Boolean Parametric Data Flow Modeling - Analyses - Implementation

Evangelos Bempelis

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 07 août 2006

Présentée par

Evangelos Bempelis

Thèse dirigée par **Alain Girault**
et codirigée par **Pascal Fradet**

préparée au sein **INRIA**
et de **École doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Modèle de calcul flot de données avec Paramétrés Entiers et Booléens

Modélisation – Analyses – Mise en œuvre

Thèse soutenue publiquement le **26-Fév-15**,
devant le jury composé de :

Prof. Marc Geilen
Technical University of Eindhoven, Rapporteur

Dr. Robert De Simone
INRIA Sophia Antipolis, Rapporteur

Prof. Alix Munier
Université de Paris 6, Examinatrice

Prof. Tanguy Risset (Président du jury)
CITI, INSA Lyon, Examinateur

Arthur Stoutchinin
STMicroelectronics, Examinateur

Dr. Alain Girault
INRIA Grenoble, Directeur de thèse

Dr. Pascal Fradet
INRIA Grenoble, Co-Directeur de thèse

As you set out for Ithaka
hope the voyage is a long one,
full of adventure, full of discovery.

— C. P. Cavafy

ACKNOWLEDGMENTS

When I started my PhD thesis, little did I know what it would be about. The final destination was vague ideas and distant. However, during the last three years, this journey got me to visit many interesting places, both physically and intellectually. From the whole experience, this is what I treasure more. The compilation of experiences during the last years that changed me and made me (I hope) a better person.

Looking back at these moments, I need to thank all the people that made my trip through knowledge a little bit more comfortable, a little bit more enjoyable and a little bit easier. I will start from Ali-Erdem Ozcan, the man who initiated the collaboration between STMicroelectronics and INRIA and made my PhD position a reality. Of course, I want to thank my managers in ST, that is Bruno Lavigueur for his tireless guidance through all the technical issues I encountered at ST and Arthur Stoutchninin for helping me during the final steps of the thesis.

My thesis advisors, Alain Girault and Pascal Fradet have been decisive factors throughout the PhD. They seemed to complement each other making my journey much easier. Alain for his insight, for providing a clean perspective of the work and for his support when things were looking grim. Pascal, for his strong character that lead to many discussions, sometimes on the verge of fighting, but always constructive enough to push the work further. He has the ability to always challenge a statement which makes every bit of progress a slow but verified step.

Moreover, I want to thank all my colleagues. For the very nice working environment they provided and the endless discussions that have been food for thought (or not) and a welcomed distraction from all the issues of the work. Hence, Dmitry, Gideon, Peter, Yoann, Adnan, Sophie, Xavier, Gregor, Jean-Bernard, thank you very much!

I would also like to thank my family, Ifigeneia, Charilaos and Vasilis for their support, even remotely, these last years. They always have been on my side and shared both the good and the nice moments of my journey.

Finally, I want to thank my girlfriend, Stefania. She has bared with me the last three years and traveled with me this journey. She carried much of the burden of my PhD being the recipient of much of my complaints and worries. Thank you very much for your support, without which my journey would be much more difficult.

Evangelos

CONTENTS

ACRONYMS

DAG	Directed Acyclic Graph
NoC	Network-on-Chip
SWPE	Software Processing Element
HWPE	Hardware Processing Element
HWPU	Hardware Processing Unit
UMA	Uniform Memory Access
DMA	Direct Memory Access
DSP	Digital Signal Processing
FIFO	First-In First-Out
HEVC	High Efficiency Video Coding
VC-1	Video Codec - 1
DVFS	Dynamic Voltage-Frequency Scaling
MoC	Model of Computation
FSM	Finite State Machine
KPN	Kahn Process Network
WEG	Weighted Event Graph
SDF	Synchronous Data Flow
HSDF	Homogeneous Synchronous Data Flow
CSDF	Cyclo-Static Data Flow
BDF	Boolean Data Flow
IDF	Integer Data Flow
PSDF	Parameterized Synchronous Data Flow
VRDF	Variable Rate Data Flow
SADF	Scenario-Aware Data Flow
SPDF	Schedulable Parametric Data Flow
SSDF	Scalable Synchronous Data Flow
SPDF	Synchronous Piggybacked Data Flow

BDDF	Bounded Dynamic Data Flow
CDDF	Cyclo-Dynamic Data Flow
HDF	Heterochronous Data Flow
MDSDF	Multi-Dimensional Synchronous Data Flow
EIDF	Enable Invoke Data Flow
CFDF	Core Functional Data Flow
PiMM	Parameterized and Interfaced Dataflow
FRDF	Fractional Rate Data Flow
PEDF	Predicated Execution Data Flow
DIF	Dataflow Intechange Format
APGAN	Acyclic Pairwise Grouping of Adjacent Nodes
RPMC	Recursive Partitioning based on Minimum Cuts
DLS	Dynamic Level Scheduling
HEFT	Heterogeneous Earliest Finish-Time
MH	Mapping Heuristic
LMT	Levelized-Min Time
TDS	Task Duplication based Scheduling
PSGA	Problem-Space Genetic Algorithm
DSM	Decision State Modeling
GST	Generalized Schedule Tree
SST	Scalable Schedule Tree
BPDF	Boolean Paramatric Data Flow
PSLC	Parametric SDF-like Liveness Checking
SMT	Satisfiability Modulo Theories
ASAP	As Soon As Possible
TNR	Temporal Noise Reduction
QoS	Quality of Service
GPU	Graphical Processing Unit
MdC	Modèles de Calcul

RÉSUMÉ EN FRANÇAIS

Les applications de gestion de flux sont responsables de la majorité des calculs des systèmes embarqués (vidéo conférence, vision par ordinateur). Leurs exigences de haute performance rendent leur mise en oeuvre parallèle nécessaire. Par conséquent, il est de plus en plus courant que les systèmes embarqués modernes incluent des processeurs multi-coeurs qui permettent un parallélisme massif.

La mise en oeuvre des applications de gestion de flux sur des multi-coeurs est difficile à cause de leur complexité, qui tend à augmenter, et de leurs exigences strictes à la fois qualitatives (robustesse, fiabilité) et quantitatives (débit, consommation d'énergie). Ceci est observé dans l'évolution de codecs vidéo qui ne cessent d'augmenter en complexité, tandis que leurs exigences de performance demeurent les mêmes.

Les Modèles de Calcul (MdC) flot de données ont été développés pour faciliter la conception de ces applications qui sont typiquement composées de filtres qui échangent des flux de données via des liens de communication. Ces modèles fournissent une représentation intuitive des applications de gestion de flux, tout en exposant le parallélisme de tâches de l'application. En outre, ils fournissent des analyses statiques pour la vivacité et l'exécution en memoire bornée. Cependant, les applications de gestion de flux modernes comportent des filtres qui échangent des quantités de données variables, et des liens de communication qui peuvent être activés / désactivés.

Dans cette thèse, nous présentons un nouveau MdC flot de données, le Boolean Paramatric Data Flow (BPDF), qui permet le paramétrage de la quantité de données échangées entre les filtres en utilisant des paramètres entiers et l'activation et la désactivation de liens de communication en utilisant des paramètres booléens. De cette manière, BPDF est capable de exprimer des applications plus complexes, comme les décodeurs vidéo modernes.

Malgré l'augmentation de l'expressivité, les applications BPDF restent statiquement analysables pour la vivacité et l'execution en memoire bornée. Cependant, l'expressivité accrue complique grandement la mise en oeuvre. Les paramètres entiers entraînent des dépendances de données de type paramétrique et les paramètres booléens peuvent désactiver des liens de communication et ainsi éliminer des dépendances de données.

Pour cette raison, nous proposons un cadre d'ordonnancement qui produit des ordonnancements de type "aussi tôt que possible" (ASAP) pour un placement statique donné. Il utilise des contraintes d'ordonnancement, soit issues de l'application (dépendance de données) ou de l'utilisateur (optimisations d'ordonnancement). Les contraintes sont analysées pour la vivacité et, si possible, simplifiées. De cette façon, notre cadre permet une grande variété de politiques d'ordonnancement, tout en garantissant la vivacité de l'application.

Enfin, le calcul du débit d'une application est important tant avant que pendant l'exécution. Il permet de vérifier que l'application satisfait ses exigences

de performance et il permet de prendre des décisions d'ordonnancement à l'exécution qui peuvent améliorer la performance ou la consommation d'énergie.

Nous traitons ce problème en trouvant des expressions paramétriques pour le débit maximum d'un sous-ensemble de BPDF. Enfin, nous proposons un algorithme qui calcule une taille des buffers suffisante pour que l'application BPDF ait un débit maximum.

1 INTRODUCTION

A Musico-Logical Offering,
An Eternal Golden Braid

— D.R.Hofstadter

Multimedia applications are widely used in the modern world. Video conferencing with multiple participants and high quality movie playback, even on mobile devices, are considered granted for modern users. Apart from our daily life, multimedia applications have changed our capabilities. Augmented reality car head-up displays are becoming available, facilitating driving with low visibility, while remote surgery allows doctors to perform surgeries over long distances. For example the Lindbergh operation, where a team of French surgeons operated over a distance of 6.230 km, from New York to Strasbourg. Hence, it is not surprising that multimedia applications already accounted for more than 90% of the computing cycles of general purpose processors, back in 1998 [39],[27],[106]. Because of their structure around the notion of "streams", multimedia applications are also referred to as *streaming* applications.

1.1 STREAMING APPLICATIONS

Streaming applications are characterized by large streams of data that are being communicated between different computation nodes (often called *actors* or *filters*). An example of a streaming application is shown in Figure 1. The application gets as input 3 streams (I_1, I_2, I_3) and outputs 2 streams (O_1, O_2), the result of processing of the input streams through 6 filters (F_1, F_2, F_3, F_4, F_5, F_6). Filter F_6 has a backward connection to filter F_1 forming an internal loop (feedback).

It is evident that the requirements of streaming applications can widely vary. For applications that run on mobile devices, low power consumption is crucial to preserve battery life. On the contrary, medical applications need to be reliable and, in the case of remote surgery, with extremely low latency. A common factor, though, is their high performance requirements, which makes their parallel implementation a necessity.

The complexity of streaming applications tends to increase, while their strict requirements remain the same or even increase. An example is HEVC, the newest video coding standard, which increases the decoding complexity by 60% in comparison to its predecessor H.264 [119], while performance demands remain at the same level or even increase: for the same frame rate of 30 frames per second and assuming their maximum resolution, 4K and 2K respectively (Figure 2), the HEVC decoder needs to process 4 times the amount of data processed by H.264 in the same amount of time due to the increase in resolution.

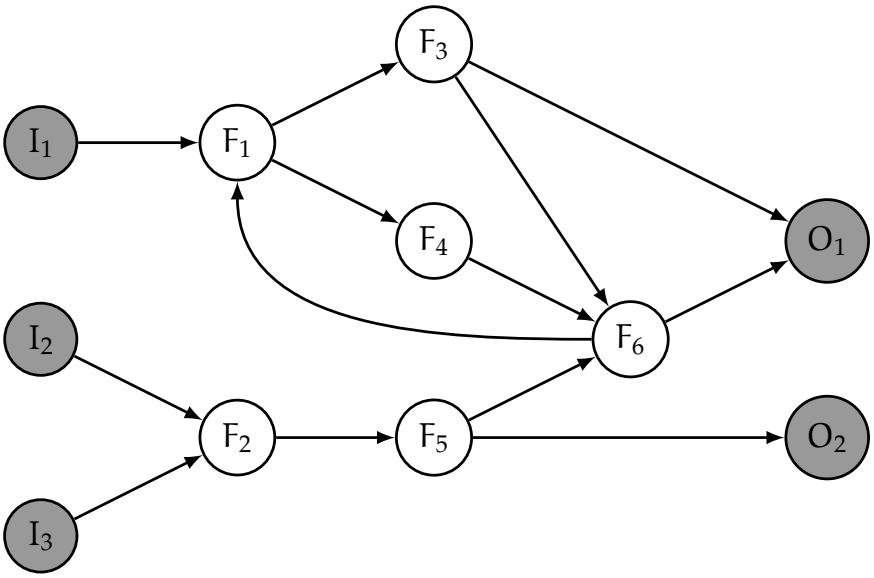


Figure 1: Structure of a streaming application.

The need for parallel implementation and the increased complexity makes the development (*i. e.,* design and implementation) of streaming applications very challenging.

A major challenge in streaming application design is hardware/software co-design. In the early development phases, an application often needs to be partitioned into components that will be implemented as specialized hardware and components that will be run in software. Moreover, it should be decided which aspects of the hardware components must be flexible enough for future reconfiguration. These decisions are hard to evaluate at such an early stage and they may lead to an implementation not meeting the desired specifications. For this reason, modeling and rapid prototyping of the application is essential in the design of streaming applications.

General-purpose languages, are not well suited for the implementation of streaming applications because they do not provide a natural way to represent streams and filters. The regular communication patterns between the different computation nodes and their parallel execution cannot be explicitly expressed in languages like C or Java. In addition, general-purpose languages are designed for the von-Neumann architectures, but highly parallel architectures do not fit the von-Neumann model as they use multiple instruction streams and distributed memory.

1.2 MODELS OF COMPUTATION

A Model of Computation (MoC) can be thought of the set of rules that govern execution [76]. It is the general guidelines within which the developer will program. A MoC allows the mathematical representation of the program and helps the verification of many of its properties. Some of these properties may be inherent to the model (*e. g.,* determinism), while others can be verified with static analyses (*e. g.,* liveness).

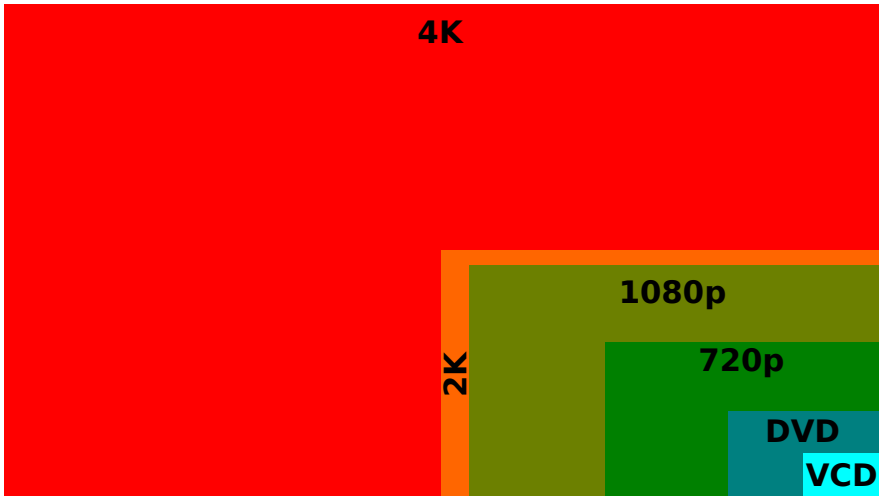


Figure 2: Comparison of different video resolutions. All sizes are scaled to 16:9 ratio.

The need for rapid prototyping and deployment of massively parallel streaming applications has led to the development of several MoCs that facilitate design and parallel implementation. Such MoCs are meant to represent the application in an intuitive manner for the programmer while exposing its parallelism.

In contrast with sequential execution, where the von-Neumann model prevailed, in concurrent execution there still has not been a universally accepted MoC. A plethora of MoCs that focus on parallelism has been developed, including Petri Nets [96], Kahn Process Networks [67], and Data Flow [35] and all its variants.

Parallelism-oriented MoCs are attractive for the prototyping of streaming applications because they allow static analyses that verify various qualitative (liveness, boundedness) and quantitative (throughput, power consumption) properties of the application early in the design process. They also make parallel implementation easier because they provide an intuitive way to represent filters and streams and allow the functional partitioning of an application exposing the available parallelism and allowing modular design.

Data flow MoCs are of particular interest because they are extensively used in industry. Lustre/Scade has been successfully used to implement flight control system in Airbus planes [20] and the signaling system of the Hong-Kong subway [82]. In addition, much of the development of data flow visual programming languages in the 80s was backed by industrial sources [66] resulting in the development of visual languages, such as LabView [65] which was successfully deployed in industry, significantly reducing development time [5]. Moreover, the existence of a plethora of data flow based programming languages like Lucid [120], Lustre [24], Signal [9], Lucid Synchrone [25], StreamIt [117] *etc.*, is another indication of its widespread use.

1.3 STREAMING APPLICATION DEVELOPMENT WITH DATA FLOW MoCs

Data Flow MoCs are well-suited to program streaming applications on many-core architectures. Models like *Synchronous Data Flow* (SDF) [78], have been

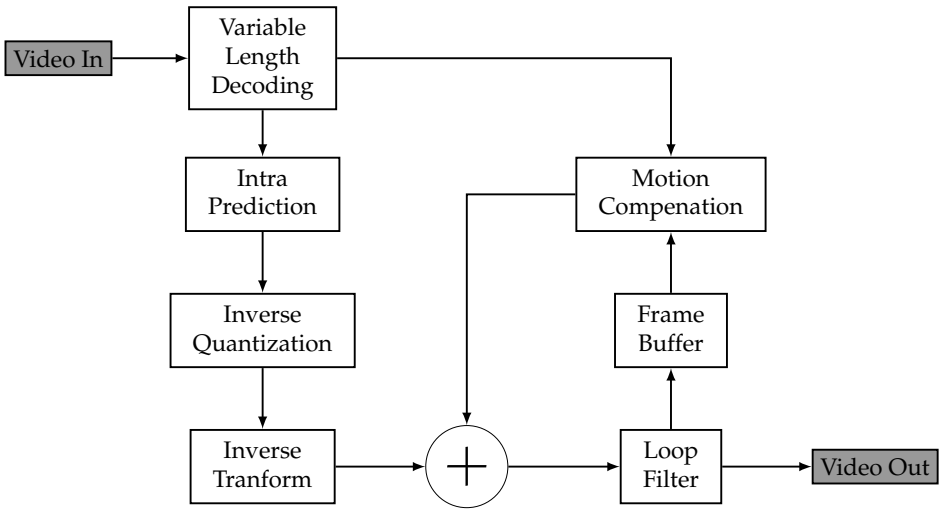


Figure 3: General structure of a video decoder.

widely used to develop Digital Signal Processing (DSP) applications. Yet, many modern streaming applications, such as high definition video codecs, show a dynamic behavior that current MoCs cannot express.

Figure 3 shows the structure of a modern video decoder. The encoded video is first processed by the *Variable Length Decoding* filter, which, based on the encoding of the input, will send data to the *Intra Prediction* filter and to the *Motion Compensation* filter. As implied by its name, the amount of data sent to these filters is data-dependent. The data is processed in parallel in the two pipelines until the two streams are merged and are processed by the *Loop Filter*.

To capture this dynamic behavior, parametric dataflow MoCs have been introduced from the early 2000s, including PSDF [11], VRDF [122], SADF [116] and SPDF [43]. These models allow the amount of data exchanged between the filters of an application to change at run-time according to the values of the manipulated data. This is achieved by using integer parameters.

However, not all parts of a video are encoded in the same way. For example, some frames may not use the *Motion Compensation* filter, or some blocks of the video may not be coded and may not need the inverse quantization or the inverse transform. To take advantage of this information, the topology of the application should be changed at run-time.

For this reason, data flow MoCs that allow the change of the graph topology at run-time have been developed, including BDF [21] and IDF [22]. However, these models lack analyzability. Moreover, current parametric data flow MoCs do not allow the topology of the application to change at run-time. Hence, there is a need for a new data flow MoC that combines parametric exchange of data with dynamic topology changes.

Increasing the expressivity of a data flow model greatly complicates its deployment on a parallel architecture. Parametric exchange of data results in parametric data dependencies, while dynamic topology changes remove and add data dependencies at run-time. Many standard implementation techniques are incompatible with this behaviour and cannot be used. Furthermore, manual parallel implementations are hard to produce and can be error-prone.

Finally, static analyses of the application for properties, such as throughput and power consumption, are crucial to verify that the application meets its requirements. They help making design decisions early in the development process. Moreover, because of the dynamic behaviour of the application, quantities such as throughput or power consumption can greatly vary at run-time. The ability to efficiently calculate the values of properties like throughput can lead to better implementations. However, existing approaches cannot be applied due to the increased expressiveness of the MoC.

1.4 CONTRIBUTIONS

The main challenge in developing a data flow MoC lies in the trade-off between analyzability and expressivity of the model. In this thesis, we propose a new data flow MoC, the *Boolean Paramatric Data Flow* (BPDF) model [7], that combines parametric data exchange between filters of the application with the ability of dynamically changing the topology of the application. A distinguished property of BPDF is to provide static analyses for the deadlock-free and bounded execution of the application, despite the increase in expressivity.

The increased expressivity of BPDF greatly complicates implementation. We address this problem with the development of a scheduling framework for the production of parallel schedules for BPDF applications [8]. Our framework facilitates the automatic production of complex parallel schedules, which are often hard to produce and error-prone, while preserving the boundedness and liveness properties of the application.

Finally, we approach the problem of throughput calculation for BPDF applications by finding parametric throughput expressions for the maximum throughput of a subset of BPDF graphs. An algorithm is proposed to calculate sufficient buffer sizes for the application to operate at its maximum throughput.

The thesis is structured as follows: In Chapter 2, the current state-of-the-art data flow MoCs are presented. Moreover, the current methods of their implementation on many-core architectures are discussed. Our data flow MoC, BPDF, along with its static analyses are described in Chapter 3. Chapter 4 contains our scheduling framework and its evaluation with a modern video decoder, VC-1, expressed using BPDF. In Chapter 5, we propose a parametric throughput analysis for BPDF. We use the analysis to express parametrically the throughput of the VC-1 decoder. Finally, the thesis is summarized in Chapter 6, where contributions and future work perspectives are discussed.

2 DATA FLOW MODELS OF COMPUTATION

> Last night what we talked about
> It made so much sense
> But now the haze has ascended
> It don't make no sense anymore
>
> — Arctic Monkeys

Modeling is essential during development as it allows the analysis and study of a system to be done indirectly on a model of the system rather than directly on the system itself. Computational systems are modeled with Models of Computation (MoCs). These are mathematical formalisms that can be used to express systems.

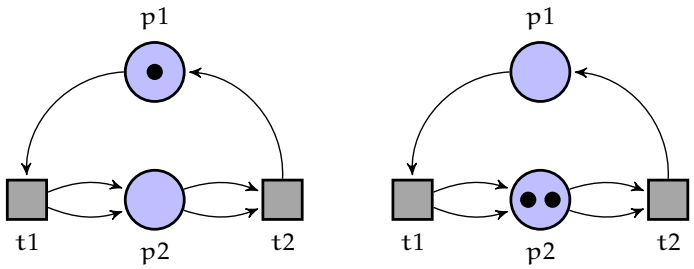
A system captured using a MoC can be analyzed and have various properties, both qualitative (*e.g.*, reliability) and quantitative (*e.g.*, performance) verified. Then, the model can be used to generate code that preserves these properties which in turn can be compiled into software or synthesized into hardware. This way, many aspects of the system can be explored rapidly before the actual development takes place. Moreover, specifications of the system can be verified and, as the human factor is limited, the procedure is less error-prone. Hence, a system can be developed and evaluated faster, cheaper and safer.

2.1 PARALLEL MODELS OF COMPUTATION

There are many MoCs developed over the years, each one focusing on different aspects of the system under development. For example, Finite State Machines (FSMs) focus on sequential execution and are widely used to design control systems while *discrete event* models focus on timing. In this thesis, our goal is to facilitate the development of parallel embedded systems, therefore we focus on MoCs that expose parallelism such as Petri Nets, Process Networks and Data Flow.

2.1.1 *Petri Nets*

Petri Nets first appeared in Carl Adam Petri dissertation "*Kommunikation mit Automaten*" [Communication with automata] [96] in 1962. Petri focused on formalizing the communication of asynchronous components of a computer system. His work drew a lot of attention and was further developed by Holt *et al.* in the final report of the Information Systems Theory Project [61] in 1968. Petri Nets have been developed since, a few books summing up the advances in the field being: Peterson's "*Petri Net Theory and the Modeling of Systems*" [95] in 1981, Brams' "*Réseaux de Petri: Théorie et pratique*" [19] in 1983 and Desel and Esparza's "*Free Choice Petri Nets*" [36] in 1995.



(a) Initial marking. (b) After t1 fires.

Figure 4: A Petri Net example.

A Petri Net is composed by two components, a *net* and a *marking*. The net is a directed bipartite multi-graph that consists of two types of nodes: places and transitions. Places are drawn as circles and represent memory, transitions are drawn as rectangles and represent computation units. Edges connect transitions with places and vice versa but nodes of the same type cannot be connected. The marking indicates the data stored currently on each place, shown as black dots inside the places.

Each transition has a set of incoming edges connecting the transition with a set of *input* places. Similarly, each transition is connected with a set of *output* edges with its outgoing edges. If all the input places of a transition have enough data the transition is called *enabled*. An enabled transition needs to have at least one data token stored in its input places for each of its incoming edges.

A Petri Net executes by *firing* enabled transitions. First the transition consumes data tokens from all its input places, on token for each of the connecting edges. Then, it computes and produces one token for each outgoing edge which is then stored in the corresponding connected place.

An example of a Petri Net is shown in Figure 4. It is composed by two transitions $(t1, t2)$ and two places $(p1, p2)$ where $p1$ has one token initially stored in it. In the example of Figure 4, transition $t1$ has enough tokens to fire. After its firing, place $p1$ is left with no tokens but place $p2$ has 2 new tokens, one for each outgoing edge of $t1$. Now $t2$ has enough tokens to fire, and so on.

Petri Net models can be used to prove system properties such as mutual exclusion (*e. g.,* two transitions cannot fire in parallel), liveness (absence of deadlock) and safety (that the system cannot reach a particular state). In Figure 4 one can easily show that $t1$ and $t2$ are mutually exclusive and that the graph will not deadlock. However, the execution of a Petri Net is non-deterministic: when multiple processes are enabled any one may fire.

2.1.2 *Process Networks*

Process networks, or Kahn Process Networks (KPNs), were introduced by Gilles Kahn in 1974 [67]. A KPN is composed of a set of processes, interconnected with communication links (channels) (Figure 5). Communication links are unidirectional and the only way processes communicate with each other. Each process may either be executing or may be blocked, waiting for data on one of its incom-

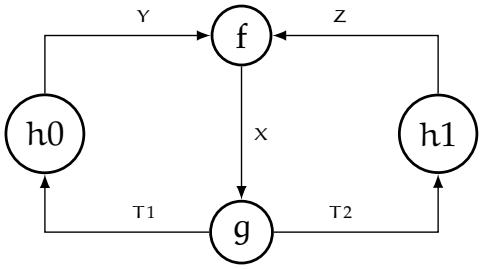


Figure 5: An example of a process network with processes $g, h0, h1, f$ and channels $T1, T2, Y, Z, X$. Figure reproduced from [67].

ing communication links. A process cannot check a whether a communication link is empty or not. Once a process finishes execution, it produces tokens on one or many of its outgoing communication links. There is no blocking mechanism that prohibits a process to write. Each communication link has a type (*e. g.,* integer, boolean float *etc.*). The sequence of data elements on a link, called its *history*, is a complete partial order.

KPNs can be formulated with a system of equations where processes are functions and communication links are variables. This set is reduced to a single fixed point equation $X = f(X)$. As the functions are continuous over complete partial orders, the equation has a unique least fixed point [67]. In this way KPNs are deterministic: the least fixed point which depends on the histories of the communication links is unique therefore, the histories produced on each communication link are independent of the execution order of the processes. Using the fixed point, a KPN can be analyzed for functional verification (*e. g.,* that the program will produce the desired output). Moreover, a KPN is *terminating* or *non-terminating* depending on whether its fixed point contains finite or infinite histories.

Although fixed point analysis gives the length of the histories of the communication links, it does not reason on the accumulation of data tokens which depends on the execution order. A KPN is *strictly bounded* if the accumulation of tokens on all its communication links is bounded by b for all possible execution orders. A KPN can be transformed so that it is strictly bounded. To do so, feedback links are added for each communication link with b initial tokens. However, the feedback links may introduce a deadlock transforming a non-terminating program into a terminating one. Boundedness of KPNs is discussed in Parks' thesis [93] and extended in [47].

2.1.3 *Data Flow*

The Data flow MoC first appeared in 1974, in a paper by Jack B. Dennis [35]. The initial goal of data flow was to lift some limitations of Petri Nets and increase their expressiveness. Dennis' data flow is expressive enough to express the source program while exposing the available parallelism.

In Dennis' data flow, applications are expressed as directed graphs. Nodes, called *actors*, are function units and edges are communication links. Actors can execute or *fire* once they have enough tokens on their input links, as in Petri

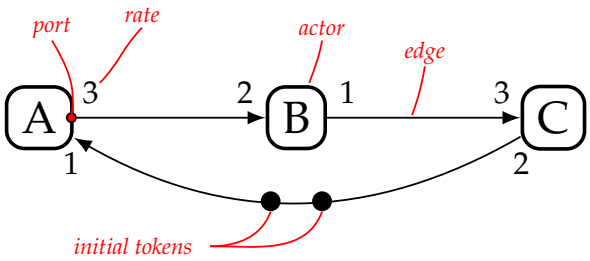


Figure 6: A simple SDF graph.

Nets. However, in contrast with Petri Nets, Dennis' data flow takes into account the value of the data on the links. So, more specialized actors are introduced to express `if-then-else` structures, boolean operations, splitting and merging of data and more. Dennis' data flow model is very expressive though and has limited analyzability. Hence, subsequent data flow models aimed at limiting expressiveness and increasing analyzability. The most influential is Synchronous Data Flow introduced in 1987 by Lee and Messerschmitt [78].

Dennis' Data flow shares the same conceptual basis with Petri Nets but where Petri Nets focuses on the modeling part, data flow takes a step further and allows reasoning on the type and the value of the data communicated as well as the amount [42], a feature that was abstracted in later data flow MoCs. Moreover, data flow execution is deterministic whereas Petri Net execution is not.

In comparison with KPNs, a major difference is that processes in KPNs can be executing by consuming data from just a subset of their inputs. In contrast, actors in data flow require data on all of their inputs. Moreover, later data flow MoCs provide analysis for properties like liveness and boundedness matching the analyzability of Petri Nets.

The deterministic behaviour and analyzability of data flow MoCs makes them attractive for use in streaming application development. This is also shown by the industrial success of data flow as discussed in Section 1.2. In this thesis, we focus on data flow MoCs, which are presented in detail in the next section.

2.2 SYNCHRONOUS DATA FLOW

In this section, we present Synchronous Data Flow (SDF) that makes the basis for most of the later data flow MoCs [78]. SDF was introduced in 1987, for the implementation of DSP applications on parallel architectures. SDF is well suited for DSP because of the ease of expression of such applications using the model.

In SDF, as in Dennis' data flow, a program is expressed as a directed graph, where nodes (*actors*) are functional units. Edges are communication links connecting actors implemented by First-In First-Out (FIFO) queues. The connection of an actor with an edge is called a *port*. Each edge is characterized by a production (*resp.* consumption) rate indicating the amount of data produced (*resp.* consumed) on (*resp.* from) the edge after the firing of the corresponding actor. As these rates also characterize the respective port, we often refer to them as *port rates*.

An SDF graph executes by firing its actors. The firing of an SDF actor has three steps:

A. Consumption of data tokens from all its incoming edges (*inputs*).
B. Execution of its internal, side-effect free function
C. Production of data tokens to all its outgoing edges (*outputs*).

The number of tokens consumed or produced on an edge is defined by its rate. An actor can fire only when *all* of its input edges have enough tokens (*i. e.,* at least the number specified by each rate), in this way its execution is *independent* of the order the actor reads its ports. Such an actor is called *fireable* or *eligible to fire*. In SDF, all rates are constant integers, therefore known at compile time.

2.2.1 *Formal Definition*

Formally, an SDF graph is defined as a 5-tuple $(\mathcal{G}, \mathtt{lnk}, \mathtt{init}, \mathtt{prd}, \mathtt{cns})$ where:

- $\mathcal{G}$ is a directed connected multigraph $(\mathcal{A}, \mathcal{E})$ (*i. e.,* there can be more than one edge connecting a pair of actors.) with $\mathcal{A}$ a set of actors, and $\mathcal{E}$ a set of directed edges.
- $\mathtt{lnk} : \mathcal{E} \to \mathcal{A} \times \mathcal{A}$ associates each edge with the pair of actors that it connects.
- $\mathtt{init} : \mathcal{E} \to \mathbb{N}$ associates each edge with a number of initial tokens.
- $\mathtt{prd} : \mathcal{E} \to \mathbb{N}^*$ associates each edge with a production rate.
- $\mathtt{cns} : \mathcal{E} \to \mathbb{N}^*$ associates each edge with a consumption rate.

For example, the simple SDF graph in Figure 6, is composed of three actors, $\mathcal{A} = \{A, B, C\}$ and three edges, $\mathcal{E} = \{\overline{AB}, \overline{BC}, \overline{CA}\}$. For convenience throughout the thesis we use the notation $e = \overline{XY}$ where X and Y represent the producer and consumer of the edge. The $\mathtt{lnk}$ function for edge $\overline{AB}$ is defined as $\mathtt{lnk}(\overline{AB}) = (A, B)$ and similarly for the rest of the edges. The $\mathtt{init}$ function returns 0 for all edges except for $\mathtt{init}(\overline{CA}) = 2$. Finally, the $\mathtt{prd}$ and $\mathtt{cns}$ functions return the production and consumption rates, for example $\mathtt{prd}(\overline{AB}) = 3$, $\mathtt{cns}(\overline{AB}) = 2$, $\mathtt{prd}(\overline{BC}) = 1$, $\mathtt{cns}(\overline{BC}) = 3$, $\mathtt{prd}(\overline{CA}) = 2$ and $\mathtt{cns}(\overline{CA}) = 1$.

An SDF graph is characterized by its *topology matrix* (Γ), which is similar to the incidence matrix from graph theory. The topology matrix has its columns assigned to the nodes of the graph, and its rows to the edges. Each entry (i, j) of the matrix corresponds to the amount of data produced on edge i from node j, *i. e.,* the corresponding port rate. Consumption of data is represented by a negative value and the absence of a link is represented by 0. If multiple edges connect two actors, the sum of the production/consumption rates is used instead. The topology matrix for the graph in Figure 6 is:

	A	**B**	**C**
AB	3	-2	0
BC	0	1	-3
CA	-1	0	2

The *state* $(\mathcal{S})$ of an SDF graph is a vector indicating the number of tokens stored on each edge of the graph at a given instant. Every time an actor fires,

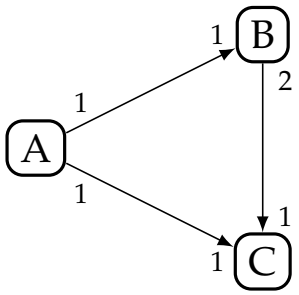


Figure 7: An inconsistent SDF graph.

it produces and consumes tokens altering the state of the graph. For instance, the initial state of the SDF graph in Figure 6 is $\mathcal{S}_{init} = [0\ 0\ 2]$. After actor A fires once, the state of the graph becomes $\mathcal{S}' = [3\ 0\ 1]$.

2.2.2 *Static Analyses*

A key property of SDF is that it can be statically analyzed for *consistency*, *boundedness* and *liveness*. Analyses of these properties are formally presented in [48].

These properties are crucial for embedded systems. Consistency ensures that the graph is valid, *i.e.*, there are no incompatible rates on the graph, opening the way to the analyses of *boundedness and liveness*. Boundedness ensures that an application operates within bounded memory. With a known memory bound, the designer can ensure that the system has sufficient memory and allocate it statically at the beginning of the execution of the application, greatly improving performance. Finally, liveness ensures the continuous operation of a system, which is generally desirable and essential for critical systems.

Consistency

To present *rate consistency*, we take the example of an inconsistent graph in Figure 7. As shown in the Figure 7, For each firing of actor A, actor B is enabled once and can produce 2 tokens on $\overline{BC}$. However, actor C is also enabled only once, and cannot consume both tokens that B produces. As a result, tokens will accumulate on edge $\overline{BC}$ and the graph is *inconsistent*. An SDF graph is *consistent* if there is a set of firings that returns the graph back to its initial state. Indeed, a repetition of such a set never leads to an accumulation of tokens on any edge and the graph is consistent.

We call such a set of firings an *iteration* of the graph, that is a non-empty set of actor firings that return the graph back to its initial state.

An iteration of the graph can be found by solving the so-called *system of balance equations*. To return to the initial state the total production of tokens on each edge should be to equal the corresponding consumption. Formally:

$$\forall e = \overline{AB} \in \mathcal{E},\quad \exists\, \#A, \#B,\quad \#A \cdot prd(e) = \#B \cdot cns(e) \tag{1}$$

where #A and #B are called solutions of actors A and B, respectively. They indicate the number of times each actor fires during the iteration.

We get one such Eq. (1) for each edge of the graph, ending up with a set of balance equations, which forms a linear system. The system can be expressed more compactly using the topology matrix:

$$\Gamma \cdot r = 0 \tag{2}$$

where r is the vector with the actor solutions. The minimum non-trivial integer solution of the system (Eq. (2)) is called the *repetition vector* and indicates the minimum number of times each actor needs to fire for the graph to return to its initial state, the *minimum iteration*. Although an iteration can be any multiple of the minimum iteration, from now on we use iteration to denote the minimum iteration, unless noted otherwise. If the system has no solution, the graph is said to be *inconsistent* as it never returns to its initial state. Hence, we can formally define consistency:

Definition 1 (Consistency)**.** *An SDF graph is consistent iff its system of balance equations has a non-trivial solution.*

For the SDF graph in Figure 6, the balance equation for, say, the edge $\overline{AB}$ is $\#A \cdot 3 = \#B \cdot 2$. The system of balance equations, using Eq. (2) is:

$$\begin{bmatrix} 3 & -2 & 0 \\ 0 & 1 & -3 \\ -1 & 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} \#A \\ \#B \\ \#C \end{bmatrix} = 0$$

The minimum solution is $r = [2\ 3\ 1]$, also written $r = [A^2\ B^3\ C]$ for better readability. The graph in Figure 6 is *consistent*, that is, there is a set of firings that returns the graph back to its initial state.

Boundedness

An SDF graph is *bounded* if it can execute in finite memory. A consistent SDF graph is inherently bounded. By definition, there is no accumulation of tokens on any edge of the graph hence, the graph operates in bounded memory. Therefore, checking an SDF graph for boundedness, amounts to check consistency by solving the system of balance equations. Formally:

Definition 2 (Boundedness)**.** *An SDF graph is bounded iff its system of balance equations has a non-trivial solution.*

An efficient algorithm to compute the repetition vector is presented in [6] with linear time complexity ($\Theta(|\mathcal{E}| + |\mathcal{A}|)$) in the number of actors and the number of edges. The algorithm first randomly sets the solution of an actor to 1 and, as the graph is connected, it finds a solution for each actor, if it exists. Then, the algorithm normalizes the solution to be the minimum integer solution. If the algorithm successfully returns a repetition vector, the graph is consistent and bounded.

Liveness

An SDF graph is live if it can execute an infinite number of time without deadlocking. Checking the liveness of a graph amounts to finding a sequence of firings that complete an iteration, a *schedule*. Finding the schedule for one iteration is sufficient for the liveness of the graph; once the schedule is executed, the graph returns to its initial state, allowing the schedule to start again and repeat indefinitely. However, not all schedules are valid. There may be schedules that cannot finish the iteration because they contain *non-eligible* firings, *i.e.*, firings of actors that do not have enough tokens on their input edges. A schedule that is composed only by *eligible* firings is called *admissible*. Hence, formally:

Definition 3 (Liveness). *An SDF graph is live iff there exists an admissible schedule.*

For the graph in Figure 6, a schedule that completes an iteration is:

$$A; A; B; B; B; C$$

It is admissible because all firings can take place, (*i.e.*, the respective actor has enough tokens on its input edges) and he graph is live. In contrast, a non-admissible schedule is:

$$A; C; B; B; B; A$$

After actor A first fires, actor C cannot fire as there are not enough tokens on its input edge.

Acyclic graphs and graphs with non-directed cycles are inherently live, as an admissible schedule can always be found, just from the topological sorting of the actors. When there are directed cycles, however, each cycle needs to have a sufficient number of initial tokens for the graph to be live.

Algorithms used to find admissible schedules are called *class-S algorithms* by Lee and Messerschmitt in [78]. In [6] a simple class-S algorithm is presented with time complexity

$$O(I f_i f_o + |\mathcal{E}|) \quad \text{where} \quad I = \sum_{X \in \mathcal{A}} \#X$$

and f_i (*resp.* f_o) the maximum number of incoming (*resp.* outgoing) edges among all actors.

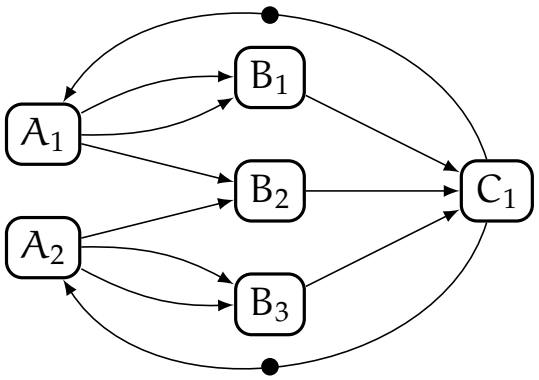
2.2.3 *Special Cases of SDF Graphs*

Some restrictive classes of SDF are worth mentioning as they are used in a variety of cases. The Homogeneous Synchronous Data Flow (HSDF) graphs are graphs where all port rates equal to 1. Formally:

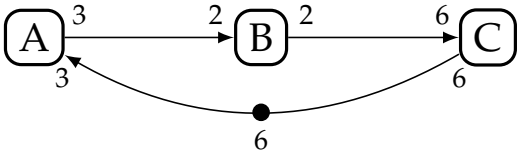
Definition 4 (Homogeneous SDF graph). *An SDF graph is homogeneous iff*

$$\forall e \in \mathcal{E}, \quad prd(e) = cns(e) = 1$$

Any SDF graph can be converted to an equivalent HSDF graph. There are many algorithms that convert SDF graphs to HSDF graphs, one widely used can be found in [111]. The main intuition behind the transformation is to replicate



(a) HSDF graph.



(b) Normalized SDF graph.

Figure 8: The HSDF and the normalized SDF graphs of the graph in Figure 6.

each actor as many times as its solution and connect the new actors according to the rates of the original SDF. The resulting graph may have an exponential increase in size.

The resulting HSDF graph from the SDF graph in Figure 6 is shown in Figure 8a. Each actor in HSDF graph corresponds to a firing in the iteration of the corresponding actor. So, for actor A that needs to be fired twice ($r = [A^2\ B^3\ C]$), the HSDF has two actors (A_1, A_2). The edges are produces accordingly: the first firing of A produces 2 tokens for the first firing of B, hence the two edges between A_1 and B_1 in the HSDF, and one token for its second firing, shown with $\overline{A_1 B_2}$.

HSDF representation is useful because it exposes all the available task parallelism. It has been successfully used to produce parallel schedules of SDF graphs and evaluate its throughput (see Section 2.4.3).

Another convenient class of SDF graphs is uniform or *normalized* SDF graphs. An SDF graph is normalized if all the ports of any given actor have the same rate. Any SDF graphs can be transformed to an *equivalent* normalized SDF graph. Two SDF graphs are equivalent if any schedule that is admissible for one is also admissible for the other.

A simple transformation is the replacement of the rates of the ports of each actor $\#A$ with

$$\frac{\mathrm{lcm}_{N \in A} \#N}{\#A}$$

The initial tokens should be adjusted accordingly to trigger the same number of firings of the consumer. The method is presented in detail in [86] for Weighted

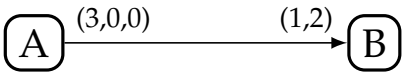


Figure 9: A CSDF graph.

Event Graphs (WEGs) which are equivalent to SDF graphs [107]. The normalized SDF of the graph in Figure 6 is shown in Figure 8b.

2.3 EXTENSIONS OF SYNCHRONOUS DATA FLOW

This section describes the more prominent of the extensions of SDF, starting with MoCs that are fully defined at compile-time like Cyclo-Static Data Flow [15] (Section 2.3.1).

We classify the more expressive models in two categories: the ones that allow the graph to change topology at run-time (*dynamic topology* models, Section 2.3.2) and the ones that allow the amount of data exchanged between actors to change at run-time (*dynamic data rate* models, Section 2.3.3).

Dynamic topology models like BDF [23] and its natural expansion IDF [22] introduce specialized actors that can change the topology of the graph at run-time using boolean or integer parameters, respectively.

Dynamic data rate models use integer parameters to parameterize the amount of data communicated between the actors of a graph. Some of these models are PSDF [11], VRDF [122], SADF [116] and SPDF [43].

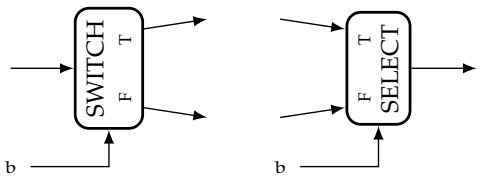
Many models presented below, allow actors to change their internal functionality at run-time. In this thesis, we disregard any dynamic change of the graph that does not affect its data flow analyses If it does not affect any of the subsequent analyses of the model and one can safely ignore it when it comes to the modeling of the application. In the SDF MoC for example, one can assume that the internal functionality of the actors change at run-time. If the rates of each port remain the same the boundedness and liveness analyses remain valid.

