A model of programming languages for dynamic real-time streaming applications

Xuan Khanh Do

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Xuan Khanh DO

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Modèle de calcul et d'exécution pour des applications flots de donnés dynamiques avec contraintes temps réel (Thesis topic : A model of programming languages for dynamic real-time streaming applications)

soutenue le 17 octobre 2016

devant le jury composé de :

M. Marco Mattavelli	Rapporteur
M. Marco Bekooij	Rapporteur
Mme. Alix Munier	Examinateur
M. Frédéric Blanc	Examinateur
M. Stéphane Louise	Encadrant
M. Albert Cohen	Directeur de thèse

Acknowledgements

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 Thierry Collette (head of CEA LIST/DACLE department), Renaud Sirdey (ex-head of LaSTRE laboratory), and Raphaël David (head of LCE laboratory), for warmly hosting me in the DACLE department, the LaSTRE and LCE laboratories, in addition for providing me all the resources for the work.

My thesis advisors, Albert Cohen and Stéphane Louise have been decisive factors throughout the PhD. They seemed to complement each other making my journey much easier. Albert for his insight, for providing a clean perspective of the work and for his support when things were looking grim. He has the ability to always challenge a statement which makes every bit of progress a slow but verified step. Stéphane, for his thorough knowledge that leads to many excellent ideas and a lot of discussions, sometimes on the verge of disagreement, but always constructive enough to push the work further. His motivation and dynamism for this work has affected me all along this successful journey. I thank you for your daily professional and personal advices. I really enjoyed working with you.

These 3 years would never pass so fast and so smooth without the great support of my colleagues. I thank you all, one by one, for the great time we share during and after the work. In particular, I would like to thank Paul, Thierry, Philippe, Paul-Antoine, Thanh Hai, who helped me to overcome a lot of technical problems related to my work, and also for our great discussions about my work. It is of great pleasure to continue this journey with you after the PhD.

A special thank to the department secretaries, in particular Annie, Sabrina, Marie-Isabelle and Julien, for taking care with an ultimate speed and lovely smile all my administrative issues.

Last but not least, a very special acknowledgement of love and gratitude to my family, the DO's. Thank you my parents Thanh and Dung, my sister Trang, for your continuous love and support. It was the key for my success throughout the challenging years of my education. I also thank my wife's family, the TRUONG's, for showing their ultimate care and sharing this joy for me.

Finally, there are no words that can express my feelings of gratitude to my wife Linh. Your big support, your great tolerance, your great daily encouragements, and your shining

love made this tough journey very special. I love you.

Abstract

There is an increasing interest in developing applications on homo- and heterogeneous multiprocessor platforms due to their broad availability and the appearance of many-core chips, such as the MPPA-256 chip from Kalray (256 cores) [34] or TEGRA X1 from NVIDIA (256 GPU and 8 64-bit CPU cores). Given the scale of these new massively parallel systems, programming languages based on the dataflow model of computation have strong assets in the race for productivity and scalability, meeting the requirements in terms of parallelism, functional determinism, temporal and spatial data reuse in these systems. However, new complex signal and media processing applications often display several major challenges that do not fit the classical static restrictions: 1) How to provide guaranteed services against unavoidable interferences which can affect real-time performance?, and 2) How these streaming languages which are often too static could meet the needs of emerging embedded applications, such as context- and data-dependent dynamic adaptation?

To tackle the first challenge, we propose and evaluate an analytical scheduling framework that bridges classical dataflow MoCs and real-time task models. In this framework, we introduce a new scheduling policy noted Self-Timed Periodic (STP), which is an execution model combining Self-Timed scheduling (STS), considered as the most appropriate for streaming applications modeled as data-flow graphs, with periodic scheduling: STS improves the performance metrics of the programs, while the periodic model captures the timing aspects. We evaluate the performance of our scheduling policy for a set of 10 real-life streaming applications and find that in most of the cases, our approach gives a significant improvement in latency compared to the Strictly Periodic Schedule (SPS), and competes well with STS. The experiments also show that, for more than 90% of the benchmarks, STP scheduling results in optimal throughput. Based on these results, we evaluate the latency between initiation times of any two dependent actors, and we introduce a latency-based approach for fault-tolerant stream processing modeled as a CSDF graph, addressing the problem of node or network failures.

For the second challenge, we introduce a new dynamic Model of Computation (MoC), called *Transaction Parameterized Dataflow* (TPDF), extending CSDF with parametric rates and a new type of control actor, channel and port to express dynamic changes of the graph topology and time-triggered semantics. TPDF is designed to be statically analyzable regarding the essential deadlock and boundedness properties, while avoiding the aforementioned restrictions of decidable dataflow models. Moreover, we demonstrate that TPDF can be used to accurately model task timing requirements in a great variety of situations and introduce a static scheduling heuristic to map TPDF to massively parallel embedded platforms. We validate the model and associated methods using a set of realistic applications and random graphs, demonstrating significant buffer size and performance

improvements (*e.g.*, throughput) compared to state of the art models including Cyclo-Static Dataflow (CSDF) and Scenario-Aware Dataflow (SADF).

Index terms— Many-core, parallelism, dataflow, embedded systems, dynamic applications, scheduling, time-constrained, fault-tolerant, latency, throughput

Contents

1 Introduction **3**
- 1.1 Thesis Motivation . . . 3
  - 1.1.1 From SoC to MPSoC: challenges of the embedded manycore . . . 4
  - 1.1.2 Programmability: how to leverage manycore processors? . . . 5
  - 1.1.3 New needs for emerging embedded real-time applications . . . 6
- 1.2 Problem Statement . . . 8
- 1.3 Contribution . . . 9
- 1.4 Outline . . . 11

2 Dataflow Models of Computation **13**
- 2.1 Parallel Models of Computation . . . 14
  - 2.1.1 Kahn Process Networks . . . 14
  - 2.1.2 Dataflow . . . 15
- 2.2 Cyclo-Static Dataflow . . . 15
  - 2.2.1 Formal Definition . . . 15
  - 2.2.2 Static Analyses . . . 17
  - 2.2.3 Scheduling Cyclo-Static Dataflow . . . 18
  - 2.2.4 Special Cases of CSDF Graphs . . . 20
- 2.3 Dynamic Extensions of Cyclo-Static Dataflow . . . 21
  - 2.3.1 Dynamic Topology Models . . . 21
  - 2.3.2 Dynamic Rate Models . . . 22
  - 2.3.3 Model Comparison . . . 28
- 2.4 Programming Languages based on Dataflow Models . . . 28
  - 2.4.1 StreamIt . . . 29
  - 2.4.2 ΣC . . . 30
  - 2.4.3 Transformation between ΣC and StreamIt . . . 38
- 2.5 Summary . . . 41

