.

```
1  #ifndef SPIDER2_ARCHI_API_H
2  #define SPIDER2_ARCHI_API_H
3
4  /* === Includes === */
5
6  #include <cstdint>
7  #include <string>
8  #include <api/global-api.h>
9
10 namespace spider {
11
12     inline uint64_t defaultC2CZeroCommunicationCost(uint32_t, uint32_t, uint64_t) {
13         return 0;
14     }
15
16     inline uint64_t defaultZeroCommunicationCost(uint64_t) {
17         return 0;
18     }
19
20     inline uint64_t defaultInfiniteCommunicationCost(uint64_t) {
21         return UINT64_MAX;
22     }
23
24     /* === Type definition(s) === */
25
26     /**
27     * @brief Memory exchange cost routine (overridable).
```

```
28      */
29      using MemoryExchangeCostRoutine = std::function<uint_least64_t(uint_least64_t /* = Number of bytes = */)>;
30
31      /**
32      * @brief Memory bus send / receive routine.
33      */
34      using MemoryBusRoutine = std::function<void(int_least64_t  /* = Size in bytes = */,
35                                                  void *,        /* = Buffer to send / receive = */
36                                                  void *         /* = Buffer to send / receive = */)>;
37
38      /**
39      * @brief Data memory allocation routine (overridable).
40      *        This should return the allocated buffer.
41      */
42      using MemoryAllocateRoutine = std::function<void *(uint_least64_t /* = Number of bytes = */)>;
43
44      /**
45      * @brief Data memory deallocation routine (overridable).
46      */
47      using MemoryDeallocateRoutine = std::function<void(void * /* = physical address to free = */)>;
48
49      /* === Function(s) prototype === */
50
51      namespace archi {
52          /**
53          * @brief Get the unique platform of the spider session.
54          * @return reference pointer to the platform.
55          */
56          Platform *&platform();
57          }
58
59          namespace api {
60
61              /* === General Platform related API === */
62
63              /**
64              * @brief Create a new Platform (only one is permitted)
65              * @param clusterCount   Number of cluster in the platform (1 by default).
66              * @param totalPECount   Total number of PE in the platform (1 by default).
67              * @return pointer to the newly created @refitem Platform
68              * @throws @refitem Spider::Exception if a platform already exists.
69              */
70              Platform *createPlatform(size_t clusterCount, size_t totalPECount);
71
72              /**
73              * @brief Set the Global Run-Time (GRT) PE.
74              * @param grtProcessingElement  Processing Element of the GRT.
75              */
76              void setSpiderGRTPE(PE *grtProcessingElement);
77
78              /* === MemoryUnit related API === */
79
80              /**
81              * @brief Create a new MemoryInterface.
82              * @param size  Size of the MemoryInterface in bytes.
83              * @return Pointer to newly created @refitem MemoryInterface.
84              */
85              MemoryInterface *createMemoryInterface(uint64_t size);
86
87              /**
88              * @brief Override the allocate routine of a given @refitem  MemoryInterface.
89              * @param interface  Pointer to the @refitem MemoryInterface.
90              * @param routine    Routine to set.
91              */
92              void setMemoryInterfaceAllocateRoutine(MemoryInterface *interface, MemoryAllocateRoutine routine);
93
94              /**
95              * @brief Override the deallocate routine of a given @refitem  MemoryInterface.
96              * @param interface  Pointer to the @refitem MemoryInterface.
97              * @param routine    Routine to set.
```

```
98              */
99              void setMemoryInterfaceDeallocateRoutine(MemoryInterface *interface, MemoryDeallocateRoutine routine);
100
101             /**
102             * @brief Creates a new @refitem MemoryBus.
103             * @param sendRoutine     Routine used for sending data on this bus.
104             * @param receiveRoutine  Routine used for receiving data on this bus.
105             * @return pointer to the created @refitem MemoryBus.
106             */
107             MemoryBus *createMemoryBus(MemoryBusRoutine sendRoutine, MemoryBusRoutine receiveRoutine);
108
109             /**
110             * @brief Override the send cost routine of a given @refitem MemoryBus.
111             * @param bus      Pointer to the @refitem MemoryBus.
112             * @param routine  Routine to set.
113             */
114             void setMemoryBusSendCostRoutine(MemoryBus *bus, MemoryExchangeCostRoutine routine);
115
116             /**
117             * @brief Override the receive cost routine of a given @refitem MemoryBus.