2.3.1 *Static Models*

In this section, we present Cyclo-Static Data Flow (CSDF) that extends the base SDF model but is still fully defined statically at compile-time. CSDF was introduced in 1995 by Bilsen *et al.* [15]. It targets applications that have predictable periodic behaviour, known at compile time. CSDF uses a series of rates that shift cyclically instead of a single fixed rate as in SDF. This way the production and consumption rates of an edge may change periodically. A rate can have a zero value, as long as the sum of the rates in a series is strictly positive.

A sample CSDF graph is shown in Figure 9. The edge $\overline{AB}$ has two sets of rates instead of two fixed rates. Each time an actor fires, all its port rates shift to the next value in the series. For the graph in Figure 9, the first time actor A fires, it produces 3 tokens and the next two, 0 tokens. On the fourth firing, the rate will shift cyclically back to 3.

In [15], a consistency analysis for CSDF is given. CSDF graphs can always be translated in HSDF graphs indicating that CSDF is not more expressive than



(a) SWITCH actor. (b) SELECT actor.

Figure 10: BDF special actors

SDF. However, such a conversion is not always practical because it requires a combinatorial explosion of the number of actors in the resulting HSDF.

2.3.2 *Dynamic Topology Models*

In this section, we present two models that focus on altering the graph topology at run-time, Boolean Data Flow (BDF) and Integer Data Flow (IDF). Joseph T. Buck introduced BDF in his thesis [21] as an extension of SDF that provides `if-then-else` functionality. BDF uses two special actors, a SWITCH and a SELECT actor (Figure 10). SWITCH has a single data input and two data outputs, whereas SELECT is the opposite with two data inputs and one data output. Both actors have a boolean control input that receives boolean tokens. Depending on the value of the boolean tokens, SWITCH (*resp.* SELECT) selects the output (*resp.* input) port that is activated.

A BDF graph is analyzed like an SDF graph except for the SWITCH (*resp.* SELECT) actors whose output (*resp.* input) ports use rates depending on the proportion of `true` tokens on their input boolean streams which can also be seen as the probability of a boolean token to be `true`. A SWITCH actor with a proportion of p `true` tokens in its boolean stream, will produce after n firings $n \cdot p$ tokens on its `true` output and $n \cdot (1 - p)$ tokens on its `false` output. A SELECT actor will consume tokens in a similar manner.

In this way, for each separate boolean stream b_i, we get a probabilistic rate p_i, forming a vector $\vec{p}$. The balance equations of a BDF graph include such probabilistic values. For the graph in Figure 11, with p_1 the proportion of `true` token for boolean stream b_1 and p_2 for b_2, respectively, we get that a non-trivial solution does not exist, unless $p_1 = p_2$. BDF describes the graphs which have non-trivial solutions for all values of $\vec{p}$ as *strongly consistent*. On the contrary, graphs that are consistent only for specific values of $\vec{p}$ are called *weakly consistent*. Weakly consistent graphs cannot have guarantees (*e.g.*, liveness, boundedness) on their execution as it depends on the values of the tokens on the boolean steams.

BDF greatly increases SDF expressiveness. In fact, it is shown in [21] that BDF is Turing complete and that the decision of whether a BDF application operates within bounded memory is undecidable as it equates to the halting problem.

BDF was extended by IDF [22]. IDF replaces BDF boolean streams with integer streams, allowing SWITCH and SELECT actors to select one port over many, instead of just two as in BDF.

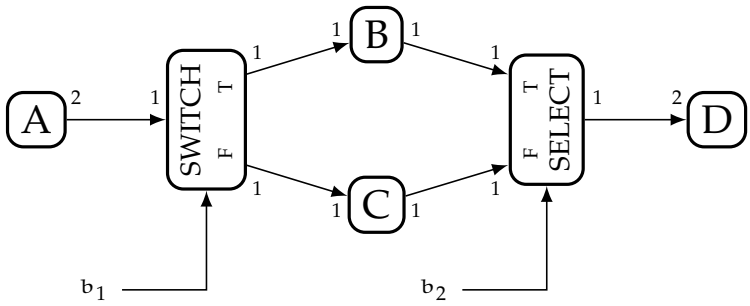


Figure 11: A BDF graph.

2.3.3 *Dynamic Rate Models*

This section presents models that may change port rates dynamically at run-time, altering the amount of data exchanged between actors at run-time.

Parametric Synchronous Data Flow

Parameterized Synchronous Data Flow (PSDF) [11] introduces a set of parameters in the actor definition of SDF. These parameters may control the internal functionality of the actors but may also affect the data flow behaviour of the graph as they can be used in the definition of port rates.

The PSDF MoC is organized in an hierarchical manner: each PSDF actor, unless it is primitive, is itself a PSDF graph, a *component*. Each component consists of three subgraphs, the main data flow subgraph (*body*) that implements the main functionality, and two auxiliary subgraphs (*init* and *subinit*), which change the parameters of the body. The links between the subgraphs of a component and its parent component can be either standard data flow links or links propagating parameter values, called *initflow*.

The *subinit* subgraph changes parameters affecting only the internal functionality of the actors in the body of the component. It can have both data flow inputs and *initflow* inputs. In this way the parameters set by the *subinit* graph can be data dependent and change values within the iteration of the component.

The *init* subgraph changes parameters that also affect data flow behaviour, such as port rates. The *init* graph can only have *initflow* inputs and its execution is not data-dependent. When the parent component invokes a child, first the *init* is fired once to set its parameters and then the iteration of the *subinit* and *body* graphs takes place. PSDF restricts the execution of the *init* graph like that, to ensure that the interface between the parent component and the child remains consistent throughout an iteration of the child. In this way, PSDF restricts changes of parametric port rates *between* iterations of a component. Both *init* and *subinit* outputs are *initflow*, carrying parameter values and not part of the main data flow.

In Figure 12 a PSDF component is shown. The component has two sets of data flow inputs, one to connected to the *subinit* graph and one to the *body*. There is also an *initflow* carrying parameter values from the parent component. In this example, the body has three functions and function f_2 is configured with two parameters g and p. g changes the functionality of f_1 while p sets its output rate.

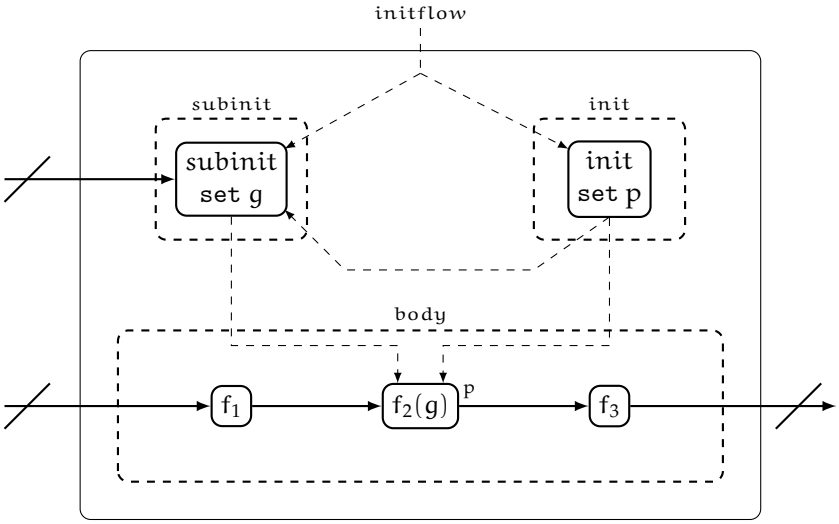


Figure 12: A PSDF component.

When the component is fired, first the *init* graph is fired and it sets parameter p and potentially other parameters. Finally, the rest of the graph executes as in the SDF model, with *subinit* fired first to set the value for g. Within the iteration *subinit* may fire multiple times to change the value of g but *init* fires only once.

PSDF is analyzed in the same fashion as SDF, but every possible configuration of its subgraphs needs to be taken into account. A configuration of a component is a possible assignment of all its parameters. Once all the parameters have taken values, the graph is reduced to an SDF graph, and all SDF analyses can be used.

Each hierarchical component is analyzed separately. A PSDF graph is consistent if of its components are consistent. If all possible configurations of a component are consistent, then the component is said to be *locally synchronous*, if all of them are inconsistent, the component is called *locally asynchronous* and in all other cases, *partially locally synchronous*. The exact methodology of analyzing a PSDF graph is not described in [11], however, an informal set of conditions is given

A. All PSDF graphs (*init*,*subinit*,*body*) must be locally synchronous.
B. Both *init* and *subinit* graphs need to produce exactly one token on each of their outputs, every time they complete an iteration.
C. The data flow inputs of *subinit* must not depend on a parameter.
D. Both the inputs and the outputs of the *body* must not depend on a parameter.

PSDF has been successfully used to parameterize existing DSP models. It has been proposed as a meta-modeling technique to be used on top of various data flow MoCs besides SDF (*e. g.*, CSDF). However, its practical parametrization of the SDF model is not formalized enough to provide strong guarantees (boundedness, liveness). Moreover, the addition of the auxiliary graphs (*init* and *subinit*) increases the design complexity and makes it less intuitive. It is not clear, for instance, when a value of a parameter is set and how often it changes.

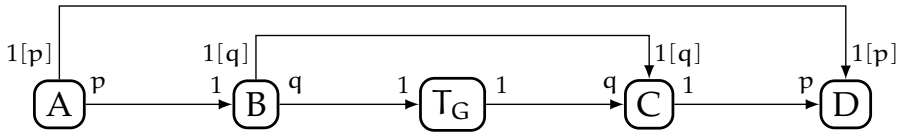


Figure 13: Example of a VRDF task graph.

Variable Rate Data Flow

Variable Rate Data Flow (VRDF) [122] adds parametric rates to the original SDF model. VRDF focuses on data flow graphs that derive from task graphs, which are Directed Acyclic Graphs (DAGs). The VRDF graph that results from a task graph, also models the sizes of the buffers on the edges, resulting in a strongly connected graph.

VRDF restricts the usage of parametric rates either on a single actor or in pairs. When in pairs, for each actor with an output port with a parametric rate, there must be another actor with an input port with a matching parametric rate. These actor pairs must be well-parenthesized: If there is a pair using parameter p, then another pair using parameter q can be nested within the actors of the first pair but cannot interleave (Figure 13).

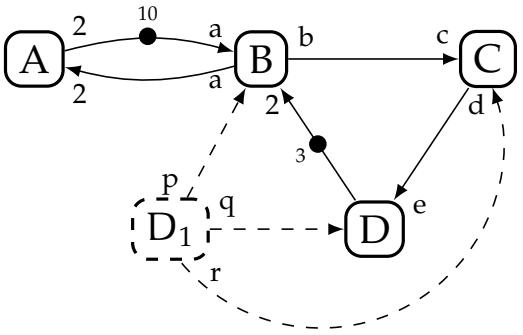
VRDF allows a parameter to take a zero value, disabling the production (*resp.* consumption) of tokens on (*resp.* from) an edge. This is a similar behaviour with BDF when a `false` boolean value deactivates the port of a SWITCH or SELECT actor. However, VRDF avoid the unboundedness problem that arises in BDF because of its requirement of parameters used in pairs as described earlier. This restriction guarantees that a consistent VRDF graph is also bounded. To check the consistency of a VRDF graph, its system of balance equations of the graph can be solved symbolically, producing a parameterized repetition vector. Parameter values propagate through extra edges between the two actors of the pair. These edges have production and consumption rates of 1 and have zero initial tokens.

Figure 13 gives an example of a VRDF task graph. Actors A and D use the parameterized rate p on their output and input ports respectively. Another pair of actors is nested between them, actors B and C using parameter q. Two edges ($\overline{AD}$, $\overline{BC}$) propagate the parametric values with production and consumption rates of 1. A task graph, T_G, is located between the the actors B and C. Solving the balance equations of the graph, we find that actors A and D have solutions of 1, actors B and C parametric solution of p while T_G has a parametric solution equal to pq.

VRDF proposes a clean solution to increase the expressiveness of SDF with parametric rates. However, it imposes many restrictions; task graphs should be acyclic and the parametric ports come only in matching pairs.

Scenario-Aware Data Flow

Scenario-Aware Data Flow (SADF) [116] is a modification of the original SDF model inspired by the concept of system scenarios [52]. SADF introduces a special type of actors, called *detectors*, and enables the use of parameters as port



Rate	s_1	s_2
a	2	1
b	3	2
c	1	1
d	2	1
e	3	1
p	1	2
q	3	4
r	2	4

Figure 14: An SADF graph with its scenarios.

rates. Detectors, detect the current scenario the application operates in and change the port rates of the graph accordingly.

Each detector controls a set of actors. These sets do not overlap, that is each actor is controlled by a single detector. The detectors are connected to each actor with a control link, a data flow edge which always has a consumption rate of 1. When a detector fires, it consumes tokens from its input edges and selects a scenario. Based on the detected scenario, the detector sets its output rates that are parameterized and produces control tokens on all output edges. When an actor fires, it first reads a token from the control link that configures the values of its parameters, and then waits to have sufficient tokens on its input edges.

The set of possible scenarios is finite and known at compile time. A scenario is defined by a set of values, one for each parameterized rate. A production (*resp.* consumption) rate of any edge can take a zero value if and only if the corresponding consumption (*resp.* production) rate takes also a zero value in the same scenario configuration.

An example of an SADF graph is given in Figure 14. For better readability rates of 1 are omitted. The graph is composed by 1 detector and 4 actors. Actors B, C and D are controlled by the detector via the control links shown with dashed lines. Actor A is not controlled by any detector which means that it always operates in the same scenario – all its port rates are static. The possible scenarios the detector may select are s_1 and s_2 (see Figure 14). Parametric rates p, q and r are the production rates of the detector on the control links for each scenario.

When the graph executes, the detector fires first choosing a scenario, say s_1. The detector will produce one control token to B, three tokens to D and two tokens to C (because of the values of p, q and r in scenario s_1). These tokens set the parameters for each actor according to the scenario. For example, actor B will execute with $a = 2$ and $b = 3$. When the control tokens are consumed, the actors wait for the detector to produce new control tokens for the next scenario.

As all scenarios are known at compile time, SADF is analyzed by analyzing all possible SDF graphs that result from each scenario. SADF requires the solutions of the detectors to be 1 for all scenario configurations. Hence, the detectors cannot change parameter values within an iteration. This restriction is true for the strongly consistent SADF, for which analyses for boundedness and liveness are provided in [116]. Weakly consistent SADF graphs may support changing of

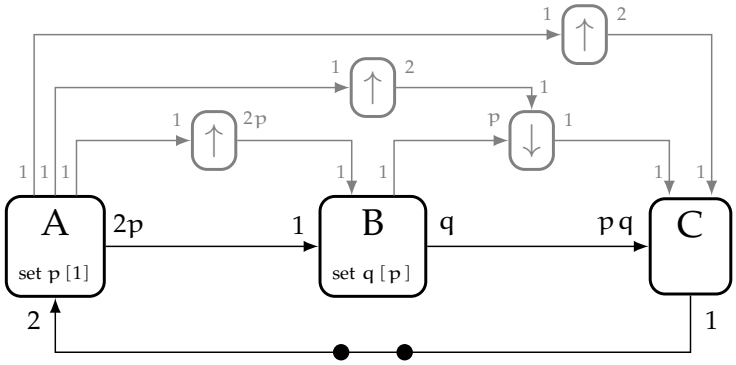


Figure 15: An SPDF graph with its parameter propagation network.

rates and topology within an iteration, but in general they are not statically analyzable. When referring to the SADF model, we will always refer to the strongly consistent version of the model.

SADF resembles CSDF in the sense that it uses a fixed set of possible rates on each port. However, it does not impose their ordering at compile time. In contrast with other models that use parametric rates, SADF does not require a parametric analysis as all configurations can be analyzed separately as in SDF at compile time but this approach may be costly when the number of scenarios is large.

Schedulable Parametric Data Flow

Schedulable Parametric Data Flow (SPDF) [43] is a data flow MoC that was developed to deal with dynamic applications where the rates of an actor may change within an iteration of the graph.

SPDF uses symbolic rates which can be products of positive integers or symbolic variables (parameters). The variable values are set by actors of the graph called *modifiers*. Actors that have parameters on their port rates or at their solutions are called *users* of the parameter. The parameter values are produced by the modifiers and propagate towards all the users through an auxiliary network of upsamplers and downsamplers.

Modifiers and users have *writing* and *reading* periods respectively. These indicate the number of times an actor should fire before producing / consuming a new value for a parametric rate. The writing periods are defined by an annotation under each modifier of the form `set param[period]`. The reading periods are calculated by analyzing the graph.

Not all writing periods are acceptable. Some may cause inconsistencies and SPDF introduces safety criteria and analyses to check whether an SPDF graph satisfies them. These analyses rely on the notion of *regions* formed by the users of each parameter. SPDF demands that for a parameter to have a safe writing period, the subgraph defined by its region needs to complete its local iteration before the parameter changes value. The parameter regions may overlap, as long as all criteria are satisfied. This is called the *period safety* criterion. There is also another safety criterion but we will not go into more details here.

MODEL	DYNAMIC RATES		DYNAMIC TOPOLOGY		STATIC ANALYSES
	BETWEEN ITERATIONS	WITHIN ITERATION	BETWEEN ITERATIONS	WITHIN ITERATION	
SDF	○	○	○	○	●
CSDF	○	○	○	○	●
BDF	○	○	●	●	○
IDF	○	○	●	●	○
PSDF	●	●	○	○	●
VRDF	●	○	●	○	●
SADF	●	○	●	○	●
SPDF	●	●	○	○	●

Table 1: Comparison table of expressiveness and analyzability of data flow models

A sample SPDF graph is shown in Figure 15. The graph has two parameters, p and q. The modifier of p is actor A with writing period 1 and the modifier of q is actor B with writing period of p. In gray is shown the auxiliary network for parameter communication that propagate the parameter values. The region of parameter p is $\{A, B, C\}$ and that of q is $\{B, C\}$.

SPDF graphs can be statically analyzed to ensure their boundedness and liveness. These analyses rely on the symbolic solution of the balance equations and the satisfaction of safety criteria mentioned above. Moreover, for liveness, SPDF checks the liveness of all directed cycles and demands that there is a directed path from each modifier to all the users.

Compared to other parametric models, SPDF provides the maximum flexibility as far as the changing of the parameter values are concerned. However, this increased expressivity makes scheduling SPDF applications very challenging because the data dependencies are parametric and can change any time during execution; in contrast with other parametric models where a schedule can be found at the beginning of an iteration, in SPDF graphs parameters may change within the iteration, demanding a constant reevaluation of the schedule.

2.3.4 *Model Comparison*

To sum up, we provide Table 1 comparing the dynamic features of the models mentioned in the previous sections. The SDF and CSDF MoCs offer no dynamism at all. Although CSDF seemingly change both rates and topology within its iteration, its translation to an HSDF graph indicates that it is not more expressive that SDF.

The BDF and IDF MoCs allow the topology graph to change, however, they do not allow changes in the rates of the graph. Moreover, they lack static analyses for boundedness and liveness.

The PSDF MoC, provides dynamic rates that may change within an iteration, *i. e.,* a child component can change its internal rates many times during the iteration of the parent. Yet, PSDF does not provide dynamic topology and its analyses are not well-defined.

The VRDF MoC as well as the SADF MoC provide both dynamic rates and dynamic topology, however, only in-between iterations. Both have other limitations not captured by the table, like the restriction of the VRDF model on dynamic rates coming in matching pairs and the requirement of SADF for all scenarios to be known and analyzed at compile time.

Finally, the SPDF MoC supports rate changing in-between and within an iteration but not support any topology change. SPDF is analyzable but due to its complexity it is difficult to schedule efficiently.

Other Dataflow Models

There are many other data flow models of lesser importance. For completeness, we mention some of the more notable ones:

- Scalable Synchronous Data Flow (SSDF) [105] adds a factor n on all the graph rates, allowing the graph to "scale" each time it starts an iteration. In this way, SSDF aims to increase the efficiency of SDF by letting actors firing on larger amounts of data.
- Synchronous Piggybacked Data Flow (SPDF) [92] goal is to allow global state updates without side effects. It introduces global states that allow the controlled, synchronous update of local states of actors throughout the graph.
- Bounded Dynamic Data Flow (BDDF) [85] was inspired by SSDF. BDDF deals with the limitations of SSDF like the absence of data-dependent rates or rates that vary in a periodic manner. BDDF allows dynamic rates as long as the amount of tokens that may accumulate on any edge is guaranteed to be bounded at compile time.
- Cyclo-Dynamic Data Flow (CDDF) [121] extends the set of rates of the original CSDF model to have variable lengths and variable rates each time they are triggered.
- DF* [29] extends the basic SDF model with data-dependent control of the graph and non-deterministic behaviour.
- Heterochronous Data Flow (HDF) [53] enhances the SDF model by adding an internal FSM to each actor that allow them to change port rates after each firing. HDF was developed in an effort to combine different MoCs in a single design.
- Multi-Dimensional Synchronous Data Flow (MDSDF) [88] changes SDF rates to have matrix dimensions, so that instead of producing and consuming tokens, actors produce and consume matrices of tokens. Such an extension makes the development of image processing application more intuitive.
- Enable Invoke Data Flow (EIDF) [99], [101] adds a set of modes to each actor that may change non-deterministically each time the actor is invoked. EIDF aims to facilitate rapid prototyping of applications.
- Core Functional Data Flow (CFDF) [99], [101] is a subset of EIDF where the changing between actor modes is deterministic.
- Parameterized and Interfaced Dataflow (PiMM) [37] is a parameterized data flow model that focuses on hierarchical composition of applications. PiMM extends the previous work on interface-based hierarchy of the SDF MoC [97]. It can be seen as an evolution of the PSDF MoC.

		SCHEDULING	
	MAPPING	ORDERING	TIMING
fully dynamic	run-time	run-time	run-time
static-assignment	compile-time	run-time	run-time
self-timed	compile-time	compile-time	run-time
fully static	compile-time	compile-time	compile-time

Table 2: Mapping and scheduling taxonomy

This list of data flow models remains non-exhaustive but the models we have described in this section illustrate the basic principles and motivation behind data flow models.

2.4 DATA FLOW APPLICATION IMPLEMENTATION

Once a data flow application is developed, it consists of a set of tasks that have data dependencies and other precedence constraints with each other. It is usually represented by *task graphs*. Task graphs are Directed Acyclic Graphs (DAGs) where each node is a task and each edge a data dependency. The implementation of such an application consists of two steps: *mapping* and *scheduling*. Scheduling is further divided into *ordering* and *timing*. The terminology for these steps varies a lot in literature, *e.g.*, sometimes mapping is called assignment. Also the whole implementation procedure is often referred to as scheduling.

Depending on when each step takes place, we get the scheduling taxonomy of Lee and Ha in [77] presented in Table 2. The order that these steps take place is not definitive: Mapping may or may not precede scheduling. In many cases these steps interleave to achieve optimal results.

2.4.1 *Mapping*

Mapping is the *allocation of tasks in space*. It assigns tasks to hardware elements, like processors or other specialized processing units. Mapping deals with the optimization of the utilization of the processing elements and their communication network which includes load balancing of tasks among processing elements, minimization of the interprocessor communication, completion time optimization, *etc*...

Load balancing aims at distributing evenly the tasks on the available processing elements. After load balancing takes place, if the processing elements are not utilized at 100%, the voltage and frequency of the processing elements can be adjusted to slow their operation and reduce their power consumption. *Interprocessor communication* needs to be taken into account as it may be very expensive both time-wise and power-wise to move large amounts of data from a processor to another. *Completion time optimization* is important when it comes to heterogeneous architectures as many tasks of an application can be significantly sped up if they are allocated in the correct processing element.

Mapping can be seen as a bin packing problem which is known to be *NP-Complete* [44]. There are many heuristics in the literature for mapping task graphs, some of them are First-Fit Decreasing, Next-Fit Decreasing, Best-Fit Decreasing and Worst-Fit decreasing [60] typically used for tight fit in low processor availability, and First-Come First-Serve mainly used when the available processors are abundant.

In the following, we consider a static mapping where each actor is mapped to a separate processing element. The motivation behind this decision is that, in streaming applications with computation intensive actors, it is a usual practice to implement actors in hardware or use dedicated processors for a single actor.

2.4.2 *Scheduling*

Scheduling is the *allocation of tasks in time*. It consists of two steps: *ordering* and *timing*. Ordering defines the execution order of the tasks. Timing assigns an integer value to each task indicating the exact time at which the task will start its execution. Scheduling mainly deals with the optimization of the application performance and memory utilization. It focuses on optimizing the performance of the application (*i. e.,* maximizing throughput and minimizing latency) as well as its memory footprint (*i. e.,* minimizing code size and the memory used for data).

Parametric data flow actors usually have data-dependent execution time and hence static timing is not an option. Hence, based on the taxonomy of Table 2, we focus on *self-timed* scheduling.

Scheduling, just like mapping, is known to be *NP-Complete* [44]. There are many scheduling techniques used in task graphs, reused from classical job-shop scheduling [26]. These techniques can also be used for SDF graphs, once the graph is converted into a DAG [79].

To convert an SDF graph into a DAG, first the graph is transformed into a homogeneous SDF graph, so that each firing of an actor is also a task, and remove the edges with initial tokens. The liveness property of the original graph ensures that the resulting task graph is acyclic. The technique is described in detail in [79].

Task Scheduling heuristics

Task scheduling heuristics are classified in various categories such as list scheduling algorithms, clustering algorithms, guided random search methods and task duplication algorithms. Here, we mention some of the heuristics developed over time.

• *List Scheduling Heuristics* assign priorities to tasks based on which tasks are ordered in a list. Some representative list scheduling algorithms are Dynamic Level Scheduling (DLS) [110], Heterogeneous Earliest Finish-Time (HEFT) [118] Mapping Heuristic (MH) [40] and Levelized-Min Time (LMT) [64].

• *Clustering Heuristics* try to place heavily communicating tasks on the same processing element, trading-off parallelism with interprocessor communication. Sample heuristics are presented in [18] and in [17].

• *Task Duplication Heuristics* try to utilize processor idle time to duplicate tasks and reduce the waiting time of the precedent tasks. A representative heuristic is Task Duplication based Scheduling (TDS) [103].

• *Guided Random Search Heuristics* are mostly genetic algorithms used for task mapping and scheduling. Examples are Push-Pull [71] and Problem-Space Genetic Algorithm (PSGA) [38].

However, these approaches are not applicable in more expressive data flow MoCs where the number of firings can be parametric.

Data Flow Scheduling

Throughout this thesis we represent schedules with strings showing the firings of the actors. We use ';' for sequential execution, '$\|$' for parallel execution, superscripts for repetition and parenthesis for nested loops over a smaller schedule. For instance, the sequential schedule

$$(A^3;\ B^2)^4;\ C^2$$

fires actor A three times followed by actor B two times. This execution is repeated four times before actor C is fired twice.

Sequential data flow schedules can be regarded as infinite loops over a series of firings completing an iteration. They are also referred to as *looped schedules*. So, once the previous schedule fires C twice, it starts over and repeats indefinitely. In the above schedule, each actor appears, lexically, only once. Such a schedule is called *Single Appearance Schedule*. Single appearance schedules are interesting because they minimize the schedule code size (see Section 2.4.3).

Parallel scheduling of data flow graphs, involves the transformation of the graph into its homogeneous counterpart (HSDF graph) [79]. In this way the available parallelism is exposed and popular techniques from task scheduling can be used.

Consider, for example, the simple SDF graph in Figure 6, and a platform of 2 processing elements (P_1, P_2). The equivalent HSDF graph is shown in Figure 8a. Let us recall that the repetition vector of the graph is $r = [A^2\ B^3\ C]$ and hence the set of tasks are $\{A_1, A_2, B_1, B_2, B_3, C_1\}$. For a mapping, where firings $\{A_1, B_1, B_2, C_1\}$ are mapped on processor P_1 and firings $\{A_2, B_3\}$ on P_2, we can get a self-timed parallel schedule taking into account precedence constraints. So, the ordering for processor P_1 is (A_1, B_1, B_2, C_1) while for P_2 it is (A_2, B_3). The precedence constraints of tasks in different processors are $B_2 > A_2$, $C_1 > B_3$. The mapping and ordering of the tasks is shown in Figure 16.

The execution on processor P_1 proceeds as follows: At time instant t_1 the processor starts execution with A_1. Once A_1 is finished it continues with B_1. At time instant t_3 the execution of B_2 starts once both B_1 and A_2 have finished execution. Similarly, C_1 starts executing at time instant t_4. If the timing of each task was known, a fully static schedule would be possible.

The above schedule fires each actor as soon as it is available and is called an As Soon As Possible (ASAP) schedule. ASAP schedules are known to provide maximal throughput [111].

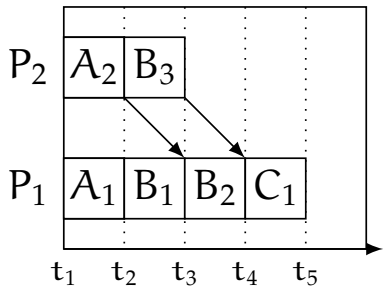


Figure 16: Mapping and ordering of the graph in Figure 6 on processors P_1 and P_2.

Generally, the scheduling of a data flow graph focuses on the firings of one iteration, which is then repeated periodically. Sometimes the repetition vector is multiplied by a factor u, the *unfolding* factor. In parallel schedules, this allows the concurrent execution of firings belonging in different iterations, enabling the exploitation of pipelining. This gain in parallelism comes at the price of a more costly in computation and/or larger in code size schedule.

We classify the self-timed schedules of data flow graphs in three categories depending on the kind of ordering decisions that take place at run-time: *static*, *quasi-static* and *dynamic*. A schedule is *static* if all ordering decisions are taken at compile-time. A schedule is *quasi-static* if the only decisions taken at run-time are decisions that depend on parameter values[1]. A schedule is *dynamic* if it is neither static nor quasi-static.

Typically, static schedules consist of series of firings and for-loops with static boundaries. Quasi-static schedules consist of firings, for-loops with static or parametric boundaries, and if-then-else structures with conditions on parameter values. Dynamic schedules may include computation taking into account external factors and variables such as power consumption to manipulate the schedule at run-time.

The term *quasi-static* has taken various definitions in the literature as is presented in the following sections. Our definition is compatible with most existing definitions. The general intuition behind the term, however, remains the same, it is a distinct class of schedules standing between purely static and dynamic schedules.

In general, static and quasi-static schedules are preferred as they often offer better performance due to lower scheduling overhead. However, a dynamic schedule may prove more flexible and in certain cases more efficient. Finally, there is a trade-off between the code size of the schedule and its dynamicity; a static schedule often occupies more memory than a dynamic one as it keeps all the precomputed information stored, instead of computing it at run-time.

2.4.3 *Scheduling Optimization Criteria*

Scheduling juggles with the optimization of many different, and often conflicting, criteria. These mainly concern the memory usage (*code* and *buffer size*) and the performance of the application (*latency, throughput*).

1 In this way, quasi-static schedules make sense only for parameterized data flow MoCs.

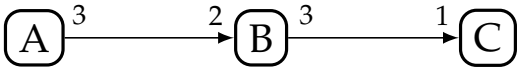


Figure 17: An SDF graph.

Throughput indicates the performance of an application. It is a very important factor, mainly in real-time applications that demand a minimum throughput to be sustained. In some cases maintaining sufficient throughput can be crucial. *Latency* is a metric that measures the response time of an application. It is important in applications like video conferencing, telephony and on-line games where it needs to be kept to a minimum. *Code size* refers to the length of the code implementing the schedule. In embedded systems memory can be sparse and minimizing the size of the schedule is sometimes crucial. Similarly, *memory used for data* needs to be optimized.

These criteria can be analyzed at compile-time for SDF graphs and help take scheduling decisions. Below we discuss the underlying trades-offs between these criteria when scheduling data flow graphs.

Code size

Code size affects other criteria as removing part of the schedule code will most certainly come in conflict with the optimization of some other aspect of the schedule.

We give below a simple example showing the trade-off between code size and buffer size of an SDF graph. Consider the graph in Figure 17. The repetition vector is $r = [A^2\ B^3\ C^9]$. One possible schedule is:

$$A^2;\ B^3;\ C^9$$

This schedule has each actor appearing only once and is called *single appearance schedule*. In this way, the code size of the schedule is minimized. However, the buffer size needed to store data on the graph is 6 tokens on edge $\overline{AB}$ and another 9 tokens on edge $\overline{BC}$. A different schedule could be

$$A;\ B;\ C^3;\ A;\ B;\ C^3;\ B;\ C^3$$

This schedule needs a buffer size of 4 tokens on edge $\overline{AB}$ and a size of 3 tokens on edge $\overline{BC}$ a total size of 7 tokens, a significant reduction on the 15 tokens needed by the previous schedule. However, the code size of the schedule is now 8 actor instances compared to the 3 instances of hte single appearance schedule. A compromise can be found by factoring out common factors on the original single appearance schedule. Factoring out 3 from actors B and C yields:

$$A;\ (B;\ C^3)^3$$

This schedule needs a buffer size of 6 tokens on edge $\overline{AB}$ and 3 tokens on $\overline{BC}$ while keeping the code size of the schedule the same. This trade-off is explored in detail by Bhattacharyya and Lee in [12] and [13].

Buffer Size

The memory usage of the graph needs to be optimized, *i.e.,* the buffers on the edges of the graph need to be minimized. The minimization of buffer sizes is a problem that has two different instances, depending on whether the implementation relies on a single shared memory or individual buffers.

If the edge buffers are implemented as a single shared memory, then the total memory usage of the graph must be minimized. This is similar to the register allocation problem. An approach using model-checking is presented in [57].

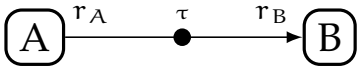


Figure 18: A generic SDF edge with production rate r_A, consumption rate r_B and τ initial tokens.

When it comes to using individual buffer between actors, then the problem is significantly different. In [6] it is proved that for any schedule, the minimum buffer size (β) of an edge $\overline{AB}$ as in Figure 18 is

$$\beta = \begin{cases} r_A + r_B - g + \tau \bmod g & \text{if } 0 \leqslant \tau \leqslant r_A + r_B - g \\ \tau & \text{otherwise} \end{cases} \tag{3}$$

where $g = \gcd(r_A, r_B)$, the greatest common divisor of the two rates. If the buffer size is less, then there is no admissible schedule for the graph and the graph is not live. The buffer size can be modeled with a backward edge, $\overline{BA}$, with $\mathit{init}(\overline{BA}) = \beta - \tau$ as shown in Figure 19.

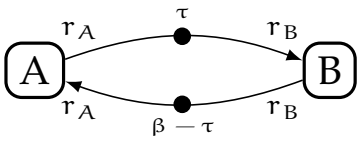


Figure 19: An SDF edge with a finite buffer of size β.

However, using the minimum buffer size has an impact on throughput as it prohibits the parallel execution of the two actors sharing the edge with the minimum buffer. Therefore, there is a trade-off between the buffer size and throughput as there is a trade-off, between buffer size and code size.

Latency

For data flow graphs, it is defined as the time delay between the beginning of the first firing of an iteration and the end of the last firing completing the same iteration as illustrated in Figure 20. Latency is the delay between the instance an input is taken, that is the start of an iteration of the graph, and the corresponding output is produced.

A technique that finds schedules minimizing latency for SDF graphs is presented in [50]. The technique creates a *latency graph* by adding one *source* and one *sink* actor on the graph. Latency is then computed based on the slowest

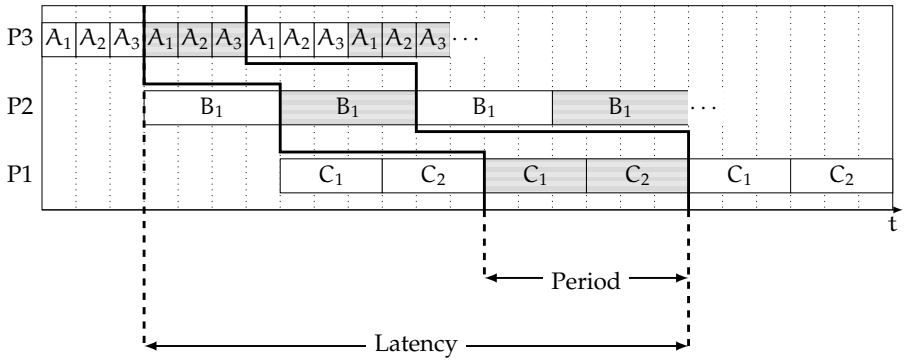


Figure 20: A pipelined schedule on three processors, showing the difference between latency and period.

path between the two actors. The authors study the trade-off between throughput and latency showing that minimizing latency does not necessarily maximize throughput and vice-versa. Moreover, the schedule minimizing latency may not be a single appearance schedule, suggesting a trade-off between latency and code size.

Throughput

Throughput measures the amount of data produced per time unit. In data flow, this corresponds to the number of iterations per time unit. At first sight, it seems that throughput is very similar to latency. However, they are quite different and often antagonistic. This becomes more apparent in Figure 20 where the pipelined execution of an SDF graph is considered. Throughput is the rate at which outputs are produced. So, if we define the *period* of one iteration to be the delay between the production of two consecutive outputs, the throughput of the graph (T_g) can be computed with:

$$T_g = \frac{1}{\text{period}} \tag{4}$$

When a data flow application is deployed on multiple processors, the throughput of an iteration can be increased arbitrarily, if the graph has no directed cycles, by replicating actor instances in different processors and executing them in parallel. When cycles are present, they impose a lower bound on throughput beyond which there is no gain in performance no matter how many processors are added.

This bound is called the *iteration bound*. Such a bound was first explored in [104] and the term *iteration bound* for data flow applications was introduced in [91]. A parallel schedule that achieves an iteration period equal to the iteration bound is said to be *rate-optimal*. Therefore a rate-optimal schedule operates at maximum throughput.

In SDF applications, one way to compute throughput is to convert the graph to the equivalent HSDF graph and to find the cycle with the *maximum cycle mean* – the critical cycle [111]. Intuitively, the cycle with the maximum cycle mean is the one that imposes the cyclic dependency that results in the iteration bound

of the graph. The cycle mean of a cycle is the sum of the execution times of the actors on the cycle, over the sum of the initial tokens on the edges of the cycle.

An algorithm calculating the MCM of a graph in $O(|\mathcal{A}||\mathcal{E}|)$ time, is given in [69]. Dasdan and Gupta [34] developed two algorithms based on a technique that unfolds the graph up to $|\mathcal{A}|$ number of times. The algorithms compute the MCM in $O(|A_U| + |E_U|)$ and $\Theta(|A|^2 + |E_U|)$ time respectively, where A_U and E_U the are actors and the edges of the unfolded graph. However, since the conversion to HSDF may greatly increase the size of the graph, sometimes exponentially, this approach is not always practical.

An alternative conversion to HSDF method is proposed by Geilen [45]. The proposed conversion produces a reduced HSDF graph (*i.e.*, it has less actors than the one produced with the original algorithm). The new HSDF graph is not functionally equivalent to the original SDF as the approach does not aim to the expansion of one actor per instance, but rather to a graph that has the same throughput and latency as the original graph.

Another throughput calculation is proposed in [49], where the execution state space of the original graph is explored until a state is visited twice. This cycle in the state space means that the execution has finished the initial transitional behaviour, the *prologue*, and reached a periodic behaviour, the *steady-state*. The duration of such a period can be used to calculate throughput.

The method is formulated using $(max, +)$ algebra [46]. A matrix G_M keeps the time delay between the production of a token in the current iteration and the production of the equivalent token in the next iteration, characterizing the behaviour of the graph in time. To calculate the values of the matrix, a symbolic execution of one iteration of the graph takes place. Then, it is shown that the eigenvalue of the G_M matrix corresponds to the MCM of the graph giving the throughput value.

This new approach performs better than many algorithms that compute the MCM of HSDF graphs [49]. Another benefit of the $(max, +)$ approach is that it can deal with graphs demonstrating more dynamism. For instance, the technique has been applied to SDF graphs with actors with parametric execution times in [51] and to the more expressive SADF model [32].

Throughput is antagonistic with buffer sizes. As mentioned earlier, buffer sizes can be modeled as backward edges forming directed cycles. These cycles may impose a lower iteration bound than the iteration bound of the graph with unlimited buffer sizes, further restricting throughput. Throughput increases as buffer sizes increase, up to an upper bound defined by the iteration bound.

Power consumption

When throughput constraints are satisfied, actors that operate faster than needed can be slowed down to achieve a reduction in power and energy consumption. The most commonly used technique is Dynamic Voltage-Frequency Scaling (DVFS). By scaling the voltage and the frequency of a processing element, one can increase (*resp.* decrease) its execution speed while increasing (*resp.* reducing) its energy and power consumption.

Typically, this scaling happens in discrete pairs of Voltage-Frequency. The adjustments of the scale can happen in various ways. They can be *regular* (interval-

based changes), they can be *fixed at compile-time* or *event based*. In the last case, the decisions are taken at run-time. Indicative control parameters that are used to select a pair on the Voltage-Frequency scale are *buffer occupancy* (typically based on cache misses when it comes to processors) and *worst-case execution times* (typically calculated at compile-time) [56]. Both metrics are indirect links to the throughput of the application which is why the efficient calculation of the throughout of the application at run-time is very beneficial: it allows the scheduler to make decisions on which Voltage-Frequency pair to use for each element in order to minimize the energy consumption, while satisfying the throughput constraints.

Energy-aware task scheduling has been extensively explored ([125], [56], [108], [81], [84], [74]). The typical procedure for throughput constrained applications, like real-time applications, starts with the scheduling of the tasks using Earliest Deadline First and then scaling down Voltage-Frequency on each processor to reclaim slack, a procedure called SimpleVS.

DVFS is explored for the SADF MoC, in [33]. The authors present a method to reduce power consumption in SADF graphs. The approach starts by using minimum frequency at the beginning of each scenario and then iteratively (if the iteration will not meet the deadline), changes the frequency of the critical path, until the throughput constraint is met.

Other Criteria

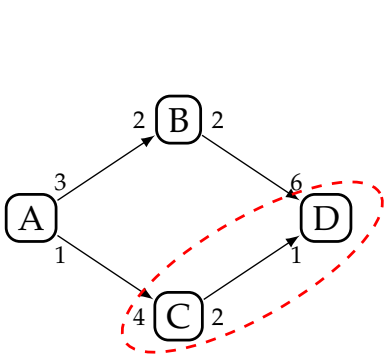
Other criteria have been considered as well. For example, Sriram and Bhattacharyya [111] optimize parallel SDF schedules by minimizing the synchronization points between processors. Also, when it comes to DVFS, lowering the frequency increases the chances of an error. So, reliability is another "criterion" that can be optimized as shown in [1] for simple data flow graphs.

2.4.4 *Scheduling Synchronous Data Flow*

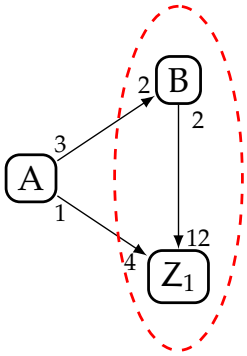
In the above, data flow graphs are first converted into HSDF graphs and then scheduled reusing standard task graph scheduling techniques. However, converting a data flow application into an HSDF graph is not always practical and when it comes to more expressive models it is not even feasible.

In the SDF case, the conversion can end up in an exponential growth in the number of actors. This issue is addressed in [98] with a technique called *clustering* that prevents actor explosion. The problem becomes even more evident in CSDF applications. In the original CSDF paper [15], a scheduling technique that avoids the conversion to HSDF is presented. Another approach [94], uses clustering to convert the CSDF graph into a SDF graph. In both cases, some of the potential parallelism expressed in the original CSDF graph cannot be exploited.

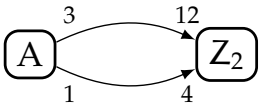
Various intermediate representations have been assumed to facilitate the scheduling of SDF graphs. In [2], an interprocessor communication graph and a synchronization graph are used to model the self-timed execution of SDF graphs. These graphs are produced based on the mapping and ordering of the tasks. Damavandpeyma *et al.* [31] propose a method called Decision State Modeling (DSM). The method models a given mapping and ordering by adding aux-



(a) Initial SDF graph.



(b) After clustering actors C and D into Z_1.



(c) After clustering actors Z_1 and B into Z_2.

(d) After clustering actors Z_2 and A into Z_3.

Figure 21: Clustering technique applied on a simple SDF graph. Eventually, the graph is clustered into a single actor

iliary actors and edges based on the *decision state space* of the given schedule (*i. e.*, a state space modeling all the possible transitions between the states of the processors based on the given ordering). The resulting graph is significantly smaller than those resulting from the conversion to HSDF.

Clustering is a standard scheduling technique for SDF graphs. The main principle is the replacement of a set of actors with a single actor. When the single actor is fired, a local schedule executes, firing the clustered actors. Clustering is usually applied iteratively until the whole graph is clustered into a single actor. It is easy to produce the schedule of the graph afterwards, by iteratively replacing the execution of each actor with the execution of its clustered actors.

A simple clustering example is shown in Figure 21. The repetition vector of the graph is $r = [A^4\ B^6\ C\ D^2]$. Initially, the actors C and D are clustered into Z_1. The consumption rates of the incoming edges of Z_1 are adjusted accordingly:

$$cns(\overline{BZ_1}) = cns(\overline{BD}) \times \#D/gcd(\#C, \#D) = 12$$

and

$$cns(\overline{AZ_1}) = cns(\overline{AC}) \times \#C/gcd(\#C, \#D) = 4$$

The local schedule of the two clustered actors is easy found to be $C; D^2$. Then, Z_1 is clustered with B in actor Z_2. Similarly, we find $cns(\overline{AZ_2}) = 12$ and $cns(\overline{AZ_2}') = 4$. The schedule of the clustered actors is $B^6; Z_1$. Finally, the two remaining actors, A and Z_2, are clustered into a single actor, Z_3. The schedule of the clustered actors is $A^4; Z_2$. The schedule of the graph is defined by replacing iteratively the schedules of the clusters:

$$Z_3 \to A^4; Z_2 \to A^4; B^6; Z_1 \to A^4; B^6; C; D^2$$

which corresponds to a single appearance schedule.