3 Self-Timed Periodic Scheduling **45**
- 3.1 Motivational Example . . . 47
- 3.2 System Model . . . 48
  - 3.2.1 Timed Graph . . . 48
  - 3.2.2 Graph Levels . . . 48
  - 3.2.3 System's model and Schedulability . . . 49
- 3.3 Self-Timed Periodic Scheduling . . . 49
  - 3.3.1 Assumptions and Definitions . . . 50

3.3.2 Latency Analysis under STP Schedule . . . 51
3.4 Evaluation Results . . . 54
3.4.1 Benchmarks . . . 54
3.4.2 Experiment: Latency comparison . . . 56
3.4.3 Experiment: Throughput comparison . . . 57
3.4.4 Discussion: Decision tree for real-time scheduling of CSDF applications . . . 58
3.5 Summary . . . 58

4 Latency-based approach for fault-tolerance 63
4.1 Motivational Example . . . 64
4.2 Hard-Real-Time Scheduling of CSDF . . . 65
4.3 Actor Dependence Function . . . 67
4.3.1 Definition . . . 67
4.3.2 Calculating ADF . . . 68
4.3.3 Illustrative example . . . 69
4.4 Latency Analysis . . . 70
4.4.1 Definition . . . 70
4.4.2 Latency Analysis under a hard-real-time scheduling . . . 70
4.5 Fault-Tolerance . . . 71
4.5.1 Data Model . . . 71
4.5.2 Support for fault-tolerance . . . 72
4.6 Evaluation results . . . 73
4.6.1 Benchmarks . . . 73
4.6.2 Experiment: Throughput comparison . . . 73
4.7 Summary . . . 75

5 Transaction Parameterized Dataflow 77
5.1 Model of Computation . . . 78
5.2 (max, +) Algebraic Semantics of TPDF . . . 80
5.3 Static Analyses . . . 82
5.3.1 Rate consistency . . . 82
5.3.2 Boundedness . . . 83
5.3.3 Liveness . . . 84
5.3.4 Throughput Analysis . . . 86
5.3.5 Scheduling . . . 88
5.4 Summary . . . 89

6 Real-Time Extension for TPDF 91
6.1 Time-Constrained Automata . . . 91
6.1.1 Chains . . . 92
6.1.2 Time-constrained trees . . . 93
6.1.3 Automata . . . 93
6.1.4 The visibility principle . . . 93
6.2 Systematic translation from TCA to TPDF . . . 94
6.3 Example . . . 96
6.4 Application . . . 100

6.4.1 QDS design . . . 100
6.4.2 TPDF design . . . 102
6.5 Summary . . . 102

7 Experimental Results **105**
7.1 Benchmarks . . . 105
7.1.1 Case-study on Edge Detection . . . 106
7.1.2 Case-study on Viola & Jones . . . 108
7.1.3 Case study on Satellite positioning . . . 109
7.1.4 Case-Study on Cognitive Radio . . . 111
7.1.5 Case-Study on VC-1 Decoder . . . 112
7.2 Analysis Tool . . . 113
7.3 Experimental Results . . . 115
7.4 Summary . . . 116

8 Conclusions **119**
8.1 Conclusions . . . 119
8.2 Open Problems and Future Research . . . 120
8.2.1 The STP scheduling policy . . . 120
8.2.2 The TPDF Model of Computation . . . 121
8.2.3 Compilation toolchain . . . 123

Acronyms **127**

List of Figures

1.1 Bridging dataflow MoCs and real-time task models . . . 10
1.2 Comparison of dataflow models of computation . . . 11

2.1 An example of a process network . . . 14
2.2 A CSDF graph of the MP3 application . . . 16
2.3 An inconsistent CSDF graph . . . 17
2.4 Illustration of latency path for the MP3 application . . . 19
2.5 BDF special actors . . . 21
2.6 A BDF graph . . . 22
2.7 A PSDF graph . . . 24
2.8 SADF model of an MPEG-4 decoder . . . 25
2.9 An SPDF graph with its parameter propagation network . . . 26
2.10 A simple BPDF graph . . . 27
2.11 Stream graph for the DCT application . . . 29
2.12 A simplified view of the MPPA chip architecture . . . 31
2.13 Process Network topology of a ΣC subgraph . . . 32
2.14 The iDCT agent with one input and one output . . . 33
2.15 Four principal stages of the compilation toolchain . . . 35
2.16 The DCT execution time of each agent and its standard error by number of cores . . . 36
2.17 The Motion Detector Graph . . . 36
2.18 The Motion Detection execution time and standard error of each agent . . . 37
2.19 Diagram of the Python program . . . 39
2.20 Throughput normalized for the BeamFormer and Parallel application . . . 41
2.21 Throughput normalized for the DCT application . . . 42

3.1 CSDF graph of the MP3 application . . . 47
3.2 Initial phase of schedule S_α . . . 53
3.3 Schedule S_∞ by pipelining the steady state S_α . . . 53
3.4 Illustration of latency path for the MP3 application . . . 55
3.5 Ratios of the latency under different scheduling policies . . . 58
3.6 Decision tree for real-time scheduling of CSDF applications . . . 60

4.1 Distributed stream graph . . . 64
4.2 Example of hard-real-time scheduling for distributed stream graph . . . 65
4.3 Example MP3 application . . . 69
4.4 Example of Late scheduling and ADF calculation . . . 70

4.5 Example of hard-real-time scheduling for the MP3 application 71
4.6 State Machine of the fault-tolerant procedure 72

5.1 Example of a TPDF graph with integer parameter p, control actor C and control channel e_5. 79
5.2 Boundedness of TPDF graph 85
5.3 Live TPDF graphs 86
5.4 State-space and its Finite State Machine 88
5.5 Scheduling of the TPDF graph example 89