118             * @param bus      Pointer to the @refitem MemoryBus.
119             * @param routine  Routine to set.
120             */
121             void setMemoryBusReceiveCostRoutine(MemoryBus *bus, MemoryExchangeCostRoutine routine);
122
123             /**
124             * @brief Set the writing speed of a given @refitem MemoryBus.
125             * @param bus    Pointer to the @refitem MemoryBus.
126             * @param value  Writing speed in bytes/s.
127             */
128             void setMemoryBusWriteSpeed(MemoryBus *bus, uint64_t value);
129
130             /**
131             * @brief Set the reading speed of a given @refitem MemoryBus.
132             * @param bus    Pointer to the @refitem MemoryBus.
133             * @param value  Reading speed in bytes/s.
134             */
135             void setMemoryBusReadSpeed(MemoryBus *bus, uint64_t value);
136
137             /**
138             * @brief Create a @refitem InterMemoryBus associated to the communication between two @refitem Cluster.
139             * @param clusterA  Pointer to cluster A.
140             * @param clusterB  Pointer to cluster B.
141             * @param busAToB   Pointer to @refitem MemoryBus in the direction A -> B.
142             * @param busBToA   Pointer to @refitem MemoryBus in the direction B -> A.
143             * @return pointer to created @refitem InterMemoryBus.
144             */
145             InterMemoryBus *createInterClusterMemoryBus(Cluster *clusterA,
146                                                         Cluster *clusterB,
147                                                         MemoryBus *busAToB = nullptr,
148                                                         MemoryBus *busBToA = nullptr);
149
150             /* === Cluster related API === */
151
152             /**
153             * @brief Create a new Cluster. A cluster is a set of PE connected to a same memory unit.
154             * @param PECount         Number of PE in the cluster.
155             * @param memoryInterface Memory interface of the memory unit of the cluster.
156             * @return pointer to the newly created @refitem Cluster.
157             */
158             Cluster *createCluster(size_t PECount, MemoryInterface *memoryInterface);
159
160             /* === PE related API === */
161
162             /**
163             * @brief Create a new Processing Element (PE).
164             * @param hwType        S-LAM user defined hardware type.
165             * @param hwID          Physical hardware id of the PE (mainly used for thread affinity).
166             * @param cluster       Cluster of the PE.
167             * @param name          Name of the PE.
```

```
168             * @param type          Spider PE type.
169             * @param affinity      Optional thread affinity.
170             * @return Pointer to newly created @refitem ProcessingElement, associated memory is handled by spider.
171             */
172             PE *createProcessingElement(uint32_t hwType, uint32_t hwID, Cluster *cluster, std::string name,
173             PEType type = PEType::LRT, int32_t affinity = -1);
174
175             /**
176             * @brief Attach a Processing Element to a managing Local RunTime (LRT).
177             * @param pe   Pointer to the PE.
178             * @param lrt  Pointer to the LRT.
179             * @remark If the either of the pointers is null, nothing happens.
180             * @remark If the pe is already attached to an LRT or is an LRT, nothing happens.
181             * @throws @refitem spider::Exception If lrt is not a valid LRT, nothing happens.
182             */
183             void attachPEToLRT(PE *pe, PE *lrt);
184
185             /**
186             * @brief Set the SpiderPEType of a given PE.
187             * @param processingElement    Pointer to the PE.
188             * @param type  Spider::PEType to set.
189             */
190             void setPESpiderPEType(PE *processingElement, PEType type);
191
192             /**
193             * @brief Set the name of a given PE.
194             * @param processingElement    Pointer to the PE.
195             * @param name  Name of the PE to set.
196             */
197             void setPEName(PE *processingElement, std::string name);
198
199             /**
200             * @brief Enable a given PE (default).
201             * @param processingElement  Pointer to the PE.
202             */
203             void enablePE(PE *processingElement);
204
205             /**
206             * @brief Disable a given PE.
207             * @param processingElement  Pointer to the PE.
208             */
209             void disablePE(PE *processingElement);
210         }
211     }
212 }
213 #endif //SPIDER2_ARCHI_API_H
```