Not all sets of actors are safe to a cluster. For instance, in the second clustering step of the previous example, actors A and Z_1 cannot be clustered, since the resulting graph would have a directed cycle that did not exist before (see Figure 22). This artificial cycle creates a deadlock, prohibiting the construction of an admissible schedule. The potential creation of directed cycles is the main weakness of the clustering technique as there are graphs that cannot be clustered.

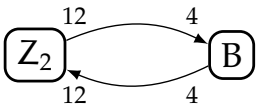


Figure 22: An invalid clustering of the graph in Figure 21a that leads to the creation of a directed cycle, making it impossible to find an admissible schedule.

Bhattacharyya describes a bottom-up approach in [14] called *Acyclic Pairwise Grouping of Adjacent Nodes* (APGAN). APGAN iteratively clusters pairs of adjacent actors until the whole graph is clustered into a single actor, similar to the clustering example we demonstrated earlier. The approach produces single appearance schedules that are optimal as far as the buffer sizes are concerned, when the graph contains no initial tokens.

A different, top-down, approach is presented in [89], called Recursive Partitioning based on Minimum Cuts (RPMC). RPMC iteratively partitions the graph into two subgraphs $\mathcal{G}_L$ and $\mathcal{G}_R$ with a schedule $S_L^{g_L}; S_R^{g_R}$ where S_L is the schedule of $\mathcal{G}_L$ and

$$g_L = \gcd\{\#A | A \in \mathcal{A}_L\}$$

The algorithm continues until each subgraph is composed of a single actor. In [14], it is shown that the two approaches complement each other in the sense that APGAN performs better with SDF with regular structures and RPMC with SDF with more irregularities.

Falk *et al.* [41] assume graphs consisting of both CSDF and SDF actors. Then, actors are clustered and FSMs are used to express the quasi-static schedule of each cluster, greatly improving performance over a dynamic implementation.

In [16], a method that produces K-periodic schedules for SDF is presented. The schedule is called K-periodic as there is a vector K that keeps the period of execution of each actor. The paper indicates how K-periodic schedules that achieve maximum throughput can be produced for SDF graphs. The benefit of K-periodic schedules is that, in contrast with other approaches there is no need for a *prologue* in the schedule.

Govindarajan *et al.* [55] present a method that minimizes the buffer requirements while keeping the rate-optimal (*i.e.*, maximum throughput) execution using linear programming. Stuijk *et al.* [114] explores the trade-off for CSDF graphs.

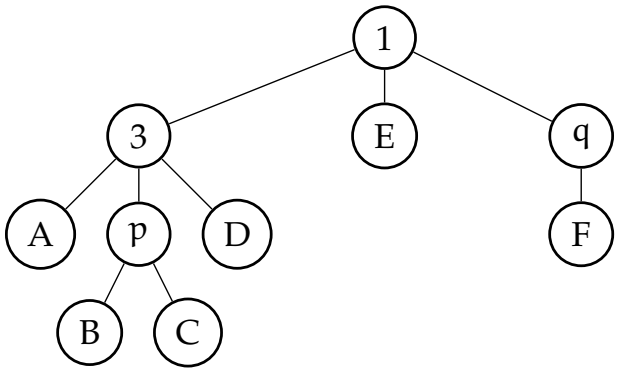


Figure 23: GST of sequential schedule $(A;(B;C)^p;D)^3;E;F^q$.

2.4.5 *Scheduling more Expressive Data Flow Graphs*

When it comes to graphs of more expressive MoCs, conversion to HSDF is not available at compile-time and it is too expensive to be used at run-time. That is either because of the parametric nature of the repetition vector or because of the dynamic topology changes. For this reason, production of schedules for more expressive MoCs is of particular interest.

Dynamic Port Rates

Lee and Ha [75][58], introduced scheduling techniques for data-dependent data flow graphs. More specifically, they coin the term *quasi-static* to refer to a schedule statically produced at compile-time where the only computations left at run-time are the absolutely necessary ones, *i. e.*, a loop based on the value of a parameter unknown at compile-time. This was the prelude to what later would be the basis for scheduling parameterized graphs; quasi-static scheduling is used in [10] to schedule PSDF graphs. This time the term is specified as *parameterized looped schedules*, and Bhattacharya describes a clustering technique for scheduling PSDF graphs.

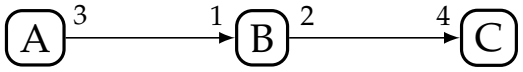
In [73], Ming-Yung Ko *et al.* introduce a representation of sequential schedules called Generalized Schedule Tree (GST). These trees have leafs as actor firings while parent nodes indicate the iteration count of its children. For instance, the sequential schedule

$$(A;(B;C)^p;D)^3;E;F^q$$

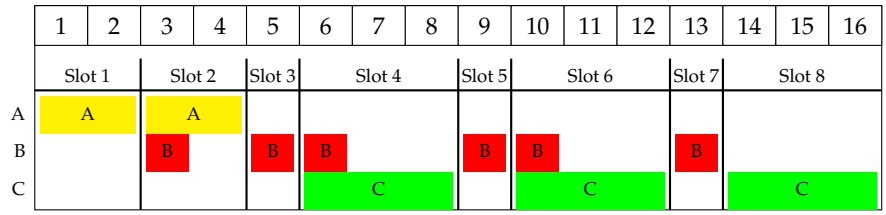
can be expressed with the GST of Figure 23.

Plishker *et al.* [100] introduces an approach to reuse the APGAN algorithm in more dynamic data flow graphs, producing schedules represented by GSTs. Kee *et al.* [70] apply the generalized schedule trees technique to optimize buffer sizes of PSDF graphs.

GSTs are expanded into Scalable Schedule Trees (SSTs) in [109]. SSTs allow nodes to have dynamic iteration count as well as the guarded execution of the leaves where the actor firing corresponding to the leaf can be skipped if there are insufficient data on the actor inputs. Finally, SST features *arrayed children nodes* where the children of a node are executed based on a pattern. They are



(a) SDF graph



(b) Slotted ASAP schedule

Figure 24: An SDF graph and its respective ASAP slotted schedule.

used to express parameterized schedules in [124]. However, SSTs as well as GSTs are limited to sequential schedules.

Apart from code and buffer size, throughput for parameterized MoCs is explored as well. In [122], minimum buffer sizes for VRDF graphs with throughput constraints are computed.

Dynamic Topology

When it comes to dynamic topology, Buck [21] explores the generation of bounded length sequential schedules for BDF graphs. He proposes a clustering technique that achieves such schedules, However, because of the nature of BDF, the resulting schedules may not strictly implement one iteration of the graph and boundedness is not always guaranteed.

Parallel scheduling of BDF graphs is also addressed by Ha and Lee [58]. They preserve the boundary of each iteration whatever the boolean values, allowing static timing to be used. This is achieved by adding slack in the relevant parts of the schedule.

Ko *et al.* considers graphs with dynamic topology [72], and proposes a method to produce multiple sequential schedules at compile-time that are then used at run-time according to the current graph topology. The approach aims at the reduction of the code size schedule and the buffer sizes considering both single and multiple appearance schedules.

Slotted Scheduling

A special type of parallel scheduling is *blocked* or *slotted scheduling* [58]. Slotted scheduling is a technique where the schedule is expressed by a sequence of slots. In each schedule slot, a set of actors fire in parallel. A slot cannot start its execution before all the actors from the previous slot have completed their firing.

Slotted scheduling is sometimes preferred, as it makes it easier to express parallel schedules. Without slots, the synchronization between processors have to be explicitly expressed. This synchronization heavily depends on the parameter values in graphs with parametric rates. Slotted scheduling simplifies the expression of quasi-static schedules.

The main drawback of slotted scheduling is that, due to the hard synchronization between slots, slack may be introduced. For this reason, the main goal of slotted scheduling is to group actor firings with similar durations in the same slot. Techniques like retiming [83], move the graph initial tokens changing the graph data dependencies, and loop winding [54], a pipelining technique similar to functional pipelining [63], can be used to reduce the introduced slack. Moreover, the slotted schedule itself can group actors with similar execution times in the same slot to further reduce the slack.

Throughout the thesis, we present slotted schedules in two ways: as a set of *slot sequences* for each actor, or as a single *schedule stream*.

Slot sequences are sequences of events that indicate whether an actor fires ($\mathcal{F}$) or remains idle ($\mathcal{E}$) during a slot. The events are separated by the ';' operator that indicates sequencial execution. Indexes are used to indicate repetition of the same event. We use parenthesis to group parts of the sequence that may iterate. For the slotted schedule in Figure 24b, the slot sequences of the actors are:

$$\begin{aligned} A &: \mathcal{F}^2 \\ B &: \mathcal{E};\mathcal{F}^6 \\ C &: \mathcal{E}^3;(\mathcal{F};\mathcal{E})^2;\mathcal{F} \end{aligned}$$

Schedule streams are produced by merging the slot sequence of each actor together, executing in parallel the events from each slot sequence in a single slot. Parallel execution is indicated with the '$\|$' operator. The merging procedure is described in detail in Appendix A.1. The merging of the schedule streams above yields:

$$A;(A\|B);(B;(B\|C))^2;B;C;$$

Assuming that the timing of the actors is known, *e.g.*, $t_A = 2$, $t_B = 1$ and $t_C = 3$ time units, we get the slotted schedule shown visually in Figure 24b. We see that when actors B and C are grouped together, actor B is delayed until the firing of actor C is completed. For instance, it would be better to trigger the 4^{th} firing of B right after its 3^{rd} firing in the 4^{th} slot instead of firing it in a separate slot, but this is not allowed by the slotted model. In this way, when actor B is grouped in the same slot with actor C, it stands idle until the slot is finished, introducing slack in the schedule.

2.5 SUMMARY

In this chapter, the current state-of-the-art of Models of Computation that focus on parallelism was presented. Two main steps of system design have been discussed: *modeling* and *implementation*.

Data flow modeling has been a natural choice for the development of highly parallel streaming applications. The intuitive design and the exposure of the underlying parallelism makes data flow a very attractive solution. Moreover, data flow MoCs provide compile-time analyses for both qualitative and quantitative properties of the system. In this way, the development procedure becomes faster and less error-prone, resulting in more reliable and high quality products.

Still, new data flow MoCs are needed as current applications get more complex and demand increased expressiveness. Proper combination of both topological and data rate dynamism has not yet been achieved in any of the existing models. These features are desirable in modern streaming applications that process encoded data. Based on the encoding, the amount of data as well as the parts of the application that process it may vary, as discussed in Section 1.3.

Implementation on many-core platforms is very challenging as there are many different conflicting parameters to take into account. Many task scheduling heuristics have been developed, and although they can be reused in less expressive models like SDF, they quickly become obsolete when more expressive models are employed. New techniques have been developed to schedule parametric data flow graphs. However, they are limited to sequential schedules. Parallel scheduling of parametric MoCs is crucial for their efficient implementation on many-core platforms.

Finally, various optimization criteria were presented. It is evident that throughput plays a prominent role in the development of an application. Most streaming applications have real-time constraints that are reflected on throughput constraints. To satisfy these constraints, appropriate buffer sizes need to be selected and power consumption needs to be fine tuned.

However, when it comes to parametric MoCs, throughput calculation at compile-time is not available since it depends on the parameters which are data-dependent. There have been some efforts to compute the throughput of more expressive MoCs like SADF but when it comes to parametric MoCs throughput computation is still lacking.

In the following, we present a new data flow MoC that provides both topological and data rate dynamism at run-time based on a combination of both integer and boolean parameters. The aim of the model is to increase the expressiveness of previous MoCs, so that streaming applications, like video decoders, can be modeled more efficiently. The new MoC preserves all the static analyses that make data flow modeling so attractive. Our new data flow MoC, Boolean Paramatric Data Flow (BPDF), is presented in Chapter 3.

Furthermore, we propose a scheduling framework that can schedule data flow MoCs on many-core platforms. The framework relies on scheduling constraints that can derive either from the application or the designer. The constraints are analyzed parametrically and a schedule is produced at compile-time when possible. When not, a dynamic scheduler is employed. The framework is presented in detail and applied to BPDF in Chapter 4.

Finally, the thesis addresses the problem of parametric throughput calculation. A method of calculating throughput for MoCs with parametric rates is presented in Chapter 5.

3 BOOLEAN PARAMETRIC DATA FLOW

> Everything flows
>
> — Heraclitus

Data flow Models of Computation (MoCs) are well-suited to develop streaming applications on many-cores. They allow the parallel development of individual parts of an application and their reuse in future applications. Data flow MoCs *expose the task parallelism* of the application, which is essential for an efficient deployment on many-core platforms.

Another feature that makes data flow modeling attractive is its *static analyzability*. Data flow programs can be analyzed to provide guarantees on their execution at compile-time. Properties like liveness and bounded execution can be ensured. It is easy to write deadlocking applications, so *liveness* is important to guarantee continuous execution. *Boundedness* is crucial for embedded platforms which often have limited memory available. Knowing the amount of memory needed by an application at compile-time, allows its static allocation, leading to more efficient implementations.

Many streaming applications have a dynamic behavior. In order to capture them efficiently, two main features of the underlying MoC are needed: *dynamic reconfiguration* of the application topology and *dynamic rates* at which the various components of the application exchange data when executing.

Still, MoCs that combine both kinds of dynamic behaviour are lacking either in expressiveness or analyzability. To overcome this shortcoming, we propose a new MoC, called *Boolean Paramatric Data Flow* (BPDF), which combines integer parameters (to express dynamic rates) and boolean parameters (to express the activation and deactivation of communication channels). Integer parameters can change between each iteration while boolean parameters can even change within an iteration.

When expressiveness increases, analyzability is usually compromised. The major challenge with such dynamic MoCs is to guarantee liveness and boundedness. For BPDF, we provide static analyses which ensure statically these properties.

The chapter is organized as follows: In Section 3.1 we present the syntax and informal semantics of BPDF. In Section 3.2, the static analyses are described. Then, in Section 3.3, some implementation details and subtleties raised by the dynamic features are discussed. Finally, Section 3.4 contains a comparison of BPDF with other parametric MoCs.

3.1 BOOLEAN PARAMETRIC DATA FLOW

BPDF extends SDF by allowing rates to be parametric and edges to be annotated with boolean conditions. This way BPDF combines dynamic port rates with dy-

namic graph topology. Unlike other models, BPDF strictly separates the role of rates, which are directly linked with the graph iteration, from the role of the boolean conditions, which are linked with disabling edges and changing the graph topology. This is achieved by restricting the integer rates of the ports to non-zero values.

3.1.1 *Parametric rates*

BPDF port rates are products of positive integers or symbolic variables. They are defined by the grammar:

$$\mathcal{R} \quad ::= \quad k \mid p \mid \mathcal{R}_1 \cdot \mathcal{R}_2$$

where $k \in \mathbb{N}^*$ and $p \in \mathcal{P}_i$, with $\mathcal{P}_i$ denoting the set of symbolic variables standing for *integer parameters*.

Although it is not needed for the analysis of the model, a maximal value for each integer parameter must be specified for implementation purposes (*e. g.*, assigning sufficient buffer sizes). We will denote the maximum of a parameter p as p_{max}. Therefore, any given parameter p varies in the interval $[1..p_{max}]$.

Unlike the rates of SDF graphs that are fixed at compile time, the parametric rates of a BPDF graph can change between iterations. The mechanism that change the integer values must guarantee that all values are set at the beginning of each iteration. The choice of such a mechanism is implementation dependent and does not interact with the analyses or compilation process.

3.1.2 *Boolean conditions*

Each BPDF edge is annotated by a boolean condition which deactivates the edge when it evaluates to false. These boolean expressions are defined by the grammar:

$$\mathcal{B} \quad ::= \quad \mathtt{tt} \mid \mathtt{ff} \mid b \mid \neg\mathcal{B} \mid \mathcal{B}_1 \wedge \mathcal{B}_2 \mid \mathcal{B}_1 \vee \mathcal{B}_2$$

where `tt` is true, `ff` is false and b belongs to $\mathcal{P}_b$, denoting the set of symbolic variables standing for *boolean parameters*.

Each boolean parameter is modified by a single actor called its *modifier*. In BPDF, a modifier may change a boolean parameter within a graph iteration, using the annotation "b@π" where b is the boolean parameter to be set and π is the *period* at which the boolean changes value. The period of a boolean parameter b is the exact (possibly symbolic) number of firings of its modifier between two successive changes. Once a new value is produced it propagates to all the actors that need it, as discussed in Section 3.3.2.

3.1.3 *Formal definition*

Formally, a BPDF graph is defined as a 10-tuple:

$$(\mathcal{G}, \mathcal{P}_i, \mathcal{P}_b, \mathtt{lnk}, \mathtt{init}, \mathtt{prd}, \mathtt{cns}, \beta, M, \pi_w)$$

where:

- $\mathcal{G}$ is a directed connected multigraph $(\mathcal{A}, \mathcal{E})$ with $\mathcal{A}$ a set of actors, and $\mathcal{E}$ a set of directed edges.
- $\mathcal{P}_i$ is the set of integer parameters.
- $\mathcal{P}_b$ is the set of boolean parameters.
- $\mathtt{lnk} : \mathcal{E} \to \mathcal{A} \times \mathcal{A}$ associates each edge with the pair of actors that it connects.
- $\mathtt{init} : \mathcal{E} \to \mathcal{R}$ associates each edge with a number of initial tokens.
- $\mathtt{prd} : \mathcal{E} \to \mathcal{R}$ associates each edge with its production rate.
- $\mathtt{cns} : \mathcal{E} \to \mathcal{R}$ associates each edge with its consumption rate.
- $\beta : \mathcal{E} \to \mathcal{B}$ associates each edge with its boolean condition.
- $M : \mathcal{P}_b \to \mathcal{A}$ returns for each boolean parameter its modifier.
- $\pi_w : \mathcal{P}_b \to \mathcal{R}$ returns for each boolean parameter its writing period.

As in SDF, we may refer to port rates instead of edge production/consumption rates.

For each boolean parameter, we define its set of users and its frequency, which are used in the analysis of the model in later sections. An actor A is a user of a boolean parameter b if A is connected by an edge whose condition involves b. The set of all the users of b is therefore defined as:

Definition 5 (Users). *The set of users of the boolean parameter* b*, written* $\mathit{Users}(b)$*, is defined as*

$$\mathit{Users}(b) = \{A, B \mid (\overline{AB}) \in \mathcal{E}, b \in \beta(\overline{AB})\}$$

The modifier, $M(b)$, writes b at period π_w, while all the actors in $\mathit{Users}(b)$ read b at period π_r. These are called *reading* and *writing* periods, respectively. The writing periods are defined in the model definition. Then, the reading periods of all actors can be calculated. Boolean parameter communication must ensure that the parameters are written and read at the right pace without introducing deadlocks. The implementation of the boolean parameter communication is discussed in Section 3.3.2. In addition of periods, there is also the notion of *frequency*, which is the number of times a parameter may change within an iteration. It is equal to the number of firings of its modifier during one iteration, divided by its writing period.

Definition 6 (Frequency). *The frequency of a boolean parameter* b *, written* $\mathit{freq}(b)$*, is defined as*

$$\mathit{freq}(b) = \frac{\#M(b)}{\pi_w(b)} \tag{5}$$

Based on the frequency of the parameter, the reading period of each user is calculated, hence for an actor $U \in \mathit{Users}(b)$:

$$\pi_r^U(b) = \frac{\#U}{\mathit{freq}(b)} \tag{6}$$

In this way, all the boolean values that are produced by the modifier within an iteration are also consumed by the users. Reading and writing periods are detailed in Section 3.2.2.

BPDF combines parametric rates and frequent topology reconfiguration as no other dataflow model proposes. Furthermore, as shown in the next section, this gain in expressiveness does not prohibit effective static analyses.

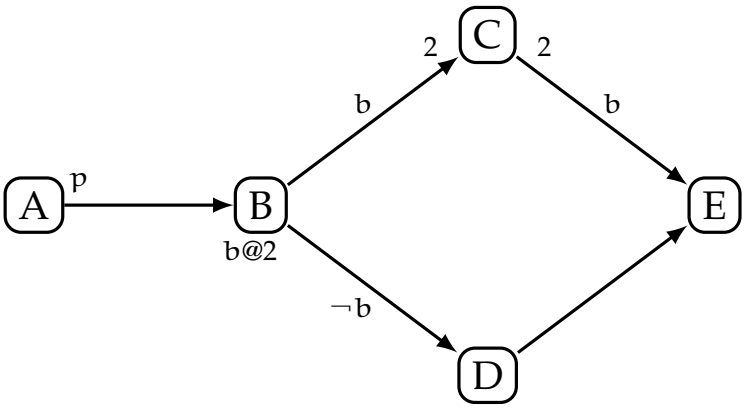


Figure 25: An example of a BPDF graph with an integer parameter p and boolean parameter b

3.1.4 *Example*

Figure 25 shows a simple BPDF graph. Omitted rates and conditions are equal to 1 and tt respectively. The graph uses one parametric rate ($prd(\overline{AB}) = p$) and a single boolean parameter, b appearing in the conditions of edges $\overline{BC}$, $\overline{BD}$ and $\overline{CE}$. The modifier of b is actor B, ($M(b) = B$), with writing period of 2, ($\pi_w(b) = 2$), annotated underneath the actor with b@2. Finally, *Users*(b) = {B, C, D, E}.

The repetition vector is $[A^2, B^{2p}, C^p, D^{2p}, E^{2p}]$, which is computed with a symbolic variant of the classical SDF algorithm (see Section 3.2.1). With the solution of the modifier (B) known, we can compute the frequency of boolean parameter b using Eq. (5):

$$freq(b) = \#B/\pi_w(b) = 2p/2 = p$$

which indicates that parameter b may take p different values within an iteration.

The conditional edges are active only when their condition (here b or ¬b) is tt. When an actor fires, it first evaluates the condition of its edges. Hence, if the actor is a modifier it sets all boolean values. Then, it consumes tokens only from the incoming edges with a boolean condition that evaluates to tt. Then, the actor executes its internal function and produces tokens on its outgoing edges that have a boolean condition that evaluates to tt.

This implies that a completely disconnected actor, *i.e.*, whose edges all evaluate to ff, still fires but does not read or write on any edge. Such a firing may be seen as *dummy* firing. The propagation of boolean values still takes place through auxiliary ports as described in Section 3.3.2. Hence, the actor can continue firing without consuming or producing tokens until one of boolean parameter changes and sets one of the edge conditions to tt.

With that in mind, a sample execution of the graph assuming p is set to 10 is the following: A fires and produces $p = 10$ tokens on edge $\overline{AB}$. Then B fires

and sets the value of boolean parameter b. If b is set to tt, B does not produce tokens on edge $\overline{BD}$. The reading period of D is (Eq. (6)):

$$\pi_r^D(b) = \frac{\#D}{freq_b} = \frac{2p}{p} = 2$$

Hence, D will fire twice without consuming tokens and then will wait for the next boolean value. B will fire a second time without changing the value of b enabling C to fire once. Finally, E will consume the tokens produced by C and D.

If b is set to ff, C is disconnected and it will fire once without producing or consuming tokens before waiting for the next boolean value ($\pi_r^C(b) = 1$). D and E will fire as expected. This continues until each actor has fired a number of times equal to its repetition count (as in SDF).

In the above, we see that BPDF execution is *determinate*. The output of the graph in Figure 25 is independent of the order of execution of the enabled actors.

3.2 STATIC ANALYSES

Static analyses of an application are very important for embedded systems. It provides guarantees, and helps the development of efficient and robust applications. As in SDF and some other data flow models, BPDF can be statically analyzed. In this section we present how a BPDF application can be checked for *consistency*, *boundedness* and *liveness*.

3.2.1 *Rate Consistency*

Like in SDF, rate consistency in BPDF is checked by solving a system of balance equations. The difficulty in BPDF is that there are parametric rates and therefore, a unique integer solution cannot be found. For this, the balance equations are solved *symbolically* instead, and in general, the resulting repetition vector is symbolic.

However, a BPDF graph with n boolean parameters may have up to 2^n configurations, each one with a different system of equations, leading to multiple repetition vectors. Furthermore, the configuration of a BPDF graph may change within an iteration. There is the possibility that the graph changes to a configuration with a different repetition vector than the previous configuration, making reasoning about an iteration very difficult.

For this reason, when checking BPDF consistency, the boolean parameters are not taken into account. This enforces the system to be rate consistent for *all* possible configurations of the graph. Indeed, if the system is rate consistent when all edges are present (enabled), then it is also consistent when one or several edges are removed (disabled). This reflects to the system of balance equations as well because, when removing edges, the resulting system of equations will be a subset of the original system of equations of the fully connected graph (*i. e.*, graph with all edges active regardless the boolean values). The solution of the superset of equations will also be a solution of all subsets corresponding

to different configurations of the graph. In this way, the graph uses the same repetition vector for all possible configurations.

The algorithm used to solve the system of balance equations is a generalized version of the one used for SDF graphs in [6], using symbolic operations instead of integer operations. The algorithm randomly chooses one actor, sets its solution to 1, and recursively solves all other actors, according to the balance equations. A normalization step is needed to yield the minimum positive symbolic integer solution.

Checking rate consistency of all edges maybe considered too strict because it does not take into account the fact that some edges may not be active at the same time (*e.g.*, two edges annotated by b and $\neg b$). On the other hand, it simplifies the understanding and implementation since a graph has a unique (although parametric) iteration vector.

Assuming the graph in Figure 25, we arbitrarily set $\#B = 1$. Then, due to the balance equations we find

$$\left.\begin{array}{rcl} p \cdot \#A &=& \#B \\ 2 \cdot \#C &=& \#B \\ \#D &=& \#B \\ \#E &=& \#D \end{array}\right\} \Rightarrow \begin{array}{rcl} \#A &=& \frac{1}{p} \\ \#B &=& 1 \\ \#C &=& \frac{1}{2} \\ \#D &=& 1 \\ \#E &=& 1 \end{array}$$

To normalize we multiply all solutions with the least common multiple of the denominators, here we get $lcm = 2p$. The resulting repetition vector is therefore:

$$[A^2\ B^{2p}\ C^p\ D^{2p}\ E^{2p}]$$

If the undirected version of the BPDF graph is acyclic, a solution to the balance equations always exists. When the BPDF graph contains an undirected cycle, the graph may be rate inconsistent. There is, however, a necessary and sufficient condition for the existence of solutions. Each undirected cycle

$$X_1, X_2, \ldots, X_n, X_1$$

should satisfy the following condition:

$$(\textit{Cycle condition}) \qquad p_1 \cdot p_2 \ldots \cdot p_n = q_1 \cdot q_2 \ldots \cdot q_n \tag{7}$$

where p_i and q_j denote respectively, the production and consumption rates of edge (X_i, X_j). This condition enforces that the product of "output" rates of a cycle should be equal to the product of "input" rates of this cycle.

Property 1 (Consistency). *A BPDF graph is rate consistent iff all its undirected cycles satisfy the cycle condition (Eq. (7)).*

Proof. To prove Property 1, we consider a cycle

$$X_1 \overset{p_1 \quad q_2}{\longrightarrow} X_2 \overset{p_2}{\longrightarrow} \ldots \overset{q_n}{\longrightarrow} X_n \overset{p_n \quad q_1}{\longrightarrow} X_1$$

We show that the solutions found for the path obtained by removing the edge $X_n \xrightarrow{p_n \quad q_1} X_1$ are also solutions for the balance equation of the suppressed edge. The solutions verify the following balance equations:

$$\#X_i \cdot p_i = \#X_{i+1} \cdot q_{i+1} \qquad \text{for } i = 1 \dots n-1$$

Multiplying all the *lhs* and *rhs* of the $n-1$ equations, yields:

$$\#X_1 \cdot \ldots \cdot \#X_{n-1} \cdot p_1 \ldots \cdot p_{n-1} = \#X_2 \cdot \ldots \cdot \#X_n \cdot q_2 \ldots \cdot q_n$$

We remove the common factors:

$$\#X_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} = \#X_n \cdot q_2 \cdot \ldots \cdot q_n$$

By multiplying both sides by q_1, we get:

$$\#X_1 \cdot q_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} = \#X_n \cdot q_1 \cdot q_2 \ldots \cdot q_n$$

Then, the cycle condition (Eq. (7)) gives:

$$\#X_1 \cdot q_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} = \#X_n \cdot p_1 \cdot p_2 \ldots \cdot p_n$$

And, by simplifying by $p_1 \cdot p_2 \ldots \cdot p_{n-1}$, we finally get:

$$\#X_1 \cdot q_1 = \#X_n \cdot p_n$$

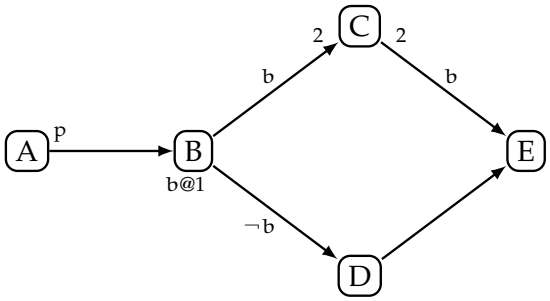
which is the balance equation of the suppressed edge. The cycle condition guarantees that the balance equation of any suppressed edge is satisfied. Hence, the generic solutions satisfy the balance equations for all edges, which guarantees rate consistency of the graph. □

Finding a solution for the system of balance equations suffices for the consistency of a BPDF graph. However, the cycle condition indicates which cycle is the culprit for the absence of a solution. It may provide useful feedback to the application developer.

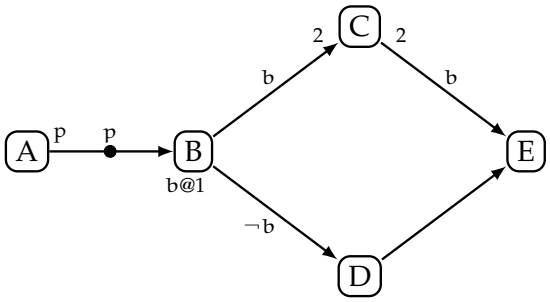
3.2.2 *Boundedness*

In the case of SDF, a rate consistent graph also guarantees bounded execution as there exists a series of firings that returns the graph to its initial state, if it is deadlock free. As far as the integer parameters are concerned, this is true for BPDF as well; if the BPDF graph returns to its initial state after each iteration, then all integer parameters can be modified at these points and boundedness is guaranteed.

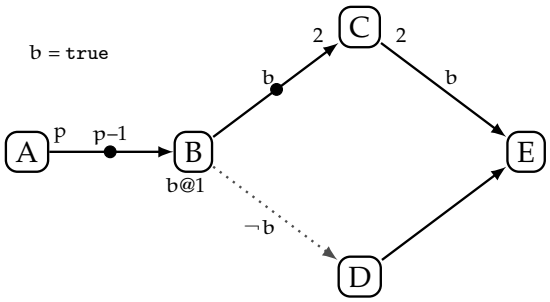
However, when boolean parameters are introduced, rate consistency alone is no longer sufficient to guarantee that the graph will return to its initial state. Consider for example the graph in Figure 26a. This is the same graph as the one in Figure 25 except for the writing period of the boolean parameter b which now is $\pi_w(b) = 1$. First actor A fires and produces p tokens on $\overline{AB}$ (Figure 26b). Then, B sets the value of b to true, and produces one token on $\overline{BC}$ as shown in Figure 26c. If actor B changes the value to false (Figure 26d), the token in $\overline{BC}$



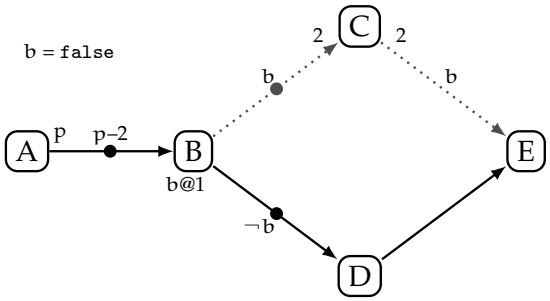
(a) BPDF graph from Figure 25 with $\pi_w(b) = 1$.



(b) Actor A produces p tokens on $\overline{AB}$.



(c) Actor B sets b to `true` and produces 1 token on $\overline{BC}$.



(d) Actor B sets b to `false` and produces 1 token on $\overline{BD}$.

Figure 26: A sample execution of a BPDF graph with a non-valid writing period. There is no guarantee that the token stored on $\overline{BC}$ will be consumed.

will not be consumed by actor C and there is no guarantee that b will take a second true value. With one token stored on $\overline{BC}$ the graph is unable to return to its initial state and its iteration is compromised. Boundedness and liveness are no longer guaranteed.

It is clear that not all periods are safe and their consistency must be checked. The criterion ensuring that parameter modification periods are safe relies on the notions of *regions* and *local iterations*.

Definition 7 (Region). *The region of a boolean parameter* b*, noted* $\mathfrak{R}(b)$*, is defined as:*

$$\mathfrak{R}(b) = M(b) \cup \mathit{Users}(b)$$

The *region* of a boolean parameter is the set containing its modifier and all its users. For example, the region of b in Figure 25 (same as in Figure 26a) is $\mathfrak{R}(b) = \{B,\ C,\ D,\ E\}$.

Definition 8 (Local solutions). *The local solution of an actor* X_i *in a subset of actors* $L = \{X_1, \ldots, X_n\}$*, written* $\#_L X_i$*, is defined as:*

$$\#_L X_i = \frac{\#X_i}{\mathrm{gcd}(\#X_1, \ldots, \#X_n)}$$

The solutions of the system of balance equations are called *global solutions* because they define the number of firings for the iteration of the whole graph. Given a subset of actors in a graph, we can extract a subgraph and accordingly a new, smaller system of balance equations. By solving this system, a potentially different solution is found for each actor, indicating the number of firings needed for the subgraph to return to its initial state. These solutions are called *local solutions* and they denote a *nested* iteration. We call the iteration of the subgraph a *local iteration* to differentiate it from the iteration of the graph called *global iteration*.

Definition 9 (Period Safety). *A BPDF graph is period safe if and only if for each boolean parameter* $b \in \mathcal{P}_b$ *and each actor* $X \in \mathfrak{R}(b)$*,*

$$\exists k \in \mathbb{N}^*,\ \#X = k \cdot \mathit{freq}(b)$$

The factor k *is the reading (or writing) period of* b *for* X*.*

Intuitively, the *period safety* criterion states that a parameter can be modified at most once per local iteration of its region. For the graph in Figure 25, b can be changed after each iteration of the subgraph formed by the actors in $\mathfrak{R}(b) = \{B, C, D, E\}$. We find that the iteration of the subgraph is $(B^2\ C\ D^2\ E^2)$, hence b can change after every 2 firings of B. This formalizes why the writing period of 2 in Figure 25 is safe and why the period of 1 in Figure 26a is not.

Another way to obtain the local iteration of a region is by factorizing the global solutions of the actors in the region of the boolean parameter by their greatest common divisor. Here, the global iteration is $A^2\ B^{2p}\ C^p\ D^{2p}\ E^{2p}$. The greatest common divisor of the solutions of the actors in $\mathfrak{R}(b)$ is p. Factorizing by p yields $A^2\ (B^2\ C\ D^2\ E^2)^p$, giving us the local iteration in the parenthesis. Therefore, actor B can write a new boolean value every 2 firings, while actors C, D, E read a new value after 1, 2, and 2 firings, respectively.

However, local iterations can only be defined when the number of firings of each actor in the region of a parameter b is a multiple of the frequency of b. This property is called *period safety*.

Period safety ensures that, during a local iteration of a region of a given boolean parameter, the number of tokens produced on any edge of this region equals the number of tokens consumed from this edge. It is ensured by a simple syntactic check on BPDF graphs.

In Figure 25, $\mathfrak{R}(b) = \{B, C, D, E\}$, $freq(b) = \#M(b)/\pi_w(b) = p$, and the repetition vector is $[A^2\ B^{2p}\ C^p\ D^{2p}\ E^{2p}]$. Each solution of the actors of $\mathfrak{R}(b)$ is a multiple of the frequency p. The annotation $b@2$ of B is thus period safe. In contrast, the graph in Figure 26a uses a writing period $\pi_w(b) = 1$. This gives a new frequency of $freq(b) = \#M(b)/\pi_w(b) = 2p$. However, the solution of actor C is not a multiple of $freq(b)$ so, $\pi_w(b) = 1$ is not a safe period.

Property 2 (Boundedness)**.** *A rate consistent and period safe BPDF graph returns to its initial state at the end of its iteration.*

Proof. To prove Property 2, we consider an arbitrary edge (Figure 27), and show that, during an iteration, X produces the same number of tokens that Y consumes. In this way, the edge returns to its initial state after an iteration. When all edges return to their initial state, so does the graph.

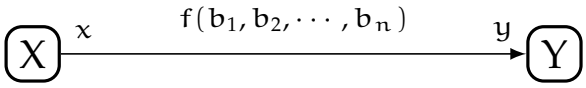


Figure 27: An arbitrary BPDF edge.

The condition $f(b_1, \cdots, b_n)$ is a boolean condition depending on n boolean parameters $b_1, b_2, ..., b_n$. Due to period safety, for each $b_i \in \{b_1, \cdots, b_n\}$ we have:

$$\exists k_i, \ell_i \in \mathbb{N}^*, \quad \#X = k_i \cdot freq(b_i) \quad \text{and} \quad \#Y = \ell_i \cdot freq(b_i) \tag{8}$$

By rate consistency, we also have

$$\#X \cdot x = \#Y \cdot y$$

Therefore, for each $b_i \in \{b_1, \cdots, b_n\}$:

$$k_i \cdot x = \ell_i \cdot y \tag{9}$$

When actor X (resp. Y) is fired, it produces x (resp. consumes y) tokens if $f(b_1, \cdots, b_n)$ and 0 otherwise. We write that X produces $prod(b_1, \cdots, b_n)$ and Y consumes $cons(b_1, \cdots, b_n)$ with

$$\begin{aligned} prod(b_1, \cdots, b_n) &= \text{if } f(b_1, \cdots, b_n) \text{ then } x \text{ else } 0 \\ cons(b_1, \cdots, b_n) &= \text{if } f(b_1, \cdots, b_n) \text{ then } y \text{ else } 0 \end{aligned}$$

During an iteration, there will be $freq(b_i)$ (potentially different) values of boolean b_i. We note these values as the vector $\vec{b_i}$. Let $\uparrow$ be the upsampling operator on vectors defined as

$$[x_1, \ldots, x_n] \uparrow k = [\underbrace{x_1, \cdots, x_1}_{k}, \cdots, \underbrace{x_n, \cdots, x_n}_{k}]$$

Then, $\vec{b_i} \uparrow k_i$ is a vector such that the boolean at rank i represents the value that is used by X at its i^{th} firing. The total production of tokens on the edge is:

$$P = \overrightarrow{\sum} (\mathcal{M}\ \ prod\ \ [\vec{b_1} \uparrow k_1, \ldots, \vec{b_n} \uparrow k_n]) \tag{10}$$

with

$$\overrightarrow{\sum} \vec{a} \ = \ \overrightarrow{\sum} [a_1, \ldots, a_n] \ = \ a_1 + \ldots + a_n$$

and

$$\begin{aligned} & \mathcal{M}\ \ f\ \ [\vec{a_1}, \ldots, \vec{a_n}] \\ = \ & \mathcal{M}\ \ f\ \ [a_{1,1}, \ldots, a_{1,m}] \ldots [a_{n,1}, \ldots, a_{n,m}] \\ = \ & [f(a_{1,1}, \ldots, a_{n,1}), \ldots, f(a_{1,m}, \ldots, a_{n,m})] \end{aligned}$$

Equivalently the total token consumption is:

$$C = \overrightarrow{\sum} (\mathcal{M}\ \ cons\ \ [\vec{b_1} \uparrow l_1, \ldots, \vec{b_n} \uparrow l_n]) \tag{11}$$

The following properties hold for these vector functions:

$$\left(\overrightarrow{\sum}\ \vec{v}\right) \uparrow k = \overrightarrow{\sum}\ (\vec{v} \uparrow k) \tag{12}$$

$$(\mathcal{M}\ \ f\ \ [\vec{v_1} \cdots \vec{v_n}]) \uparrow x = \mathcal{M}\ \ f\ \ [\vec{v_1} \uparrow x \cdots \vec{v_n} \uparrow x] \tag{13}$$

$$(\vec{v} \uparrow x) \uparrow y = \vec{v} \uparrow (x \cdot y) \tag{14}$$

Proofs of these properties are provided in Appendix A.2. By rate consistency:

$$\overrightarrow{\sum} (\mathcal{M}\ \ prod\ \ \vec{v})\ x = \overrightarrow{\sum} (\mathcal{M}\ \ cons\ \ \vec{v})\ y \tag{15}$$

Then, starting from Eq. (10):

$$\begin{aligned} P = \ & \overrightarrow{\sum} \left(\mathcal{M}\ \ prod\ \ \left[\vec{b_1} \uparrow k_1, \ldots, \vec{b_n} \uparrow k_n\right]\right) \\ = \ & \frac{1}{x} \cdot x \left(\overrightarrow{\sum} \left(\mathcal{M}\ \ prod\ \ \left[\vec{b_1} \uparrow k_1, \ldots, \vec{b_n} \uparrow k_n\right]\right)\right) \\ & \text{by Eq. (12), Eq. (13), Eq. (14):} \\ = \ & \frac{1}{x} \left(\overrightarrow{\sum} \left(\mathcal{M}\ \ prod\ \ \left[\vec{b_1} \uparrow (k_1 \cdot x), \ldots, \vec{b_n} \uparrow (k_n \cdot x)\right]\right)\right) \\ & \text{by Eq. (9):} \\ = \ & \frac{1}{x} \left(\overrightarrow{\sum} \left(\mathcal{M}\ \ prod\ \ \left[\vec{b_1} \uparrow (l_1 \cdot y), \ldots, \vec{b_n} \uparrow (l_n \cdot y)\right]\right)\right) \\ & \text{by Eq. (14), Eq. (13), Eq. (12):} \\ = \ & \frac{y}{x} \left(\overrightarrow{\sum} \left(\mathcal{M}\ \ prod\ \ \left[\vec{b_1} \uparrow l_1, \ldots, \vec{b_n} \uparrow l_n\right]\right)\right) \\ & \text{by Eq. (15), Eq. (11):} \\ = \ & \frac{y}{x} \cdot \frac{x}{y} \left(\overrightarrow{\sum} \left(\mathcal{M}\ \ cons\ \ \left[\vec{b_1} \uparrow l_1, \ldots, \vec{b_n} \uparrow l_n\right]\right)\right) \\ = \ & C \end{aligned}$$

Therefore, for any edge and any successive boolean values, the number of produced tokens is equal the number of consumed tokens in an iteration. Hence, all edges return to their initial state after one iteration and therefore, so does the graph. □

Rate consistency and period safety are crucial to ensure this property. However, we assume that actors can be fired in the right order to respect dataflow and parameter communication constraints. This holds only when the graph is live and the next section shows how liveness is checked.

3.2.3 *Liveness*

In SDF, checking liveness is performed by finding a schedule for a basic iteration. Since each actor must be fired a fixed number of times in each iteration, this can be done by an exhaustive search. A class of algorithms that finds schedules for SDF is class-S algorithms discussed in Section 2.2.2. The situation is more complex in BPDF for two reasons:

- First, boolean parameters have to be communicated within the iteration, from modifiers to users. This introduces new constraints between firings of modifiers and users which may introduce deadlocks.
- Second, actors may have to be fired a parametric number of times during an iteration. Finding a schedule may, in general, involve some inductive reasoning.

Boolean Parameter Communication

Boolean parameter communication is implemented by adding new edges to the original BPDF graph. For each parameter boolean b, we add between its modifier $M = M(b)$ and each user $U \in \mathit{Users}(b)$, an edge $e = \overline{MU}$, with $prd(e) = u$ and $cns(e) = m$, with u and m being the local solutions of U and M in the region of b:

$$u = \#_{\mathfrak{R}(b)} U \quad \text{and} \quad m = \#_{\mathfrak{R}(b)} M$$

In other words, M and U occur as $(\dots\ M^m \dots\ U^u \dots)$ in the local iteration corresponding to the region of b. It is easy to see that the solutions of the balance equations of the original graph are also valid for these new edges. During a local iteration, M will produce $m \cdot u$ copies of the value of b, which will all be read by U during this local iteration. The sample BPDF graph from Figure 25 with its additional boolean propagation edges is shown in Figure 28. In Section 3.3.2 we present a refinement of this implementation, which sends only one copy for each new boolean value.

This implementation allows modifiers to change the value of a boolean parameter even when the previously sent value has not been read. In this context, the effect of boolean parameters might be better described as disabling ports instead of edges. Indeed, at a given instant the input and output ports of an edge may be in a different state (*i. e.,* one enabled and one disabled).