6.1 Constrained chain 92
6.2 Chain of Figure 6.1 with relative labeling. The relative date is calculated from the previous after node. 92
6.3 Time-constrained tree 93
6.4 Unfolding of an automaton 93
6.5 Observable values of a temporal variable 94
6.6 Send of message 94
6.7 Modeling of TCA node as a TPDF graph 95
6.8 Synchronisation using control clock 96
6.9 Periodic tasks 96
6.10 Periodic tasks with period 2, relative deadline 1 and phase 1. The blue dotted line is used to cluster the region with a phase equal to 1. 97
6.11 Illustration of a periodic task with period 5 97
6.12 Two periodic tasks of period $25ms$ and $15ms$ 99
6.13 Constrained chain 100
6.14 System architecture of a QDS software 101
6.15 A TPDF model of the QDS software 104

7.1 TPDF graph of the Edge Detection application 107
7.2 Results of different Edge Detection methods 108
7.3 edge-contours detected using different thresholds 109
7.4 TPDF graph of the Viola & Jones application 110
7.5 Person detection with the Viola Jones algorithm 110
7.6 TPDF graph of the Satellite application 111
7.7 TPDF model of a OFDM demodulator 112
7.8 Minimum buffer size of a TPDF graph example 113
7.9 The VC-1 decoder captured in TPDF. Three possible modes of operation can be distinguished: *Intra only*, *Inter only*, *Intra and Inter*. 114
7.10 An XML representation of the TPDF example graph in Figure 5.1. 114
7.11 Ratios of the throughput under different degrees of pipelining 116
7.12 Average analysis times in ms for random graphs. 117

8.1 A TPDF graph with data duplication on edge 122
8.2 A TPDF multi-graph 122
8.3 TPDF code of a simple application 124
8.4 Graph representation and its XML file 125

List of Tables

2.1 Comparison table of expressiveness and analyzability of dataflow models . 28
2.2 StreamIt benchmark suite used for evaluation 39
2.3 StreamIt vs. ΣC 40

3.1 Proposed STP Schedules 49
3.2 Benchmarks used for evaluation 54
3.3 Results of Latency comparison 56
3.4 Results of Throughput comparison 59

4.1 Dependence between actor's executions in the MP3 application 69
4.2 Benchmarks used for evaluation 73
4.3 Results of Throughput comparison 74

6.1 Period parameter of the QDS and their typical values 102

7.1 Benchmarks from different sources to check the expressiveness of TPDF and its performance results 106
7.2 Throughput obtained and the improvement of TPDF 115

Résumé en français

Il y a un intérêt croissant pour le développement d'applications sur les plates-formes multiprocesseurs homo- et hétérogènes en raison de l'extension de leur champ d'application et de l'apparition des puces many-core, telles que Kalray MPPA-256 (256 cœurs) ou TEGRA X1 de NVIDIA (256 GPU et 8 cœurs 64 bits CPU). Étant donné l'ampleur de ces nouveaux systèmes massivement parallèles, la mise en oeuvre des applications sur ces plates-formes 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).

Dans ce contexte, les Modèles de Calcul (MdC) flot de données ont été développés pour faciliter la conception de ces applications. Ces MdC sont par définition composées de filtres qui échangent des flux de données via des liens de communication. Ces modèles fournissent une représentation intuitive des applications flot de données, tout en exposant le parallélisme de tâches de l'application. En outre, ils fournissent des capacités d'analyse statique pour la vivacité et l'exécution en memoire bornée. Cependant, de nouvelles applications de signalisation et de traitement des médias complexes présentent souvent plusieurs défis majeurs qui ne correspondent pas aux restrictions des modèles flot de données statiques classiques: 1) Comment fournir des services garantis contre des interférences inévitables qui peuvent affecter des performances temps réel ?, et 2) Comment ces langages flot de données qui sont souvent trop statiques pourraient répondre aux besoins des applications embarquées émergentes, qui nécessitent une exécution plus dynamique et plus dépendante du contexte ?

Pour faire face au premier défi, nous proposons un ordonnancement hybride, nommé Self-Timed Periodic (STP), qui relie des MdC flot de données classiques et des modèles de tâches temps réel. Cet ordonnancement peut aussi être considéré comme un modèle d'exécution combinant l'ordonnancement classique dirigé seulement par les contraintes de dépendance d'exécution appelé Self-Timed Scheduling (STS), évalué comme le plus approprié pour des applications modélisées sous forme de graphes flot de données, avec l'ordonnancement périodique: STS améliore les indicateurs de performance des programmes, tandis que le modèle périodique capture les aspects de synchronisation. Nous avons évalué la performance de notre ordonnancement sur un ensemble de 10 applications et nous avons constaté que dans la plupart des cas, notre approche donne une amélioration significative de la latence par rapport à un ordonnancement purement périodique ou Strictly Periodic Scheduling (SPS), et rivalise bien avec STS. Les expériences montrent également que, pour presque tous les cas de test, STP donne un débit optimal. Sur la base de ces résultats, nous avons évalué la latence entre le temps d'initiation de tous les deux acteurs dépendants, et nous avons introduit une approche basée sur la latence pour le traitement des flux à tolérance de pannes modélisée comme un graphe CSDF, dans le

but d'aborder des problèmes de défaillance de nœud ou de réseau.

Pour le deuxième défi, nous présentons un nouveau MdC flot de données, le Transaction Parameterized Dataflow (TPDF), une extension de CSDF avec des paramètres entiers et un nouveau type d'acteur, channel et port de contrôle qui permettent d'exprimer les changements dynamiques de la topologie du graphe et d'imposer des contraintes temporelles dans le modèle. Malgré l'augmentation de l'expressivité, TPDF est conçu pour rester statiquement analysable pour leur propriété de vivacité et d'exécution en mémoire bornée tout en évitant des restrictions mentionnées ci-dessus des modèles flot de données statiques. De plus, nous démontrons que TPDF peut être utilisé pour modéliser des contraintes temps réel dans une grande variété de situations, où une plate-forme d'ordonnancement peut être appliqué pour ordonnancer et partitionner des applications modélisées par TPDF sur des architectures massivement parallèles. Nous avons validé le modèle et des méthodes associées en utilisant un ensemble d'applications et de graphes aléatoires, qui démontre une réduction significative de la taille du tampon et des améliorations en terme de performance (e.g., le débit) par rapport à des modèles de pointe, y compris le Cyclo-Static Dataflow (CSDF) et le Scenario-Aware Dataflow (SADF).