Listing B.1 – Platform creation API of Spider 2.0.

List of Figures

List of Tables

Listings

Glossary

πSDF Parameterized and Interfaced Synchronous DataFlow 17, 31, 36–40, 42 f., 55 f., 59 f., 62, 73, 79, 81–86, 88, 93–99, 105, 107, 109, 113, 116–119, 121 f., 125, 127 f., 130, 132, 134–140, 142, 144, 146, 148 ff., 152, 154, 156, 158–163, 170–174

SAD State-Aware Dataflow . . . . . 59 f., 66 f., 71, 74, 78, 88

Cacla Continuous Actor Critic Learning Automaton . . . . . 87 f.

Fifo First-In First-Out Queue 24 f., 29 ff., 34 f., 37, 60 f., 63 f., 66–69, 71 f., 75 f., 78 f., 81, 86, 99, 104, 106, 111, 136, 140, 155, 165, 188

FPGA Field Programmable Gate Array . . . . . 75, 77 f., 132, 162, 172

Preesm Parallel Real-time Embedded Executives Scheduling Method . . . 52, 59, 117, 128 f., 138 f.

Spider Synchronous Parameterized and Interfaced Dataflow Embedded Runtime 17, 19, 53, 55, 59, 92 f., 96, 98, 117 ff., 122–125, 128, 130, 136–139, 149, 151, 153, 159 f., 162 f., 169

Spider 2.0 Synchronous Parameterized and Interfaced Dataflow Embedded Runtime 2.0 . . . . . 17, 53, 56, 128–142, 144, 147, 149–154, 159–163, 165 f., 176, 179, 186

ADF Affine DataFlow . . . . . 31 f.

AHSDF Acyclic Homogeneous SDF 53

AI Artificial Intelligence 13

APEG Acyclic Precedence Expansion Graph 55, 93

API Application Programming Interface 50, 55 f., 96, 115, 128–131, 133 ff., 160, 176, 179, 186

BPDF Boolean Parametric Dataflow 40

CEG Consistency Equivalent Graph 66–70, 77

CLEG Consistency and Liveness Equivalent Graph . . . . . 66 f., 69, 77

CPS Cyber-Physical System 161

CPU Central Processing Unit . . . . . 13, 54 f., 131 ff., 160, 164

CSDF Cyclo-Static Dataflow . . . . . 31 f., 59, 73, 92, 127

DAG Directed Acyclic Graph . . . . . 54, 91 ff., 97, 115 f., 118, 125

DFT Discrete Fourier Transform 157

DPN Dataflow Process Network . . . . . 24 f., 28 f., 63

DSE Design Space Exploration 14, 16, 128, 136

DSP Digital Signal Processing 46

DSSF Deterministic SDF with Shared Fifos 34

FFT Fast-Fourier Transform 32, 46, 157

FIR Finite Impulse Response 32

FSM Finite-State Machine 24, 188

FSM-SADF Finite-State Machine (FSM) based SADF 127

GPD Globally Persistent Delay . . . . . 79 f., 82 f.

GPP General Purpose Processor . . . . . 45 f., 52 f., 56

GPU Graphics Processing Unit . . . . . 46, 52–56, 131 ff., 160

GRT Global RunTime . . . . . 130 f., 135, 139, 150 ff.

HEFT Heterogeneous Earliest-Finish-Time 54

HI-HTGS HMBE Integrated HTGS 96

HLS High-Level Synthesis 192

HMBE HTGS Model-Based Engine 53, 55

HMPSoC Heterogeneous Multi-Processor System on Chip . . . . . 91, 96 f., 125, 192

HPC High-Performance Computing 54, 97

HTGS Hybrid Task Graph Scheduler 55

RV Repetition Vector . . . . . 30, 38, 66 f., 69 f.

S-LAM System-Level Architecture Model . . . . . 128 f.