For example, in Figure 28, actor B may fire all $2p$ firings and produce all p boolean values for parameter b on the boolean propagation edges, before actors C,D and E start executing. When these actors execute they will enable

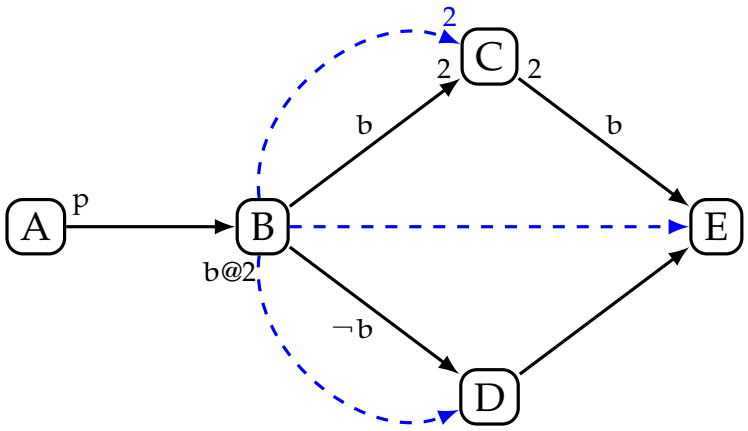


Figure 28: BPDF graph with its boolean propagation edges (dashed edges).

and disable their input ports depending on the boolean value that they read from their boolean propagation edges. The safety period criterion guarantees each actor will use the correct boolean values and the all tokens produced will also be consumed.

The BPDF MoC ensures that the same number of tokens will be produced and consumed on each edge during an iteration (see Property 2).

The newly added communication edges may introduce new cycles in the graph, and subsequently deadlocks. As a consequence, liveness analysis must be performed on the BPDF graph augmented with all its communication edges. The augmented graph includes all possible constraints. Therefore, if it is live, all resulting subgraphs with less edges (and thus less constraints) are live as well.

Acyclic Graphs and Saturated Edges

Acyclic BPDF graphs are always live. In such case, the topological order of the DAG defines a single appearance schedule [6]. The graph in Figure 28 is acyclic and we easily find the schedule

$$A^2; B^{2p}; C^p; D^{2p}; E^{2p}$$

which shows that the graph is live.

Moreover, graphs where each cycle has at least one *saturated* edge is live. An edge is said to be saturated if it contains enough initial tokens for its consumer X to fire #X times. Since this edge has at least the total number of tokens consumed by X in a complete iteration, it does not introduce any constraints and can be ignored. If each cycle has a saturated edge, then the graph can be considered as acyclic (by removing this saturated edge) and therefore live. The single appearance schedule corresponding to the topological order of the DAG obtained by removing all the saturated edges is also a schedule for the original graph.

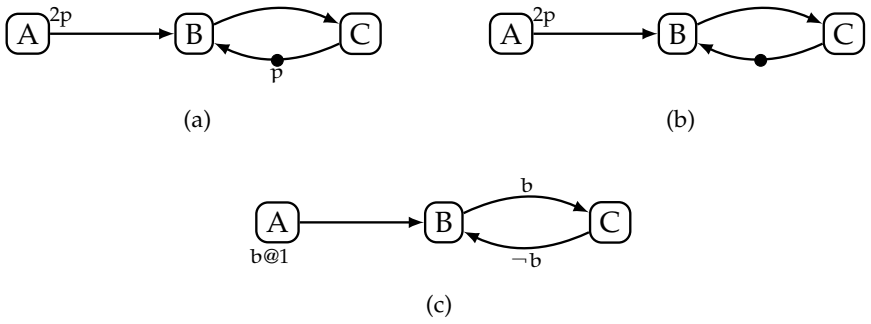


Figure 29: Various cases of live BPDF graphs.

Parametric SDF-like Liveness Checking

When there are cycles without any saturated edge, we adapt the approach taken in SDF. Checking the liveness of cyclic SDF graphs is done by computing an iteration by abstract execution. Since the total number of firings is fixed, all possible orderings of firings can be tested. We adapt this approach to BPDF by

- ignoring booleans (all edges are assumed to be always enabled);
- testing only schedules where each s consecutive firings of an actor A represents an integer fractional of its solution, *i.e.*, $\exists k \in \mathbb{N}^*, \#A = k \cdot s$

Ignoring boolean conditions is safe since it maximizes constraints. If a schedule is found by assuming that all conditional edges are enabled, it will also be valid if some of these edges are disabled. In this way, a global schedule is found, admissible for all possible configurations of the graph.

By considering only occurrences of the form A^s with $\#A = k \cdot s$, we bound the number of such occurrences. For instance, if $\#A = \alpha \cdot p$, then at most α occurrences will be considered in the abstract execution. In other words, if an actor needs to be fired a parametric number of times, only parametric number of firings will be considered. With this constraint, the liveness algorithm of SDF can be reused. We refer to this algorithm as Parametric SDF-like Liveness Checking (PSLC).

The PSLC algorithm is shown in Alg. 1. It is composed by a `while loop` over the firings in the repetition vector and a `for loop` over the set of actors. Each time the algorithm checks whether an actor has firings left, if it does, the algorithm tries to fire it. If the firings left are parametric the algorithm tries to fire the actor a parametric number of times. If the actor can fire the required number of firings it is added to the schedule and both the graph status and the repetition vector are updated. The algorithm continues till a repetition vector worth of firings are scheduled or if no firing was added to the schedule after the end of the `for-loop`. This indicates that there is a *deadlock*.

For instance, the graph in Figure 29a has a repetition vector of $[A\ B^{2p}\ C^{2p}]$. The sole cycle $\mathcal{C} = \{B, C\}$ of the graph has no saturated edges; to be saturated the edge $\overline{CB}$ would need $2p$ tokens instead of only p tokens. Still, the PSLC algorithm finds the schedule $A;(B^p;C^p)^2$ and this graph is live.

Yet, there are cycles for which this approach is not sufficient. There are graphs where actors with parametric solutions need to be considered in non-parametric occurrences. Consider, for example, the graph of Figure 29b. Its

Algorithm 1 Parametric SDF-like Liveness Checking (PSLC) algorithm

```
procedure PSLC(𝒜, r)
    while HasFirings(r) do
        progress = false
        for ∀a ∈ 𝒜 do
            firings = GetFirings(a)
            if firings == 0 then continue
            end if
            if IsParam(firings) then
                times ← GetParam(firings)
            else
                times ← 1
            end if
            if CanFire(a, times) then
                progress ← true
                AddFiring(a, times, schedule)
                UpdateGraphStatus(a, times)
                UpdateRepVector(a, times, r)
            end if
        end for
        if progress == false then
            return(deadlock)
        end if
    end while
    return(schedule)
end procedure
```

repetition vector is also $[A\ B^{2p}\ C^{2p}]$ and it is clearly live with the schedule $A;(B;C)^{2p}$. However, the PSLC algorithm cannot find it. In this case, a simple inductive reasoning would suffice. However, such an inductive approach gets complex to define for general BPDF graphs. We therefore propose two improvements of the PSLC algorithm.

Cycle Clustering

To deal with such problematic cycles, we use the standard clustering technique described in [6] and presented in Section 2.4.3. Clustering a subgraph $\mathcal{G}'$ of a graph $\mathcal{G}$ involves replacing $\mathcal{G}'$ by a single actor Z. The new actor Z is connected to the same external ports as $\mathcal{G}'$ was, but the port rates must be adjusted. The port rate r of an actor $A \in \mathcal{G}'$ is replaced by $r \cdot \#_{\mathcal{A}'}A$, where $\#_{\mathcal{A}'}A$ is the local solution of A in the set of actors $\mathcal{A}'$ of $\mathcal{G}'$.

Formally, given a $\mathcal{G} = (\mathcal{A}, \mathcal{E})$ clustering replaces a set of actors $\Psi \subseteq \mathcal{A}$ with actor Z resulting in graph $\mathcal{G}' = (\mathcal{A}', \mathcal{E}')$ where $\mathcal{A}' = \mathcal{A} - \Psi + \{Z\}$ and $\mathcal{E}' = \mathcal{E} - \{e | lnk(e) = (A, B), A \in \Psi \vee B \in \Psi\} + \mathcal{E}''$ where $\mathcal{E}''$ is a set of new modified edges connected the new actor Z with the rest of the graph.

So, for each $e = \overline{A}\overline{B} \in \mathcal{E}$ with $A \in \Psi$ and $B \notin \Psi$ we get an edge $e' = \overline{Z}\overline{B}$ with the same initial tokens and consumption rate but with

$$prd(e') = prd(e) \times \#A/\#_\Psi A$$

with $\#_\Psi A = \#A/gcd(\{\#X | X \in \Psi\})$. Similarly, we get edges with Z as consumer. for each $e = \overline{A}\overline{B} \in \mathcal{E}$ with $A \notin \Psi$ and $B \in \Psi$ we get an edge $e' = \overline{A}\overline{Z}$ with the same initial token and production rate but with

$$cns(e') = cns(e) \times \#B/\#_\Psi B$$

In general, clustering arbitrary subgraphs may introduce cycles. Here, by clustering only cycles, we avoid introducing new ones.

For each cycle $C = X_1, \ldots, X_n$, our PSLC algorithm finds a local schedule. If the fraction of the total number of firings of each actor over its number of firings in this local schedule is parametric, the cycle is clustered into a new actor Z. The rate r of each port of an actor X_i connected to the rest of the graph is replaced by $r \cdot \#_C X_i$. It follows that a firing of Z corresponds to the firings $X_1^{k_1}, \ldots, X_n^{k_n}$ with $k_i = \#_C X_i$.

For instance, the local schedule for the cycle of the graph of Figure 29b is $B; C$. The fraction of the total number of firings over the number of firings in the local schedule is parametric for each actor ($\frac{2p}{1} = 2p$ in both cases). The cycle is clustered into a new actor Z to get the new graph in Figure 30.

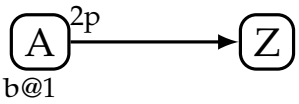


Figure 30: Actors B and C from Figure 29b clustered into actor Z.

The PSLC algorithm now finds the schedule $A; Z^{2p}$ which corresponds to $A; (B; C)^{2p}$ for the original graph, which is therefore correctly found to be live.

False Cycles

A final refinement is needed to take into account *false cycles*. For instance, the previous algorithm would fail to find a schedule for the cycle in Figure 29c since it does not have any initial tokens. However, it is clear that one of its two edges is always disabled; in other words, this cycle is false. We also deal with this issue by using clustering.

False cycles are detected using a truth table for all the conditions of the cycle or using an SMT solver. If, for each combination of values of the boolean parameters, at least one condition is false, then the cycle is false. The subgraph forming cycle C can be clustered, as for each set of values of the boolean parameters, a – *potentially different* – schedule can always be found because in all cases the subgraph is acyclic (one edge of the cycle is always disabled). Firing the resulting actor corresponds to executing a different schedule depending on the boolean conditions. However, in all cases, the local schedule fires the actors the same number of times (*i. e.*, each actor X_i is fired $\#_C X_i$ times).

The false cycle of Figure 29c can be clustered into a new actor Z. A firing of Z corresponds to the schedule if $b = \texttt{tt}\ B; C$ else $C; B$. The global schedule $A; Z^{2p}$ corresponds to the schedule

$$A; (\text{if } b \text{ then } B; C \text{ else } C; B)^{2p}$$

To summarize, liveness checking proceeds by adding to the graph all the required boolean parameter communication edges, by suppressing the saturated edges, by detecting and clustering all false cycles, by clustering all true cycles whose local schedule does not fire actors a fractional part of their global solutions, and finally by finding a schedule using the PSLC algorithm. If the last step succeeds, a sequential schedule has been found and the graph is live.

The above analysis is incomplete and there are live BPDF graphs that will be rejected, but it is sufficient in practice[1]. Note that the above analysis does not try to estimate the minimal number of initial tokens for the graph to be live. The goal of the analysis is only to verify that the graph is live, given an amount of initial tokens.

3.3 IMPLEMENTATION OF BPDF APPLICATIONS

In this section, we deal with the subtleties that occur when it comes to the implementation of a BPDF application. Namely, we deal with the boolean parameters communication, the detailed firing of the actors and the scheduling of the application graph.

3.3.1 *Actor Firing*

In SDF and static MoCs, an actor fires as follows: once all its input edges of an actor have enough tokens, the actor reads the data, executes an internal function and then produces tokens on all its output edges.

In BPDF, the execution of an actor is similar but the boolean parameters need to be taken into account. Some incoming edges may be disabled, enabling the actor to fire even though those edges have insufficient data. So, before reading data, an actor needs to evaluate the boolean conditions associated with its incoming edges. Similarly, the actor needs to evaluate the boolean conditions of its outgoing edges to produce data only on the enabled ones.

The propagation of the boolean values and their periodic production/consumption must be handled automatically to guarantee that the implementation respects the model, eliminating the error-prone human factor. Moreover, in this way the programmer does not need to take care of such tedious work.

For this reason, each BPDF actor uses a wrapper that deals with the boolean parameters. This wrapper implements the reading and writing (in the case of modifiers) of boolean parameters from and to the boolean propagation edges. Generally, the firing of a BPDF actor consists of the following steps:

A. Set boolean values. (Modifiers only)
B. *(Wrapper)* Read values from boolean propagation edges.
C. *(Wrapper)* Read data from active incoming edges.

1 In fact, we have not encountered such a graph yet.

D. Execute internal function.
E. *(Wrapper)* Produce data on active output edges.
F. *(Wrapper)* Produce values on boolean propagation edges.

In this implementation, the modifier is unable to produce data dependent values for its boolean parameters. Instead, the boolean parameters can be data independent or depend on previous input values of the modifier. This is a significant drawback because it is more practical for many applications to use boolean parameters based on the data of the current firing. An alternative sequence (Figure 31) can be used to remedy this problem:

A. *(Wrapper)* Read values from boolean propagation edges.
B. *(Wrapper)* Read data from active incoming edges.
C. Execute internal function.
D. Set boolean values. (Modifiers only)
E. *(Wrapper)* Produce data on active output edges.
F. *(Wrapper)* Produce values on boolean propagation edges.

For example actor X in Figure 32, is the user of boolean parameter b_1 and the modifier of boolean parameter b_2. Every time the actor fires, it first reads a value for b_1. If $f(b_1) = \mathtt{tt}$, it reads r tokens from the edge $\overline{RX}$. Then, it executes its internal function and produces a value for the boolean parameter b_2. If $g(b_1, b_2) = \mathtt{tt}$, it produces w tokens on the edge $\overline{XM}$. Finally, it produces $\#_L M$ copies of the value of b_2 on the boolean propagation edge $\overline{XM}$.

With such an implementation, boolean parameters can take values dependent on input data of the current firing. However, a modifier cannot have an incoming edge with a condition that contains boolean parameters it modifies. The reason is that the values of the boolean parameters are produced *after* reading data from the input edges which may lead to the actor reading data from an edge that would otherwise be disabled.

Both implementations do not affect the analyses of the model and can be used. In our case, as we aim for data dependent parameters, we assume that modifiers do not change boolean parameters that appear in the boolean conditions of their input edges.

3.3.2 *Parameter Communication*

Since integer parameters can change their values only between iterations, their communication is naturally synchronized by them and can be centralized. Hence, we focus here on boolean parameter communication, which requires synchronization between the firings of modifiers and users.

In Section 3.2.2, we implemented this synchronization by connecting each modifier - user $(M - U)$ pair with an edge $e = \overline{MU}$ with $prd(e) = u$ and $cns(e) = m$, where u and m are the periods of reading (by U) and writing (by M) of the boolean parameter.

With this pure data flow implementation, if $u < m$, then each user U must wait for several firings of M before it can read the parameter and fire itself. This is more constrained than needed because the user could read the new value of a parameter just after the first firing of its modifier. We therefore propose a less constrained and more efficient implementation.

WRAPPER

```
/*************** LEGEND ********************
 * br1 - brL: Boolean parameters to read
 * bw1 - bwK: Boolean parameters to write
 * e_in1 - e_inN: Input edges
 * e_out1 - e_outM: Output edges
 * count_<param>: Counter for <param>
 * read(<edge>): Reads data from <edge>
 * write(<data>,<edge>): Writes <data> on <edge>
 * cond(<edge>): Evaluates the condition of the <edge>
 ******************************************/

// Reading values of boolean parameters
br_1 = read (e_br1);
...
br_L = read (e_br1);

// Reading tokens from enabled incoming edges
if (cond(in1)) then in1 = read(e_in1)
...
if (cond(inN)) then inN = read(e_inN)
```

CORE

```
// Main function
fire();

// Set up new boolean values
bw1 = newValue_bw1();
...
bwK = newValue_bwK();
```

```
// Writing tokens to enabled outgoing edges
if (cond(e_out1)) then write(out1,e_out1)
...
if (cond(e_outM)) then write(outM,e_outM)

// Writing values of boolean parameters to all users

// Write values to all (i) users of bw1
write(bw1,e_bw1U1);
...
write(bw1,e_bw1Ui);
...
// Write values to all (j) users of bwK
write(bwK,e_bwKU1);
...
write(bwK,e_bwKUj);
```

Figure 31: Generic wrapper for a BPDF modifier that can set boolean parameters based on the input of its current firing (*Alternative solution*).

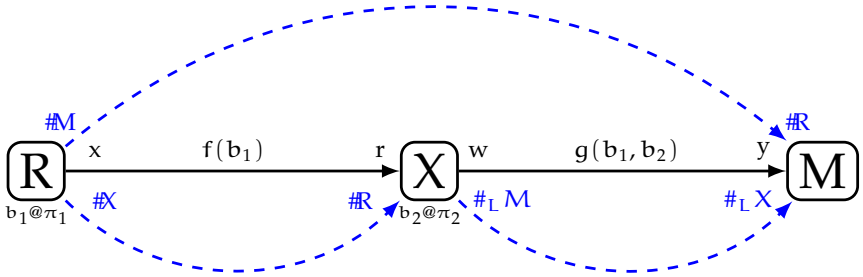


Figure 32: A BPDF graph expanded with its boolean propagation edges.

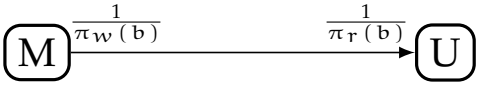
Consider the graph in Figure 32. Actor X is the modifier of b_2 and a user of b_1; it reads and writes tokens to and from conditional edges with a reading period π_r for b_1, and a writing period π_w for b_2. The code in Listing 1 implements one firing of X.

First, actor X must read the value of parameter b_1 every $\pi_r = \#X/freq(b_1)$ firings (the period safety ensures that this number is an integer). Counter `count_b1` implements this periodic reading. If the input edge $\overline{RX}$ is active, *i.e.*, $f(b_1) = \texttt{tt}$, then X consumes r tokens. Then, the actor executes its internal function `fire()`.

Actor X is also the modifier of b_2, so it must send a new value to each user of b_2 every π_w firings. Counter `count_b2` keeps track of the writing period. The edge between X and user M is denoted by `e_b2M`. Finally, if $g(b_1, b_2) = \texttt{tt}$, then X produces w tokens on edge $\overline{XW}$.

Code wrappers implementing this periodic behaviour of reading and writing boolean parameters can easily be produced automatically as shown in the generic wrapper in Figure 33. Their role is to synchronize the firings of modifiers and users for parameter communication. A user will be blocked waiting for the modifier to produce the new boolean value that it needs. Thus, if a completely disconnected actor can fire without constraint, it will eventually have to wait in order to read (or write) a new boolean value.

Actually, this implementation can be described in a dataflow-like manner by adding edges between modifiers and users of the form



where $\pi_r(b)$ denotes the reading period of b by U and a fractional rate, $\frac{a}{b}$, means "produce/consume a tokens each b firings". This representation is related to an extension of SDF: the Fractional Rate Data Flow (FRDF) model [90].

3.3.3 *Scheduling*

In Section 3.2.3, we described a way to find sequential schedules for BPDF applications. However, our objective is to use BPDF to implement streaming applications on many-cores with highly parallel schedules. In such cases, it is

WRAPPER

```
/*************** LEGEND ********************
 * br1 - brL: Boolean parameters to read
 * bw1 - bwK: Boolean parameters to write
 * e_in1 - e_inN: Input edges
 * e_out1 - e_outM: Output edges
 * count_<param>: Counter for <param>
 * read(<edge>): Reads data from <edge>
 * write(<data>,<edge>): Writes <data> on <edge>
 * cond(<edge>): Evaluates the condition of the <edge>
 ******************************************/

// Reading parameters values at the right periods
if (count_br1 == 0) then br_1 = read(e_br1);
...
if (count_brL == 0) then br_L = read(e_brL);
```

CORE

```
// Reading tokens from enabled incoming edges
if (cond(in1)) then in1 = read(e_in1)
...
if (cond(inN)) then inN = read(e_inN)

// Main function
fire();

// Set up new boolean values
bw1 = newValue_bw1();
...
bwK = newValue_bwK();

// Writing tokens to enabled outgoing edges
if (cond(e_out1)) then write(out1,e_out1)
...
if (cond(e_outM)) then write(outM,e_outM)
```

```
// Writing parameter values at the right periods
if (count_bw1=0) then {
    // Write values to all (i) users of bw1
    write_p(bw1,e_bw1U1);
    ...
    write_p(bw1,e_bw1Ui);
}
...
if (count_bwK=0) then {}
    // Write values to all (j) users of bwK
    write_p(bk,e_bwKU1);
    ...
    write_p(bk,e_bwKUj);
}

/* Increment the counters modulo the
 writing and reading periods */
count_bw1 = (count_bw1+1) % pw_bw1;
...
count_bwK = (count_bwk+1) % pw_bwK;

count_br1 = (count_br1+1) % (#X / freq(br1));
...
count_brL = (count_brL+1) % (#X / freq(brL));
```

Figure 33: Generic wrapper for a BPDF actor using periodic reading and writing of parameters.

Listing 1: Sample code of one firing of actor X of Figure 32

```
// Read b1 at the right period
if (count_b1 == 0) {b1 = read(e_b1);}
// Read tokens from (e_RX) if enabled
if (f(b1)) {
    for (i = 0; i < r;i++) {in[i] = read(e_RX);}
}

// Main function
fire();

// Set value of b2
b2 = newValue_b2();

// Write tokens on (e_XW) if enabled
if (g(b1,b2)) {
    for (i = 0; i < w;i++) write(out[i],e_XW);
}

// Write b2 on the propagation edge to M at the right period
if (count_b2 ==   0) {write (b2,e_b2M);}

// Increment the counters modulo the writing and reading periods
count_b1=(count_b1+1) % #X / freq(b1);
count_b2=(count_b2+1) % pw_b2;
}
```

usual to use ASAP scheduling. When it comes to the production of ASAP schedules, complexity increases quickly. BPDF adds to the complexity because of the boolean parameters and the parametric rates as well. In this section, we discuss the main difficulties that arise in the production of ASAP parallel schedules for BPDF applications. We will not go into much details, however, as we present our scheduling approach in Chapter 4.

Implications due to Integer Parameters

An ASAP schedule needs to fire each actor as soon as its firing constraints are satisfied. In simpler models like SDF, this means that there are enough data on all the input edges of an actor and it can be found easily at compile time. However, this is not as simple in BPDF.

Consider the simple BPDF edge in Figure 34. with production/consumption rates p and q, boolean guard c, and t initial tokens. In order to find when there are enough tokens on the edge for actor B to fire, we need to now the values of the parameters. This means that the scheduling needs to take place at run time except for some cases when we may deduct a quasi-static expression for the schedule.

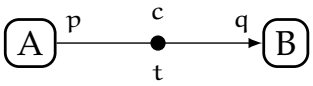


Figure 34: A simple BPDF edge.

However, the generation and analysis of a task graph of a given SDF graph cannot be afforded at run-time. For this reason, in Chapter 4, we introduce a scheduling framework that handles the generation of such a schedule in an efficient way.

Implications due to Boolean Parameters

Boolean parameters affect the scheduling procedure in two ways: first, they add data dependencies due to the parameter communication edges, and secondly they remove data dependencies at run-time based on the values of the boolean parameters.

In the case of the parameter communication edges, they can readily be dealt with as edges of the graph that introduce additional data dependencies. However, if the refined edges (Section 3.3.2) are used, the constraints need to be refined as well.

When an edge is deactivated, the scheduler should take it into account as it may remove a data dependency, thereby allowing the consumer to fire earlier. Moreover, an actor may get completely disconnected and still fire. A firing with no input may be totally acceptable by the main functionality of the actor, and so it fires normally. However, for some actors it may not make sense to execute without input. In these cases, the actor performs a *dummy firing*, *i.e.,* a firing without calling its internal function, just to keep synchronized with the rest of the BPDF graph. These firings just adjust the internal counters of the actor.

3.3.4 BPDF Compositionality

So far, we have considered BPDF graphs that are flat, *i.e.,* each actor is a primitive functional unit. However, one of the benefits of data flow models is modularity and the ability of developing complex applications from the composition of multiple simple ones. Using composites has many benefits in applications development. It allows the separate development of parts of the application from different teams, the reusability of previously developed modules, and the scalability of the applications (by dealing with several smaller ones instead of a single large application).

In such a setting, a graph may have actors that themselves are graphs. Such actors are called *composites*. MoCs like PSDF [11], PiMM [37] and interfaced SDF [97] focus on such hierarchical compositions of applications.

In this section, we discuss the subtleties that arise when BPDF graphs are used hierarchically. Each graph in a lower hierarchical level should have an interface with the higher hierarchical level, *i.e.,* edges that connect the actors of the graph with the ports of the composite actor. Such an actor is shown in Figure 35. The composite has two inputs, connected to actors A and B with rates p and 2, and one output from actor D with rate of 1. We refer to these actors as *interface actors* and the edges as *interface edges*. The rates of the interface of the composite are computed based on the rates of the interface actors and their solution in the composite. For an actor A with solution #A and port rate r_A, we get the rate r_{CA} of the composite using:

$$r_{CA} = \#A \cdot r_A \tag{16}$$

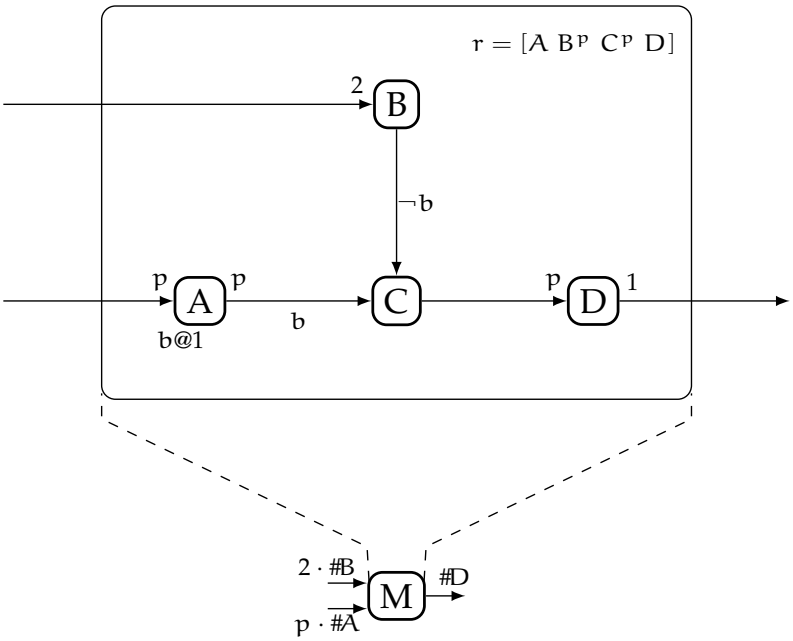


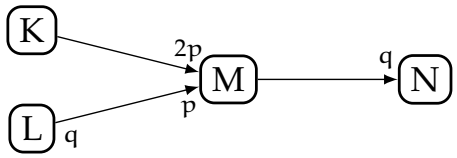
Figure 35: A BPDF composite.

In this way, one firing of the composite actor corresponds to the execution of one iteration of the graph within the composite. In Figure 36a the composite actor M is used in a BPDF graph. The equivalent flattened version of the graph is shown in Figure 36b.

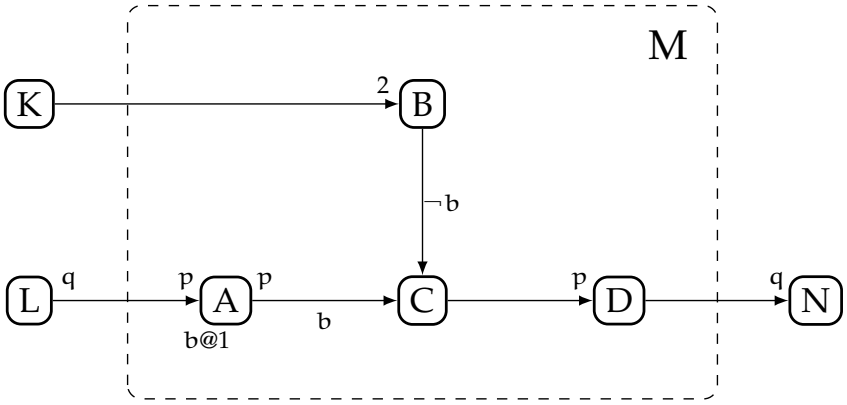
An issue that arises in this scheme, is the usage of integer parameters. If the interface actors have parametric solutions, then parent and child should share these integer parameters. This limits the child to change these integer values between iterations of the parent graph. However, if the interface of the child has only fixed rates, or there are integer parameters that do not appear in the solutions of the interface actors, then these integer parameters of the child can change their values between iterations of the child graph, *i.e.*, potentially multiple times within the iteration of the parent. As a consequence, such an hierarchical BPDF graph cannot be flattened into a BPDF graph.

Boolean parameters need to be considered too. Boolean parameters modified by an actor in the parent graph, can be safely used in the edges of the child as their values will remain fixed during the iteration of the child. In this way, they will have a frequency of 1 and all reading periods will equal the solution of the actor, satisfying the safety period. However, the definition of the region of the boolean parameters needs to be modified to include composite actors whose graph include users of the boolean parameter.

It is best to prohibit modifiers of boolean parameters of the parent graph to be actors within a composite. Although the composite may act as a modifier for the parent, there may be cases that the modifier within the composite changes the boolean parameter multiple times during an execution of the composite which raises the question of which value will be used in the parent. For this, boolean parameters should be limited on one hierarchical level at a time or ensure that a modifier in a lower level produces unique values. However, propagation of



(a) Composite actor M inside its top-level BPDF graph.



(b) Equivalent BPDF graph without composites.

Figure 36: An hierarchical BPDF graph and its flattened equivalent.

boolean values across different levels of the hierarchy complicates the design and should be avoided.

3.4 MODEL COMPARISON

BPDF is a MoC that combines dynamic rates with dynamic topology. There are other models that have at least one of these two features as well. The main difference between these MoCs is the trade-off between expressivity and analyzability. In this section we compare BPDF with models that have similar expressivity.

3.4.1 *Boolean Data Flow*

We single out the BDF MoC [21] from other MoCs because, despite its lack of dynamic rates – all rates in BDF are fixed, it is the only model that focuses directly on dynamic topology. So, we will compare only this aspect with BPDF.

BDF uses two special actors that have as input a stream of boolean values. The first one is the SWITCH actor which has one data input and two data outputs as well as a boolean input stream. Depending on the incoming boolean value, SWITCH activates the appropriate output. The second actor is called SELECT and has two data inputs and one data output and a boolean input stream. The boolean value selects which input is active in this case.

BDF is shown to be Turing Complete in [21]. The problem arises in the boundedness analysis of BDF. There can be cases where it is undecidable whether a graph is operating within bounded memory or not. Buck in his thesis [21] details how a Universal Turing Machine can be built using BDF actors. Here, we give an intuitive example that illustrates the problem. The graph in Figure 37

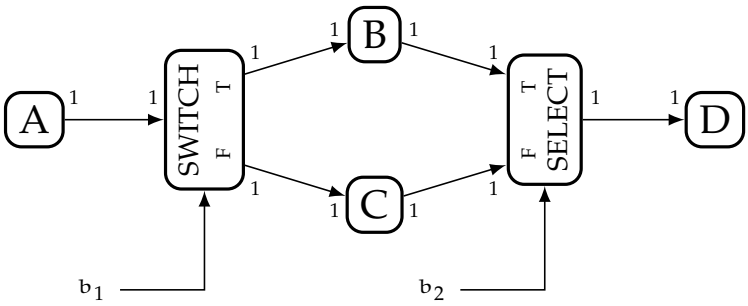


Figure 37: A BDF graph that may operate in unbounded memory.

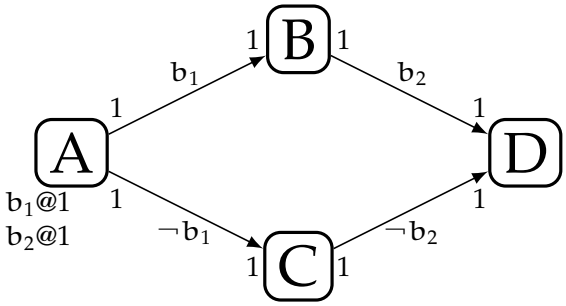


Figure 38: A BPDF graph similar to the BDF graph in Figure 37.

uses two different boolean streams to control the inputs and outputs of the SWITCH and SELECT actors respectively. Thus, assuming that the stream b_1 is always true, and the stream b_2 is always false, it is easy to observe that there will be an accumulation of tokens on the output of actor B. In order to guarantee boundedness, one needs to show the equivalence of the two streams a problem that Buck shows to be undecidable [21].

In some cases Buck provides a clustering technique to find a bounded schedule but the graph iteration is sacrificed. For example, the graph in Figure 39. The SWITCH actor provides tokens to actors B and C depending on the values of the boolean stream b. Actor B needs two tokens to fire. Assuming that b takes a `tt` value, allowing SWITCH to produce a token for actor B, and then takes `ff` values producing tokens to actor C, there is no guarantee that the boolean stream will take a second `tt` value so that the token stored on the input edge of B will be consumed. This means that there is no guarantee that the graph will ever return to its initial state.

When it comes to BPDF, both these issues are dealt with thanks to the *safety period* criterion and the fact that disconnected actors still fire without producing or consuming data. Safety period restricts BPDF expressivity so that it remains analyzable. Let us consider two examples of how BPDF would capture similar functionality as the two problematic BDF graphs.

In Figure 38 the two different boolean parameters, even if they are stuck at opposite values, do not cause unboundedness. This is because actors B and C will produce tokens regardless of the boolean value (*i. e.*, even when disconnected), eventually enabling actor D to fire.

Finally, in 3Figure 40, the safety period criterion forces the boolean parameter to change value every two firings of actor A. This way if there is a `tt` value, it

will last for two firings of A that will produce two tokens for actor B to fire. As a result, when the boolean parameter changes value, the input edge of B returns back to its initial state preserving the iteration of the graph.

3.4.2 *Schedulable Parametric Data Flow*

Just like BDF, SPDF [43] focuses only on one aspect of dynamism, namely the parametric rates. The model does not support dynamic topology and we compare SPDF with BPDF only as far as the parametric rates are concerned.

SPDF uses the notion of regions, just like in the case of BPDF, to calculate changing periods of integer parameters within an iteration. Changing integer parameters within an iteration adds a lot to the expressiveness of the model but also greatly increases its scheduling complexity. BPDF, on the other hand, allows integer parameters to change only between iterations. This way all rates are known at the beginning of an iteration, reducing the scheduling complexity.

SPDF may potentially be combined with BPDF into a single MoC where both integer and boolean parameters change within an iteration. The resulting model will be very difficult to schedule apart from simple sequential and trivial parallel schedules.

3.4.3 *Scenario-Aware Data Flow*

Another model that stands out is SADF [116]. SADF uses special actors, called detectors, to change the rates of part of the graph. SADF allows dynamic topology by allowing production and consumption rates of zero. Each different configuration is called a *scenario*.

In SADF, the set of possible scenarios that the graph may operate in needs to be defined. To capture a BPDF graph in SADF, all possible configurations of the graph need to be expressed and stored at compile time. In many cases this is impractical, as the number of possible configurations of even a small BPDF graph using a couple of parameters can be prohibitive.

On the contrary, the enumeration of the scenarios in SADF has some advantages. BPDF demands a graph to be consistent when all of its edges are active. In many cases, however, there are guards that use opposite boolean values and the relevant edges are never active simultaneously. BPDF does not support an alternative analysis where the graph is partially consistent, depending on the

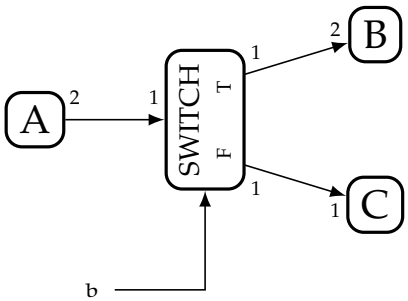


Figure 39: A BDF graph with problematic iteration.

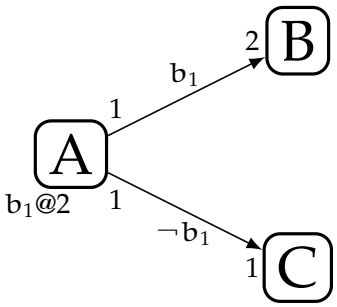


Figure 40: A BPDF graph similar to the BDF graph in Figure 39. The safety criterion guarantees that the graph will always return to its initial state.

values of the boolean parameters. The main reason behind this decision is to keep the analysis of the graph independent of the values of the parameters which may lead to an explosion of different cases to verify.

SADF, on the other hand, renders such analysis feasible, by taking individual scenarios. Instead of analyzing the application for all possible values of the parameters, SADF focuses only in the much smaller subset of the values of the scenarios. In this sense, there may be SADF graphs that cannot be captured in BPDF, because they will be found to be inconsistent.

3.4.4 *Other Models of Computation*

There are other MoCs that combine dynamic rates and topology of lesser importance. As already mentioned in Chapter 2 models like CSDF and VRDF allow rates to take zero values effectively deactivating the port. However, in the case of CSDF the sequence of change of the rates – and consecutively the activation and deactivation of the ports, is determined at compile time. VRDF allows only parameters that come in matching pairs, that is if there is a parametric production rate on a path, there must be the equivalent consumption rate on the same path. Moreover, VRDF is limited to acyclic graphs only.

3.5 SUMMARY

In this chapter, we presented the BPDF MoC. The goal of this MoC is to provide the increased expressiveness needed for the efficient implementation of modern streaming applications while still providing static analyses for consistency, boundedness and liveness.

BPDF accomplishes this by introducing integer and boolean parameters that allow dynamic production and consumption rates on the edges of the graphs and dynamic topology changes. Integer parameters are changing once per iteration, allowing a symbolic solution of the balance equations to be found and most of the existing SDF techniques to be adapted to handle parameters. Boolean parameters are allowed to change periodically within an iteration of the graph by special actors called modifiers. BPDF introduces a supplementary criterion, namely the period safety criterion, that ensures bounded and live operation of the graph regardless of the values of the boolean parameters.

BPDF provides analyses for consistency, boundedness and liveness that are independent of specific values of both the integer and the boolean parameters. For this reason, the model imposes additional constraints on the graph that could be otherwise lifted, like the demand of consistency on the graph when all its edges are enabled.

We analyzed the subtleties that arise in the implementation of the model. The implementation choices, however, do not affect its static analyzes. Such choices are, for example, the actions that compose the firing of a BPDF actor or the exact implementation of the propagation of the boolean parameters.

Moreover, we saw that, due to the extra expressiveness, BPDF increases the complexity of the scheduling of the model. Dynamic rates change the number of times that each actor should fire and dynamic topology may enable actors earlier. Disconnected actors may fire, at least conceptually, without any inputs or outputs, to keep track with the advancement of the iteration. All these factors need to be taken into account by a scheduler to lessen overhead. We detail the scheduling of BPDF applications in Chapter 4.

Finally, we compared BPDF to other data flow MoCs of similar expressiveness. Compared to BDF that focuses on the boolean, and SPDF that focuses on the integer parameters, BPDF finds a trade-off between the two, where both types of parameters are restricted on how often they can change values. This restriction makes BPDF analyzable and schedulable. A similar model, SADF, provides an alternative solution useful in specific applications that cannot be captured by BPDF due to its restrictive consistency criteria. On the other hand, BPDF is preferable to SADF in applications where the large number of parameter combinations prohibits the use of the latter. One can say that the two models complement each other.

4 SCHEDULING FRAMEWORK

> It's who I am and what I feel
> My life is automatic
> Up in the air, it's what I breathe
> and it is never static
>
> — Amaranthe

The development of more expressive data flow models enables more complex applications, such as video decoders, to be captured, but considerably increases the complexity of scheduling. As discussed in Chapter 3, parametric rates and dynamic topology both add challenges when it comes to parallel scheduling. Such dynamism prohibits the reuse of existing techniques; a DAG cannot be derived because of the parametric number of firings and the conditional connections between them. In addition, manual production of such schedules would be time consuming and error-prone.

This chapter describes a scheduling framework that allows the automatic production of parallel schedules for the deployment of BPDF applications on many-core platforms. The framework relies on the extraction of parametric scheduling constraints from the application. To allow optimization and manual manipulation of the schedule, the framework also allows the user to add constraints that affect the ordering and the parallel execution of the actors. These constraints are analyzed to preserve the boundedness and liveness guarantees of the application. The resulting set of scheduling constraints is used to produce the parallel schedule.

The chapter starts with the overview of the underlying platform along with its mapping and scheduling assumptions in Section 4.1. Then, the proposed framework is presented in Section 4.2. Finally, the experimental evaluation of the framework is presented and discussed in Section 4.5.

4.1 UNDERLYING PLATFORM

BPDF is designed with streaming applications, like video decoders, in mind. Such applications are composed of computationally intensive functions and very often have time restrictions. For example, a video needs to be decoded at a certain frame rate (*i.e.*, 24 frames per second) to maintain an acceptable QoS. To meet such requirements, streaming applications are typically developed in specialized hardware, sometimes with a software layer that allows the partial configuration of each hardware component.

We target the STHORM many-core platform [87], formerly known as P2012. Developed by STMicroelectronics, STHORM is a representative of a modern many-core platform.

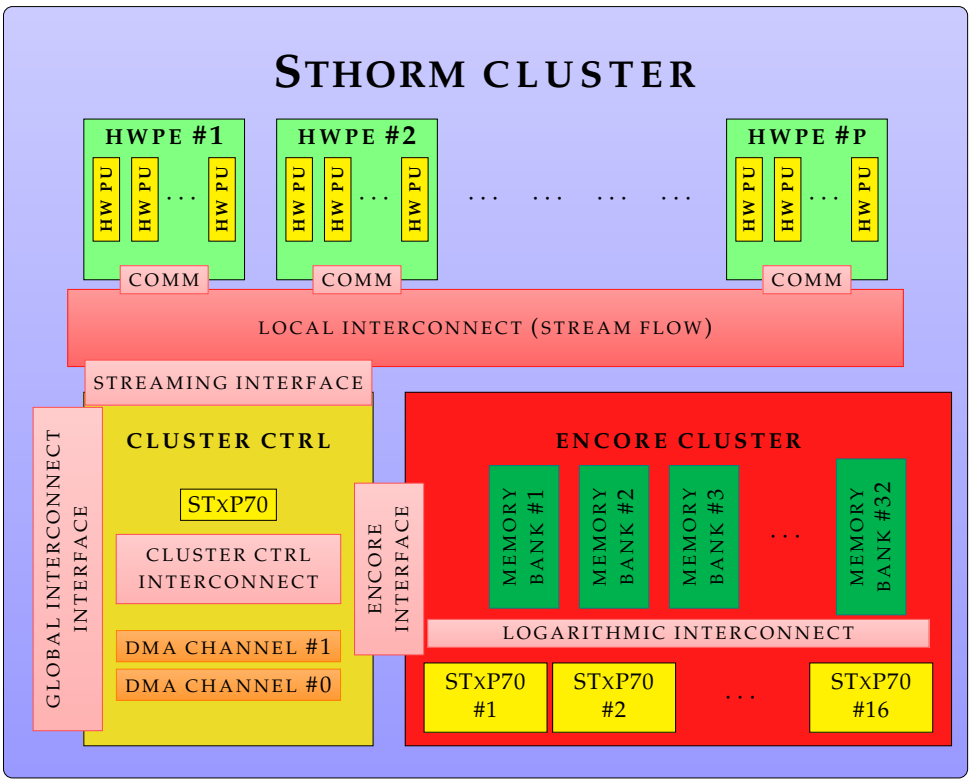


Figure 41: Architecture of a cluster of the STHORM platform.

STHORM is composed by up to 32 clusters interconnected by a global asynchronous NoC. Each cluster is composed by by a Cluster Controller, a multi-core computing engine called ENCore, and may optionally have a number of specialized Hardware Processing Elements (HWPEs).

The Cluster Controller manages the intra- and inter-cluster communication. It has a Global Interconnect Interface to access the Global NoC, a communication interface with the ENCore engine and a Streaming Interface with the HWPEs. It also has a DMA module, allowing efficient communication of the ENCore engine and the HWPEs with each other and the global interconnect.

The ENCore engine can have up to 16 Software Processing Elements (SWPEs), which are general-purpose STxP70-V4 32-bit RISC processors. The STxP70-V4 processors use a centralized shared memory (UMA architecture) composed by multiple banks that allow simultaneous access from all processors.

The HWPEs are further partitioned in Hardware Processing Units (HWPUs) that share the communication interface of the hosting HWPEs. The architecture of the STHORM cluster is shown in Figure 41.

STHORM includes a native programming model, Predicated Execution Data Flow (PEDF), that simplifies the parallel implementation of streaming applications. PEDF uses *filters* to implement applications. A filter can be:

- A *primitive filter* is a filter that applies a well defined function to a set of input data in order to produce a set of output data. It is implemented as a hardware or software processing element and is the building block of PEDF model.