Mots-clés— Many-core, parallélisme, dataflow, systèmes embarqués, applications dynamiques, ordonnancement, contraintes de temps, tolérant aux fautes, latence, débit

Chapter 1

Introduction

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."

— Lewis Carroll, *Alice in Wonderland*

Contents

1.1 Thesis Motivation 3
1.1.1 From SoC to MPSoC: challenges of the embedded manycore 4
1.1.2 Programmability: how to leverage manycore processors? 5
1.1.3 New needs for emerging embedded real-time applications 6
1.2 Problem Statement 8
1.3 Contribution 9
1.4 Outline 11

There is an increasing interest in developing applications on multiprocessor platforms due to their broad availability and the appearance of many-core chips, such as the MPPA-256 chip from Kalray (256 cores) [34] or Epiphany from Adapteva (64 cores). Given the scale of these new massively parallel systems, programming languages based on the dataflow model of computation have strong assets in the race for productivity and scalability, toward meeting the requirements in terms of parallelism, functional determinism, temporal and spatial data reuse in these systems. However, new complex signal and media processing applications often display dynamic behavior and real-time constraints that do not fit the usual static restrictions. This thesis addresses these problems by providing guaranteed services against unavoidable interferences which can affect real-time performance of dataflow applications and proposes a new model of computation, which allows topology changes and time constraints enforcement.

The remainder of this chapter is organised as follows. Section 1.1 introduces our motivation for undertaking the research work detailed in this dissertation. In Section 1.2, we state the problems addressed in this thesis. Section 1.3 discusses our contributions. Finally, the organization of this dissertation is presented in Section 1.4

1.1 Thesis Motivation

In this section, the motivation of this thesis is detailed from both the hardware and software points of views.

1.1.1 From SoC to MPSoC: challenges of the embedded manycore

Single core has been making faster for decades, by increasing clock speed, increasing cache and pipeline size and exploiting Instruction-Level Parallelism (ILP) [21]. However, this progress reached a plateau at the end of the 20th century, since the progress of electronics lead to an execution close enough to the practical limits of the Instruction Level Parallelism [58], that no substantial improvements of processor performances was achievable without being a burden with regards to the number of transistors and power consumption. Since the single core practical limits were so close, the only way left to improve performance without impacting power consumption is the use of higher level parallelism, *i.e.* multiply the number of processors on the chip in conjunction with multitask or multithread programming. This is the multicore era since the beginning of the 21th century, which opens not only new opportunities but introduce also significant challenges.

The memory wall With so much processing power, the first issue awaiting, is the problem of memory wall, initially defined as the gap between the speed of processors and the speed of memory access, has slowly morphed into a different problem: the latency gap became smaller, but with the increase in the number of cores, the need for memory bandwidth has increased. Interconnects became the other issue that needed attention: existing solutions became either too power hungry as the transistor count went up (the case of bus or ring solutions) or led to higher latency (mesh networks), as the number of cores continues to increase. Finally, the larger number of cores starts to question the emphasis on core usage efficiency and ideas relying on using some of the cores (or at least hardware threads) as helpers have popped up recently. On software side, as the number of cores increased, the cost of pessimistic, lock-based synchronization of access to shared memory got higher, prompting the search for new mechanisms.

To overcome this limitation, multi- and manycore architectures have undergone several changes. One of the most popular ways is to find a kind of hierarchical memory architecture that will permit very high and sustained bandwidths between processing elements close to one another and an important but more scarce communication means between distant elements. This kind of architecture is already met in several embedded manycores: Intel Xeon Phi, Phytium Mars and Kalray's MPPA [34] platforms. For example, the last one uses clustered architectures with several (typically 16) cores on a shared memory, and clusters being bound to a 2D Network-on-Chip (NoC) (either a mesh or a torus). The weak point of this architecture as a path to the future, is that migrations between clusters and therefore virtual clustering is hard because communications between close neighbor clusters is the same as faraway clusters. This is acknowledged by hardware designers who work on possible 3D heterogeneous stacking with an interconnection and memory layer [43].

The Instruction Level Parallelism wall Secondly, a programmer who would want to take the utmost advantage of the computing power of a chip should be able to express a parallelism that can feed at least several thousands of computing units each cycle. It requires some scaling: Programming thousands of tasks, making sure to synchronize them efficiently, feeding them with enough data to munch in a way that is manageable by human programmers is a challenge waiting ahead. Threads are not a very good means to express parallelism because they lack sufficient determinism to be manageable and permitting

debug, that is also a big challenge [75].

These arising issues that will appear with massive multicore chips will at the same time put a pressure on the programmability issues. It means that programming languages need to evolve to free the programmers from the hardware constraints, but also to permit an easy expression of efficient applications, *i.e.* using the whole chip computing power if required (and otherwise being as power efficient as possible). This problem will be discussed in the next section.

1.1.2 Programmability: how to leverage manycore processors?

Programming concepts used by usual programming models and tools (*e.g.*, OpenMP), do not really fit the architectural constraints of many-core systems. In fact, because of the memory wall, programming languages for manycore architectures should provide an efficient and intuitive path to prefetch data on on-chip memories, hence the programmer should be imposed to explicit each data element that would be required for a given computation kernel. On these a priori prefetched elements, simple data parallelism should be easy to express , but should avoid to permit referencing non-constant data elements that are not explicitly prefetched. It means that data-parallelism should not be as transparent as with current implementations of OpenMP since, with OpenMP, data sharing is mostly implicit. Nonetheless, it should be nearly as easy as OpenMP shared data to implement data parallelism, provided that execution determinism can still be achieved.

On that front, dataflow paradigms are good candidates for programing multi- and manycore systems. CUDA [31] and OpenCL [54] are two ongoing industrial efforts to bring dataflow principles to ordinary (C-like) languages. They are quite well fitted to heterogeneous computing, but in their current form, remain focused on GPU-like models of computation. Moreover, memory movements being completely explicit, it makes them very low level on the programmer side and sometimes challenging for scheduling complex set of heterogeneous tasks in the execution support of these languages (see *e.g.* [113]).