SA-πSDF State-Aware Parameterized and Interfaced Synchronous DataFlow 60, 79, 84–89

SADF Scenario-Aware DataFlow 127, 188

SaMM State-Aware dataflow Meta-Model . . . 17, 19, 43, 59 ff., 63, 65–69, 71, 74, 76, 79 f., 83 f., 86, 88 f., 128, 135, 138 f., 162, 169, 172 f., 192

SDF Synchronous DataFlow . . . . 15, 24, 28–34, 36, 40, 53, 55, 59 ff., 66, 68, 71 ff., 82, 92 ff., 96 f., 99, 101, 105, 115, 125, 127, 154, 162, 168, 188, 190

SDFG SDF Graph . . . . . 30, 33 f., 93 f., 96–99, 101, 105

SE Symbolic Execution 30, 67

SoC System on Chip 15

SPDF Schedulable Parametric Dataflow . . . . . 31, 36, 40 f., 92, 95

SR-DAG Single-Rate Directed Acyclic Graph . 93 f., 96–100, 104, 109, 115–122, 128, 149 f., 154, 159

UMA Uniform Memory Access 47

WCET Worst Case Execution Time 53, 55, 135

WSDF Windowed Synchronous DataFlow 55, 59, 96

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Extension et Analyse des Modèles de Calcul Flux de Données pour Systèmes Embarqués

Nom Prénom de l'auteur : ARRESTIER FLORIAN

Membres du jury :
- Madame MUNIER-KORDON Alix
- Monsieur PACALET Renaud
- Monsieur SEROT Jocelyn
- Monsieur MENARD Daniel
- Monsieur DESNOS Karol
- Monsieur RISSET Tanguy
- Monsieur JUAREZ Eduardo

Président du jury :

Date de la soutenance : 10 Novembre 2020

Reproduction de la these soutenue

☒ Thèse pouvant être reproduite en l'état
☐ Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 10 Novembre 2020

Signature du président de jury

Mme Alise Munier-Kordon

Le Directeur,

M'hamed DRISSI

DOCTORAT BRETAGNE LOIRE / MATHSTIC

INSA RENNES

Titre : Extension et Analyse des Modèles de Calcul de Type Flux de Données pour Gestionnaires de Ressources Embarqué.

Mot clés : Modèles de calcul flux de données, Système Embarqué, Modélisation d'application

Résumé : Ces dernières années, les Modèles de Calcul (MdCs) de type flux de données ont été utilisés pour la modélisation d'applications dans un large éventail de domaines tels que le traitement vidéo et audio, les télécommunications, la vision par ordinateur et l'apprentissage machine. Ces applications s'exécutant sur des Systèmes Hétérogènes Multi-Processeurs sur Puce, elles nécessitent une allocation et une gestion efficaces des ressources disponibles, à la fois lors de la compilation mais aussi lors de l'exécution. Afin de gérer des applications dynamiques dont le comportement ne peut être entièrement prévu lors de la compilation, les gestionnaires d'exécution doivent pouvoir prendre des décisions à la volée, tout en minimisant l'impact sur les performances des applications. Dans cette thèse, des contributions sont faites à la fois dans le domaine des MdCs de type flux de données et dans le domaine de la gestion des ressources des systèmes embarqués, avec l'introduction d'une nouvelle sémantique pour les MdCs flux de données et la proposition d'une nouvelle représentation intermédiare des graphes flux de données. La nouvelle sémantique proposée ajoute aux MdCs de type flux de données améliore la modélisation d'applications avec un niveau de parallélisme à fin grain ce qui peut bénéficier aux outils de synthèse de haut niveau. Enfin, la représentation intermédiaire des graphes flux de données proposée améliore grandement la consommation de mémoire et la rapidité d'exécution lors de la gestion d'applications dynamiques basées flux de données.

Title: Extension and Analysis of Dataflow Models of Computation for Embedded Runtimes.

Keywords: Dataflow Models of Computation, Embedded Runtime, Application Modeling

Abstract: In the recent years, dataflow Models of Computation (MoCs) have been commonly used to model stream processing applications in a wide range of domains such as video and audio processing, telecommunications, computer vision and machine learning. Stream processing applications running on Heterogeneous Multi-Processor System on Chips require efficient resource allocation and management, both at compile-time and at runtime. To cope with modern adaptive applications whose behavior can not be exhaustively predicted at compile-time, runtime managers must be able to take resource allocation decisions on-the-fly, with a minimum overhead on applications performance. In this thesis, contributions are made to address issues in both dataflow MoCs and runtime resource management with the introduction of the State-Aware dataflow Meta-Model (SaMM) and with a novel intermediate representation of dataflow graphs, respectively. SaMM adds an explicit semantics for fine grained parallelism modeling to dataflow MoCs which can benefit to High-Level Synthesis tools. Secondly, the proposed intermediate representation offers significant memory gains and runtime speed-up when used in a dynamic and embedded dataflow-based runtime.