- A *module* is a composite filter. A module can contain one or more primitive filters or modules. Modules allow the development of applications hierarchically, facilitating the design.
- A *controller* which schedules the firing of the filters in a module and controls the configuration parameters for each filter. Each module contains its own controller. If a module is fired, its controller is activated and executes its internal schedule, which fires the filters of the module.

4.1.1 *Mapping*

The scheduling framework focuses on the production of parallel schedules given a static mapping and does not deal with mapping decisions.

In our implementation, a BPDF graph is implemented in PEDF using a single module, hence the hierarchical composition of modules is not used. This is because we assume BPDF graphs that do not use hierarchy. We discussed BPDF compositionality in Section 3.3.4. Such an hierarchical model can take advantage of PEDF modules where each module is a BPDF graph.

The module contains one PEDF primitive filter for each BPDF actor, which corresponds to a separate (hardware or software) processing element. Integer parameter and boolean parameters communication takes place through the controller of the module by changing corresponding configuration parameters of each filter. The controller also controls the execution of the application.

This is a simple yet realistic mapping scheme. Although, in this mapping scheme, each actor is mapped on a separate processing element, the framework can handle any other kind of static assignment, where actors may share resources as discussed in Section 4.4.2.

4.1.2 *Scheduling*

The goal of our scheduling framework is the generation of the controller that controls the schedule of each BPDF actor. The PEDF model uses *slotted scheduling* (Section 2.4.5) to schedule the firing of the filters. At the beginning of a slot, the controller selects several filters to be fired and their execution takes place concurrently. When these executions are completed, the next slot may start. The controller may compute the composition of the next slot while the filters of the current slot execute, therefore the hardware pipeline is not slowed down.

We produce slotted schedules that can be directly implemented using this model. Such a framework can also be used by other state-of-the-art many-core platforms. For instance, modern GPUs support a similar execution model. In mainstream GPU programming models, such as CUDA [28] and OpenCL [112], the host processor, equivalent to the controller of STHORM, creates a task group, loads it on the GPU, and gets the results when all tasks have finished their execution. In parallel with the execution of the task group, the host processor may determine tasks to be executed next.

Although it relies on the STHORM platform, our scheduling framework can be easily adapted to produce non-slotted schedules, as discussed in Section 4.4.1.

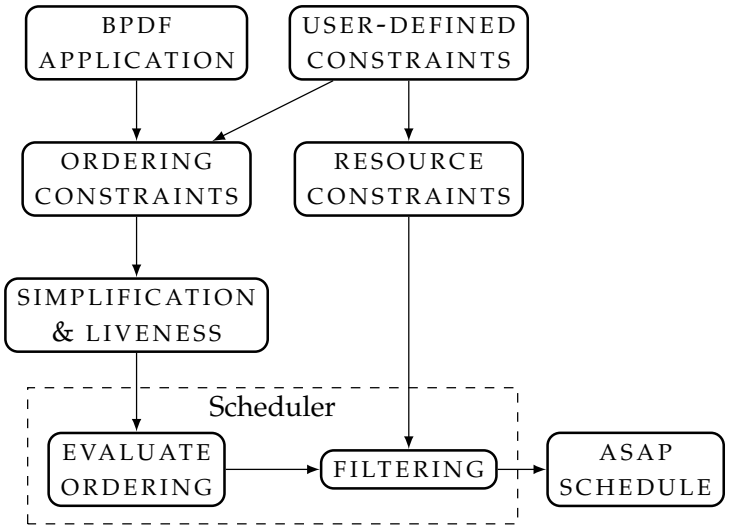


Figure 42: Our Scheduling Framework.

4.2 SCHEDULING FRAMEWORK

The goal of our framework is to produce *ASAP parallel* schedules. ASAP scheduling is chosen because it is considered to be the best strategy when timing information about the execution of the actors is unknown [111]. Unknown timing is typical for BPDF applications where the execution time of the actors may vary a lot, based on the values of the integer and boolean parameters.

Our framework also aims to be *flexible* and *expressive* in order to support a multitude of platforms and different optimization criteria. Our framework uses a fixed scheduling algorithm (here ASAP) and allows the schedule to be manipulated by adding new scheduling constraints that affect the ordering and the parallel execution of the actors.

An overview of the framework is shown in Figure 42. The framework receives the BPDF application along with an optional set of user-defined constraints. From these, a set of ordering constraints and a set of resource constraints are produced. The resulting ordering constraints (*i. e.*, those extracted from the BPDF graph and those added by the user) are checked for liveness and, if possible, simplified. Then, the ASAP scheduler produces the ASAP parallel schedule meeting the constraints one slot at a time. Each slot is produced in two stages; first the ordering constraints are evaluated to produce a set of fireable actors, then, the set of fireable actors is filtered, according to the resource constraints, to produce a subset of fireable actors that will eventually fire in the slot. The scheduler continues until the iteration is completed.

Scheduling constraints derive both from the *application* and express the dependencies of the dataflow graph, or from the *user* expressing platform specificities (*e. g.*, resource limitations), or optimizing some criteria (*e. g.*, power consumption, buffer sizes). They can be of two distinct types: *Ordering constraints* that restrict individual actor firings, and *resource constraints* that control parallel execution (*e. g.*, limiting the level of parallelism). Application constraints can only be ordering constraints. User constraints can be both resource and ordering constraints. They are defined by the programmer for a specific application

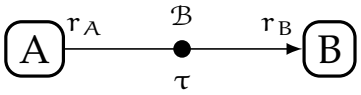


Figure 43: A generic BPDF edge.

or platform. The two types of constraints are presented in detail in Section 4.3 and Section 4.4.

4.3 ORDERING CONSTRAINTS

An ordering constraint is a relationship between the firings of two actors of the form:

$$A_i > B_{f(i)}$$

where A_i denotes the i^{th} firing of actor A and $B_{f(i)}$ denotes the $f(i)^{th}$ firing of actor B (where f is any total function from $\mathbb{N}^*$ to $\mathbb{Z}$). A null or negative value for $f(i)$ means that the instance A_i does not depend on B.

4.3.1 *Application Constraints*

Application constraints (or data flow constraints) are automatically derived from dependencies between actors. These constraints can either be *data dependencies* that derive from the edges of the graph, or *boolean propagation constraints* that derive from the implicit boolean propagation edges.

One data dependency is extracted from each edge of the graph. For an edge between actors A and B with production/consumption rates r_A and r_B respectively, initial tokens τ, and boolean guard $\mathcal{B}$, as in Figure 43, the following ordering constraint is generated:

$$B_i > A_{f(i)} \quad \text{where} \quad f(i) = \left\lceil \frac{r_B \cdot i - \tau}{r_A} \right\rceil \tag{17}$$

Function $f(i)$ expresses the number of times actor A should have fired so that there are enough tokens on the edge for the i_{th} firing of actor B, noted B_i, to take place. B_i needs A to have produced $r_B \cdot i - \tau$ tokens. To produce that many tokens, A needs to fire a number just greater than $\frac{r_B \cdot i - \tau}{r_A}$ times, that is:

$$\left\lceil \frac{r_B \cdot i - \tau}{r_A} \right\rceil$$

The data dependency in Eq. (17) does not depend on the boolean guard $\mathcal{B}$. However, the scheduler takes boolean guards into account by disregarding ordering constraints of disabled edges at run-time.

Boolean parameters introduce constraints due to the communication of their values between modifiers and users. A user needs to read a new value according to its reading period (π_r). The modifier produces a new value according to its writing period (π_w). Therefore, we get the following ordering constraint for each user (U) and modifier (M) of the same boolean:

$$U_i > M_{f(i)} \quad \text{where} \quad f(i) = \pi_w \cdot \left\lfloor \frac{i-1}{\pi_r} \right\rfloor + 1 \tag{18}$$

Function $f(i)$ expresses the number of times that the modifier should fire to produce the value of the boolean parameter used by the user on its i^{th} firing (U_i). Actor U reads a new boolean value every π_r firings, so U_i will need the k^{th} value of the boolean parameter to fire, where k is

$$k = \lfloor (i-1)/\pi_r \rfloor + 1 \tag{19}$$

The modifier produces a value every π_w firings, hence the k^{th} boolean value is produced on its m^{th} firing, where m is.

$$m = \pi_w \cdot (k-1) + 1 \tag{20}$$

Therefore, we replace k in Eq. (20) by the required value of Eq. (19), yielding:

$$\pi_w \cdot \left(\left\lfloor \frac{i-1}{\pi_r} \right\rfloor + 1 - 1 \right) + 1 \quad = \quad \pi_w \cdot \left\lfloor \frac{i-1}{\pi_r} \right\rfloor + 1$$

which indicates the number of times the modifier needs to have fired to produce the needed value. The constraint restricts the user to wait for the production of a boolean value but does not restrict the modifier. Indeed, the modifier may produce a new boolean value (or all the boolean values) before the user has finished using the previous one. The user will use the new values later, based on its reading period, when they are needed.

It is worth to note here that this is the weakest constraint that the boolean propagation imposes on the user of a parameter. As discussed in Section 3.3.2, the actual implementation may impose stronger constraints. If the propagation of the boolean values is implemented with simple data flow edges, as suggested in Section 3.2.3, then the boolean propagation constraints can be derived as data dependencies from the additional edges.

Finally, as we consider each actor executing on a single processing element (hardware or software), we have the implicit self-constraint for each actor X:

$$X_i > X_{i-1} \tag{21}$$

which expresses the fact that there can not be multiple instances of an actor executing in parallel and that the firings of the same actor take place in sequence.

The self dependency expressed in Eq. (21) is derived from the given mapping decision that requires each actor on a different processing element. In Section 4.4.2, we demonstrate how alternative static mapping schemes, using data parallelism, can be expressed.

4.3.2 *User Ordering Constraints*

Additional user ordering constraints can be used to optimize various criteria (*e.g.*, power consumption, buffer size). Here we give a few examples of user constraints that achieve various improvements.

Consider again the generic BPDF graph of Figure 43. For simplicity and without loss of generality, we consider the rates r_A and r_B to be co-primes. Hence,

the repetition vector of the graph is $[A^{r_B}\ B^{r_A}]$ and A will fire r_B times without any constraints. Since A fires r_B times consecutively, if B does not consume enough tokens, there will be an accumulation of tokens on the FIFO buffer from A to B.

If the programmer wants to restrict this FIFO buffer to be of a certain size, say k tokens, the execution of A needs to be regulated so that it fires only when there is enough space left on the edge buffer. Adding the following backwards constraint from B to A:

$$A_i > B_{f(i)} \quad \text{where} \quad f(i) = \left\lceil \frac{r_A \cdot i - k + \tau}{r_B} \right\rceil \tag{22}$$

enforces the buffer size to be at most k.

Typically, in data flow graphs a buffer constraint of k tokens is modeled by adding a backward edge from B to A with $k - t$ initial tokens, so that the total number of tokens on the AB cycle will be k. Extracting the data dependency from such an edge, gives the constraint expressed in Eq. (22).

Additional user-defined constraints may introduce a deadlock if they are not compatible with the data dependency constraints extracted from the BPDF application. For instance, in the previous example, it should be checked that k is large enough so that A can trigger all r_A firings of B. This verification step is done using a deadlock detection algorithm presented in Section 4.3.3.

4.3.3 *Liveness Analysis*

Inconsiderate user-defined constraints must be checked statically for liveness. A set of ordering constraints may introduce deadlock when they imply (by transitivity) a constraint of the form:

$$(A_i > A_j) \wedge (i \leqslant j) \tag{23}$$

which requires that the ith firing of an actor A must take place after the jth firing where j is the same or a future firing ($j \geqslant i$). All cyclic constraints from an actor to itself must be checked. The complexity of liveness analysis is linear to the number of cyclic constraints. To ensure liveness, it must be shown that each cycle of the form:

$$A_i > B_{f_1(i)} > \ldots > C_{f_n(i)} > A_{f_{n+1}(i)}$$

that is to say

$$A_i > A_{f_1(\ldots(f_n(f_{n+1}(i)))\ldots)}$$

satisfies

$$i > f_1(\ldots(f_n(f_{n+1}(i)))\ldots) \tag{24}$$

We consider all ordering constraints to detect such cycles. Typically, the expression $f_1(\ldots(f_n(f_{n+1}(i)))\ldots)$ contains parameters and ceiling functions. In general, only an upper bound can be computed. Parameters are replaced by their maximum or minimum values and ceilings $\lceil \frac{a}{b} \rceil$ by $\frac{a}{b} + 1$ or $\frac{a}{b} - 1$ depending on their sign and position. The expression $f_1(\ldots(f_n(f_{n+1}(i)))\ldots)$ is then simplified to get an upper bound. If the condition in Eq. (24) is true for all cycles,

then the liveness of the schedule is guaranteed. Otherwise, since we have computed an over-approximation, we cannot guarantee that the set of constraints is *not* live but we still reject the graph.

Consider, for instance, the simple BPDF graph in Figure 44.

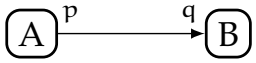


Figure 44: A simple BPDF graph.

The data dependency we obtain from the edge is:

$$B_i > A_{f(i)} \qquad \text{with} \quad f(i) = \left\lceil \frac{q \cdot i}{p} \right\rceil$$

When the user wants to limit the edge buffer to k tokens, she introduces the following constraint:

$$A_i > B_{g(i)} \qquad \text{with} \quad g(i) = \left\lceil \frac{p \cdot i - k}{q} \right\rceil$$

Together they form a cyclic constraint:

$$A_i > A_{f(g(i))}$$

In practice, such a limit (k) as well as the maximum values of parameters (p_{max}, q_{max}) are actual integers. Here, we illustrate the verification process using symbolic values. Based on Eq. (24), to ensure liveness we must verify that:

$$\begin{aligned}
i > f(g(i)) \quad &\Leftrightarrow \quad i > \left\lceil \frac{q \cdot \left\lceil \frac{p \cdot i - k}{q} \right\rceil}{p} \right\rceil \\
&\Leftarrow \quad i > \frac{q \cdot \left(\frac{p \cdot i - k}{q} + 1 \right)}{p} + 1 \\
&\Leftrightarrow \quad i > i + \frac{q - k}{p} + 1 \\
&\Leftrightarrow \quad k > p + q \\
&\Leftarrow \quad k > p_{max} + q_{max}
\end{aligned}$$

So, if the limit placed on the buffer size k is greater than $p_{max} + q_{max}$, a live schedule is ensured. This is only a *sufficient* condition, because of the approximation incurred by removing the ceiling functions.

In general, if there exists a cycle that does not satisfy Eq. (24), then the involved user constraints are rejected. Actually, this cycle condition can be relaxed by taking boolean guards into account. The scheduler takes into account the boolean values, hence if two constraints of a cycle depend on on contradictory boolean guards, then the cycle is live as it cannot be formed.

Consider the simple BPDF graph in Figure 45. The graph contains a false cycle $\mathcal{C} = \{A, B\}$. As discussed already in Section 3.2.3, such cycles do not introduce a deadlock because they are never formed at run-time. However, the cycle has two ordering constraints:

$$A_i > B_i \quad \text{and} \quad B_i > A_i$$

that form the cyclic dependency:

$$A_i > B_i > A_i$$

which obviously does not satisfy the criterion in Eq. (24).

Checking for false cycles in the cycles formed by scheduling constraints is not the same as in the liveness analysis because instead of actors we have instances of actors. Hence, one should take into account the reading periods of the actors, to make sure that the involved instances use the same boolean value of the contradicting parameter.

4.3.4 *Scheduler*

For a set of constraints, the goal is to find the actors that will fire in each slot or, equivalently, to find the slot at which each firing takes place. The assignment of a slot to each firing must be a valid solution to the set of constraints.

A function that assigns the firings of an actor to slots is called a *firing function*. For the firings of actor A, A_i, we get the firing function $\Phi(A_i) : \mathcal{F}(A_i) \to \mathcal{L}$, where $\mathcal{F}(A_i)$ is the set of firings of actor A and $\mathcal{L}$ is the set of the schedule slots.

As we want to get the ASAP schedule meeting the constraints, each firing A_i should be scheduled at the earliest slot possible, that is the first slot after all the actor instances A_i depends on, have fired. A firing A_i, with a set of constraints $\mathcal{C}(A_i)$, depends on a set of firings $\mathcal{F}(A_i)$ that derive from $\mathcal{C}(A_i)$. If $\mathcal{C}(A_i)$ is:

$$\mathcal{C}(A_i) = \left\{A_i > B_{1\,f_1(i)},\ A_i > B_{2\,f_2(i)},\ \ldots, A_i > B_{n\,f_n(i)}\right\}$$

then we get the set of firings:

$$\mathcal{F}(A_i) = \left\{B_{1\,f_1(i)},\ B_{2\,f_2(i)},\ \ldots, B_{n\,f_n(i)}\right\}$$

Therefore, the firing function for A_i is:

$$\Phi(A_i) = \begin{cases} \max\limits_{k\in[1,n]} \left(\Phi\left(B_{k\,f_k(i)}\right)\right) + 1 & \text{if } i > 0 \\ 0 & \text{if } i \leqslant 0 \end{cases} \tag{25}$$

Ordering constraints may be deactivated depending on boolean conditions. The firing function takes into account these conditions as well. So, if $\mathcal{F}(A_i)$ depends on n boolean conditions $(L_1, L_2, \ldots, L_n)$ the firing function given in Eq. (25) becomes:

$$\Phi(A_i) = \begin{cases} \max\limits_{k\in[1,n]} \left(L_k\ ?\ \Phi\left(B_{k\,f_k(i)}\right)\ :\ 0\right) + 1 & \text{if } i > 0 \\ 0 & \text{if } i \leqslant 0 \end{cases} \tag{26}$$

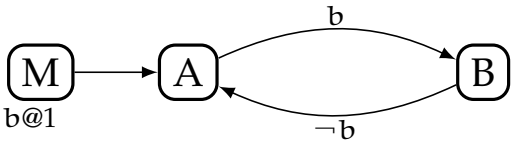


Figure 45: A BPDF graph with a false cycle.

where $L_k \ ? \ \Phi\left(B_{kf_k(i)}\right) : 0$ means that if condition L_k is true, then the firing function of $B_{kf_k(i)}$ is considered normally and if it is false 0 is used instead.

For readability, we suppress the 0 part and write: $L_k \ ? \ \Phi\left(B_{kf_k(i)}\right)$ instead. Moreover, we will define firing functions using only the part where $i > 0$.

As all actors have the implicit self-constraint of Eq. (21) in their set of constraints, their previous firing is in the set of firings ($A_{i-1} \in \mathcal{F}(A_i)$). For this reason, all firing functions are strictly monotonic, that is:

$$\Phi(A_i) > \Phi(A_{i-1}) \tag{27}$$

For example, consider an actor A with ordering constraints

$$A_i > B_{f(i)} \ \text{ and } \ A_i > C_{g(i)}$$

with constraint $A_i > B_{f(i)}$ depending on boolean parameter b, and the self-constraint $A_i > A_{i-1}$, has

$$\mathcal{F}(A_i) = \{B_{f(i)}, C_{g(i)}, A_{i-1}\}$$

and will have the firing function:

$$\Phi(A_i) = \max\left(\Phi\left(b \ ? \ B_{f(i)}\right), \ \Phi\left(C_{g(i)}\right), \ \Phi\left(A_{i-1}\right)\right) + 1 \ \text{ if } \ i > 0$$

A scheduler is used to execute the graph (Alg. 2). It takes as input the repetition vector (R) and the set of actors ($\mathcal{A}$) along with their firing functions (Φ). Moreover, the scheduler gets the reading and writing periods of the actors (Π).

The scheduler uses a *status vector* (V_s) that keeps track of the number of times an actor has fired, a *next slot vector* (V_n) that indicates the slot number that the next firing of an actor will take place, and a *fire vector* (V_f) that indicates the actors that fire in the current slot. The current slot number is kept in the variable ℓ.

The scheduler is an infinite loop that schedules one iteration at a time. At the beginning of an iteration, the V_s vector and the current slot number ℓ are initialized respectively to $\vec{0}$ and 0. Then, the scheduler reads the current values of the integer parameters ($\mathcal{I}$) and evaluates the repetition vector, the firing functions, and the writing and reading periods.

The core of the scheduler is a `for-loop` over the set of actors, which selects the eligible actors for the current slot. Before entering the loop, the current slot number is increased by 1 and V_f is initialized to $\vec{0}$. The `for-loop` first updates the next slot (V_n) value of each actor using its firing function, then checks whether the actor is eligible to fire in the current slot. If the actor A is eligible *and* still has firings left for the current iteration (*i. e.*, $V_s[A] < R[A]$), it is marked to be fired in the firing vector ($V_f[A] = 1$) and its status $V_s[A]$ value is increased by 1.

Firing functions depend on their previous values as well as on values of other firing functions. For this reason, the computed values of each firing function are kept in memory to avoid re-computation.

If a firing generates a new boolean value, the values of the boolean parameters ($\mathcal{B}$) are updated. These are kept in lists (one list for each parameter) which are initially empty. The boolean values are needed for the evaluation of the

Algorithm 2 ASAP Scheduler that uses firing functions

```
procedure SCHEDULER(R, 𝒜, Φ, Π)
    while true do
        V_s = 0⃗
        ℓ = 0
        ℐ = GET INT VALUES()
        R = EVALUATE(R, ℐ)
        Φ = EVALUATE(Φ, ℐ)
        Π = EVALUATE(Π, ℐ)
        while V_s ≠ R do
            ℓ = ℓ + 1
            V_f = 0⃗
            for ∀X ∈ 𝒜 do
                V_n[X] = Φ(X, V_s[X] + 1, ℬ)
                if V_n[X] == ℓ ∧ V_s[X] < R[X] then
                    V_f[X] = 1
                    V_s[X] = V_s[X] + 1
                    if NEWBOOL() then
                        ℬ = GET BOOL VALUES()
                    end if
                end if
            end for
            FIRE(V_f)
        end while
    end while
end procedure
```

firing functions. The reading periods are used to get the correct value of the boolean parameter from the corresponding list. For a firing A_i using boolean b, with writing period $\pi_r^A(b)$, the k^{th} value of boolean parameter is used, where:

$$k = \left\lceil \frac{i}{\pi_r^A(b)} \right\rceil$$

Once the `for-loop` is finished, the firing vector is issued to fire. The `for-loop` is nested within a `while-loop` over the iteration of the graph. The outer loop ends once the status vector equals the repetition vector of the graph. Liveness analysis guarantees that at least one firing will take place every time the `for-loop` executes and that the `while-loop` will end.

4.3.5 *Constraint Simplification*

Once the constraints are verified to preserve liveness, they are simplified. The simplification step consists of two parts:

- **Constraint trimming**, where redundant constraints are removed.
- **Constraint resolution**, where constraints are resolved into slot sequences at compile time if possible.

Constraint Trimming

Constraint trimming checks the firing functions (Φ) in the max function of Eq. (25) and removes the firing functions that can be compared at compile time. For each pair of firings

• When the firing functions to be compared are of the same actor, because of the monotonicity of the firing function (Eq. (27)), we only need to compare their indexes.

For an actor A with

$$\Phi(A_i) = \max\left(\Phi(B_{f_1(i)}), \Phi(B_{f_2(i)}), \dots\right) + 1$$

if $f_1(i) > f_2(i)$, then $\Phi(B_{f_2(i)})$ is suppressed.

• When the firing functions are of different actors, then comparison may be possible by transitivity through a third firing function.

For an actor A with

$$\Phi(A_i) = \max\left(\Phi(B_{f(i)}), \Phi(C_{g(i)}, \dots)\right) + 1$$

and a firing function of C depending on B:

$$\begin{aligned} &\Phi(C_i) = \Phi(B_{k(i)}) + 1 \\ \Rightarrow &\Phi(C_{g(i)}) = \Phi(B_{k(g(i))}) + 1 \end{aligned}$$

if $k(g(i)) \geqslant f(i)$ then $\Phi(B_{f(i)})$ is suppressed from $\Phi(A_i)$.

When constraint trimming takes place, the boolean parameters should be taken into account. When a constraint $\Phi(B_j)$ suppresses constraint $\Phi(C_k)$ in firing function $\Phi(A_i)$, it should be checked whether the constraint $\Phi(B_j)$ depends on a boolean parameter. Trimming can not take place when constraint $\Phi(B_j)$ can be suppressed due to boolean parameters because then constraint $\Phi(C_k)$ is no longer redundant.

Constraint Resolution

In some cases, we can solve the firing functions of each or some of the actors, at compile time. Solving a firing function means to define it in terms of a simple non recursive function of its index and other firing functions that are already solved. Such definitions derive inductively from Eq. (25) when the max function can be evaluated at compile time. The evaluation of the max function is possible in many cases. Here, we provide a non-exhaustive list of the more prominent cases:

• For source actors, the max function of the firing function contains only the previous firing of the actor:

$$\Phi(A_i) = \Phi(A_{i-1}) + 1$$

It is easy to show that the firing function of such actors is the identity function:

$$\Phi(A_i) = i \tag{28}$$

• When the actor depends on a single firing that has already been defined as a constant:

$$\Phi(A_i) = \max(c, \Phi(A_{i-1})) + 1$$

then the firing function can be simplified into:

$$\Phi(A_i) = i + c \tag{29}$$

• Another interesting case is the simple data dependency between a consumer and a producer when the firing function of the producer is already solved. Consider an edge as the one in Figure 44 (p. 80). The data dependency between the two actors is $B_i > A_{f(i)}$, where $f(i) = \lceil \frac{qi}{p} \rceil$. Based on Eq. (25), the firing function of B is:

$$\Phi(B_i) = \max\left(\Phi\left(A_{\lceil \frac{qi}{p} \rceil}\right), \Phi(B_{i-1})\right) + 1$$

We show that the firing function in this case is:

$$\Phi(B_i) = \begin{cases} \Phi\left(A_{\lceil \frac{qi}{p} \rceil}\right) + 1 & \text{if } q > p \\ \\ \Phi(A_1) + i & \text{if } q \leqslant p \end{cases} \tag{30}$$

Proof. By induction. For the case where $q > p$, the expression holds for $i = 1$:

$$\Phi(B_1) = \max\left(\Phi\left(A_{\lceil \frac{q}{p} \rceil}\right), \Phi(B_0)\right) + 1 = \Phi\left(A_{\lceil \frac{q}{p} \rceil}\right) + 1$$

Assuming that

$$\Phi(B_n) = \Phi\left(A_{\lceil \frac{qn}{p} \rceil}\right) + 1$$

we show that

$$\Phi(B_{n+1}) = \Phi\left(A_{\lceil \frac{q(n+1)}{p} \rceil}\right) + 1$$

Indeed,

$$\begin{aligned} \Phi(B_{n+1}) &= \max\left(\Phi\left(A_{\lceil \frac{q(n+1)}{p} \rceil}\right), \Phi\left(B_n\right)\right) + 1 \\ &= \max\left(\Phi\left(A_{\lceil \frac{q(n+1)}{p} \rceil}\right), \Phi\left(A_{\lceil \frac{qn}{p} + 1 \rceil}\right)\right) + 1 \end{aligned}$$

Furthermore:

$$q \geqslant p \quad \Rightarrow \quad \left\lceil \frac{qn + q}{p} \right\rceil \geqslant \left\lceil \frac{qn + p}{p} \right\rceil = \left\lceil \frac{qn}{p} \right\rceil + 1$$

and since all firing functions are strictly monotonic (Eq. (27)), this yields:

$$\Phi\left(A_{\lceil \frac{q(n+1)}{p} \rceil}\right) \geqslant \Phi\left(A_{\lceil \frac{qn}{p} \rceil + 1}\right)$$

Hence,

$$\Phi(B_{n+1}) = \Phi\left(A_{\lceil \frac{q(n+1)}{p} \rceil}\right) + 1$$

The case $q \leqslant p$ is proved in a similar way. □

With the above simplification cases, we end up with a variety of different definitions of the firing functions.

In the particular case of SDF, they can be defined using only indexes. In this case, one can compute the schedule slots for each actor A by evaluating the firing function for $i \in [1..\#A]$. In this way, static slot sequences can be produced for each actor.

Firing function definitions may also include integer parameters as is the case of Eq. (30). In this case, we may be able to produce parameterized slot sequences for each actor and produce a single schedule stream using the procedure in Appendix A.1.

To illustrate the various cases, we consider again the simple graph in Figure 44 (p. 80). Actor A is a source actor and its firing function is (Eq. (28)):

$$\Phi(A_i) = i \quad \text{with} \quad i \in [1, q]$$

which corresponds to the slot sequence $\mathcal{F}^q$, meaning that A fires for q consecutive slots. For actor B we distinguish the following cases:

CASE $q \leqslant p$: Using Eq. (30) we get

$$\Phi(B_i) = i + 1 \quad \text{with} \quad i \in [1, p]$$

Hence, the sequence of slots of actor B is $\mathcal{E}\mathcal{F}^p$ meaning that B is idle during the first slot and then fires at each slot. The execution will proceed as follows: A fires in the first slot and for each subsequent slot, A will fire in parallel with B until it has fired a total of $\#A = q$ times. This totals to $q - 1$ firings of B so there remains to fire B another $p - (q - 1)$ times. Hence, the two slot sequences can be combined in a slotted schedule (see Appendix A.1 for an algorithm that merges such expressions):

$$A; (A \| B)^{q-1}; B^{p-q+1}$$

CASE $q > p$: Two sub-cases must be considered:

SUB-CASE $q = k \cdot p$: If q is a multiple of p, using Eq. (30) with $q > p$:

$$\Phi(B_i) = \left\lceil \frac{qi}{p} \right\rceil + 1 = \left\lceil \frac{kpi}{p} \right\rceil + 1 = ki + 1 \quad \text{with} \quad i \in [1, p]$$

Each firing of B occurs after k firings of A and the corresponding slot sequence is $\mathcal{E}(\mathcal{E}^{k-1}\mathcal{F})^p$. The two slot sequences can be combined in a slotted schedule:

$$A; (A^{k-1}; (B \| A))^{p-1}; A^{k-1}; B$$

SUB-CASE $q = k \cdot p + r$ WITH $0 < r < p$: In this case, Eq. (30) yields:

$$\Phi(B_i) = \left\lceil \frac{qi}{p} \right\rceil + 1 \quad \text{with} \quad i \in [1, p]$$

The slot sequence of B cannot be expressed as before because the ceiling in the firing function cannot be resolved and needs to be computed at run-time.

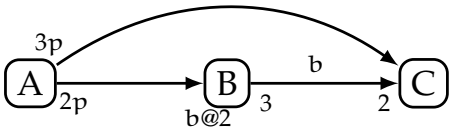


Figure 46: A BPDF graph whose constraints can be solved symbolically.

In the last case, the firing functions could not be expressed as a slotted schedule. Still avoiding the evaluation of the set of constraints at run-time and using a fixed firing function instead is a large improvement. One can observe that even when the relation between the parameters is unknown, with a single condition at runtime we can effectively use simplified results. This makes the schedule faster, as we reduce the calculations done at run-time, but may increase the schedule code size. So, we have a trade-off between performance and memory.

The above cases are just a small sample of the possible firing functions that can be solved at compile time. Other cases and simplifications are presented in [30].

Simplification Example

Consider the graph in Figure 46, whose iteration is $[A\ B^{2p}\ C^{3p}]$ and whose dataflow constraints are:

$$B_i > A_{\lceil \frac{i}{2p} \rceil}, \quad i \in [1, \#B]$$
$$C_j > B_{\lceil \frac{2j}{3} \rceil}, \quad C_j > A_{\lceil \frac{j}{3p} \rceil}, \quad j \in [1, \#C]$$

plus the implicit $X_i > X_{i-1}$ for all actors. Moreover, actor C, as a user of the boolean parameter b, is constrained by (from Eq. (18)):

$$C_j > B_{2\lfloor \frac{j-1}{3} \rfloor + 1} \quad \Rightarrow \quad C_j > B_{2\lceil \frac{j}{3} \rceil - 1}$$

because of the floor property:

$$\left\lfloor \frac{n}{m} \right\rfloor = \left\lceil \frac{n-m+1}{m} \right\rceil = \left\lceil \frac{n+1}{m} \right\rceil - 1$$

This transformation is useful because all the constraints involve only ceiling functions.

We notice that although actor C has two constraints from actor B that seem redundant, they are not trimmed because, as explained in the previous section, the prevailing constraint is depending on the boolean parameter b.

Actor A is a source actor and from Eq. (28), we have:

$$\Phi(A_i) = i \quad \text{with} \quad i = 1 \quad (\#A = 1)$$

So, we schedule A_1 in the first slot ($\Phi(A_1) = 1$) and the slot sequence of actor A is $\mathcal{F}$.

The constraint for actor B becomes $B_i > A_1, i \in [1, \#B]$, hence:

$$\Phi(B_i) = \max(\Phi(A_1), \Phi(B_{i-1})) + 1 = \max(1, \Phi(B_{i-1})) + 1$$

Actor B is depending on a constant firing so, from Eq. (29), we have:

$$\Phi(B_i) = i+1, \quad \text{with} \quad i \in [1, 2p]$$

and the slot sequence of B is $\mathcal{E}\mathcal{F}^{2p}$. Finally, the three constraints on actor C yield the following firing function:

$$\begin{aligned}\Phi(C_i) &= \max\left(\Phi(A_1), b?\Phi\left(B_{\lceil\frac{2i}{3}\rceil}\right), \Phi\left(B_{2\lceil\frac{i}{3}\rceil-1}\right), \Phi(C_{i-1})\right)+1 \\ &= \max\left(1, b?\left(\left\lceil\frac{2i}{3}\right\rceil+1\right), 2\left\lceil\frac{i}{3}\right\rceil, \Phi(C_{i-1})\right)+1, \quad i \in [1\ldots 3p]\end{aligned}$$

- When b is `tt`, $\lceil\frac{2i}{3}\rceil+1$ is not suppressed, and since:

$$\forall i, \quad \left\lceil\frac{2i}{3}\right\rceil+1 > 2\left\lceil\frac{i}{3}\right\rceil > 1$$

we get:

$$\Phi(C_i) = \max\left(\left\lceil\frac{2i}{3}\right\rceil+1, \Phi(C_{i-1})\right)+1, \quad i \in [1, 3p]$$

where we have the case of a firing function depending on single data dependency. Hence, from Eq. (30) we get:

$$\Phi(C_i) = \left\lceil\frac{2i+3}{3}\right\rceil + i = i+2, \quad i \in [1, 3p]$$

- When b is `ff`, the firing function becomes:

$$\Phi(C_i) = \max\left(2\left\lceil\frac{i}{3}\right\rceil, \Phi(C_{i-1})\right)+1, \quad i \in [1, 3p]$$

Again, we have the form of a single data dependency and, from Eq. (30), we get:

$$\Phi(C_i) = 2\left\lceil\frac{1}{3}\right\rceil + i = i+2, \quad i \in [1, 3p]$$

We notice that in both cases, the solution for actor C is:

$$\Phi(C_i) = i+2, \quad i \in [1, 3p]$$

As a result, the value of the boolean parameter does not influence the solution. The slot sequence of actor C is $\mathcal{E}^2\mathcal{F}^{3p}$. Having found the slot sequences of all actors:

$$\begin{aligned}A &: \mathcal{F} \\ B &: \mathcal{E}\mathcal{F}^{2p} \\ C &: \mathcal{E}^2\mathcal{F}^{3p}\end{aligned}$$

we can merge them into a single schedule stream using the procedure presented in Appendix A.1:

$$A; B; (B \| C)^{2p-1}; C^{p+1}$$

4.4 RESOURCE CONSTRAINTS

Resource constraints are used to regulate the parallel execution of actors. Such constraints can be used to limit the degree of parallelism or to enforce mutual exclusion between (groups of) actors. They can be seen as filter functions applied to the set of fireable actors $\mathcal{S}$ and returning a subset $\mathcal{T}$ of $\mathcal{S}$. Any such function f must satisfy the two following conditions:

$$\forall \mathcal{S},\ f(\mathcal{S}) = \mathcal{T} \quad \Rightarrow \quad \mathcal{T} \subseteq \mathcal{S} \tag{31}$$

$$\forall \mathcal{S},\ f(\mathcal{S}) = \mathcal{T} \quad \Rightarrow \quad \mathcal{T} \neq \varnothing \tag{32}$$

The condition in Eq. (31) ensures that the function is safe (only firable actors can be selected), while the condition in Eq. (32) ensures that it preserves liveness (at least one actor is selected to be fired).

Many languages can be used to express such constraints. Since they are functions over finite domains, one may even consider expressing them exhaustively as tables. Here, we use rewrite rules on sets inspired from the Gamma formalism [3]. The general form of a resource constraint is:

$$\textbf{replace}\ S_A\ \textbf{by}\ S_B\ \textbf{if}\ \textit{condition} \tag{33}$$

where S_A and S_B are nonempty sets of enabled actors such that $S_B \subseteq S_A$. It can be read as "replace S_A by S_B if `condition` is true". When the `condition` is always true it can be omitted. For example, the rule

$$\textbf{replace}\ A,\ B\ \textbf{by}\ A \tag{34}$$

can be read as "if the actors A and B are fireable, then replace them by A (*i. e.*, remove B)". It prevents actors A and B from being fired in the same slot and gives priority to A.

Rewrite rules can use pattern variables to match arbitrary actors. For instance, the rule

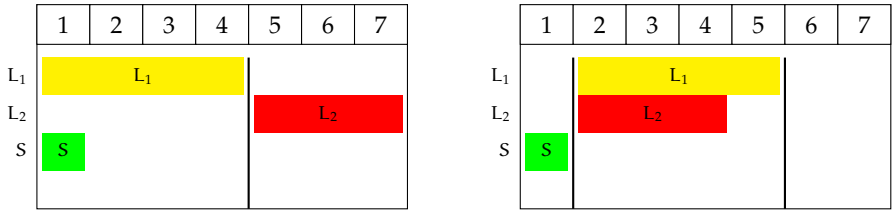
$$\textbf{replace}\ w, x, y, z\ \textbf{by}\ w, x, y \tag{35}$$

can be read as "select four arbitrary fireable actors and suppress one of them". It limits the level of parallelism to 3 actors. Indeed, rewriting rules apply until no match can be found. Therefore, Rule (35) above applies as long as there are more than three firable actors.

Rules can also depend on a condition. For instance, assuming that the two predicates *short* and *long* denote whether an actor takes a short or long time to execute, the rule

$$\textbf{replace}\ x,\ y\ \textbf{by}\ x\ \textbf{if}\ \textit{short}(x) \wedge \textit{long}(y) \tag{36}$$

prevents short and long actors from being fired within the same slot (priority is given to short ones). This rule may improve the overall computation time. Indeed, if S is a "short" actor while L_1 and L_2 are two "long" actors such that S and L_1 are fireable at the same slot and firing S enables L_2, then it is better to fire first S alone and then L_1 and L_2 in parallel. The effect of the constraint in this case is shown in Figure 47, where Figure 47a is the slotted schedule before applying Rule (36) and Figure 47b after.



(a) Initial schedule.

(b) Schedule after rule is applied.

Figure 47: Possible effect of the resource constraint. (Rule (36))

In a similar manner we can limit the power consumption during a slot. Assuming two predicates that classify actors into high (H) or low (L) power consumers, we can limit power consumption by firing at most one H actor either alone or along with at most one L actor. The following set of rules implements such a limitation:

$$\begin{aligned} &\textbf{replace } x,\ y && \textbf{by } x && \textbf{if } H(x) \wedge H(y) \\ &\textbf{replace } x,\ y,\ z && \textbf{by } x,\ y && \textbf{if } H(x) \wedge L(y) \wedge L(z) \end{aligned} \tag{37}$$

Several rules can also be combined in sequence or in parallel. The semantics of parallel composition enforces that rules applied in parallel act on disjoint sets of actors. For example, the sequential combination of Rule (36) followed by Rule (35) limits the possible parallel firings to one actor, two short actors, or two long actors. The application of Rule (35) followed by Rule (36) impose the same limitation but the results will differ as Rule (36) followed by Rule (35) gives priority to short actors.

It is very easy to check that such rules preserve boundedness and liveness. It is sufficient to check that each rule obeys the conditions in Eq. (31) and Eq. (32).

Resource constraints have to be applied dynamically. They can either be resolved by an iterative application of the rules until no rule can be applied, or they can be statically compiled, considering all sets of fireable actors, into constant time selection operations. The two approaches trade-off run-time performance with memory usage.

4.4.1 *Alternative Scheduler*

When only ordering constraints are used, we can define the firing function of each actor as described in Section 4.3.4, potentially simplify (Section 4.3.5) them, and use the scheduler in Alg. 2. However, resource constants dynamically shift firings in later slots, effectively changing the firing functions of the actors at run-time. For this reason we propose an alternative scheduler that handles directly actor constraints.

This alternative scheduler takes as input the repetition vector (R), the set of actors ($\mathcal{A}$), the ordering and the resource constraints ($\mathcal{C}, \mathcal{G}$), and the reading and writing periods of the actors (Π). It evaluates the constraints and produces a slotted schedule (Figure 48). The scheduler stops when an iteration is finished and is reset to begin the next iteration.

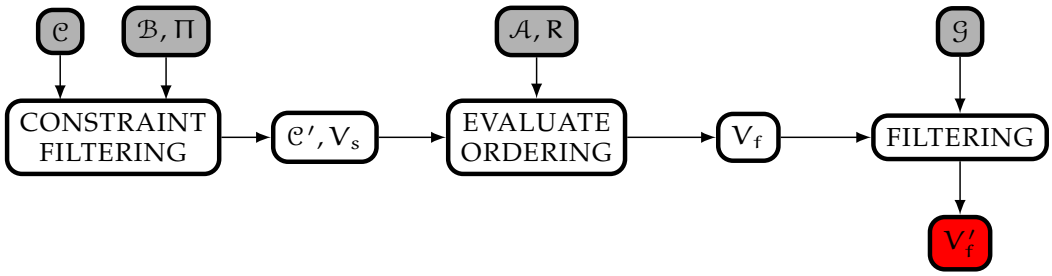


Figure 48: Scheduler overview of the calculation of the actors in a slot.

Similar to the previous scheduler, the alternative scheduler is an infinite loop scheduling one iteration at a time. It uses two auxiliary vectors: the *firing vector* (V_f) and the *status vector* (V_s). V_f indicates the fireable actors of each slot. V_s keeps track of the number of times each actors has fired. It is reset to 0 at the beginning of each iteration. At the beginning of an iteration, the scheduler gets the values of the integer parameters ($\mathcal{I}$) and evaluates the repetition vector and the ordering constraints.

The core of the scheduler is a `while-loop` over the iteration of the graph. Each execution of the loop schedules one slot. At the beginning of the loop, the firing vector is set to 0, and the values of the boolean parameters are updated. The boolean values are kept in lists, one list for each boolean parameter. Each time the procedure GET BOOL VALUES is called, the scheduler checks whether there has been a new boolean value generated by a modifier, and if so, it adds the value in the corresponding list. These lists are needed because different actors may be using different values of the same boolean parameter, depending on their reading periods and the number of times they have been fired within the iteration.

Once the boolean values are updated, the loop executes three functions: CONSTRAINT FILTERING that filters ordering constraints based on the current boolean values, EVALUATE ORDERING that evaluates the ordering constraints and produces a set of firables actors, and finally FILTERING that filters the set of fireable actors to a subset based on the resource constraints.

CONSTRAINT FILTERING takes as input the set of ordering constraints ($\mathcal{C}$), the values of the boolean parameters ($\mathcal{B}$) and the reading and writing periods (Π). For each constraint deriving from an edge of the graph, the function evaluates the corresponding boolean condition and, if the guard is false, removes the constraint from the set of ordering constraints. The procedure returns a reduced set of ordering constraints ($\mathcal{C}'$). The boolean values used for each constraint are based on the read and write periods of the constrained actor. For a constraint $A_i > B_{f(i)}$ depending on boolean parameter b, with read period $\pi_r^A(b)$, the k^{th} value of b will be used, where k is:

$$k = \left\lceil \frac{i}{\pi_r^A(b)} \right\rceil$$

EVALUATE ORDERING takes as input the reduced set of constraints $\mathcal{C}'$ produced by CONSTRAINT FILTERING, the status vector V_s, the set of actors $\mathcal{A}$, and the repetition vector R. The procedure returns the fireable vector V_f that flags the fireable actors for the next slot.

Algorithm 3 ASAP scheduler that handles constraints directly

```
procedure SCHEDULER(R, 𝒜, 𝒞, 𝒢, Π)
    while true do
        V_s = 0⃗
        ℐ = GET INT VALUES()
        R = EVALUATE(R, ℐ)
        𝒞 = EVALUATE(𝒞, ℐ)
        while V_s ≠ R do
            ℬ = GET BOOL VALUES()
            𝒞' = CONSTRAINT FILTERING(𝒞, ℬ, Π)
            V_f = EVALUATE ORDERING(𝒞', V_s, 𝒜, R)
            V_f = FILTERING(V_f, 𝒢)
            V_s = V_s + V_f
            FIRE(V_f)
        end while
    end while
end procedure
```

This procedure is shown in Alg. 4. We denote $\mathcal{C}(X)$ the set of constraints imposed on X (*i.e.*, all constraints of the form $X_i > \ldots$). The procedure initializes the firing vector to 0 and iterates over the set of actors. If all the constraints imposed on an actor are satisfied and the actor has firings left then it is flagged as eligible to fire in the firing vector.