Lately, the concept of stream programming (*e.g.* StreamIt [105], Brook [25] or ΣC [52]) is making huge steps because the framework of stream programming offers to the programmer a path toward an easy and manageable way to express and describe massive parallelism. It also renders obvious the data-path and data-dependency all along the processing chain, both to the programmer and to the compiler without making it completely explicit. Therefore the compiler takes an important role: It can place and route the data migration path and automatically discard data that are not relevant any longer at a given point of processing. Stream programming can offer a deterministic execution model which permits to discriminate between sane parallel programs and unmanageable ones at compile-time (see [42]). It also offers a manageable memory and data placement with regards to the compilers.

The bases of stream programming rely on Kahn Process Networks (KPN, [65]), more precisely on their derivations, like Data Process Networks ([76]), as well as their more restrictive variants such as Synchronous Dataflow (SDF, [74]) Cyclo-Static Dataflow (CSDF, [19]). Any process can take one or several channels as inputs and the same as outputs. Input channels are read-only and output channels are write-mostly. Channels are the only communication means between processes (dataflow paradigm) and reading is blocking so any task that misses some data on any of its input channel is not permitted to pursue its execution until all channels have enough data to provide. These dataflow MoCs are attractive for the prototyping of streaming applications because they allow static analy-

ses that verify various qualitative (liveness, boundedness determinism) 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. In addition, much of the development of data flow visual programming languages in the 80s was backed by industrial sources [63] resulting in the development of visual languages, such as LabView [62] which was successfully deployed in industry, significantly reducing development time [10].

However, the appearance of new classes of applications requires increasing the expressivity of dataflow models, that greatly complicates its deployment on a parallel architecture. For example, parametric exchange of data results in parametric data dependencies, or dynamic topology changes remove and add data dependencies at run-time. Many standard implementation techniques (*e.g.*, the compilation toolchain of StramIt or ΣC) are incompatible with this behaviour and cannot be used. Furthermore, manual parallel implementations are hard to produce and can be error-prone, so a further research to improve the dataflow models in the needs of new emerging applications is necessary. However, before knowing what to improve, a survey of requirements of possible future applications can provide some insights.

1.1.3 New needs for emerging embedded real-time applications

New classes of applications are arising. For example, multimedia applications with higher definitions 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. The future of these emerging applications relies mostly on the fact that enough computing power (as stated by Gustafson's Law [55]) will be available with enough versatility, but also good power performance. Power issues advocates the use of manycores in the embedded world since more parallelism can (at least theoretically) be translated as more power efficiency.

Moreover, the structure of these new embedded applications often revolves around the notion of "streams". For example, both LTE (4G) [97] and the H265 [3] drafts were evaluated on the MPPA platform by using the stream programming model, which can be characterized by large streams of data that are being communicated between different computation nodes (often called *actors* or *filters*). However, 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.

To capture the limitations of stream programming concepts, we must focus on applications that are outside the scope of digital processing or still in infancy, then try and see what the stream programming paradigm is lacking in nowadays implementations to face the challenge of popularizing them and offering new leverage for parallelizing applications. Let us study several simple examples of such applications.

Software Radio (SW) and Cognitive Radio (CR) The Software Radio ultimate goal would be to replace all but the unavoidable analog stages of an ordinary radio as a digital platform. It means that in the ideal case, the analog amplifiers would be directly plugged into an Analog to Digital Converter (ADC) for the reception side and in a Digital to Analog Converter (DAC) for the emission side. This goal is only wishful thinking, but for narrower bands, *i.e.* without the High Frequency Modulation stage, this is realistic, if enough processing power is available. Therefore the protocol management up to the base modulation can be done entirely in software, provided that the computation can be performed in real-time (which sometimes requires astonishing performances, *e.g.* WiMax or LTE). Nonetheless, Software Radio is mostly about replacing protocol dependent Application-Specific Integrated Circuits (ASICs) and a part of the analog stage with software. This is something usual programming languages, and especially Stream Programming Languages are well fitted to.

The situation goes beyond with Cognitive Radio [59]. The most basic difference between Software Radio and Cognitive Radio is the context dependence, and the fact that configuration of a Cognitive Radio adapts itself from the knowledge of the context. This can be to switch the protocols from Mobile bands and protocols to Wifi communications when the system is stationary and a Wifi access point is available. This can be changing protocols when changing countries, or using in real-time known blank times in protocols to transmit information without interfering with official and licensed transmissions. Context and learning based adaptation is why manycore systems with parallel programming are more relevant than FPGA for such applications.

What appears as important differences between Software Radio and Cognitive Radio paradigms, is that Cognitive Radio makes Software Radio more intelligent, by being sensitive to context and past history, by recognizing it, and by adapting to it in real-time. Concretely, it requires database and real-time (depending on the application, either soft or hard) capabilities, and a configuration that can change dynamically, according to the context of execution. CR can use a mix of mode switching, hard real-time constraints and low-level and high-level data processing which make them a good application scheme for manycore systems. It can also "learn" from the past, and adapt dynamically its present state in accordance with the context and whatever solution was seen as correct in its more or less recent past.

Virtual and Augmented Reality On December 28, 2015, Google filed a new application with the Federal Communications Commission of the United States for a new version of the Google Glass, after failure of the first prototype. It is still hard to predict what will come out of this new product, nonetheless, what is sure is that Virtual Reality and Augmented Reality always important roles to take in the future, either in the medical field to help surgeons in their work (e.g. [96]), and for more mundane applications, for touring/-museum visit guidance (*e.g.* [44]), or even for walkers guidance [91] (either for disabled or valid persons). It can also be recreational (*e.g.* Google Ingress). Such systems are also available since a long time for aircraft pilots, and start to appear for driver assistance in high-end cars and trucks. These applications quite naturally require real-time adaptation to context, and often a good computational power, because, especially for augmented reality, the system must recognize the context which can be a challenge by itself, then apply as accurately as possible an added and relevant piece of information. This result may require physical simulations, database matching, *etc.*, all working together in a single

application.