EVAL evaluates the constraints of an actor $\mathcal{C}(X)$ according to the current status vector (V_s). More precisely, for each constraint

$$X_i > Y_{f(i)}$$

EVAL simply checks whether:

$$f(V_s[X] + 1) \leqslant V_s[Y]$$

which corresponds to the satisfaction of the data dependency. Indeed, $V_s[X] + 1$ is the index of the next firing of X and $f(V_s[X] + 1)$ indicates the number of firings that actor Y should have achieved before the $V_s(X) + 1^{th}$ instance of X is fireable. If the current number of firings of Y (*i.e.*, $V_s[Y]$) is greater than the number of firings of Y needed for the next firing of X (*i.e.*, $f(V_s[X] + 1)$) then the constraint

$$X_i > Y_{f(i)} \quad \text{with} \quad i = V_s[X] + 1$$

is satisfied. If all the constraints in $\mathcal{C}(X)$ are satisfied, then EVAL returns true and X is eligible to be fired in the current slot. Otherwise, it returns false and X will not be fired.

Apart from the ordering constraints, the repetition vector is also checked, ($V_s[X] < R[X]$), to determine whether the actor needs to be fired again in the current iteration or has finished its execution. If both conditions are satisfied, $V_f[X]$ is set to 1. After all actors have been considered, the fireable vector is produced. Since each actor is selected as soon as its constraints are met, the procedure produces an ASAP schedule *w.r.t.* the given constraints.

Algorithm 4 Evaluation of ordering constraints

```
procedure EVALUATE ORDERING(R, 𝒞, 𝒜, V_s)
    V_f = 0⃗
    for ∀X ∈ 𝒜 do
        if EVAL(𝒞(X), V_s) ∧ V_s[X] < R[X] then
            V_f[X] = 1
        end if
    end for
    return(V_f)
end procedure
```

Finally, FILTERING takes as input the firing vector along with the resource constraint matrix $\mathcal{G}$. FILTERING is just a lookup procedure that finds V_f in the constraint table and returns the entry of the table for that vector, so we do not provide its pseudo code. The procedure returns the reduced firing vector containing the actors to be fired in the next slot.

At the end of `while-loop`, the status vector V_s is updated based on the actors flagged in the firing vector. Finally, the firing vector is issued to fire. Concurrent to the execution of the actors, the scheduler computes the firing vector for the next slot. The scheduler is summarized in Alg. 3.

Liveness analysis of the ordering constraints (Section 4.3.3) and the liveness condition of resource constraints (Eq. (32)) guarantee that EVALUATE ORDERING and FILTERING procedures will issue at least one actor to fire in each slot, as long as the the repetition vector is not reached. Hence, the inner loop always terminates when the iteration of the graph is complete.

4.4.2 *Framework Extensions*

The proposed framework imposes constraints on the execution of the application, sometimes leading to inefficient implementations. This is because of the slotted execution, which introduces slack, (see Figure 47a) and the hard synchronization of the application at the completion of each iteration.

In this section we discuss how the framework can be extended with *pipelined* and *non-slotted* execution, to produce more efficient schedules.

Pipelined Execution

One could argue that more parallelism can be achieved if we allowed the scheduler to span over multiple iterations. Although this is more efficient, without the synchronization at the end of an iteration, fast executing producers will overflow the buffers on the edges if the consumers are slower. In other words, the dataflow analysis will no longer guarantee the boundedness of the application.

For the propagation of the integer parameters, the scheduler would keep track the value each actor uses and calculate constraints accordingly, in a similar way that the scheduler takes into account boolean parameters. This would ensure that each actor uses the correct value of the parameter even if it lags

behind. To ensure boundedness, a buffer constraint for each edge would be added to prevent any token accumulation on the edges.

Non-Slotted Execution

The slotted scheduling model may introduce a lot of slack in the produced schedule because of the explicit synchronization after every slot. This is inherent to the model but can be mitigated using constraints to group actors with similar timings in slots (see Section 4.5.2).

The slotted model was prescribed by our target platform but we should point out that our framework can also be used to produce non-slotted schedules. In our context, where each actor is implemented by a separate processing element, the ASAP non-slotted schedule is optimal *w.r.t.* to time and constraints.

A non-slotted scheduler needs to fire new actors at the moment they become available. To do so, the evaluation of ordering constraints and the filtering process need to be adjusted.

For the ordering constraints, the scheduler main-loop must update the status vector each time an actor ends its firing (instead of at the end of each slot). The scheduler re-evaluates constraints each time an actor ends, finds new enabled actors and fires them.

Moreover, an extra vector recording the active (*i.e.*, currently executing) actors is needed. It is used to prevent active actors from being considered during constraint evaluation. This vector is also used for the evaluation of resource constraints which now apply on the enabled *and* the active actors.

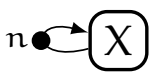
This scheduler may introduce extra overhead at run-time depending on the implementation. However, it is possible to pre-compute the actors to be fired, so that the overhead is kept to a minimum or is completely overlaid by the execution of the actors as is the case of the slotted execution.

Alternative Mapping

Our framework does not focus on mapping the actor firing on processing elements. So far, we have assumed a simple scheme with each actor mapped to a separate processing element. To take into account the mapping restrictions we have introduced for each actor an ordering constraint of the form:

$$X_i > X_{i-1}, \quad i \in [1, \#X]$$

Our scheduling framework can take into account different static mappings by adding the corresponding constraints in the framework. For example, an actor that may have instances executing in parallel on more than one processing elements can be modeled with a self edge with a number of initial tokens equal to the number of actor instances that can fire in parallel:



which in turn introduces the ordering constraint:

$$X_i > X_{i-n}, \quad i \in [1, \#X]$$

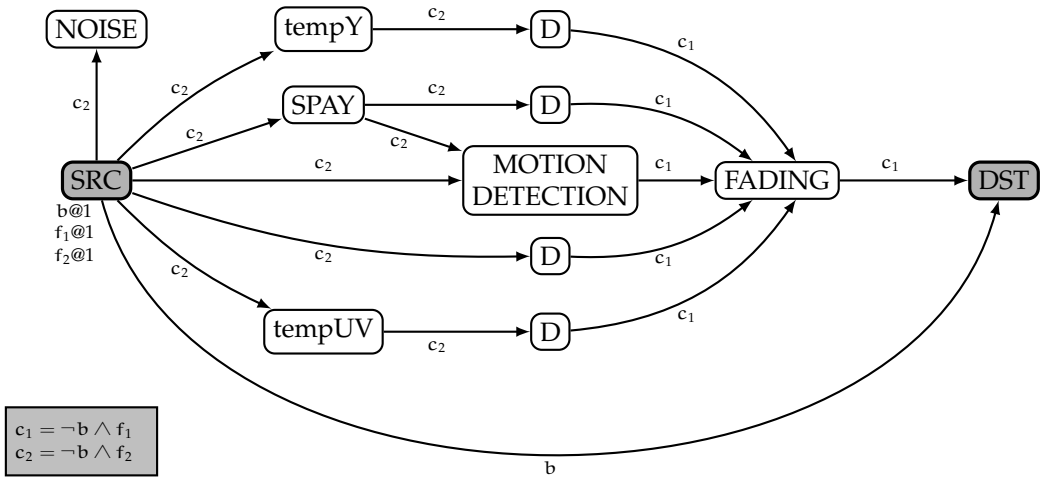


Figure 49: BPDF graph for TNR.

When multiple actors share a processing element, then filtering constraints can be used to express their mutual exclusion. For instance, Rule (34) can be used when actors A and B share the same processor. Rule (34) gives priority to actor A. A fairer sharing of the processor could be to use two resource constraints that alternate the priority with conditions:

$$\textbf{replace } A,\ B \textbf{ by } A \textbf{ if } (V_S[A] + V_S[B])\ \%2 == 1$$

$$\textbf{replace } A,\ B \textbf{ by } B \textbf{ if } (V_S[A] + V_S[B])\ \%2 == 0$$

where $V_S[A]$ is the value of the status vector for actor A indicating the number of times A has fired. The sum $V_S[A] + V_S[B]$ is the total number of times both actors have fired. In this way, the priority between the two actors alternates after either actor fires.

4.5 SCHEDULING EXPERIMENTS

In this section we evaluate the capabilities of our scheduling framework with experiments. The first experiment aims at estimating the scheduler overhead using a Temporal Noise Reduction (TNR) algorithm as a use-case. The second experiment shows how constraints can be used to manipulate the schedule using the VC-1 video decoder [68] as a use-case.

4.5.1 *Scheduler Overhead Evaluation*

TNR is an algorithm applied after the video decoding process to reduce the noise of each frame. We implemented TNR using the transaction level SystemC model of the STHORM platform, which gives accurate performance results. The application was captured as a BPDF graph and the scheduling framework was used to schedule it. The BPDF graph of TNR processes one frame per iteration. STMicroelectronics TLM simulator was used to simulate the execution of the

application on the platform and to measure the performance of each actor and each different schedule configuration. Table 3 shows the average cycles used by the processor that schedules the graph to process one frame.

We consider three different cases: the original manual implementation of the scheduler for TNR in STMicroelectronics (*Manual Schedule*), our scheduler without any simplification of constraints (*General Scheduler*) and our BPDF quasi-static schedule produced after simplification of constraints (*Generated Schedule*). For comparison, column 2 shows the corresponding performance of the fastest actor of TNR (*i. e.*, the delay actors D).

We can see that the general scheduler is much slower than the original manual schedule (more than three times more costly in cycles). Once the constraints are simplified though, the scheduler is reduced to a quasi-static schedule comparable with the manual schedule.

Even in the case of the general scheduler, however, the scheduler does not introduce overhead, since even the fastest actor of the application is considerably slower than the scheduler. Since the scheduler runs in parallel with such coarse grain actors, the application performance remains unaffected. In fact the scheduler could afford taking another 1.000.000 cycles before affecting performance. In that context, a general scheduler is realistic and allows the use of additional constraints to optimize various criteria.

4.5.2 *Use Case: VC-1 Decoder*

The VC-1 decoder [68] is a good example of a demanding codec. Its resemblance with the more recent and widely used H.264 [80] as well as with future generation codecs like HEVC [115], makes it especially relevant. The BPDF implementation of VC-1 is shown in Figure 50.

The decoder is composed of two main pipelines, the inter and the intra. The inter pipeline is composed of actor MC (Motion Compensation), while the intra pipeline is composed of actors MBB (MacroBlock to Block), INTRA (Intra prediction), and IQIT (Inverse Quantization and Inverse Transform). These two paths are combined and produce the final decoded slice in the LOOP (Loop filter). Actors SMB (Slice to MacroBlock) and MBB (MacroBlock to Block) are auxiliary actors that are used as modifiers of the boolean parameters. For easier reference, each actor is assigned a letter (shown in parenthesis).

The inter pipeline reconstructs data based on motion between different frames. For this, it fetches data from previous or future frames and, based on motion vectors, compensates the motion for the current macroblock. The intra pipeline reconstructs the data that depends on macroblocks in the neighborhood of the decoding macroblock. The intra prediction actor calculates coefficients based

	FASTEST ACTOR PERFORMANCE	GENERAL SCHEDULER	GENERATED SCHEDULE	MANUAL SCHEDULE
CYCLES / FRAME	2.140.000	1.100.000	360.000	340.000

Table 3: Schedule overhead for different schedules of TNR.

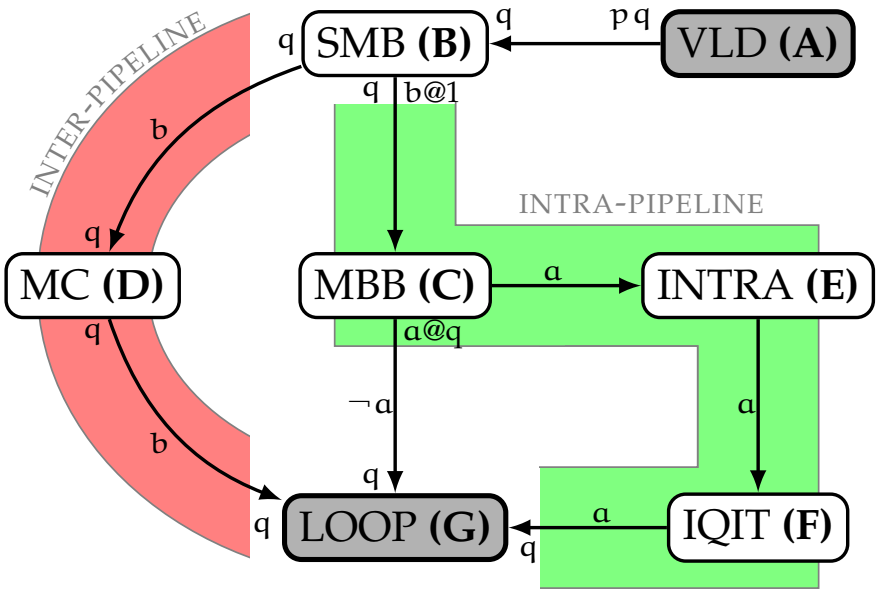


Figure 50: BPDF graph for VC-1 decoder.

on this information, IQIT applies inverse transformations to complete the decoding of the data. Finally, the residues of both pipelines are combined and smoothed in the loop filter.

The decoder makes use of two integers and two boolean parameters. The integer parameters are: p, which denotes the slice size in macroblocks, and q, which denotes the macroblock size in blocks. Each iteration of the graph processes a single slice. The boolean parameters capture whether a block is using intra (a) or inter (b) information. With these two boolean parameters, three possible modes of operation can be distinguished:

$$\begin{aligned} a \wedge \neg b &: \quad \text{Intra only} \\ \neg a \wedge b &: \quad \text{Inter only} \\ a \wedge b &: \quad \text{Intra and Inter} \end{aligned}$$

In the *Intra only* case, the value of the current block depends only on the values of the surrounding blocks. The inter pipeline is disabled. In the *Inter only* case, the value of the current block depends on the value of another block from a previous frame, as defined with a motion vector. Only the inter pipeline is used. Finally, in the *Intra and Inter* case, both pipelines are used. Note that the case where both boolean parameters are false is not functionally possible and it does not make sense as it means that there is no encoding at all. If it happens though, both pipelines are disabled and the data simply passes through to the loop filter.

By solving the balance equations, we get the repetition vector:

$$[A\ B^p\ C^{pq}\ D^p\ E^{pq}\ F^{pq}\ G^p]$$

ACTOR	EXECUTION TIME (CYCLES/FIRING)	ASAP SEQUENCES
VLD(A)	7400	$\mathcal{F}$
SMB(B)	10	$\mathcal{E}\mathcal{F}^p$
MBB(C)	10	$\mathcal{E}^2\mathcal{F}^{pq}$
MC(D)	1937	$\mathcal{E}^2\mathcal{F}^p$
INTRA(E)	288	$\mathcal{E}^3\mathcal{F}^{pq}$
IQIT(F)	365	$\mathcal{E}^3\mathcal{R}_F(pq)$
LOOP(G)	4074	$\mathcal{E}^3\mathcal{R}_G(p)$

Table 4: ASAP sequences of VC-1 decoder.

The graph is first scheduled with no additional constraints, as explained in the previous section.

The resulting schedule cannot be expressed as a single sequence using the notation introduced in Section 4.4.1. It is possible though to express the schedule using individual execution slot sequences for each actor, as shown in the third column of Table 4. Each one represents the sequence of slots of the iteration where either the actor is fired (written $\mathcal{F}$) or it remains idle (written $\mathcal{E}$). The possible idle slots after the last firing of actors are omitted.

In the case of actors IQIT(**F**) and LOOP(**G**), the schedule depends on the boolean value of a and shows an increased dynamicity. To express the sequence of firings, we use recursive functions. The function $\mathcal{R}_F$ for F is defined as:

$$\mathcal{R}_F(n) = a\ ?\ \ \mathcal{E}\ \mathcal{F}^n\ \ :\ \ \mathcal{F}^q\ \mathcal{R}_F(n-q)$$

The sequence associated to F is $\mathcal{E}^3\mathcal{R}_F(pq)$. It means that F remains idle in the first 3 slots, and then if a is `tt`, it waits one more slot and fires consecutively the remaining fires of the iteration. If a is `ff`, F fires q times in consecutive slots and then checks again the value of a by calling $\mathcal{R}_F$ with appropriately reduced number of firings.

The recursive function $\mathcal{R}_G$ for G is defined as:

$$\mathcal{R}_G(n) = a\ ?\ \ \mathcal{E}^{q+1}\mathcal{F}\ \ \mathcal{R}_{tt}(n-1)\ \ :\ \ \mathcal{E}^{q-1}\mathcal{F}\ \ \mathcal{R}_{ff}(n-1)$$

with:

$$\mathcal{R}_{tt}(n) = a\ ?\ \ \mathcal{E}^{q-1}\mathcal{F}\ \mathcal{R}_{tt}(n-1)\ \ :\ \ \mathcal{E}^{q-3}\mathcal{F}\ \mathcal{R}_{ff}(n-1)$$
$$\mathcal{R}_{ff}(n) = a\ ?\ \ \mathcal{E}^{q+1}\mathcal{F}\ \mathcal{R}_{tt}(n-1)\ \ :\ \ \mathcal{E}^{q-1}\mathcal{F}\ \mathcal{R}_{ff}(n-1)$$

The execution sequence of G is a more complex one, as it depends not only on the boolean values but also on their sequence. We consider the slot at which C is fired and sets the first value for boolean parameter a. If a is `ff`, the intra pipeline is bypassed and G should only wait $q-1$ firings of C *i.e.*, $q-1$ slots,

indicated by the *else* part of function $\mathcal{R}_G$. If a is tt, G needs to wait two more slots because of the intra pipeline (the *then* part of $\mathcal{R}_G$). For the subsequent values of a, we have the following cases:

- If a keeps the same value, the firings of G are $q-1$ slots apart, either waiting for C or for F. This is shown in the *then* (*resp. else*) part of function $\mathcal{R}_{tt}$ (*resp.* $\mathcal{R}_{ff}$).
- a switches from tt to ff: In this case, the G waits two less slots (*i. e.,* q-3) because of the absence of the intra pipeline dependency (*else* part of $\mathcal{R}_{tt}$).
- a switches from ff to tt: In this case, the G waits two more slots (*i. e.,* q+1) because of the addition of the intra pipeline dependency (*then* part of $\mathcal{R}_{ff}$).

The complete schedule is the parallel combination of all schedule streams of Table 4. It exhibits a high level of parallelism and a sample execution (produced manually) starts with:

$$A; B; (B \| C \| D); (B \| C \| D \| E); (B \| C \| D \| E \| F); \ldots$$

The produced schedule has a maximum span of $pq + 5$ slots in the worst case, where all values of a is tt. It has a minimum span of $pq + 3$ slots in the corresponding best case where all values of a is ff.

By adding user-constraints, we can modify the ASAP schedule to improve it or to satisfy some given criteria. In the following, examples of ordering and resource constraints that improve performance are given. For our experiments, we reused the VC-1 performance on STHORM based on the implementation presented in [4]. The execution time of each actor firing is shown in the second column of Table 4 (number of cycles).

To evaluate the performance of each schedule of the decoder, we developed a simulator in Java that takes as input a BPDF graph as well as a set of user constraints and simulates the slotted execution of the application based on the given execution times. When the experiments of VC-1 took place, the STHORM platform and its simulation environment were no longer available to have a more accurate performance evaluation. In the following, the overhead due to the scheduler is not taken into account as it is considered negligible.

Improving Buffer Size

In the VC-1 implementation, the inter-prediction path processes one macroblock at a time whereas the intra-prediction path processes one block at a time. Consequently, actors in the intra-prediction are fired a total number of pq times, whereas D fires only p times. This results into actor D firing in the early slots and producing a lot of tokens on the edge $\overline{DG}$. However, actor G cannot consume these tokens because the intra-prediction pipeline does not keep up and eventually blocks the actor from firing.

In particular, D will finish its iteration in the first $p+2$ slots producing a total of pq tokens. In the meanwhile, the F actor will fire at most $p+1$ times enabling G only $\left\lfloor \frac{p+1}{q} \right\rfloor$ times, leading to a consumption of at most $p+1$ tokens from edge $\overline{DG}$. Therefore, there is an accumulation of $pq-(p+1)$ tokens on $\overline{DG}$ that could be avoided.

We can limit the buffer size of the edge $\overline{DG}$ and prevent the accumulation of data in the inter-prediction path by using the buffer size restricting constraint.

In this way, we delay the inter-prediction path by constraining the actor D to wait until G has consumed q tokens. From Eq. (22) we get:

$$D_i > G_{\lceil \frac{q \cdot i - q}{q} \rceil} \Rightarrow D_i > G_{i-1}$$

This constraint adds idle intervals of $q - 1$ slots between any two consecutive firings of D. This redistribution of the firings of D has the additional benefit of a more evenly distributed power consumption, and subsequently a smaller temperature.

Although the schedule span of actor D increases, we observed only a slight increase of 2% to the total schedule time, so the total schedule span effectively remains the same. This significant change on the graph schedule is achieved by adding a single constraint. It demonstrates the flexibility of our scheduling framework.

Reducing Slack in Slots

When slotted scheduling is used, the goal is to minimize the introduced slack because of the synchronization after each slot. For this reason, we try to cluster together the more cycle-demanding actors. In Table 4, we notice that, apart from actor A that fires only once, the most costly actors are D and G. An obvious optimization is to fire them in the same slots.

We can use a resource constraint to achieve this goal. By looking at the actor schedule streams of Table 4, we see that all firings of D, after the first one, are fired in parallel with E. The following constraint can be used:

$$\textbf{replace } D, \ E \textbf{ by } E \ \textbf{ if } \neg \texttt{fireable}(G)$$

This constraint suppresses D when G is not present, effectively clustering the two actors together. The resource constraint:

$$\textbf{replace } D, \textbf{ by } \varnothing \ \textbf{ if } \neg \texttt{fireable}(G)$$

is a more straightforward way to achieve clustering, but it returns an empty set and therefore is an invalid constraint because it may cause a deadlock (*e.g.*, when the only fireable actor is D). The clustering of the two actor leads to an improvement of 15% in the total schedule time.

The resource constraint examples in Section 4.4 can be used in VC-1 as well. With precise information about actors (that we do not currently have), total power consumption could be better controlled, *e.g.*, bounded by a specified limit. Further experimentation is needed to demonstrate the way VC-1 schedule can be altered and optimized using these constraints, however the platform is not available to us yet.

Non-Slotted Execution

In Section 4.4.1, we presented a way to adjust the framework to support non-slotted execution. A non-slotted schedule is optimal *w.r.t.* timing in our context. When VC-1 is scheduled in a non-slotted manner, the total execution time is reduced by 40% from the slotted schedule without any optimization, which is a 30% improvement to the optimized slotted schedule of the previous section.

4.6 SUMMARY

The complexity of BPDF graphs raises the need for automated ways to produce efficient parallel schedules. For this reason, we introduce a scheduling framework that shifts the design focus from the production of a functional and correct schedule to the optimization of the schedule. The framework facilitates the automatic production of complex schedules that can be as efficient as the manual ones, which are often hard to produce and error-prone.

The framework specifies and implements bounded, live, and highly parallel schedules for BPDF graphs. Scheduling is made flexible by the use of user constraints that allow the framework to adapt to new execution platforms, to express optimizations and to regulate parallel firings. Static checks can ensure that constraints preserve the existence of bounded and live schedules. Although, only BPDF was considered, the framework can be adapted to schedule other data flow models as long as their data flow constraints can be expressed in the proposed constraint language.

The assumed slotted scheduling model introduces a lot slack in the resulting schedule. However, it facilitates the expression and production of quasi-static parallel schedules. In any case, although the framework was designed with slotted scheduling in mind, it can easily be used for non-slotted scheduling, thereby producing potentially more efficient schedules.

An aspect that has been ignored is mapping. The framework has been developed with demanding hardware actors in mind, so each functional actor in the BPDF application corresponds to a distinct processing element. We show that the framework can handle different static mapping schemes but still the approach does not focus on taking mapping decisions.

The scheduling framework focuses on the back-end of the scheduling procedure. Its main foal is not to produce optimal quasi-static schedules but to propose a flexible and correct by construction approach that can easily express different schedules and scheduling strategies. It allows the user to optimize the schedule at a higher level and avoid dealing with tedious implementation details of parallel schedules. The framework can then be used to explore the various scheduling possibilities and to optimize the schedule *w.r.t.* various criteria (such as latency, throughput, power consumption *etc.*).

5 THROUGHPUT ANALYSIS

I'm never gonna let you win
No I will not surrender,
Even if I start to fall
I swear to you I'll rise again

— Black Veil Brides

In this chapter, we focus on the parametric throughput calculation for the BPDF MoC. Throughput is a very important property of an application, especially when it comes to streaming. Many applications have explicit design specifications that impose strict throughput bounds at which the application must be capable to operate. Such constraints are needed to achieve real-time behaviour and an acceptable QoS. Data flow computing makes no exception and hence, there have been many studies on throughput calculation and optimization, as discussed in Section 2.4.3.

Previous work on computation of throughput for data flow does not cover the more expressive models. When it comes to a parametric model like BPDF, the existing techniques do not suffice neither to calculate nor to optimize the throughput of an application.

Knowing the throughput of a data flow graph can be beneficial both at compile time and at run-time:

- *At compile time*, the developer of the application can find and optimize the actors responsible for the bottleneck of the application. Moreover, throughput is heavily affected by the buffer sizes used and the number of initial tokens placed on directed cycles. Throughput analysis at compile time allows the developer to tune both buffer sizes and initial tokens, to meet the application requirements.
- *At run-time*, one can take decisions based on the calculated value. Once throughput is calculated, existing techniques can be reused to optimize various criteria of the application. For example, if the execution is slow, more processing elements may be allocated to increase the parallelism, or the frequency of the processing elements can be adjusted to meet the throughput requirements. On the contrary, if the application performance exceeds the throughput requirements, then power consumption can be lowered by slowing down actors (*e. g.,* by decreasing their operating voltage and frequency).

The challenge of computing the throughput for BPDF graphs is to find a *parametric* expression of the throughput at compile-time. This expression can then be evaluated efficiently at run-time. However, finding such an expression is difficult. Here, we focus on finding an expression for the *maximum achievable throughput* of a BPDF graph. Maximum throughput is of particular interest as it is the throughput the graph operates in when ASAP scheduling is used. We describe the evaluation of parametric expression of maximum throughput in Section 5.1.

However, the throughput is affected by the buffer sizes of the edges of the graph. For the graph to achieve maximum throughput, we need to ensure that the buffer sizes suffice. In Section 5.2 we revisit the throughput calculation technique for SDF graphs using a conversion of the graph to HSDF and discuss some properties of the HSDF that results from a directed cycle of two actors. We use these properties in Section 5.3 to calculate the sufficient buffer sizes that guarantee that the BPDF graph will not be slowed down and that it will operate at its maximum throughput.

We limit our approach to acyclic BPDF graphs. The current work is still in progress. It is the first steps towards a more complete approach for parametric throughout calculation.

5.1 THROUGHPUT CALCULATION

To calculate throughput, we assume an ASAP parallel pipelined execution of the graph. We are interested in the throughput the graph achieves once it enters the *steady state*, as the initial phase (typically referred to as the *prologue*) lasts a very small amount of time compared to the total operation time of the application.

We assume, as before, that each actor is executed in a different processing element and that the concurrent execution of different firings of the same actor is not allowed. We first define the terminology and the notation that is used throughout the chapter.

5.1.1 *Definitions*

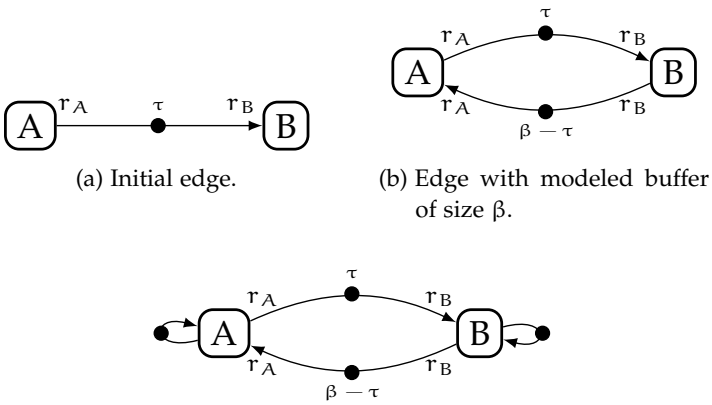
As already mentioned in Section 2.4.3, throughput indicates the performance of an application. In data flow, this corresponds to the number of completed iterations per time unit. It is also interesting to evaluate the throughput of each individual actor. This way a bottleneck can be found and the application can be further optimized.

Definition 10 (Actual actor throughput)**.** *Actual actor throughput (T_A) is defined as the number of firings the actor completes per time unit in a given execution.*

Factors that limit throughput (*i. e.,* throughput constraints) are the execution time of the actors, the data dependencies between actors, the buffer sizes of the graph edges and, finally, the directed cycles. As we assume only graphs without cycles, we focus on the first three factors. In the following, we distinguish throughput values depending on the throughput constraints taken into account for their calculation.

As actors cannot have multiple instances executing in parallel, for any given actor, there is an upper bound on the throughput the actor can achieve. This limit corresponds to the unconstrained execution of the actor, the only limiting factor being its execution time. It equates to the execution of the actor without any slack (the actor never stays idle between firings).

Definition 11 (Theoretical throughput upper bound)**.** *The theoretical throughput upper bound (T_{U_A}) of an actor A with execution time t_A is the throughput A can achieve when executing without any constraints. It is defined as:*



(a) Initial edge.

(b) Edge with modeled buffer of size β.

(c) Edge with self-loops modeling mapping dependencies.

Figure 51: Edge $\overline{AB}$ with the buffer size (β) modeled as a backward edge (51b) and the mapping dependencies as self loops (51c).

$$Tu_A = \frac{1}{t_A} \tag{38}$$

Finally, the maximum achievable throughput (or maximum throughput) of an actor is the throughput the actor achieves when the graph dependencies are taken into account. Buffers are considered to have sufficient capacity so that they do not to impose any constraint on the throughput of the graph.

Definition 12 (Maximum throughput)**.** *The maximum throughput (T_{M_A}) of an actor A is the throughput it achieves when graph dependencies are taken into account.*

The calculation of maximum throughput does not need the actual buffer sizes. If the buffer sizes are not sufficient, then the graph (and its actors) operate at a throughput strictly lower than the maximum throughput.

5.1.2 *Maximum Throughput Calculation*

We model finite buffers using backward edges as discussed in Section 2.4.3. For each edge $\overline{AB}$ of the graph, we add a backward edge $\overline{BA}$, with initial tokens:

$$init(\overline{BA}) = \beta - init(\overline{AB})$$

where β is the buffer size. In this way, the graph with the additional edges is always strongly connected.

For a strongly connected graph, boundedness means that the rate of data produced should equal the rate of data consumed on each edge. So, for the edge in Figure 51, considering that the execution times of actors A and B are t_A and t_B respectively, we get the equation:

$$T_A \cdot r_A = T_B \cdot r_B \tag{39}$$

If data is produced faster than it is consumed, there will be saturation of the backward edge and the producer will be slowed down to achieve equilibrium.

If data is consumed faster than it is produced, then an empty buffer on the edge will slow down the consumer to achieve equilibrium. In the steady state, this equilibrium has taken place and for each edge of the graph we get the balance equation (Eq. (39)).

The system of equations that results from this equilibrium is the same as the standard system of balance equations, but the unknown values are the maximum throughput values of each actor instead of the repetition counts. This means that a vector containing the throughput values of the actors of the graph, is also a solution of the system of balance equations. As a consequence, the ratio between the throughput values of two actors should be equal to the ratio of their solutions:

$$\frac{T_A}{T_B} = \frac{\#A}{\#B}$$

Moreover, throughput constraints need to be taken into account. For maximum throughput, throughput constraints are only the upper bounds of each actor and the data dependencies of the graph. The latter are captured with the balance equations, the former are expressed as extra constraints on the solution of the system. For a graph $\mathcal{G}(\mathcal{A}, \mathcal{E})$ we get the following system to solve:

$$\begin{aligned} T_A \cdot r_A &= T_B \cdot r_B \quad &, \forall\ \overline{AB} \in \mathcal{E} \\ T_A &\leqslant T_{u_A} \quad &, \forall\ A \in \mathcal{A} \end{aligned} \tag{40}$$

Unlike the original balance equations that seek the minimum integer solution, in this case we want the maximum solution that satisfy all constraints. Graph consistency guarantees that such a solution exists.

Intuitively, the solution of the system will have at least one actor operating at its upper bound. This actor will be the slowest actor imposing its constraint to all the others. With this in mind, we solve the system of throughput equations as follows:

1. *Select one of the actors that take the longest time to complete an iteration* (i. e., *the slowest actor)*. For a graph $\mathcal{G} = (\mathcal{A},\mathcal{E})$ we get actor X such that:

$$t_X \cdot \#X \geqslant t_Z \cdot \#Z, \forall\ Z \in \mathcal{A} \tag{41}$$

   where t_Z is the execution time of actor Z and $\#Z$ its solution.
2. *Set throughput of* X *to its upper bound throughput.* Here, we get:

$$T_X = T_{u_X} = \frac{1}{t_X}$$

   When the graph reaches maximum performance, the slowest actor needs to operate without any slack. If the slowest actor has slack, it can be sped up and subsequently the whole graph can operate at a higher throughput.
3. *Find the throughput values of the rest of the actors, based on the balance equations.* For any two actors X, Y we get:

$$\frac{\#X}{T_X} = \frac{\#Y}{T_Y} \tag{42}$$

   because both the repetition vector and the throughput values are solutions of the same linear system.

We now show that the solution found with the above steps satisfies the throughput constraints and that it is indeed the maximum solution that satisfies the constraints (*i. e.*, there is no greater solution).

Constraint satisfaction. For actor X, we get $T_X = \frac{1}{t_X}$ so the solution of X satisfies its throughput constraint ($T_X \leqslant T_{U_X}$). For any other actor Y with execution time t_Y, it is easy to show that:

$$T_Y \leqslant T_{U_Y} \quad \forall Y \in \mathcal{A} - \{X\}$$

where

$$T_Y = \frac{1}{t_X} \cdot \frac{\#Y}{\#X} \quad \text{and} \quad T_{U_Y} = \frac{1}{t_Y} \tag{43}$$

Indeed, from Eq. (41), we get:

$$t_X \cdot \#X \geqslant t_Y \cdot \#Y \Rightarrow \frac{1}{\#X \cdot t_X} \leqslant \frac{1}{\#Y \cdot t_Y} \Rightarrow \frac{\#Y}{\#X \cdot t_X} \leqslant \frac{1}{t_Y}$$
$$\text{(by Eq. (43))} \Rightarrow T_Y \leqslant T_{U_Y}$$ □

Maximum solution. By definition, the slowest actor X has the maximum possible solution, $T_X = \frac{1}{t_X}$. We need to prove that all other actors also get the maximum possible solution that satisfies the constraints. For this, we assume that there exists an actor Y that has a solution $T'_Y > T_Y = \frac{\#Y}{t_X \cdot \#X}$.

Any actor K sharing an edge with Y will have a solution of:

$$T'_K \cdot r_K = T'_Y \cdot r_Y \tag{44}$$

but the previous solution of K, computed with Eq. (42) was:

$$T_K \cdot r_K = T_Y \cdot r_Y < T'_Y \cdot r_Y = T'_K \cdot r_K \Rightarrow$$
$$T'_K > T_K$$

The above equation shows that an actor K sharing an edge with Y, is such that $T'_K > T_K$. Since the graph is connected, X will also share an edge with an actor with a bigger solution. But X cannot have a bigger solution because its solution is already the maximum. So, Y cannot have a solution $T'_Y > T_Y$. □

Parametric Rates

We assume that integer parameters do not change values frequently so that the graph eventually enters steady-state. Moreover, when parametric rates (similarly parametric times) are considered, comparisons between the $t_Z \cdot \#Z$ expressions may be parametric. Hence, evaluating the slowest actor is not always possible. To select one of the actors, we need to assume different cases based on the values of the parameters. For each quantity that is not comparable with the others, a different case is assumed. In the worst case, where all actors have incomparable expressions, we end up with $|\mathcal{A}|$ different cases, each one marking a different actor as the slowest one.

An algorithm that generates the different cases and their conditions is shown in Alg. 5. The algorithm takes as input the set of actors ($\mathcal{A}$), along with the

Algorithm 5 Algorithm generating all possible cases of the slowest actor along with the corresponding conditions for each case.

```
procedure GenerateCases(𝒜, R, t)
    for ∀X ∈ 𝒜 do
        for ∀Y ∈ 𝒜\{X} do
            c = compare(t[X] · R[X], t[Y] · R[Y])
            if c == 'incomparable' then
                continue
            else if c == 'lesser' then
                𝒜 = 𝒜 − {X}
                break
            else if c == 'greater' then
                𝒜 = 𝒜 − {Y}
            end if
        end for
    end for
    for ∀X ∈ 𝒜 do
        for ∀Y ∈ 𝒜\{X} do
            cond[X].add(t[X] · R[X] ⩾ t[Y] · R[Y])
        end for
    end for
    return(cond)
end procedure
```

repetition vector (R) and the vector of their execution times (t). The algorithm is consists of two pairs of nested `for-loops`. In the first pair of loops, all the comparable actors are removed from the actor set. In the second pair, only the incomparable actors remain, and for each actor X a set of conditions is generated such that:

$$t_X \cdot \#X \geqslant t_Y \cdot \#Y, \quad \forall Y \in \mathcal{A} \backslash \{X\}$$

Each symbolic expression has a numeric part ($\mathcal{N}$) and a symbolic part ($\mathcal{S}$). Hence, the previous inequality will be of the form:

$$\mathcal{N}_1 \cdot \mathcal{S}_1 \geqslant \mathcal{N}_2 \cdot \mathcal{S}_2$$

and the generated symbolic condition will be:

$$\frac{\mathcal{S}'_1}{\mathcal{S}'_2} \geqslant \left\lceil \frac{\mathcal{N}_2}{\mathcal{N}_1} \right\rceil$$

where $\mathcal{S}'_1, \mathcal{S}'_2$ are $\mathcal{S}_1, \mathcal{S}_2$ without their common factors.

Boolean Parameters

In the above calculation, the boolean values of the BPDF graph are not taken into account. The throughput calculation is done on the fully connected graph, leading to a possible underestimation of the actual throughput.

This is because removing edges from the graph reduces the scheduling constraints and may only result in actors executing *earlier* than in the fully connected graph case. Moreover, in the case an actor gets disconnected and is fired conceptually so that it keeps track of the boolean values, its execution time is considerably smaller than its regular execution time. For this reason, when boolean parameters are taken into account, throughput can only improve.

One can try to calculate throughput more accurately by adding boolean parameters in the throughput expressions. However, as boolean parameters may change multiple times inside the iteration, the schedule shape may greatly vary and it is no longer possible to argue that the graph is in steady-state.

Graph Throughput

The graph throughput, T_G, is the number of graph iterations that are completed per time unit. It can be calculated by selecting any actor, A, and dividing its throughput by its solution:

$$T_G = \frac{T_A}{\#A} \tag{45}$$

In this way, we can get up to $|\mathcal{A}|$ parametric expressions of the maximum throughput of the graph.

The maximum throughput is also the actual throughput when the graph fulfills the maximum throughput prerequisites. We need to ensure that all buffer sizes of the graph suffice for maximum throughput (Section 5.3).

Throughput Underestimation

If the integer parameters change values frequently and steady-state execution cannot be assumed, one can safely underestimate the throughput by assuming a non-pipelined slotted execution. In this case, the schedule streams computed in Chapter 4 can be used to find the maximum span (s) of the schedule in slots (Section 4.5.2). Then a safe underestimation of the execution time of an iteration is the product of the number of slots times the longest execution time:

$$t_{total} = s \cdot \max_{X \in \mathcal{G}} t_X$$

This is a gross underestimation, which can be further refined. In the case where the slot sequences of each actor are known, they can be combined in a single schedule stream using the method described in Appendix A.1. The period of a single iteration can then be evaluated by adding the maximum execution time of each slot times its index k. Hence, the total execution time of one iteration is:

$$t_{total} = \sum_{\ell \in \mathcal{L}} k \cdot max_{X \in \ell} t_X$$

where ℓ is a slot and $\mathcal{L}$ is the set of slots. For example, for the slotted schedule:

$$A^2; (A \| B)^p; C^q;$$

The total execution time then is:

$$t_{total} = 2 \cdot t_A + p \cdot \max(t_A, t_B) + q \cdot t_C$$

p	q	X	T_A	T_B	T_C	T_D	T_E	T_F	T_G
$p = 1$	$q > 20$	F	$\frac{1}{365 \cdot pq}$	$\frac{1}{365 \cdot q}$	$\frac{1}{365}$	$\frac{1}{365 \cdot q}$	$\frac{1}{365}$	$\mathbf{\frac{1}{365}}$	$\frac{1}{365 \cdot q}$
	$q < 21$	A	$\mathbf{\frac{1}{7400}}$	$\frac{p}{7400}$	$\frac{pq}{7400}$	$\frac{p}{7400}$	$\frac{pq}{7400}$	$\frac{pq}{7400}$	$\frac{p}{7400}$
$p > 1$	$q < 12$	G	$\frac{1}{4074 \cdot p}$	$\frac{1}{4074}$	$\frac{q}{4074}$	$\frac{1}{4074}$	$\frac{q}{4074}$	$\frac{q}{4074}$	$\mathbf{\frac{1}{4074}}$
	$q > 11$	F	$\frac{1}{365 \cdot pq}$	$\frac{1}{365 \cdot q}$	$\frac{1}{365}$	$\frac{1}{365 \cdot q}$	$\frac{1}{365}$	$\mathbf{\frac{1}{365}}$	$\frac{1}{365 \cdot q}$

Table 5: Parametric throughput values for the VC-1 Decoder.

In all cases, the throughput of the application is calculated with:

$$T_G = \frac{1}{t_{total}}$$

5.1.3 *Throughput Calculation Example*

Let us demonstrate the approach using the VC-1 decoder from Figure 50. Table 4 gives the execution time of each actor. With a repetition vector of

$$[A\ B^p\ C^{pq}\ D^p\ E^{pq}\ F^{pq}\ G^p]$$

For each actor, we get the total execution time it needs for one iteration ($t_N \cdot \#N$):

$$\begin{aligned} t_A \cdot \#A &= 7400 \cdot 1 \\ t_B \cdot \#B &= 10 \cdot p \\ t_C \cdot \#C &= 10 \cdot pq \\ t_D \cdot \#D &= 1937 \cdot p \\ t_E \cdot \#E &= 288 \cdot pq \\ t_F \cdot \#F &= 365 \cdot pq \\ t_G \cdot \#G &= 4074 \cdot p \end{aligned}$$

When evaluating the expressions to find the slowest actor X (first part of algorithm in Alg. 5), we end up with three actors (A, F and G) that cannot be compared with each other. Actors E, C, D, B are discarded because:

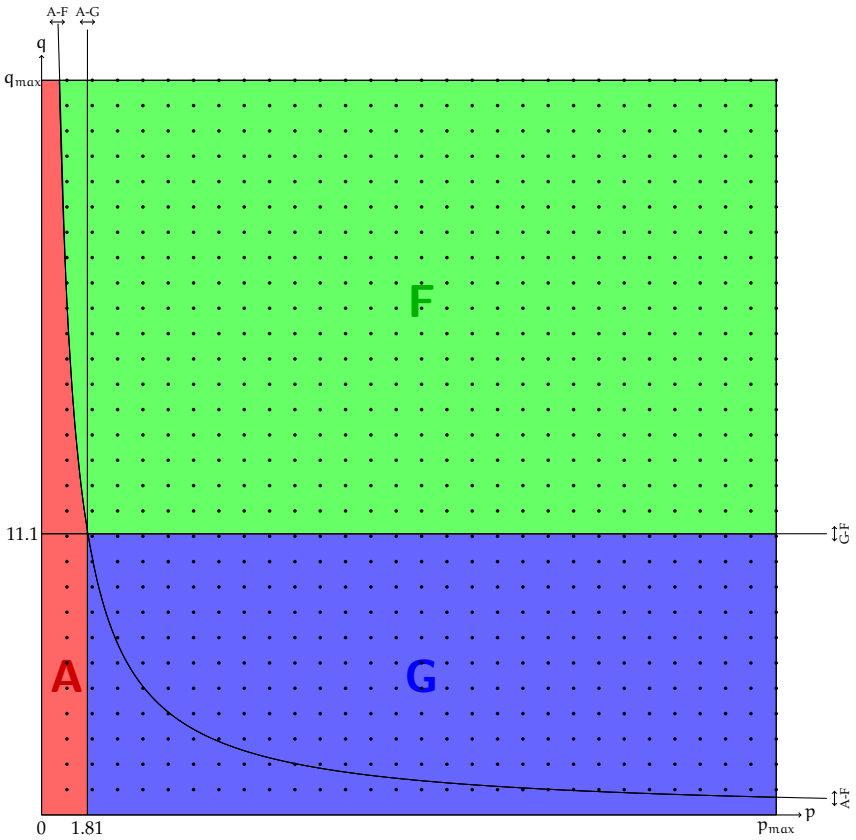
$$t_F \cdot \#F > t_E \cdot \#E > t_C \cdot \#C \quad \text{and} \quad t_G \cdot \#G > t_D \cdot \#D > t_B \cdot \#B$$

We obtain three different cases, each one for a separate actor, and a set of conditions on the parameters for each actor (the conditions of each case have been further simplified for a more readable result):
CASE $\mathbf{X = A}$ IF $(\mathbf{p = 1 \wedge q < 21})$ Indeed:

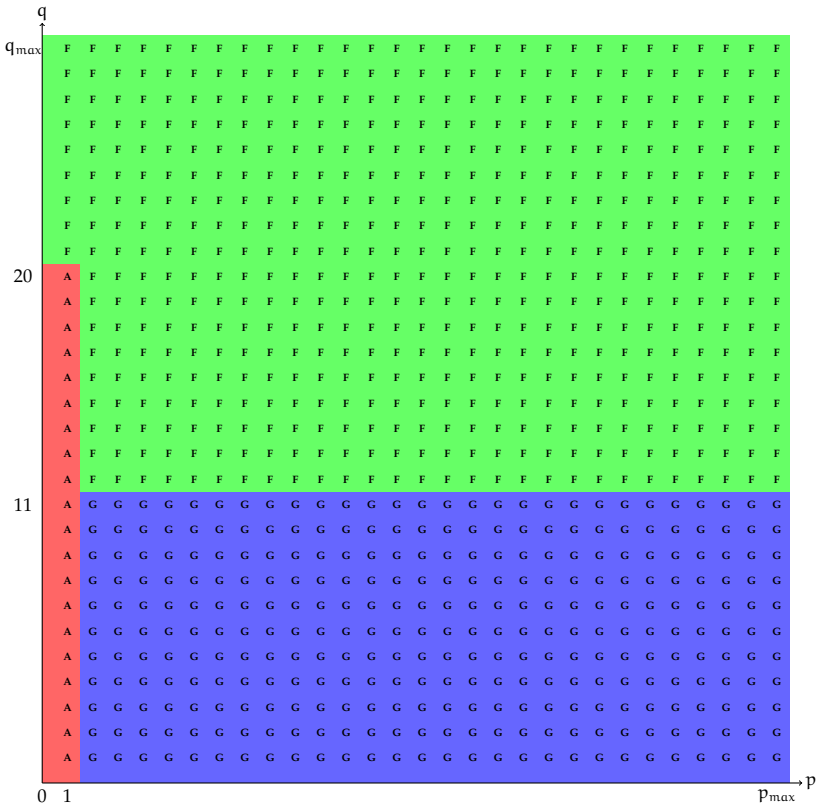
$$t_A \cdot \#A > t_G \cdot \#G \Leftrightarrow 7400 > 4074 \cdot p \Leftrightarrow p < 1.81 \Leftrightarrow \mathbf{p = 1} \quad (46)$$

$$t_A \cdot \#A > t_F \cdot \#F \Leftrightarrow 7400 > 365 \cdot pq \overset{(46)}{\Leftrightarrow} q < 20.3 \Leftrightarrow \mathbf{q < 21}$$

since $p, q \in \mathbb{N}^*$. Then we set $T_A = \frac{1}{7400}$, and compute the rest of the actor throughput values with Eq. (42):



(a) Continuous throughput regions.



(b) Discretized throughput regions.

Figure 52: VC-1 throughput regions for different pair of values of parameters p, q.

$$\begin{aligned} T_B &= \frac{1}{7400} \cdot \frac{p}{1} &&= \frac{p}{7400} \\ T_C &= \frac{1}{7400} \cdot \frac{pq}{1} &&= \frac{pq}{7400} \\ T_D &= \frac{1}{7400} \cdot \frac{p}{1} &&= \frac{p}{7400} \\ T_E &= \frac{1}{7400} \cdot \frac{pq}{1} &&= \frac{pq}{7400} \\ T_F &= \frac{1}{7400} \cdot \frac{pq}{1} &&= \frac{pq}{7400} \\ T_G &= \frac{1}{7400} \cdot \frac{p}{1} &&= \frac{p}{7400} \end{aligned}$$

CASE $\mathbf{X = G}$ IF $(\mathbf{p > 1 \wedge q < 12})$:

$$\#G \cdot t_G > \#A \cdot t_A \Leftrightarrow 4074 \cdot p > 7400 \Leftrightarrow \mathbf{p > 1}$$
$$\#G \cdot t_G > \#F \cdot t_F \Leftrightarrow 4074 \cdot p > 365 \cdot pq \Leftrightarrow q < 11.1 \Leftrightarrow \mathbf{q < 12}$$

since $p, q \in \mathbb{N}^*$. We find the throughput of the graph the graph as before by setting $T_G = \frac{1}{4074}$ and the throughput of the rest of the actors using Eq. (42). The solutions are summarized in Table 5.

CASE $\mathbf{X = F}$ IF $(\mathbf{p = 1 \wedge q > 20}) \vee (\mathbf{p > 1 \wedge q > 11})$:

$$\#F \cdot t_F > \#G \cdot t_G \Leftrightarrow 365 \cdot pq > 4074 \cdot p \Leftrightarrow \mathbf{q > 11}$$
$$\#F \cdot t_F > \#A \cdot t_A \Leftrightarrow 365 \cdot pq > 7400 \Leftrightarrow pq > 20.2 \Leftrightarrow \mathbf{pq > 20}$$

since $p, q \in \mathbb{N}^*$. Hence, we get:

$$(\mathbf{p = 1 \wedge q > 20}) \vee (\mathbf{p > 1 \wedge q > 11})$$

We find the throughput of the graph by setting $T_F = \frac{1}{365}$ and the throughput of the rest of the actors using Eq. (42). The solutions are summarized in Table 5.

In this way, the parameter space is partitioned in different regions as shown in Figure 52a. The figure shows the three boundaries that separate the parameter space into different regions, where the slowest actor differs. The boundaries are the following:

$$\begin{aligned} A - F \quad & t_A \cdot \#A = t_F \cdot \#F \Leftrightarrow q = \frac{20.3}{p} \\ A - G \quad & t_A \cdot \#A = t_G \cdot \#G \Leftrightarrow p = 1.81 \\ G - F \quad & t_G \cdot \#G = t_F \cdot \#F \Leftrightarrow q = 11.1 \end{aligned}$$

The figure also shows the pairs of $p - q$ values (black dots). Because the parameters are integer, the regions are discretized resulting in the Figure 52b.

When the parameters change values at run-time, throughput can be re-evaluated quickly by finding the region the graph operates in and then solving the corresponding parametric expressions of Table 5.

5.2 THROUGHPUT CALCULATION VIA CONVERSION TO HSDF

An acyclic BPDF graph achieves maximum throughput only if throughput is not constrained by the edge buffer sizes. Each time the parameters (both integer and boolean) of a BPDF graph take new values, they instantiate an SDF graph.

To study the requirements of buffer sizes for maximum throughput, we first look how throughput is calculated for SDF graphs by converting them to HSDF

graphs. Then we focus on the HSDF graph of the cycle that models buffer size in Figure 51. We find the properties the graph should have to achieve maximum throughput. We use these properties in Section 5.3 to calculate sufficient buffer sizes for BPDF graphs.

One way to compute throughput is to convert the graph to the equivalent HSDF graph and find the cycle with the maximum cycle mean (MCM) – the *critical cycle* [111]. For a cycle $\mathcal{C}$, with actors $\{A_1, A_2, \cdots, A_n\}$ and t_{A_i} the execution time of actor A_i, we get the cycle mean as the ratio of its total execution time (*i. e.,* the sum of execution times of its actors) over its total delay (*i. e.,* the sum of the initial tokens on its edges):

$$CM(\mathcal{C}) = \frac{execTime(\mathcal{C})}{delay(\mathcal{C})} \tag{47}$$

where

$$execTime = \sum_{i=1}^{n} t_{A_i} \quad \text{and} \quad delay = init(\overline{A_n A_1}) + \sum_{j=1}^{n-1} init(\overline{A_j A_{j+1}})$$

The MCM of a graph $\mathcal{G}$ is the MCM of all its cycles. Formally:

$$MCM(\mathcal{G}) = \max_{\mathcal{C} \in \mathcal{G}} CM(\mathcal{C}) \tag{48}$$

Once the MCM has been found, the graph throughput $T_{\mathcal{G}}$ is:

$$T_{\mathcal{G}} = \frac{1}{MCM(\mathcal{G})} \tag{49}$$

There are plenty of algorithms for finding the critical cycle, as discussed in Section 2.4.3, but in the case of cycles like the one in Figure 51, the structure of the resulting HSDF graph is very specific. The resulting generic HSDF graph (Figure 53) is composed of two main chains of actors: The one on the left consists of $\#A$ instances of actor A, and the one on the right consists of $\#B$ instances of actor B. Each of these chains has a cyclic loop back to its first actor due to the self loops added in Figure 51c. Self loops have been added to each actor of the edge to indicate the restriction that multiple instances of the actor cannot execute concurrently. In the following we assume that $t_A \cdot \#A \geqslant t_B \cdot \#B$, without loss of generality.

There are edges from the instances of actor A to those of actor B depending on the rates of the actors. These edges will have tokens on them depending on the number of initial tokens of the edge (τ). Similarly, there will be backward edges from instances of B to those of A. Some of these edges may have initial tokens depending on the buffer size (β).

When the buffer size is increased, more tokens appear on the backward links from instances of B to instances of A. Finally, maximum throughput is achieved when the MCM corresponds to the cycle formed by the actors of instances of A, $(A_1, A_2, \ldots, A_{\#A})$. Indeed, when the buffer size increases, the cycle mean of all cycles decrease in value except for the two cycles caused by the self loops. We assumed that $\#A \cdot t_A \geqslant \#B \cdot t_B$, so when the cycle formed by the instances of actor A has the MCM, maximum throughput is achieved since the MCM can no longer decrease by increasing the buffer size.

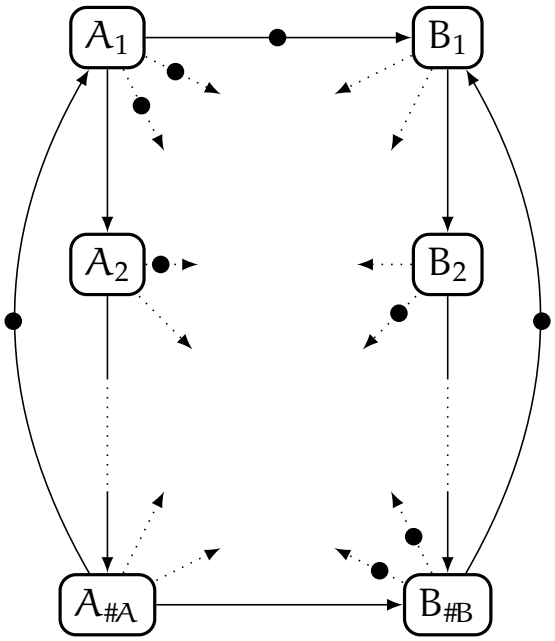


Figure 53: Generic structure of the HSDF graph resulting from the actor pair in Figure 51.

The buffer size beyond which the throughput of the graph remains unaffected is the minimum buffer size needed for the graph to achieve maximum throughput.

5.2.1 *Influence and Range*

In the next section, we estimate the minimum buffer sizes using two notions, namely *influence* and *range*, describing the dependencies of actors in the generic HSDF graph of Figure 53. We introduce these notions here.

To define influence and range we take the dependency graph that derives from the generic HSDF graph. The dependency graph spans infinitely and has no directed cycles (Figure 54). This alters the throughput analysis of the HSDF graph. The MCM corresponds to the cycle that needs the longest sequential execution between iterations. A cycle from actor A_i back to A_i in the generic HSDF graph corresponds to a path from A_i to $A_{i+\#A}$ in the dependency graph. This way, we shift the focus from trying to find cycles to finding the *longest path* from an instance of A in the first iteration to the corresponding instance of A in the next iteration. This path is the *critical path* of the graph, which also corresponds to the longest sequential execution between iterations.

The advantage of focusing on paths in the dependency graph instead of cycles in the HSDF graph is that it is easier to express buffer sizes parametrically as shown in the next section.

The dependency graph has a very distinctive structure of alternating columns of actor instances. We can represent the graph by two columns, each one corresponding to each actor, interconnected. We introduce two concepts concerning an actor instance on a particular column: *influence* and *range*.

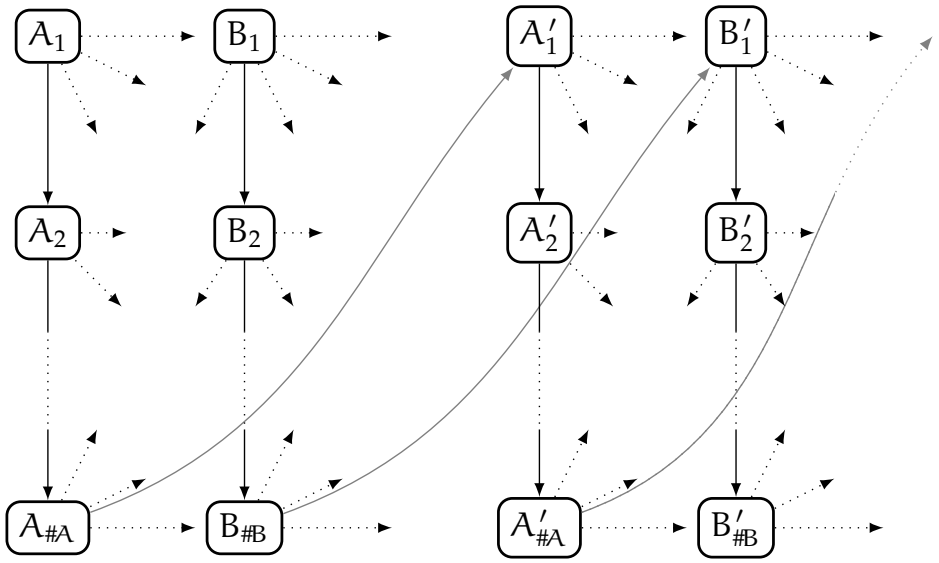


Figure 54: Unfolded graph resulting from the HSDF graph in Figure 53, the initial tokens have been removed.

Definition 13 (Influence). *For an instance A_i of actor A preceding actor B, we call* influence *of A_i, denoted $\mathcal{I}(A_i)$, the earliest instance B_j of actor B, that depends on A_i (*i. e., *the edge $\overline{A_iB_j}$ in the dependency graph exists but the edge $\overline{A_iB_{k-1}}$ does not).*

The influence of an instance A_i can be computed based on the rates and the initial tokens of the edge on the original SDF graph:

$$\mathcal{I}(A_i) = B_j \quad \text{with} \quad j = \left\lfloor \frac{r_A(i-1)+\tau}{r_B} \right\rfloor + 1 \tag{50}$$

This expression is very similar to the application constraints in Section 4.3.1. The floor gives the last instance of B that is fireable after A_{i-1} fires, that is the last instance of B that *does not* depend on A_i. By adding 1 we get the first instance of B that depends on A_i.

The importance of influence is that, as actor instances are connected vertically, when jumping between different columns the earliest available instance should be selected to get the longest path.

Before jumping to the next column however, the longest path within the current column must be traversed. To find this distance, the notion of *range* is used.

Definition 14. *The range of an instance A_i with influence B_k, is the number of consecutive instances of A following A_i, that have the same influence.*

The range is computed with:

$$R(A_i) = \left\lceil \frac{r_B - \left(r_A(i-1)+\tau\right) \bmod r_B)}{r_A} \right\rceil \tag{51}$$

Intuitively, the range expresses the number of firings of A needed for B_k to fire. The modulo in the numerator indicates the tokens provided to B_k by A_{i-1}. It is subtracted from r_B to give the remaining tokens needed for B_k to fire.

Finally, the ceiling gives the additional firings of A needed to get enough tokens, including A_i. Because $x \bmod y = x - y \left\lfloor \frac{x}{y} \right\rfloor$, we can rewrite Eq. (51) as:

$$
\begin{aligned}
R(A_i) &= \left\lceil \frac{r_B - \left(r_A(i-1) + \tau\right) + r_B \left\lfloor \frac{r_A(i-1)+\tau}{r_B} \right\rfloor}{r_A} \right\rceil \quad \Rightarrow \\
R(A_i) &= \left\lceil \frac{r_B}{r_A} - (i-1) - \frac{\tau}{r_A} + \frac{r_B}{r_A}(j-1) \right\rceil \quad \Rightarrow \\
R(A_i) &= \left\lceil \frac{r_B}{r_A} \cdot j - \frac{\tau}{r_A} \right\rceil - (i-1) \qquad (52)
\end{aligned}
$$

where j is the index from $\mathcal{I}(A_i) = B_j$. We use this expression to find the critical path that corresponds to the throughput of the graph of actors A and B (Figure 51b). Starting from the first instance of A and traversing the column of A, we check if there is a costlier path (*i. e.*, a path with larger cumulative execution time) through the instances of B. Earlier, we assumed that $\#A \cdot t_A \geqslant \#B \cdot t_B$. If this is not the case, the approach can be adapted by traversing the instances of B instead of A.

For an instance A_i with $\mathcal{I}(A_i) = B_k$, the path through the instances of B starts at B_k with $\mathcal{I}(B_k) = A_m$ and returns back to A at A_m. Hence, the two competing paths that lead to instance A_m are the direct path from A_i to A_m which costs:

$$P_A = (m - i) \cdot t_A$$

and the path through B which costs:

$$P_B = R(A_i) \cdot t_A + R(B_k) \cdot t_B$$

For each instance of A_i, a maximum cost from A_i to $A_{i+\#A}$ is found. Finally, the critical path is the path with the maximum cost among these paths.

In the next section, we use influence and range to find the buffer size needed so that there is no costlier path going through the instances of actor B and hence, the graph operates at its maximum throughput.

5.3 MINIMIZING BUFFER SIZES FOR MAXIMUM THROUGHPUT

For an acyclic BPDF graph to achieve maximum throughput, its edges need to have sufficient buffer sizes. In this section, we calculate the minimum buffer sizes that guarantee maximum throughput.

5.3.1 *Parametric Approximation of Buffer Sizes*

We model the constraints imposed by finite buffer sizes by adding backward edges with $\beta - \tau$ initial tokens where β is the size of the corresponding buffer. So, for an edge $\overline{AB}$, we get the cycle shown in Figure 51b. We consider t_A and t_B to denote the execution times of actors A and B respectively, and we assume that:

$$t_A \cdot \#A \geqslant t_B \cdot \#B$$

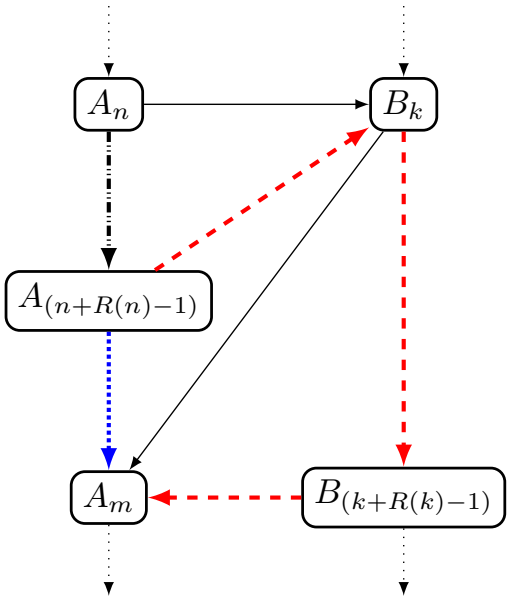


Figure 55: The two competing paths in the HSDF graph.

Without loss of generality, we assume that r_A and r_B are co-primes so that $\#A = r_B$ and $\#B = r_A$. When they are not, the solutions are divided by the $\gcd(r_A, r_B)$.

To achieve maximum throughput, we want the critical path found in Section 5.2.1 to consist only of instances of actor A, the slowest actor. Hence, we want all the paths going through B to cost less than the direct path through instances of A. Figure 55 shows the two competing paths. The dotted (blue) path is the direct path through A, while the dashed (red) path uses instances of B. The path from A_n to $A_{(n+R(A_i)-1)}$ is common to both paths, so we can discard it in the comparison. Note that the index has $R(A_i) - 1$ because the range also counts instance A_n.

The dashed path is the longest path that uses instances of B to reach A_m. That is because it starts from B_k, the first instance of B accessible from A_i ($\mathcal{J}(A_i) = B_k$) and ends at the latest instance of B that is connected with A_m. It starts with a firing of A to change to instance B_k, the first instance of B depending on A_n, where k (Eq. (50)) is:

$$k = \left\lfloor \frac{r_A(n-1)+\tau}{r_B} \right\rfloor + 1 \tag{53}$$

Then the path continues with $R(B_k)$ instances of B before reaching instance A_m, the earliest instance of actor A depending on B_k. So, by Eq. (50):

$$m = \left\lfloor \frac{r_B(k-1)+(\beta-\tau)}{r_A} \right\rfloor + 1 \tag{54}$$

where β is the buffer size to find. Therefore, the total cost of the dashed path is one firing of A and $R(B_k)$ firings of B:

$$t_A + t_B \cdot (R(B_k))$$

The total cost of the dotted path is $m - n - R(A_n)$ firings of A to reach instance A_m:

$$t_A(m - n - R(A_n) + 1)$$

As we want the dotted path to always be more expensive than the dashed one we get the following inequality:

$$\begin{aligned} t_A(m - n - R(A_n) + 1) \geqslant t_A + t_B \cdot (R(B_k)) \quad &\Leftrightarrow \\ t_A(m - n - R(A_n)) \geqslant t_B \cdot (R(B_k)) \end{aligned} \tag{55}$$

The inequality should hold for all instances of A, hence $\forall n \in [1, \#A]$. It is difficult to get an exact value because of the nested floors and ceilings, so we aim at an over-approximation. Replacing the range using Eq. (52) we get:

$$\begin{aligned} &(\text{Eq. (55)}) \Leftrightarrow \\ &t_A\left(m - n - \left(\left\lceil \frac{r_B}{r_A} \cdot k - \frac{\tau}{r_A} \right\rceil - (n-1)\right)\right) \geqslant t_B\left(\left\lceil \frac{r_A}{r_B} \cdot m - \frac{\beta - \tau}{r_B} \right\rceil - (m-1)\right) \\ &\Leftrightarrow \quad t_A\left(m - \left\lceil \frac{r_B}{r_A} \cdot k - \frac{\tau}{r_A} \right\rceil + 1\right) \geqslant t_B\left(\left\lceil \frac{r_A}{r_B} \cdot m - \frac{\beta - \tau}{r_B} \right\rceil - m + 1\right) \\ &\Leftrightarrow \quad t_A \cdot m - t_A \left\lceil \frac{r_B}{r_A} \cdot k - \frac{\tau}{r_A} \right\rceil + t_A \geqslant t_B \left\lceil \frac{r_A}{r_B} \cdot m - \frac{\beta - \tau}{r_B} \right\rceil - t_B \cdot m + t_B \\ &\Leftrightarrow \quad t_A \cdot m + t_B \cdot m - t_B \left\lceil \frac{r_A}{r_B} \cdot m - \frac{\beta - \tau}{r_B} \right\rceil \geqslant t_B + t_A \left\lceil \frac{r_B}{r_A} \cdot k - \frac{\tau}{r_A} \right\rceil - t_A \end{aligned} \tag{56}$$

Over-approximating the ceilings (minimizing the left side and maximizing the right) by $\lceil x \rceil \leqslant x + 1$ yields:

$$\begin{aligned} &(\text{Eq. (56)}) \Leftarrow t_A \cdot m + t_B \cdot m - t_B\left(\frac{r_A}{r_B} \cdot m - \frac{\beta - \tau}{r_B} + 1\right) \geqslant t_B + t_A\left(\frac{r_B}{r_A} \cdot k - \frac{\tau}{r_A} + 1\right) - t_A \\ &\Leftrightarrow \quad t_A \cdot m + t_B \cdot m - t_B \cdot \frac{r_A}{r_B} \cdot m + t_B \cdot \frac{\beta - \tau}{r_B} - t_B \geqslant t_B + t_A \cdot \frac{r_B}{r_A} \cdot k - t_A \cdot \frac{\tau}{r_A} \\ &\Leftrightarrow \quad (t_A + t_B - t_B \cdot \frac{r_A}{r_B}) \cdot m + t_B \cdot \frac{\beta - \tau}{r_B} - t_B \geqslant t_B + t_A \cdot \frac{r_B}{r_A} \cdot k - t_A \cdot \frac{\tau}{r_A} \end{aligned} \tag{57}$$

Minimizing the floor in m (Eq. (54)) using $\lfloor x \rfloor \geqslant x - 1$ we get:

$$m_{min} = \frac{r_B(k-1) + (\beta - \tau)}{r_A}$$

Replacing m with m_{min} on the left side of Eq. (57) gives:

$$\begin{aligned} &(t_A + t_B - t_B \cdot \frac{r_A}{r_B}) \cdot m + t_B \cdot \frac{\beta - \tau}{r_B} - t_B \\ = \; &(t_A + t_B - t_B \cdot \frac{r_A}{r_B}) \cdot \frac{r_B(k-1) + (\beta - \tau)}{r_A} + t_B \cdot \frac{\beta - \tau}{r_B} - t_B \\ = \; &(t_A + t_B) \cdot \frac{r_B(k-1) + (\beta - \tau)}{r_A} - t_B \cdot (k-1) - t_B \cdot \frac{\beta - \tau}{r_B} + t_B \cdot \frac{\beta - \tau}{r_B} \\ = \; &(t_A + t_B) \cdot \left(\frac{r_B(k-1)}{r_A} + \frac{\beta - \tau}{r_A}\right) - t_B \cdot (k-1) \end{aligned} \tag{58}$$

Hence:

$$\begin{aligned}&(\text{Eq. }(57)), (\text{Eq. }(58)) \Rightarrow \\ &(t_A + t_B) \cdot \left(\frac{r_B(k-1)}{r_A} + \frac{\beta - \tau}{r_A}\right) - t_B \cdot (k-1) \geqslant t_B + t_A \cdot \frac{r_B}{r_A} \cdot k - t_A \cdot \frac{\tau}{r_A}\end{aligned} \tag{59}$$

Minimizing the floor in k (Eq. (53)) using $\lfloor x \rfloor \geqslant x - 1$ and similarly maximizing using $\lfloor x \rfloor \leqslant x$ gives:

$$k_{min} = \frac{r_A(n-1) + \tau}{r_B} \quad \text{and} \quad k_{max} = \frac{r_A(n-1) + \tau}{r_B} + 1$$

Replacing k in Eq. (59) with k_{min} and k_{max} accordingly (*i. e.,* minimizing the left part and maximizing the right part) gives:

$$\begin{aligned}(\text{Eq. }(59)) \Leftarrow\ & (t_A + t_B) \cdot \left(\frac{r_B(k_{min}-1)}{r_A} + \frac{\beta - \tau}{r_A}\right) - t_B \cdot (k_{max} - 1) \\ & \geqslant t_B + t_A \cdot \frac{r_B}{r_A} \cdot k_{max} - t_A \cdot \frac{\tau}{r_A} \\ \Leftrightarrow\ & (t_A + t_B) \cdot \left(\frac{r_B}{r_A} \cdot \left(\frac{r_A(n-1)+\tau}{r_B} - 1\right) + \frac{\beta - \tau}{r_A}\right) - t_B \cdot \frac{r_A(n-1)+\tau}{r_B} \\ & \geqslant t_B + t_A \cdot \frac{r_B}{r_A} \cdot \left(\frac{r_A(n-1)+\tau}{r_B} + 1\right) - t_A \cdot \frac{\tau}{r_A} \\ \Leftrightarrow\ & (t_A + t_B) \cdot \left(n - 1 + \frac{\tau}{r_A} - \frac{r_B}{r_A} + \frac{\beta}{r_A} - \frac{\tau}{r_A}\right) - t_B \cdot \frac{r_A(n-1)+\tau}{r_B} \\ & \geqslant t_B + t_A \cdot (n-1) + t_A \cdot \frac{\tau}{r_A} + t_A \cdot \frac{r_B}{r_A} - t_A \cdot \frac{\tau}{r_A} \\ \Leftrightarrow\ & t_A \cdot (n-1) - t_A \cdot \frac{r_B}{r_A} + t_A \cdot \frac{\beta}{r_A} + t_B \cdot (n-1) - t_B \cdot \frac{r_B}{r_A} + t_B \cdot \frac{\beta}{r_A} \\ & \geqslant t_B + t_A \cdot (n-1) + t_A \cdot \frac{r_B}{r_A} + t_B \cdot \frac{r_A(n-1)+\tau}{r_B} \\ \Leftrightarrow\ & \beta \cdot \frac{t_A + t_B}{r_A} \geqslant t_B - t_B \cdot (n-1) + t_B \cdot \frac{r_B}{r_A} + 2 \cdot t_A \cdot \frac{r_B}{r_A} + t_B \cdot \frac{r_A(n-1)+\tau}{r_B}\end{aligned} \tag{60}$$

With $n_{max} = \#A = r_B$ and $n_{min} = 1$ we get:

$$\begin{aligned}(\text{Eq. }(60)) \Leftarrow\ & \beta \cdot \frac{t_A + t_B}{r_A} \geqslant t_B \cdot (2 - n_{max}) + t_B \cdot \frac{r_B}{r_A} \\ & \qquad + 2 \cdot t_A \cdot \frac{r_B}{r_A} + t_B \cdot \frac{r_A(n_{min}-1)+\tau}{r_B} \\ \Leftrightarrow\ & \beta \cdot \frac{t_A + t_B}{r_A} \geqslant 2 \cdot t_B - t_B \cdot r_B + (t_A + t_B) \cdot \frac{r_B}{r_A} + t_A \cdot \frac{r_B}{r_A} + t_B \cdot \frac{\tau}{r_B} \\ \Leftrightarrow\ & \beta \geqslant \frac{2 \cdot t_B \cdot r_A}{t_A + t_B} - \frac{t_B \cdot r_A \cdot r_B}{t_A + t_B} + r_B + \frac{t_A \cdot r_B}{t_A + t_B} + \frac{t_B \cdot r_A \cdot \tau}{r_B \cdot (t_A + t_B)}\end{aligned} \tag{61}$$

$t_A \cdot r_B$ is the total time A needs to finish an iteration and similarly $t_B \cdot r_A$ is the total time needed by B. To simplify the expression we set:

$$\sigma_X = \frac{t_X \cdot \#X}{\sum_{Z \in e} t_Z}$$

where e is the given edge and we get:

$$\begin{aligned}(\text{Eq. (61)}) &\Leftrightarrow \beta \geqslant 2 \cdot \sigma_B - r_B \cdot \sigma_B + r_B + \sigma_A + \sigma_B \cdot \frac{\tau}{r_B} \\ &\Leftrightarrow \beta \geqslant \sigma_B \left(2 + \frac{\tau}{r_B} - r_B\right) + r_B + \sigma_A \end{aligned} \tag{62}$$

Experiments show that doubling the minimum buffer size required for liveness (Eq. (3)):

$$\beta_{min} = \begin{cases} r_A + r_B - g + \tau \bmod g & \text{if } 0 \leqslant \tau \leqslant r_A + r_B - g \\ \tau & \text{otherwise} \end{cases}$$

suffices for maximum throughput. Here, the buffer size is over-approximated with the benefit that it can be expressed parametrically. The over-approximation of Eq. (62) can be adapted accordingly when B is the slowest actor. In this case, the direct path would consist of instances of actor B and the alternative path of instances of A.

5.3.2 *Exact Calculation of Buffer Sizes*

Instead of over-approximating, we can calculate the exact buffer size that suffices for maximum throughput. The algorithm in Alg. 6 calculates the exact buffer size for an edge with fixed rates (SDF). For parametric rates (BPDF), an exhaustive search for all possible parametric values of the rates of an edge could be used to find the worst case and to evaluate the exact buffer size needed.

The algorithm takes as input a pair of actors along with their maximum throughput from Section 5.1.2. Actors are modeled with a structure that contains four fields: *execution time*, *port rate*, *remaining time* and *active*. The first two are the classical fields that define the actor. The last two are auxiliary fields used by the algorithm:

- *remaining time* keeps the time units left for the current firing to finish.
- *active* is a boolean value indicating whether the actor is currently executing or not.

For a given edge $\overline{AB}$, the algorithm FindMinBuffer initializes the buffer size of the connecting edge to the middle point between the over-approximation (Eq. (62)) and the minimum buffer size for liveness (Eq. (3)):

$$\beta_{init} = \left\lceil \frac{\beta_{over} + \beta_{min}}{2} \right\rceil$$

Then it checks whether the slowest actor stays idle while the other actor is active (*i. e.*, whether there is *slack* or not) using the HasSlack (Alg. 7) procedure. While the slack is non-zero, the buffer is increased to the middle value between the previous value and the over-approximation. When the slack is zero, the buffer decreases until it reaches a non-zero value, thereby resulting in the minimum buffer size with zero slack.

HasSlack takes as input a pair of actors and the buffer size of their connecting edge, and simulates the execution of the pair. The buffer size is modeled with a backward edge as in Figure 51. In the following, actor A is assumed to

Algorithm 6 Algorithm finding the minimum buffer size for maximum throughput

```
procedure FindMinBuffer(A, B)
    β = ⌈(β_over + β_min)/2⌉
    while HasSlack(A, B, β) do
        β = ⌈(β_over + β)/2⌉
    end while
    while ¬HasSlack(A, B, β) do
        β = β − 1
    end while
    β = β + 1
    return(β)
end procedure
```

be the slowest ($t_A \cdot \#A \geqslant t_B \cdot \#B$). HasSlack uses a set of simple functions to simulate their execution:

HasFinished(X, A) is a function that checks whether actor A has finished X number of firings. X is a large multiple of the solution of the actor, guaranteeing that the simulation lasts until the execution of the graph has entered steady state.

CanFire(A) checks whether actor A can fire. If the buffer contains enough tokens and the actor is not active, it returns true.

Fire(A) sets the *active* field of actor A to true and its remaining time equal to its execution time. Then it updates the status of its input buffer by removing the consumed tokens.

IsActive(A) returns the value of the *active* field of actor A.

AdvanceTime advances time to the next termination of an actor. It decreases the remaining time of each active actor by the equivalent amount of time. If the remaining time reaches zero, the actor finishes execution, it produces tokens on its output edge, and its *active* field is set to false.

HasSlack simulates until the slowest actor (here A) reaches the desired number of firings of until A has slack. The simulation advances from the termination of a firing to the next termination using the AdvanceTime function. Each time the algorithm checks whether any actor is eligible to fire and sets him to fire. If the slowest actor is idle *while* the other actor is active, then the algorithm returns true because the slowest actor has slack in its execution.

This approach returns the exact buffer size needed for maximum throughput execution of a single SDF edge. However, the approach does not take into account the fact that some actors will not operate at their maximum throughput value based on the given edge because they may be further restricted by another edge. In other words, the algorithm finds local solutions but not global solutions for the minimization of the buffers of a whole SDF graph.

Another drawback of the algorithmic solution is that it cannot handle integer parameters. For BPDF, one should first find the worst combination of rates between the two actors, an analysis that is beyond the scope of this work.

Algorithm 7 Algorithm finding whether the slowest actor has slack in its operation

```
procedure HasSlack(A, B, β)
    while ¬HasFinished(A) do
        if CanFire(A) then
            Fire(A)
        end if
        if CanFire(B) then
            Fire(B)
        end if
        if IsActive(B) ∧ ¬IsActive(A) then
            return(true)
        end if
        AdvanceTime()
    end while
    return(false)
end procedure
```

5.4 SUMMARY

In this chapter, the problem of throughput calculation of parametric data flow graphs was discussed. Throughput calculation is important as throughput values can be taken into account to improve the design at compile time and optimize the execution at run-time.

Most throughput calculation techniques are limited to SDF graphs or consider only the worst case throughput. We focused on calculating throughput parametrically, so that it can be easily reevaluated at run-time based on changes of parameter values.

Finding a parametric expression for throughput is not an easy task. Hence, our proposal focuses on the parametric expression of the maximum achievable throughput which is easier to express. However, maximum throughput may not be achieved due to too small buffers or directed cycles with not enough tokens.

In Section 5.3, we explored the buffer requirements for maximum throughput. We proposed two ways to calculate sufficient buffer sizes, one parametric overapproximation and one algorithmic approach that finds the exact buffer sizes but cannot be applied to edges with parameters belonging to large intervals.

Our approach is limited to graph without cycles. To extend the approach to cyclic graphs, the cycle requirements for sufficient initial tokens need to be defined. This problem is similar to the buffer size problem, as buffer sizes are modeled as cycles of size 2.

Finally, we do not take into account the impact of the boolean parameters on throughput. Indeed, boolean parameters can impact the performance of a graph by disabling parts of it. However, this behaviour greatly complicates throughput calculation. The fully connected graph gives always a safe underestimation of the actual throughput value of any BPDF graph.

6 CONCLUSIONS

> Aggressively we all defend the role we play
> Regrettably time's come to send you on your way
>
> — The Killers

6.1 CONCLUSIONS

We developed BPDF, a novel parametric data flow Model of Computation (MoC) that allows changing the port rates and the topology of a graph at run-time. Despite the increase in expressiveness, BPDF remains statically analyzable. In this way, qualitative properties of an application, such as bounded and deadlock-free execution can be verified at compile-time. We believe that BPDF finds a balance point between expressiveness, analyzability and schedulability. It is expressive enough to efficiently capture modern streaming applications, while providing static analyses and moderate schedulability.

BPDF uses a combination of integer and boolean parameters that makes its parallel scheduling challenging. In Chapter 4, we developed a scheduling framework that facilitates parallel scheduling of BPDF applications. The framework takes as input a set of constraints which are then compiled into a parallel schedule or processed by a scheduler at run-time. Constraints are data dependencies deriving from the topology of the BPDF graph, but can also be user constraints that are used to restrict the buffer sizes on the graph edges or the level of parallelism achieved by the application.

We have proposed static analyses to guarantee that the set of constraints preserve the liveness and boundedness of the application. Constraints can be used to explore various scheduling possibilities and to optimize the schedule *w.r.t.* various criteria. Our framework does not aim at producing an optimal quasi-static schedule but at providing a flexible and correct-by-construction approach that can easily express different schedule policies.

Finally, the parametric expression of the throughput of a BPDF graph is important, both at compile-time and at run-time, because it helps taking design and scheduling decisions to optimize an application. In Chapter 5, we focused on the parametric expression of the maximum throughput for acyclic BPDF graphs as well as buffer size requirements that guarantee that the graph will operate at maximum throughput at run-time. Our approach on parametric throughput calculation is limited to a small subset of BPDF graphs but this is just a first step towards a more complete calculation procedure.

(a) A BPDF graph with data duplication on edge $\overline{BC}$.

(b) Equivalent BPDF graph with fractional rates to suppress data duplication.

Figure 56: Actor B needs to produce a single token. In Figure 56a, the token is duplicated p times because of consistency.

6.2 FUTURE WORK

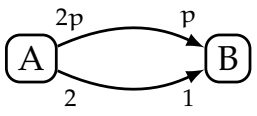
The development of the BPDF MoC along with the scheduling framework and the parametric throughput analysis can be further extended for applications and implementations that do not fit our current assumptions.

6.2.1 *The BPDF Model of Computation*

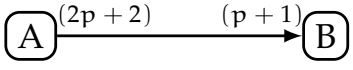
BPDF can be extended in many ways. Allowing integer parameters to change values *within* an iteration is a feature that can be used in many applications. Currently, such functionality can be achieved with composite BPDF actors as discussed in Section 3.3.4. BPDF can benefit from other models, such as SPDF, which allows such changes. However, this extension significantly increases the scheduling complexity, so changing periods should be further restricted in comparison with SPDF.

Another extension would allow port rates to be *fractional* (already used in parameter propagation edges, see Section 3.3.2) and/or *polynomial*. Fractional rates can be used to avoid duplication of the same value, as is the case with the boolean values of BPDF. Moreover, it provides a more intuitive representation of some applications. An example is given in Figure 56. Actor B needs to produce only a single token on edge $\overline{BC}$. However, for the graph to be consistent, B fires p times producing p tokens (Figure 56a). Data duplication can be avoided with a fractional rate of $\frac{1}{p}$ (Figure 56b). In this way, B fires p times but produces only a single token on edge $\overline{BC}$.

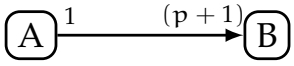
Polynomial rates are implicitly used in BPDF when a multi-rate graph is used, as shown in Figure 57a. When the multiple edges are reduced to one (Fig-



(a) A BPDF multi-graph.



(b) Equivalent BPDF graph with the multiple edges reduced to one.



(c) Graph not supported by BPDF.

Figure 57: BPDF graphs with polynomial rates.

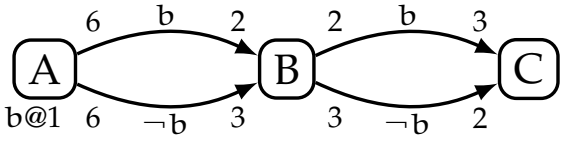


Figure 58: A BPDF graph that could be consistent with multiple repetition vectors.

ure 57b), the graph has parametric rates. However, graphs like the on in Figure 57c are not supported by the model. They are useful though, in many applications, where the first token contains some configuration information which is followed by p tokens containing data.

Static analyses of BPDF do not take into account boolean values to avoid the explosion of different combinations that occur. The model can be adjusted to support such analyses. In this way, BPDF graphs that would otherwise be rejected as inconsistent can be taken into account. Such a graph is shown in Figure 58. The graph is inconsistent because of the BC cycle. However, if the boolean parameters are taken into account, one can end up with two different repetition vectors based on the value of the boolean parameter b:

$$r = \begin{cases} [A\ B^3\ C^2] & b = \texttt{tt} \\ [A\ B^2\ C^3] & b = \texttt{ff} \end{cases}$$

Using repetition vectors based on the values of the boolean parameters raises a lot of problems though. If a repetition vector depends on a boolean parameter that changes values within an iteration, then the vector can change values in the middle of an iteration, greatly complicating analyses and scheduling. Such a functionality can be achieved with the use of sub-repetition vectors that correspond to the boolean periods.

Finally, BPDF has been implemented as a standalone model in Java. Such an implementation makes comparison with other existing data flow MoCs difficult. BPDF needs to be integrated in larger frameworks such as Ptolemy [102] and its extensions PeaCE [59] and Open RVC-CAL [123] as well as SDF^3 [113] and Dataflow Intechange Format (DIF) [62], to make such a comparison possible.

6.2.2 *Scheduling Framework*

In Section 4.4.2, we mentioned some possible extensions of the scheduling framework. More precisely, we discussed how to *pipeline* execution by adding buffer constraints on all edges and disregarding the repetition vector, and how *non-slotted* execution can be achieved by adapting the scheduler to fire actors every time an actor finishes execution. Moreover, we briefly discussed how alternative static mapping schemes can be taken into account with the addition of corresponding ordering and resource constraints. However, we were not able to implement this kind of functionality and to compare the schedules produced by our framework with schedules produced by different approaches.

The framework can be further extended to allow dynamic addition and removal of constraints at run-time. Dynamic addition of ordering constraints

should guarantee liveness. If all the possible constraints are known at compile-time, then our current analysis suffices to guarantee liveness. However, this may be a gross over-approximation. The analysis can be more precise if only the possible *combinations* of constraints are taken into account; the cost of evaluating all the possible combinations can be high but since it is done off-line it hardly matters.

Resource constraints always guarantee liveness by construction but, to avoid dynamic evaluation, the static compilation of the constraints, discussed in Section 4.4, needs to take into account the different possible sets of resource constraints that may occur at run-time.

This dynamic behaviour can be used for dynamic mapping: changing mapping scheme can take place by removing the constraints of the previous mapping scheme and adding the corresponding constraints for the new one. Because the set of constraints for each mapping do not overlap, using our liveness analysis on each different set suffices to guarantee liveness. Resource constraints can be optimized in a similar manner, by providing a different lookup table for each different set of resource constraints. In this way, the framework can support a wider implementation range that combines both mapping and scheduling.

6.2.3 *Parametric Throughput Analysis*

Our parametric throughput analysis in Chapter 5 focuses only on the maximum achievable throughput for BPDF graphs. However, it needs to be checked that the graph will operate at its maximum throughput. Hence, we provided a method to calculate buffer sizes in the case of acyclic graphs.

When it comes to BPDF graphs with directed cycles, it should also be checked whether there are enough initial tokens on a cycle to achieve maximum throughput. An approximation can be made though, by clustering the actors of a cycle until the cycle is composed of 2 actors and our algorithm for computing the buffer sizes can be used. In general, the number of initial tokens of an application cannot be changed because this affects also the functionality of the application. Nevertheless, such an analysis can be useful to recognize bottlenecks and possibly alter the port rates or the timing of the actors in the cycle.

The initial ambition behind our throughput calculation approach was to formulate a set of throughput constraints based on the edges of the graph, which would then be taken into account to solve the system of balance equations and to find parametric throughput expressions for the actual throughput of the graph. For the calculation of maximum throughput, for example, we considered only the throughput constraints imposed by the timing of the actors.

Another formulation could take into account the limitations on throughput of an actor due to the buffer sizes of its edges. Experiments show that such a formulation can be possible for acyclic graphs. In the general case, though, it may not be possible to find a formulation of throughput constraints that reduces throughput calculation to a linear system.

Another limitation of our approach is the requirement that the graph enters steady state execution. If the integer parameters change values often, then

our approach may over-approximate the throughput of the application. To deal with the transient phase, one can under-estimate throughput and gradually improve the estimation using an unfolding factor u. Every time the integer parameters change, u is set to 1 and each iteration that the parameters remain the same, u increases. The throughput value used is the throughput of the non-pipelined execution of one iteration with an unfolding factor u. Finally, after u has reached a value high enough to assume steady state execution, our method can be used.

In Section 5.1.2, we briefly discussed the effect of boolean parameters on throughput. In our analyses, we consider the BPDF graph with all its edges activated. A better approximation could be made, if the matrix holding the throughput values (see Table 5), apart from the conditions on the integer parameters also has conditions on the boolean values. Care should be taken though, because the constant change of boolean parameters within the iteration means that the initial assumption of the graph entering steady state execution no longer holds.