Autonomous Vehicles and Advanced Cruise Control and Assistance Autonomous vehicles like Google cars [88], or their ancestors, the DARPA car challenge candidates [107]; or smart agile drones like the Quadrotors from the University of Pennsylvania [71], are raising increasing interests in the world and not only for military applications.

Such applications rely on several captors and actuators. For autonomous vehicles, the actuators are quite simple (steering, breaking, accelerating, for the most important parts, since the other parts like car lights, or motor fine control, are already automatically activated in cars). The controls are simple, but the instrumentation (captors) is plural, with usually high throughput (*e.g.* video cameras), but the relevant information (position on the road, obstacles, pedestrians, or any kind of danger) is hard to extract from the rough data provided by the captors. Often information from several sources must be combined (usually in a probabilistic way, because the conclusion from a single captor is not always reliable or sufficient by itself), with hard real-time constraints but mixed criticality (*e.g.* turning at the wrong crossroad, as long as the safety of the vehicle and others is preserved, is not the same as missing a red light, or mistaking the street from the sidewalk).

Even without thinking about fully autonomous vehicles, modern days high-end cars and trucks are provided with new types of helps and monitors for drivers: Lane detection and involuntary lane-change alerts, road signs detection and recognition, surrounding vehicle monitoring and emergency brake assistance, to cite some of them. These driver help appliances are the corner stones of autonomous vehicles, and even if their failures is less critical than for autonomous vehicles, we can think that drivers will rely on them increasingly as they become more pervasive and they become more reliable. The computation and software architecture issues for these applications are the same as for the autonomous vehicles.

As has been seen, these examples of applications are highly dependent on the context (usually of physical world), sometimes past learned strategies, and often show soft or hard real-time constraints. This context dependency requires a versatile and agile execution environment that can fit the challenge of real-time reconfiguration of the application according to the variations in the current context. This is an incremental but difficult constraint on the programmability requirements of manycore systems.

Moreover, this is not one model fits all for a given application, as some parts are better expressed as real-time tasks or communication, whereas others can be a good fit for Kahn Process Networks, and others fit more client-server or transactional models. All these aspects are necessary to harness a very high level of potential parallelism within applications so that the available processing power of multicores can be efficiently exploited. The issue is to present all these aspects in a coherent way to an embedded system programmer.

1.2 Problem Statement

Although stream programming provides a sane programming base for the manycore future, it lacks the versatility required for highly dynamic applications as seen in Section 1.1.2 and 1.1.3. The main issue of these stream languages and and its model of computation is that it is often too static to meet the needs of emerging embedded applications, such as real-time, context- and data-dependent dynamic adaptation. Moreover, its compilation toolchain is mostly meant for static instantiation (the set of tasks in the application is in

most cases fixed at compilation-time). For instance, an MPEG decoder uses dynamism to route i-frames and p-frames along different paths, but the operators on those paths that process the frames all have static rates. Financial computations require identifying events in data-dependent windows, but static rate operators process those events. A network monitor recognizes network protocols dynamically, but then identifies security violations by applying a static rate pattern matcher. For other types of applications, as it would be the case with augmented reality -see Section 1.1.3, it requires real-time interface which are not often well taken into account within stream processing. These are several elementary examples of why stream programming, while it has good properties for embedded systems is not a universal solution.

To summarize, we have to solve two major challenges of emerging complex signal and media processing applications: 1) How to provide guaranteed services against unavoidable interferences which can affect real-time performance?, and 2) How these streaming languages which are often too static could meet the needs of emerging embedded applications, such as context- and data-dependent dynamic adaptation? Let us introduce what we can contribute to solving these issues in the next section.

1.3 Contribution

To find a solution that meets the application requirements, we must extend the state of the art in many aspects, including scheduling policy for classical dataflow models (*e.g.*, CSDF) to solve the first challenge as mentioned above, dynamic expressivity and analyzability for new dynamic models for the second one. This section lists the main contributions of this thesis as follows:

Contribution 1: Proposing and evaluating a scheduling framework that bridges dataflow MoCs and real-time task models We propose an analytical scheduling framework (see [36, 37, 39]) that bridges classical dataflow models and classical real-time task models as shown in Figure 1.1. The first arrow represents decidability and expressiveness for popular dataflow MoCs: HSDF, SDF [74], CSDF [19], BDF [27], KPN [65], DDF [28] and RPN [50]. The arrows between the MoCs indicate that the MoC on the left-side is a subset of the one on the right-side. For example, SDF is a subset of CSDF. The dotted vertical line represents the borderline between decidable and undecidable models. The second arrow represents popular real-time task models and the complexity of their feasibility tests: L&L [81], GMF [12], RRT [13], NRRT [11], DRT [100], EDRT [101] and TA [45]. The arrows between the models indicate that the model on the left-side is a subset of the one on the right-side. For example, L&L is a subset of GMF. The dashed line indicates that any acyclic CSDF can be scheduled as a set of L&L tasks.

In this framework, we introduce a new scheduling policy noted Self-Timed Periodic (STP), which is an execution model combining Self-Timed scheduling (STS), considered as the most appropriate for streaming applications modeled as data-flow graphs, with periodic scheduling: STS improves the performance metrics of the programs, while the periodic model captures the timing aspects. We evaluate the performance of our scheduling policy for a set of 10 real-life streaming applications and find that in most of the cases, our approach gives a significant improvement in latency compared to the Strictly Periodic Schedule (SPS), and competes well with STS. The experiments also show that, for more than 90% of the benchmarks, STP scheduling results in optimal throughput. Based on these results, we evaluate the latency between initiation times of any two dependent

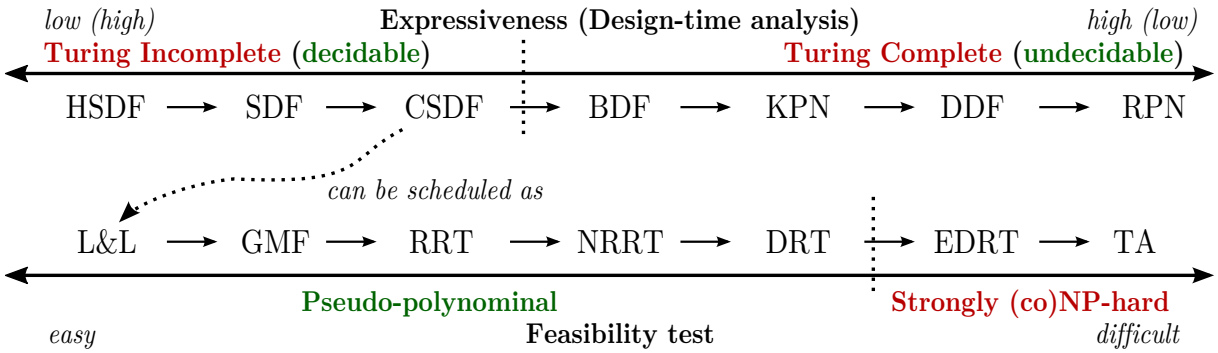


Figure 1.1: Bridging dataflow MoCs and real-time task models through the proposed scheduling framework. The link indicates that any acyclic CSDF can be scheduled as a set of L&L tasks.

actors, and we introduce a latency-based approach for fault-tolerant stream processing modeled as a CSDF graph, addressing the problem of node or network failures. We view this work as an important first step to provide a failure-handling strategy for distributed real-time streaming applications.

Contribution 2: Based on classical models, introducing a new dynamic Model of Computation, allowing topology changes and time constraints enforcement Decidable dataflow models in the SDF or CSDF family are useful for their predictability, formal abstraction, and amenability to powerful optimization techniques. However, complex signal and media processing applications, as seen in Section 1.1.3, often display dynamic behavior that do not fit the classical static restrictions; typical challenges include variable data rate processing, multi-standard or multi-mode signal processing operation, and data-dependent forms of adaptive signal processing behavior. For this reason, we introduce a new dynamic Model of Computation (MoC), called *Transaction Parameterized Dataflow* (TPDF), extending CSDF with parametric rates and a new type of control actor, channel and port to express dynamic changes of the graph topology and time constraints semantics. TPDF is designed to be statically analyzable regarding the essential deadlock and boundedness properties, while avoiding the aforementioned restrictions of decidable dataflow models. Figure 1.2 visualizes a comparison between TPDF and the most important dataflow MoCs on the three mentioned aspects of expressiveness, analyzability and implementation efficiency. It is our personal assessment based on our experiences when modeling applications, because it is difficult, if not impossible, to formalize this comparison.

We also introduce a scheduling policy to map TPDF applications to massively parallel embedded platforms. We validate the model and associated methods using a set of realistic applications and random graphs, demonstrating significant buffer size and performance improvements (*e.g.*, throughput) compared to state of the art models including Cyclo-Static Dataflow (CSDF) and Scenario-Aware Dataflow (SADF).

Moreover, several tools are also developed for this new dynamic model of computation. The first one is an analysis tool written in C++, implementing the TPDF model and its algorithms to check automatically the consistency, boundedness, liveness and analyse the worst-case throughput of dynamic reconfigurable applications. This work makes TPDF not far away from larger frameworks such as Ptolemy [94] and its extensions PeaCE [56] or Open RVC-CAL [112] as well as dataflow visual programming languages, such as Lab-

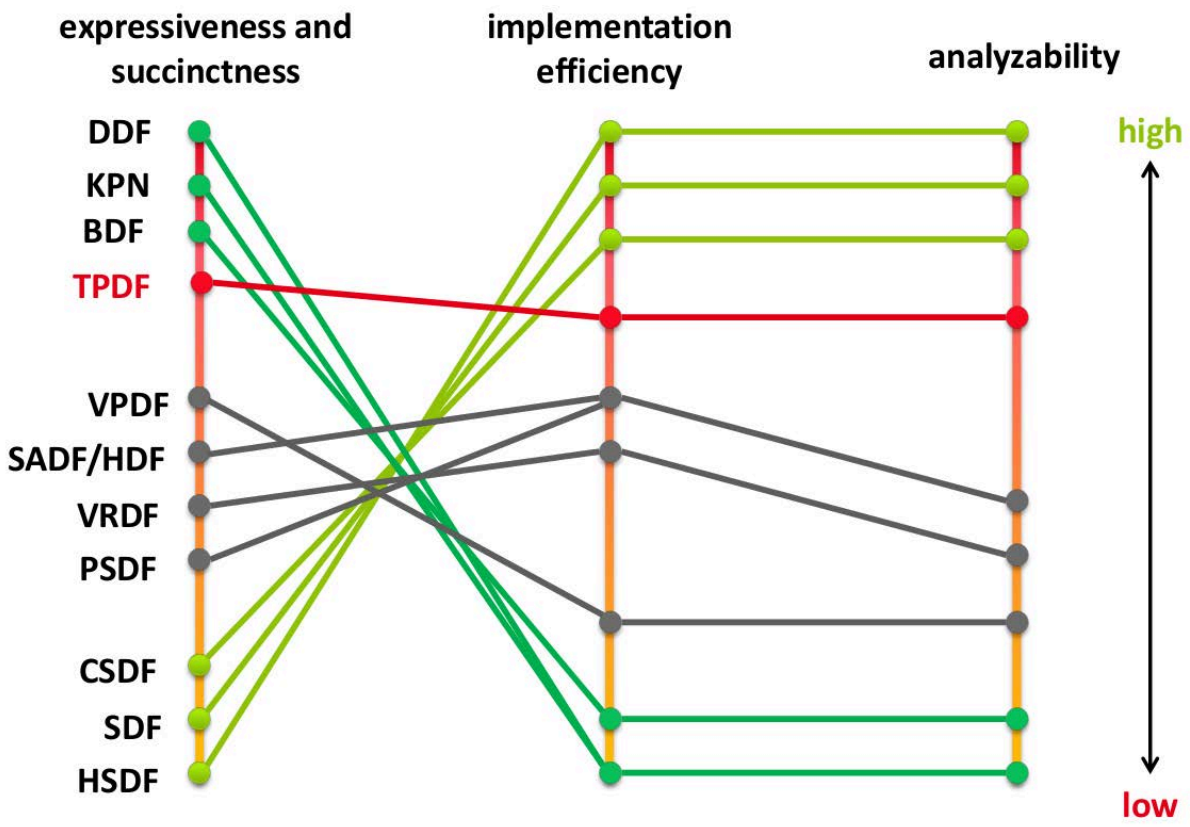


Figure 1.2: Comparison of dataflow models of computation.Our aim is to build TPDF as a Model of Computation with expressivity better than static models (*e.g.*, SDF or CSDF) and anazability betten than dynamic models (*e.g.*, BDF).