Moreover, when calculating buffer sizes, we always compute the buffer size that guarantees maximum throughput for each edge, effectively over-estimating the buffer size needed. This is because the algorithm we use does not take into account the fact that most actors (apart from the slowest ones) will not operate at their theoretical upper bound but rather at a lower value. Calculating the exact buffer size for BPDF is a challenging problem yet to be solved.

Once a parametric expression for throughput is known, it can be used at runtime to optimize the power consumption of the application. There are many ways to reduce power consumption using DVFS as briefly discussed in Section 2.4.3. In our approach, the scheduler could lower the Voltage-Frequency scale of each hardware processing element stopping just before the application fails to meet its QoS requirements, effectively reducing the power consumption of each actor.

Data flow MoCs have great potential to shape the way we develop systems. They enable modular design and static analyses, in the design phase, and code generation and automation in the implementation phase. The resulting systems are developed faster and cheaper and at the same time they are more reliable and maintainable. In this way, not only more complex designs can be conceived by experienced developers, but also development is made accessible to less knowledgeable ones. Hence, more ideas are likely to come to fruition.

Part I

APPENDIX

A

APPENDIX

A.1 SCHEDULE STREAMS

The *schedule stream* of an actor is a sequence of events where the actor is either idle ($\mathcal{E}$) or firing ($\mathcal{F}$). The events are separated with the ';' operator that indicates sequential execution. Consecutive events of the same type are clustered together using indexes. The execution advances from the left to the right of the stream. For example, stream S_1:

$$S_1 = \mathcal{E}; \mathcal{F}; \mathcal{F}; \mathcal{F}; \mathcal{E}; \mathcal{E}; \mathcal{F}$$

is written:

$$S_1 = \mathcal{E}; \mathcal{F}^3; \mathcal{E}^2; \mathcal{F}$$

For each schedule stream, a vector with its indexes indicates the *steps* at which the stream change from $\mathcal{E}$ to $\mathcal{F}$ and vice-versa. This vector (V) is called the *step vector*. For stream S_1 this vector is

$$V_1 = [1\ 3\ 2\ 1]$$

Similarly, an event vector (E) is a vector with just the events of a schedule stream:

$$E_1 = [\mathcal{E}\ \mathcal{F}\ \mathcal{E}\ \mathcal{F}]$$

The merging of multiple schedule streams into a *composite schedule stream* produces also a sequence of events, where an event can be: idle ($\mathcal{E}$), firing of an actor (X_1) or parallel firing of multiple actors, indicated by the '$\|$' operator ($X_1 \| X_2 \| \ldots \| X_n$).

The main intuition behind the merging of multiple actor streams into a composite stream is the traversal of all the streams one step (σ) at a time, and the addition of the parallel execution of all the events across the multiple streams as a single event on the composite stream that repeats σ times. The step σ needs to be the minimum step till the next event change in across all streams:

$$\sigma = \min V_X[1],\ \forall X \in \mathcal{A}$$

However, integer parameters appear in the step vectors and finding the minimum step is not possible if the indexes are incomparable. In this case, all possible cases are considered and multiple composite streams are produced, one for each case.

The algorithm Alg. 9 is a recursive procedure that takes a set of actor step vectors (V) along with the corresponding set of event vectors (E), a composite stream (S), a set of conditions (C) and the actor set ($\mathcal{A}$) and produces set of composite streams (Σ) advancing the given stream by one step. Each different composite stream is linked with a set of conditions on the integer parameters (C).

Algorithm 8 Algorithm returning the set of actors with incomparable steps

```
procedure FINDMIN(V, 𝒜, C)
    𝒜_r = 𝒜
    for X ∈ 𝒜_r do
        for Y ∈ 𝒜', X ≠ Y do
            if V_X[1] ⩾ V_Y[1] then
                𝒜_r = 𝒜_r − {X}
                break
            end if
            if V_X[1] < V_Y[1] then
                𝒜_r = 𝒜_r − {Y}
            end if
        end for
    end for
    return(𝒜_r)
end procedure
```

```
procedure UPDATE VECTORS(X)
    SHIFT LEFT(V'_X)
    SHIFT LEFT(E'_X)
    if V'_X == ∅ then
        𝒜' = 𝒜 − {X}
        V' = V' − {V_X}
        E' = E' − {E_X}
    end if
end procedure
```

First, Σ is set to $\emptyset$. Then, FINDMIN returns the actor with the minimum step or a set of incomparable actors. FINDMIN gets the set of step vectors, the actor set and the set of current conditions on the integer parameters. It is composed of two nested `for-loops` which compare the first entry in the step vector of each actor. The comparable actors with larger steps are removed from the actor set. In the comparison, the current conditions are taken into account. Finally, the reduced actor set with the actor with the smaller step or the set of incomparable actors is returned.

STREAMMERGE proceeds with a `for-loop` over the reduced set of actors ($\mathcal{A}_r$). For each incomparable actor X, a different step will be used and a different composite stream will be generated. In each case, all sets $(V, E, S, C, \mathcal{A})$ are copied to $(V', E', S', C', \mathcal{A}')$ to be altered separately. Then, σ is calculated as $V_X[1]$ and event e, to be placed in the composite stream, is initialized to $E_X[1]$.

UPDATE VECTORS is a function that updates the vectors of an actor. In particular, V_X and E_X are updated by shifting all values to the left and removing the first value. If V_X become empty, the actor (X) and its corresponding vectors (V_X,E_X) are removed from the relevant sets.

A nested `for-loop` is used to advance the step vectors of all the other actors and add their corresponding events to event e. Moreover, the set of conditions

Algorithm 9 Algorithm merging a set of actor schedule streams into a composite schedule streams

```
procedure STREAM MERGE(V, E, S, C, 𝒜)
    Σ = ∅
    𝒜_r = FINDMIN(V, 𝒜, C)
    for X ∈ 𝒜_r do
        V' = V
        E' = E
        S' = S
        C' = C
        𝒜' = 𝒜
        σ = V'_X[1]
        e = E'_X[1]
        UPDATE VECTORS(X)
        for Y ∈ 𝒜, Y ≠ X do
            C' = C' + (σ ⩽ V'_Y[1])
            V'_Y[1] = V'_Y[1] − σ
            e = e‖E'_Y[1]
            if V'_Y[1] == 0 then
                UPDATE VECTORS(Y)
            end if
        end for
        e = e^σ
        S' = S' + E
        if 𝒜' ≠ ∅ then
            Σ = Σ + STREAM MERGE(V', E', S', C', 𝒜')
        else
            Σ = Σ + (S', C')
        end if
    end for
    return(Σ)
end procedure
```

needed for σ to be the smaller step is generated and added to the set of conditions.

After the loop, e is repeated σ times and added to the composite stream. If the actor set is not empty, STREAMMERGE is called again with the updated sets $(V', E', S', C', \mathcal{A}')$. Else the current stream is added to the set of composite streams (Σ) along with its conditions. Finally, the algorithm returns the set of composite streams, each one related with a set of conditions.

Merging example

Consider the set of schedule streams:

$$S_A = \mathcal{F}^2; \mathcal{E}^2; \mathcal{F}^p$$
$$S_B = \mathcal{E}^5; \mathcal{F}^q;$$
$$S_C = \mathcal{E}^3; \mathcal{F}^p; \mathcal{E}; \mathcal{F}^p$$

with

$$V_A = [2\ 2\ p] \qquad E_A = [\mathcal{F}\ \mathcal{E}\ \mathcal{F}]$$
$$V_B = [5\ q] \qquad E_B = [\mathcal{E}\ \mathcal{F}]$$
$$V_C = [3\ p\ 1\ p] \qquad E_C = [\mathcal{E}\ \mathcal{F}\ \mathcal{E}\ \mathcal{F}]$$

STREAM MERGE will start with $V = [V_A\ V_B\ V_C]$, $E = [E_A\ E_B\ E_C]$, $S = \varnothing$, $C = \varnothing$ and $\mathcal{A} = [A\ B\ C]$. In the first run, FINDMIN will return actor A because all the first entries in V are comparable. The composite stream will be updated with the parallel execution of all events (here only A fires):

$$S = \mathcal{F}_A^2$$

also written as:

$$S = A^2$$

and all vectors will be updated:

$$V_A = [2\ p] \qquad E_A = [\mathcal{E}\ \mathcal{F}]$$
$$V_B = [3\ q] \qquad E_B = [\mathcal{E}\ \mathcal{F}]$$
$$V_C = [1\ p\ 1\ p] \qquad E_C = [\mathcal{E}\ \mathcal{F}\ \mathcal{E}\ \mathcal{F}]$$

The procedure will call it self with the updated sets. This time actor C returns and the step is 1. All actors are idle, hence:

$$S = A^2; \mathcal{E}$$

and

$$V_A = [1\ p] \qquad E_A = [\mathcal{E}\ \mathcal{F}]$$
$$V_B = [2\ q] \qquad E_B = [\mathcal{E}\ \mathcal{F}]$$
$$V_C = [p\ 1\ p] \qquad E_C = [\mathcal{F}\ \mathcal{E}\ \mathcal{F}]$$

Next, FINDMIN will return both A and C which are incomparable. Actor B is removed because of A. The algorithm will split the streams in two based on whether $1 \leqslant p$ or $p \leqslant 1$ [1]. In both cases, the composite stream will be:

$$S = A^2; \mathcal{E}; C$$

but the vectors will differ: for $p > 1$ they are:

$$V_A = [p] \qquad E_A = [\mathcal{F}]$$
$$V_B = [1\ q] \qquad E_B = [\mathcal{E}\ \mathcal{F}]$$
$$V_C = [p - 1\ 1\ p] \qquad E_C = [\mathcal{F}\ \mathcal{E}\ \mathcal{F}]$$

1 In the case of equality any stream will do so we do not differentiate.

but for $p \leqslant 1$ we get:

$$V_A = [p] \qquad E_A = [\mathcal{F}]$$
$$V_B = [1\ q] \qquad E_B = [\mathcal{E}\ \mathcal{F}]$$
$$V_C = [1\ p] \qquad E_C = [\mathcal{E}\ \mathcal{F}]$$

Both cases will call STREAM MERGE with different sets. Assuming the case where $p \leqslant 1$, in the next run FINDMIN will return A and stream S will advance as:

$$S = A^2; \mathcal{E}; C; A$$

with vectors:

$$V_A = [] \qquad E_A = []$$
$$V_B = [q] \qquad E_B = [\mathcal{F}]$$
$$V_C = [p] \qquad E_C = [\mathcal{F}]$$

As $V_A = \varnothing$, A is removed. In the next iteration, FINDMIN return both B and C as incomparable. In the first case, the condition $q \leqslant p$ is generated and the stream advances as follows:

$$S = A^2; \mathcal{E}; C; A; (B \| C)^q$$

with vectors:

$$V_B = [] \qquad E_B = []$$
$$V_C = [p - q] \qquad E_C = [\mathcal{F}]$$

In the opposite case we get the condition $p \leqslant q$ and the streams:

$$S = A^2; \mathcal{E}; C; A; (B \| C)^p$$

with vectors:

$$V_B = [q - p] \qquad E_B = [\mathcal{F}]$$
$$V_C = [] \qquad E_C = []$$

Finally, in both cases the last actor with the rest of his firing is added and we get:

$$S = A^2; \mathcal{E}; C; A; (B \| C)^q; B^{q-p} \qquad p \leqslant q \wedge p \leqslant 1$$
$$S = A^2; \mathcal{E}; C; A; (B \| C)^p; B^{p-q} \qquad q \leqslant p \wedge p \leqslant 1$$

The algorithm proceeds in a similar manner for the case $1 \leqslant p$.

A.2 VECTOR ALGEBRA

Consider vector $\vec{b} = [x_1, x_2, \ldots, x_n]$ with size $|\vec{b}| = n$. We define the up-sampling operator $\uparrow$ as follows:

$$\vec{b} \uparrow \alpha = [\underbrace{x_1, \ldots, x_1}_{\alpha}, \ldots, \underbrace{x_n, \ldots, x_n}_{\alpha}]$$

Hence, the size of the up-sampled vector is:

$$|\vec{b} \uparrow \alpha| = |\vec{b}| \cdot \alpha$$

Moreover, we define the vector sum $\overrightarrow{\sum}$, the sum of all the values within a vector:

$$\overrightarrow{\sum} \vec{b} = \sum_{i=1}^{|\vec{b}|} x_i$$

Naturally, as the up-sampling operator duplicates the values of the vector we get:

$$\overrightarrow{\sum}(\vec{b} \uparrow \alpha) = \alpha \cdot \overrightarrow{\sum} \vec{b}$$

Consider a function $f(x_1, x_2, \ldots, x_m)$ with m arguments. We define mapping function $\mathcal{M}$, a function that gets m vectors of size n and calls function f, n times as follows:

$$\begin{aligned} \mathcal{M}\ f\ \left[\vec{b_1}, \vec{b_2}, \ldots, \vec{b_m}\right] = \Big[& f(b_1[1], b_2[1], \ldots, b_m[1]), \\ & f(b_1[2], b_2[2], \ldots, b_m[2]), \\ & \ldots, f(b_1[n], b_2[n], \ldots, b_m[n])\Big] \end{aligned}$$

Property 3. *The mapping of m vectors $(\vec{b_1}, \vec{b_2}, \ldots, \vec{b_m})$ of size n, up-sampled by α, on a function f equals to the mapping of the vectors on f upsampled by α.*

$$\mathcal{M}\ f\ \left[\vec{b_1} \uparrow \alpha, \vec{b_2} \uparrow \alpha, \ldots, \vec{b_m} \uparrow \alpha\right] = \left(\mathcal{M}\ f\ \left[\vec{b_1}, \vec{b_2}, \ldots, \vec{b_m}\right]\right) \uparrow \alpha \tag{63}$$

Proof. To prove Property 3, we separately calculate the left hand side and the right hand side of Eq. (63). For the *lhs* we get:

$$\begin{aligned} & \mathcal{M}\ f\ \left[\vec{b_1} \uparrow \alpha, \vec{b_2} \uparrow \alpha, \ldots, \vec{b_m} \uparrow \alpha\right] \\ = \ & \mathcal{M}\ f\ \Big[\underbrace{b_1[1], \ldots, b_1[1]}_{\alpha}, \underbrace{b_1[2], \ldots, b_1[2]}_{\alpha}, \ldots, \underbrace{b_1[n], \ldots, b_1[n]}_{\alpha}, \\ & \quad \ldots, \underbrace{b_m[1], \ldots, b_m[1]}_{\alpha}, \underbrace{b_m[2], \ldots, b_m[2]}_{\alpha}, \ldots, \underbrace{b_m[n], \ldots, b_m[n]}_{\alpha}\Big] \\ = \ & \Big[\underbrace{f\left(b_1[1], b_2[1], \ldots b_m[1]\right), \ldots, f\left(b_1[1], b_2[1], \ldots b_m[1]\right)}_{\alpha} \\ & \quad \underbrace{f\left(b_1[2], b_2[2], \ldots b_m[2]\right), \ldots, f\left(b_1[2], b_2[2], \ldots b_m[2]\right)}_{\alpha}, \\ & \quad \ldots, \underbrace{f\left(b_1[n], b_2[n], \ldots b_m[n]\right), \ldots, f\left(b_1[n], b_2[n], \ldots b_m[n]\right)}_{\alpha}\Big] \end{aligned} \tag{64}$$

Then for the *rhs* we get:

$$
\begin{aligned}
&\left(\mathcal{M}\ \ f\ \ \left[\vec{b_1}, \vec{b_2}, \ldots, \vec{b_m}\right]\right) \uparrow \alpha \\
= \ &\Big[f\left(b_1[1], b_2[1], \ldots, b_m[1]\right), \\
&\ f\left(b_1[2], b_2[2], \ldots, b_m[2]\right), \\
&\ldots, f\left(b_1[n], b_2[n], \ldots, b_m[n]\right)\Big] \uparrow \alpha \\
= \ &\Big[\underbrace{f(b_1[1], b_2[1], \ldots b_m[1]), \ldots, f(b_1[1], b_2[1], \ldots b_m[1])}_{\alpha}, \\
&\ \underbrace{f(b_1[2], b_2[2], \ldots b_m[2]), \ldots, f(b_1[2], b_2[2], \ldots b_m[2])}_{\alpha}, \\
&\ldots, \underbrace{f(b_1[n], b_2[n], \ldots b_m[n]), \ldots, f(b_1[n], b_2[n], \ldots b_m[n])}_{\alpha}\Big]
\end{aligned}
\tag{65}
$$

From Eq. (64) and Eq. (65), we see that the sides of Eq. (63) are equal. □

As the two vectors are equal their vector sums are also equal therefore:

$$
\overrightarrow{\sum}\left(\mathcal{M}\ \ f\ \ \left[\vec{b_1} \uparrow \alpha, \vec{b_2} \uparrow \alpha, \ldots, \vec{b_m} \uparrow \alpha\right]\right) = \overrightarrow{\sum}\left(\mathcal{M}\ \ f\ \ \left[\vec{b_1}, \vec{b_2}, \ldots, \vec{b_m}\right]\right) \uparrow \alpha \tag{66}
$$

Property 4. *A vector upsampled by a product equals to the consecutive upsampling of the vector by each product factor:*

$$
\vec{x} \uparrow (a_1 \cdot a_2 \cdot \ldots \cdot a_n) = (\ldots((\vec{x} \uparrow a_1) \uparrow a_2) \ldots \uparrow a_n) \tag{67}
$$

Proof. By induction. Considering vector $\vec{x} = [x_1, x_2, \cdots x_m]$, we first show that the property holds for $n = 2$. Then, assuming that the property holds for $n - 1$, we prove that it holds for n. Hemce, for $n = 2$ we get:

$$
(\vec{x} \uparrow a_1) \uparrow a_2 = \vec{x} \uparrow (a_1 \cdot a_2) \tag{68}
$$

Proof for $n = 2$:

$$
\begin{aligned}
\vec{x} \uparrow a_1 = \ &\Big[\underbrace{x_1, x_1, \ldots, x_1}_{a_1}, \underbrace{x_2, x_2, \ldots, x_2}_{a_1}, \ldots, \underbrace{x_m, x_m, \ldots, x_m}_{a_1}\Big] \\
(\vec{x} \uparrow a_1) \uparrow a_2 = \ &\Big[\underbrace{\underbrace{x_1, x_1, \ldots, x_1}_{a_1}, \underbrace{x_1, x_1, \ldots, x_1}_{a_1}, \ldots, \underbrace{x_1, x_1, \ldots, x_1}_{a_1}, \ldots,}_{a_2} \\
&\ \underbrace{\underbrace{x_m, x_m, \ldots, x_m}_{a_1}, \underbrace{x_m, x_m, \ldots, x_m}_{a_1}, \ldots, \underbrace{x_m, x_m, \ldots, x_m}_{a_1}}_{a_2}\Big] \\
= \ &\Big[\underbrace{x_1, x_1, \ldots, x_1}_{a_1 \cdot a_2}\ \underbrace{x_2, x_2, \ldots, x_2}_{a_1 \cdot a_2} \ldots, \underbrace{x_m, x_m, \ldots, x_m}_{a_1 \cdot a_2}\Big] \\
= \ &\vec{x} \uparrow (a_1 \cdot a_2)
\end{aligned}
$$

Now, assuming that Eq. (67) holds for $n-1$, that is

$$\vec{x} \uparrow (a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1}) = (\ldots((\vec{x} \uparrow a_1) \uparrow a_2) \ldots \uparrow a_{n-1}) \tag{69}$$

we show that it holds for n. Starting from the *rhs*:

$$\begin{aligned}
& (\ldots((\vec{x} \uparrow a_1) \uparrow a_2) \ldots \uparrow a_n) \\
= \; & (\ldots(((\vec{x} \uparrow a_1) \uparrow a_2) \ldots \uparrow a_{n-1}) \uparrow a_n) \\
(\text{by Eq. (69)}) \; = \; & (\vec{x} \uparrow (a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1})) \uparrow a_n \\
= \; & \Big[\underbrace{\underbrace{x_1, x_1, \ldots, x_1}_{a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1}}, \underbrace{x_1, x_1, \ldots, x_1}_{a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1}}, \ldots, \underbrace{x_1, x_1, \ldots, x_1}_{a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1}}}_{a_n}, \ldots, \\
& \quad \underbrace{\underbrace{x_m, x_m, \ldots, x_m}_{a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1}}, \underbrace{x_m, x_m, \ldots, x_m}_{a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1}}, \ldots, \underbrace{x_m, x_m, \ldots, x_m}_{a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1}}}_{a_n} \Big] \\
= \; & \Big[\underbrace{x_1, x_1, \ldots, x_1}_{a_1 \cdot a_2 \cdot \ldots \cdot a_n}, \underbrace{x_2, x_2, \ldots, x_2}_{a_1 \cdot a_2 \cdot \ldots \cdot a_n}, \ldots, \underbrace{x_m, x_m, \ldots, x_m}_{a_1 \cdot a_2 \cdot \ldots \cdot a_n} \Big] \\
= \; & \vec{x} \uparrow (a_1 \cdot a_2 \cdot \ldots \cdot a_n)
\end{aligned} \tag{70}$$

which proves Property 4. □

Finally, as the two vectors are equal, their vector sums are also equal:

$$\overrightarrow{\sum} \left(\vec{x} \uparrow (a_1 \cdot a_2 \cdot \cdots \cdot a_n) \right) \; = \; \overrightarrow{\sum} \left((\ldots((\vec{x} \uparrow a_1) \uparrow a_2) \ldots \uparrow a_n) \right) \tag{71}$$

BIBLIOGRAPHY

[1] Ismail Assayad, Alain Girault, and Hamoudi Kalla. Tradeoff exploration between reliability, power consumption, and execution time. In *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 437–451, Naples, Italy, 2011. Springer-Verlag. ISBN 978-3-642-24269-4. URL http://dl.acm.org/citation.cfm?id=2041619.2041662.

[2] Neal Bambha. Intermediate representations for design automation of multiprocessor DSP systems. In *Design Automation for Embedded Systems*, pages 307–323. Kluwer Academic Publishers, 2002.

[3] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of ACM*, 36(1):98–111, 1993.

[4] Massimo Bariani, Paolo Lambruschini, and Marco Raggio. VC-1 decoder on STMicroelectronics P2012 architecture. In *Proceedings of 8th Annual Interantional Workshop 'STreaming Day'*, September 2010. doi: http://stday2010.uniud.it/stday2010/stday_2010.html.

[5] Ed Baroth and Chris Hartsough. Visual object-oriented programming. chapter Visual Programming in the Real World, pages 21–42. Manning Publications Co., Greenwich, CT, USA, 1995. ISBN 0-13-172397-9. URL http://dl.acm.org/citation.cfm?id=213388.213393.

[6] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. ISBN 0792397223.

[7] Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–10, Sept 2013. doi: 10.1109/EMSOFT.2013.6658581.

[8] Vagelis Bebelis, Pascal Fradet, and Alain Girault. A framework to schedule parametric dataflow applications on many-core platforms. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, pages 125–134, Edinburgh, United Kingdom, 2014. ACM. ISBN 978-1-4503-2877-7. doi: 10.1145/2597809.2597819. URL http://doi.acm.org/10.1145/2597809.2597819.

[9] Albert Benveniste, Patricia Bournai, Thierry Gautier, Michel Le Borgne, Paul Le Guernic, and Hervé Marchand. The SIGNAL declarative synchronous language: controller synthesis and systems/architecture design. In *Proceedings of the 40th IEEE Conference on Decision and Control*, volume 4, pages 3284–3289 vol.4, 2001. doi: 10.1109/.2001.980328.

[10] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 84–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0668-2. URL http://dl.acm.org/citation.cfm?id=827261.828219.

[11] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001. ISSN 1053-587X. doi: 10.1109/78.950795. URL http://dx.doi.org/10.1109/78.950795.

[12] Shuvra S. Bhattacharyya and Edward A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing Systems*, 6(3):271–288, December 1993. ISSN 0922-5773. doi: 10.1007/BF01608539. URL http://dx.doi.org/10.1007/BF01608539.

[13] Shuvra S. Bhattacharyya and Edward A. Lee. Looped schedules for dataflow descriptions of multirate signal processing algorithms. *Formal Methods System Design*, 5(3):183–205, December 1994. ISSN 0925-9856. doi: 10.1007/BF01383830. URL http://dx.doi.org/10.1007/BF01383830.

[14] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Design Automation for Embedded Systems*, 2(1):33–60, 1997. ISSN 0929-5585. doi: 10.1023/A:1008806425898. URL http://dx.doi.org/10.1023/A%3A1008806425898.

[15] Greet Bilsen, Marc Engels, Rudy Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 3255–3258 vol.5, May 1995. doi: 10.1109/ICASSP.1995.479579.

[16] Bruno Bodin, Alix Munier Kordon, and Benoît Dupont de Dinechin. K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 152–159, 2012.

[17] Cristina Boeres, Jose Viterbo Filho, and Vinod E.F. Rebello. A cluster-based strategy for scheduling task on heterogeneous processors. In *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 214–221, October 2004. doi: 10.1109/SBAC-PAD.2004.1.

[18] Nicholas S. Bowen, Christos N. Nikolaou, and Arif Ghafoor. On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems. *IEEE Transactions on Computers*, 41(3):257–273, March 1992. ISSN 0018-9340. doi: 10.1109/12.127439. URL http://dx.doi.org/10.1109/12.127439.

[19] G.W. Brams. *Réseaux de Petri: Théorie et pratique*. Number v.1 in Réseaux de Petri. Masson, 1983.

[20] D. Brière, D. Ribot, D. Pilaud, and J.-L. Camus. Methods and specifications tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, UK, December 1994. ERA Technology.

[21] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley, 1993. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1993/2429.html.

[22] Joseph T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 508–513 vol.1, October 1994. doi: 10.1109/ACSSC.1994.471505.

[23] Joseph T. Buck and Edward A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 1, pages 429–432 vol.1, April 1993. doi: 10.1109/ICASSP.1993.319147.

[24] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 178–188, Munich, West Germany, 1987. ACM. ISBN 0-89791-215-2. doi: 10.1145/41625.41641. URL http://doi.acm.org/10.1145/41625.41641.

[25] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Lucid synchrone: un langage pour la programmation des systèmes réactifs. In *Systèmes temps réel*. Lavoisier, 2007.

[26] Edward G. Coffman. *Computer and Job Shop Scheduling Theory*. Wiley, New York, 1976.

[27] Thomas M. Conte, Pradeep K. Dubey, Matthew D. Jennings, Ruby B. Lee, Alex Peleg, Salliah Rathnam, Mike Schlansker, Peter Song, and Andrew Wolfe. Challenges to combining general-purpose and multimedia processors. *Computer*, 30(12):33–37, December 1997. ISSN 0018-9162. doi: 10.1109/2.642799. URL http://dx.doi.org/10.1109/2.642799.

[28] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, 4.1 edition, April 2012.

[29] Nathalie Cossement, Rudy Lauwereins, and Francky Catthoor. DF*: An extension of synchronous dataflow with data - dependency and non-determinism. In *Forum on Design Languages*, September 2000.

[30] Khann Huu The Dam. Scheduling of parametric dataflow applications on many-core systems. Master's thesis, Université Joseph Fourier, June 2013.

[31] Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. Modeling static-order schedules in synchronous dataflow graphs. In *Proceedings of the Conference on Design, Automation Test in Europe (DATE)*, pages 775–780, 2012.

[32] Morteza Damavandpeyma, Sander Stuijk, Marc Geilen, Twan Basten, and Henk Corporaal. Parametric throughput analysis of scenario-aware dataflow graphs. In *IEEE International Conference on Computer Design (ICCD)*, pages 219–226, 2012.

[33] Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. Throughput-constrained DVFS for scenario-aware dataflow graphs. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 175–184, 2013.

[34] Ali Dasdan and Rajesh K. Gupta. Faster maximum and minimum mean cycle algorithms for system- performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, October 1998. ISSN 0278-0070. doi: 10.1109/43.728912.

[35] Jack B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag. ISBN 3-540-06859-7. URL http://dl.acm.org/citation.cfm?id=647323.721501.

[36] Jörg Desel and Javier Esparza. *Free Choice Petri Nets*. Cambridge University Press, New York, NY, USA, 1995. ISBN 0-521-46519-2.

[37] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S. Bhattacharyya, and Slaheddine Aridhi. PiMM: parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 41–48, Agios Konstantinos, Samos Island, Greece, July 2013. doi: 10.1109/SAMOS.2013.6621104. URL http://dx.doi.org/10.1109/SAMOS.2013.6621104.

[38] Muhammad K. Dhodhi, Imtiaz Ahmad, Anwar Yatama, and Ishfaq Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *Journal of Parallel Distributed Computing*, 62(9):1338–1361, September 2002. ISSN 0743-7315. doi: 10.1006/jpdc.2002.1850. URL http://dx.doi.org/10.1006/jpdc.2002.1850.

[39] Keith Diefendorff and Pradeep K. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, September 1997. ISSN 0018-9162. doi: 10.1109/2.612247. URL http://dx.doi.org/10.1109/2.612247.

[40] Hesham El-Rewini and Ted G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal on Parallel Distributed Computing*, 9(2):138–153, June 1990. ISSN 0743-7315. doi: 10.1016/0743-7315(90)90042-N. URL http://dx.doi.org/10.1016/0743-7315(90)90042-N.

[41] Joachim Falk, Joachim Keinert, Christian Haubelt, Jürgen Teich, and Shuvra S. Bhattacharyya. A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT)*, pages 189–198, Atlanta, GA, USA, 2008. ACM. ISBN 978-1-60558-468-3. doi: 10.1145/1450058.1450084. URL http://doi.acm.org/10.1145/1450058.1450084.

[42] Robert E. Filman and Daniel P. Friedman. *Coordinated Computing: Tools and Techniques for Distributed Software*. McGraw-Hill, Inc., New York, NY, USA, 1984. ISBN 0-07-022439-0.

[43] Pascal Fradet, Alain Girault, and Peter Poplavkoy. SPDF: A schedulable parametric data-flow MoC. In *Proceedings of the Conference on Design, Automation Test in Europe (DATE)*, pages 769–774, 2012.

[44] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

[45] Marc Geilen. Reduction techniques for synchronous dataflow graphs. In *Design Automation Conference*, pages 911–916, 2009.

[46] Marc Geilen. Synchronous dataflow scenarios. *ACM Transactions on Embedded Computing Systems*, 10(2):16:1–16:31, January 2011. ISSN 1539-9087. doi: 10.1145/1880050.1880052. URL http://doi.acm.org/10.1145/1880050.1880052.

[47] Marc Geilen and Twan Basten. Requirements on the execution of kahn process networks. In *Proceedings of the 12th European Symposium on Programming, ESOP*, pages 319–334. Springer Verlag, 2003.

[48] Amir Hossein Ghamarian, Marc Geilen, Twan Basten, Bart D. Theelen, Mohammad Reza Mousavi, and Sander Stuijk. Liveness and boundedness of synchronous data flow graphs. In *FMCAD*, pages 68–75, 2006.

[49] Amir Hossein Ghamarian, Marc Geilen, Sander Stuijk, Twan Basten, Bart D. Theelen, Mohammad Reza Mousavi, A.J.M. Moonen, and Marco Bekooij. Throughput analysis of synchronous data flow graphs. In *ACSD*, pages 25–36, 2006.

[50] Amir Hossein Ghamarian, Sander Stuijk, Twan Basten, Marc Geilen, and Bart D. Theelen. Latency minimization for synchronous data flow graphs. In *Euromicro DSD*, pages 189–196, 2007.

[51] Amir Hossein Ghamarian, Marc Geilen, Twan Basten, and Sander Stuijk. Parametric throughput analysis of synchronous data flow graphs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 116–121, Munich, Germany, 2008. ACM. ISBN 978-3-9810801-3-1. doi: 10.1145/1403375.1403407. URL http://doi.acm.org/10.1145/1403375.1403407.

[52] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 14(1), 2009.

[53] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[54] E. Girczyc. Loop winding - a data flow approach to functional pipelining. In *International Symposium of Circuits and Systems*, 1987.

[55] Ramaswamy Govindarajan, Guang R. Gao, and Palash Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing Systems*, 31(3):207–229, July 2002. ISSN 0922-5773. doi: 10.1023/A:1015452903532. URL http://dx.doi.org/10.1023/A:1015452903532.

[56] Philippe Grosse, Yves Durand, and Paul Feautrier. Methods for power optimization in SoC-based data flow systems. *ACM Transactions on Design Automation of Electronic Systems*, 14(3), 2009.

[57] Nan Guan, Zonghua Gu, Wang Yi, and Ge Yu. Improving scalability of model-checking for minimizing buffer requirements of synchronous dataflow graphs. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 715–720, Yokohama, Japan, 2009. IEEE Press. ISBN 978-1-4244-2748-2. URL http://dl.acm.org/citation.cfm?id=1509633.1509795.

[58] Soonhoi Ha and Edward A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, 40(11):1225–1238, November 1991. ISSN 0018-9340. doi: 10.1109/12.102826.

[59] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. PeaCE: A hardware-software codesign environment for multimedia embedded systems. *ACM Transactions Design Automation of Electronic Systems*, 12(3):24:1–24:25, May 2008. ISSN 1084-4309. doi: 10.1145/1255456.1255461. URL http://doi.acm.org/10.1145/1255456.1255461.

[60] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., Boston, MA, USA, 1997. ISBN 0-534-94968-1.

[61] Anatol W. Holt, H. Saint, R. Shapiro, and Stephen Warshall. Final report on the information systems theory project. Technical Report RADC-TR-68-305, Griffiss Air Force Base, New York, 1968.

[62] Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the*

2005 Workshop on Software and Compilers for Embedded Systems, SCOPES '05, pages 37–49, Dallas, Texas, 2005. ACM. ISBN 1-59593-207-0. doi: 10.1145/1140389.1140394. URL http://doi.acm.org/10.1145/1140389.1140394.

[63] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin. Scheduling for functional pipelining and loop winding. In *Proceedings of the 28th ACM/IEEE Design Automation Conference (DAC)*, pages 764–769, San Francisco, California, USA, 1991. ACM. ISBN 0-89791-395-7. doi: 10.1145/127601.127766. URL http://doi.acm.org/10.1145/127601.127766.

[64] Michael A. Iverson, Füsun Özgüner, and Gregory J. Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *4th Heterogeneous Computing Workshop (HCW)*, pages 93–100, 1995.

[65] Gary W. Johnson. *LabVIEW Graphical Programming*. McGraw-Hill, Inc., New York, NY, USA, 4th edition, 2006. ISBN 0071451463.

[66] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004. ISSN 0360-0300. doi: 10.1145/1013208.1013209. URL http://doi.acm.org/10.1145/1013208.1013209.

[67] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[68] Hari Kalva and Jae-Beom Lee. The VC-1 video coding standard. *IEEE MultiMedia*, 14(4):88–91, October 2007. ISSN 1070-986X. doi: 10.1109/MMUL.2007.86. URL http://dx.doi.org/10.1109/MMUL.2007.86.

[69] Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309 – 311, 1978. ISSN 0012-365X. doi: http://dx.doi.org/10.1016/0012-365X(78)90011-0. URL http://www.sciencedirect.com/science/article/pii/0012365X78900110.

[70] Hojin Kee, Chung-Ching Shen, Shuvra S. Bhattacharyya, Ian C. Wong, Yong Rao, and Jacob Kornerup. Mapping parameterized cyclo-static dataflow graphs onto configurable hardware. *Signal Processing Systems*, 66(3):285–301, 2012.

[71] Sang Cheol Kim and Sunggu Lee. Push-pull: guided search DAG scheduling for heterogeneous clusters. In *International Conference on Parallel Processing (ICPP)*, pages 603–610, June 2005. doi: 10.1109/ICPP.2005.66.

[72] Dong-Ik Ko and Shuvra S. Bhattacharyya. Dynamic configuration of dataflow graph topology for DSP system design. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 69–72, 2005.

[73] Ming-Yung Ko, Claudiu Zissulescu, Sebastian Puthenpurayil, Shuvra S. Bhattacharyya, Bart Kienhuis, and Ed F. Deprettere. Parameterized looped schedules for compact representation of execution sequences in dsp hardware and software implementation. *IEEE Transactions on*

Signal Processing, 55(6):3126–3138, June 2007. ISSN 1053-587X. doi: 10.1109/TSP.2007.893964.

[74] Fanxin Kong, Wang Yi, and Qingxu Deng. Energy-efficient scheduling of real-time tasks on cluster-based multicores. In *Proceedings of the Conference on Design Automation Test in Europe (DATE)*, pages 1135–1140, 2011.

[75] Edward A. Lee. Recurrences, iteration, and conditionals in statically scheduled block diagrams languages. In *VLSI Signal Processing III*, chapter 31, pages 330–340. IEEE Press, 1988.

[76] Edward A. Lee. Embedded software. In *Advances in Computers*, volume 56, London, 2002. Academic Press.

[77] Edward A. Lee and Soonhoi Ha. Scheduling strategies for multiprocessor real-time DSP. In *IEEE Global Telecommunications Conference and Exhibition (GLOBECOM)*, pages 1279–1283 vol.2, November 1989. doi: 10.1109/GLOCOM.1989.64160.

[78] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. ISSN 0018-9219. doi: 10.1109/PROC.1987.13876.

[79] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, January 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.5009446. URL http://dx.doi.org/10.1109/TC.1987.5009446.

[80] Jae-Beom Lee and Hari Kalva. *The VC-1 and H.264 Video Compression Standards for Broadband Video Services*. Springer, 2008.

[81] Wan Yeon Lee. Energy-saving DVFS scheduling of multiple periodic real-time tasks on multi-core processors. In *Distributed Simulation and Real-Time Applications (DS-RT)*, pages 216–223, 2009.

[82] G. LeGoff. Using synchronous languages for interlocking. In *International Conference on Computer Application in Transportation Systems*, 1996.

[83] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[84] Junyang Lu and Yao Guo. Energy-aware fixed-priority multi-core scheduling for real-time systems. In *Real-Time Computing Systems and Applications (RTCSA)*, pages 277–281, 2011.

[85] S. Ritz M. Pankert, O. Mauss and Heinrich Meyr. Dynamic data flow and control flow in high level DSP code synthesis. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages II.449–II.452, Adelaide, South Australia, April 1994. IEEE.

[86] Olivier Marchetti and Alix Munier-Kordon. A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research*, 197(2):532 – 540, 2009. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/j.ejor.2008.07.037. URL http://www.sciencedirect.com/science/article/pii/S0377221708005900.

[87] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, pages 1137–1142, San Francisco, California, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228568. URL http://doi.acm.org/10.1145/2228360.2228568.

[88] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, August 2002. ISSN 1053-587X. doi: 10.1109/TSP.2002.800830.

[89] Praveen K. Murthy, Shuvra S. Bhattacharyya, and Edward A. Lee. Combined code and data minimization for synchronous dataflow programs. Memorandum UCB/ERL M94/93, University of California at Berkeley, Electronics Research Laboratory, November 1994. URL http://ptolemy.eecs.berkeley.edu/papers/jointCodeDataMinimize/.

[90] Hyunok Oh and Soonhoi Ha. Fractional rate dataflow model and efficient code synthesis for multimedia applications. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 12–17, Berlin, Germany, 2002. ACM. ISBN 1-58113-527-0. doi: 10.1145/513829.513834. URL http://doi.acm.org/10.1145/513829.513834.

[91] Keshab K. Parhi and David G. Messerschmitt. Rate-optimal fully-static multiprocessor scheduling of data-flow signal processing programs. In *IEEE International Symposium on Circuits and Systems*, pages 1923–1928 vol.3, May 1989. doi: 10.1109/ISCAS.1989.100746.

[92] Chanik Park, Jaewoong Chung, and Soonhoi Ha. Extended synchronous dataflow for efficient DSP system prototyping. In *Proceedings of IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 196–201, July 1999. doi: 10.1109/IWRSP.1999.779053.

[93] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California, Berkeley, 1995. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/2926.html.

[94] Thomas M. Parks, José Luis Pino, and Edward A. Lee. A comparison of synchronous and cycle-static dataflow. In *Conference Record of the Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 204–210 vol.1, October 1995. doi: 10.1109/ACSSC.1995.540541.

[95] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. ISBN 0136619835.

[96] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[97] Jonathan Piat, Shuvra S. Bhattacharyya, and Mickaël Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS)*, pages 145–150, Tampere, Finland, October 2009. doi: 10.1109/SIPS.2009.5336240. URL http://dx.doi.org/10.1109/SIPS.2009.5336240.

[98] José Luis Pino, Shuvra S. Bhattacharyya, and Edward A. Lee. A hierarchical multiprocessor scheduling system for DSP applications. In *Conference Record of the Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 122–126 vol.1, October 1995. doi: 10.1109/ACSSC.1995.540525.

[99] William Plishker, Nimish Sane, Mary Kiemb, Kapil Anand, and Shuvra S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of IEEE International Workshop on Rapid System Prototyping (RPS)*, pages 17–23, 2008.

[100] William Plishker, Nimish Sane, and Shuvra S. Bhattacharyya. A generalized scheduling approach for dynamic dataflow applications. In *Proceedings of the Conference on Design, Automation Test in Europe (DATE)*, pages 111–116, April 2009. doi: 10.1109/DATE.2009.5090642.

[101] William Plishker, Nimish Sane, Mary Kiemb, and Shuvra S. Bhattacharyya. Heterogeneous design in functional DIF. *Transactions on HiPEAC*, 4:391–408, 2011.

[102] Claudius Ptolemaeus. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL http://ptolemy.org/books/Systems.

[103] Samantha Ranaweera and Dharma P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *Proceedings of 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 445–450, Cancun, Mexico, May 2000. doi: 10.1109/IPDPS.2000.846020.

[104] Raymond Reiter. Scheduling parallel computations. *Journal of ACM*, 15 (4):590–599, October 1968. ISSN 0004-5411. doi: 10.1145/321479.321485. URL http://doi.acm.org/10.1145/321479.321485.

[105] Sebastian Ritz, Matthias Pankert, and Heinrich Meyr. High level software synthesis for signal processing systems. In *Proceedings of the International Conference on Application Specific Array Processors*, pages 679–693, August 1992. doi: 10.1109/ASAP.1992.218536.

[106] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 3–13, Dallas, Texas, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3. URL http://dl.acm.org/citation.cfm?id=290940.290946.

[107] Yves Robert and Frederic Vivien. *Introduction to Scheduling*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009. ISBN 1420072730, 9781420072730.

[108] Euiseong Seo, Jinkyu Jeong, Seon-Yeong Park, and Joonwon Lee. Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Transactions on Parallel Distributed Systems*, 19(11):1540–1552, 2008.

[109] Chung-Ching Shen, Shenpei Wu, Nimish Sane, Hsiang-Huang Wu, William Plishker, and Shuvra S. Bhattacharyya. Design and synthesis for multimedia systems using the targeted dataflow interchange format. *IEEE Transactions on Multimedia*, 14(3-1):630–640, 2012.

[110] Gilbert C. Sih and Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel Distributed Systems*, 4(2):175–187, 1993.

[111] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 1st edition, 2000. ISBN 0824793188.

[112] John E. Stone, David Gohara, and Guochun Shi. OpenCL: a parallel programming standard for heterogeneous computing systems. *IEEE Design & Test*, 12(3):66–73, May 2010. ISSN 0740-7475. doi: 10.1109/MCSE.2010.69. URL http://dx.doi.org/10.1109/MCSE.2010.69.

[113] Sander Stuijk, Marc Geilen, and Twan Basten. SDF^3: SDF for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD)*, pages 276–278, Turku, Finland, June 2006. doi: 10.1109/ACSD.2006.23. URL http://doi.ieeecomputersociety.org/10.1109/ACSD.2006.23.

[114] Sander Stuijk, Marc Geilen, and Twan Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, October 2008. ISSN 0018-9340. doi: 10.1109/TC.2008.58. URL http://dx.doi.org/10.1109/TC.2008.58.

[115] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, December 2012. ISSN 1051-8215. doi: 10.1109/TCSVT.2012.2221191.

[116] Bart D. Theelen, Marc Geilen, Twan Basten, Jeroen Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *MEMOCODE*, pages 185–194, 2006.

[117] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: a language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK,

UK, 2002. Springer-Verlag. ISBN 3-540-43369-4. URL http://dl.acm.org/citation.cfm?id=647478.727935.

[118] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions Parallel Distributed Systems*, 13(3):260–274, 2002.

[119] Marko Viitanen, Jarno Vanne, Timo D. Hamalainen, Moncef Gabbouj, and Jani Lainema. Complexity analysis of next-generation HEVC decoder. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 882–885, May 2012. doi: 10.1109/ISCAS.2012.6272182.

[120] William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-729650-6.

[121] Piet Wauters, Marc Engels, Rudy Lauwereins, and J.A. Peperstraete. Cyclo-dynamic dataflow. In *Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing (PDP)*, pages 319–326, January 1996. doi: 10.1109/EMPDP.1996.500603.

[122] Maarten Wiggers, Marco Bekooij, and Gerard J. M. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 183–194, 2008.

[123] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software code generation for the RVC-CAL language. *Journal of Signal Processing Systems*, 63(2):203–213, May 2011. ISSN 1939-8018. doi: 10.1007/s11265-009-0390-z. URL http://dx.doi.org/10.1007/s11265-009-0390-z.

[124] Shenpei Wu, Chung-Ching Shen, Nimish Sane, Kelly Davis, and Shuvra S. Bhattacharyya. Parameterized scheduling for signal processing systems using topological patterns. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1561–1564, March 2012. doi: 10.1109/ICASSP.2012.6288190.

[125] Xiaodong Wu, Yuan Lin, Jian-Jun Han, and Jean-Luc Gaudiot. Energy-efficient scheduling of real-time periodic tasks in multicore systems. In *Network and Parallel Computing (NPC)*, pages 344–357, 2010.