View [62] or Simulink, which was successfully deployed in industry, significantly reducing development time but lacking static guarantees as provided in TPDF. Furthermore, this analysis tool can also be used as optimization mechanism for a new compilation toolchain, developed to use the TPDF as the model of programmation to not only homogeneous architectures such as MPPA-256 but also new emerging heterogeneous architectures.

1.4 Outline

The outline of this thesis is as follows. We first further put our problem statement and approach in context by a detailed discussion of the current state-of-the-art dataflow MoCs. From this discussion, we find two challenges that must be solved by these existing dataflow models and streaming languages to meet the needs of emerging complex signal and media processing applications: 1) How to provide guaranteed services against unavoidable interferences which can affect real-time performance?, and 2) How these streaming languages which are often too static can meet the needs of emerging embedded applications, such as context- and data-dependent dynamic adaptation?

For the first challenge, we propose and evaluate in Section 3 four classes of STP schedules based on two different granularities and two types of deadline: implicit and constrained. We evaluate the proposed STP representation using a set of 10 real-life applications and show that it is capable of achieving significant improvements in term of latency

(with a maximum of 96.6%) compared to the SPS schedule and yielding the maximum achievable throughput obtained under the STS schedule for a large set of graphs. From these results, we investigate the applicability of the theory of hard-real-time scheduling for periodic tasks in the context of streaming applications. More specifically, we consider Cyclo-Static Dataflow (CSDF) graphs with variable interprocessor communication (IPC) times, and real-time constraints imposed by hardware devices or control engineers. As a result, we establish a latency constraint on the initiation times of predecessor actors on which a given actor is dependent. Based on this constraint, we show how to guarantee real-time services and reduce inconsistencies in a CSDF application by introducing a fault-tolerant procedure.

For the second challenges, we propose in Section 5 Transaction Parameterized Dataflow (TPDF) and its extension in Section 6, a new model of computation combining integer parameters—to express dynamic rates—and a new type of control actor—to allow topology changes and time constraints enforcement. We present static analyses for liveness and bounded memory usage. We also introduce a static scheduling heuristic to map TPDF to massively parallel embedded platforms. We validate the model and associated methods in Section 7 using a set of examples that illustrate how the modelling techniques and corresponding analysis can be applied in practice. After which we conclude this thesis, with summarising the approach and discussing future work.

The thesis is structured as follows: In Chapter 2, the current state-of-the-art dataflow MoCs are presented. Moreover, the current methods of their implementation on streaming languages and manycore architectures are discussed. Chapter 3 discusses the scheduling policy for classical models to meet hard-real-time requirements. An introduction to a new dataflow model is provided in Chapter 5 , while Chapter 6 extends this model of computation to capture real-time requirements. In Chapter 7 the applicability of our analysis approach is illustrated with a number of examples, while this thesis concludes in Chapter 8.

Chapter 2

Dataflow Models of Computation

Everything flows
— Heraclitus

Contents

2.1 Parallel Models of Computation 14
2.1.1 Kahn Process Networks 14
2.1.2 Dataflow 15
2.2 Cyclo-Static Dataflow 15
2.2.1 Formal Definition 15
2.2.2 Static Analyses 17
2.2.3 Scheduling Cyclo-Static Dataflow 18
2.2.4 Special Cases of CSDF Graphs 20
2.3 Dynamic Extensions of Cyclo-Static Dataflow 21
2.3.1 Dynamic Topology Models 21
2.3.2 Dynamic Rate Models 22
2.3.3 Model Comparison 28
2.4 Programming Languages based on Dataflow Models 28
2.4.1 StreamIt 29
2.4.2 ΣC 30
2.4.3 Transformation between ΣC and StreamIt 38
2.5 Summary 41

Modelling 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 modelled with Models of Computation (MoCs). These are mathematical formalisms that can be used to express systems [77].

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.

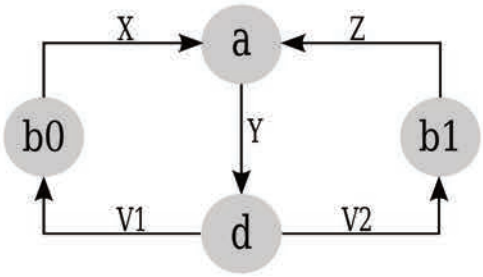


Figure 2.1: An example of a process network with processes $a, b0, b1, d$ and channels $V1, V2, X, Y, Z$. Figure reproduced from [64].

There are many Models of Computation (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, to that aim we chose to we focus on MoCs that expose parallelism such as Kahn Process Networks (KPNs) and Dataflow.

2.1 Parallel Models of Computation

2.1.1 Kahn Process Networks

Process networks, or Kahn Process Networks (KPNs), were introduced by Gilles Kahn in 1974 [64]. A KPN is composed of a set of processes, interconnected with communication links (channels) (Figure 2.1). 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 incoming communication links. A process cannot check a whether a communication link is empty or not. Once a process finishes execution, it produces atomic pieces of data, named *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 [64]. 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 [87] and extended in [49].

2.1.2 Dataflow

The Dataflow MoC first appeared in 1974, in a paper by Jack B. Dennis [35]. 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. 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. However, because Dennis' dataflow takes into account the value of the data on the links, this model is very expressive though and has limited analyzability. Hence, subsequent data flow models aimed at limiting expressiveness and increasing analyzability for properties like liveness and boundedness. Two of the most influential is Cyclo-Static Dataflow (CSDF [19]) and Synchronous Dataflow (SDF [74]), which are presented in detail in the next sections.

2.2 Cyclo-Static Dataflow

In this section, we present CSDF [19], one of the reference dataflow MoC for embedded streaming applications. CSDF was introduced in 1995, for the implementation of Digital Signal Processing (DSP) applications on parallel architectures. CSDF is well suited for DSP because of the ease of expression of such applications using the model.

2.2.1 Formal Definition

In CSDF, a program is defined as a directed graph $G = \langle A, E \rangle$, where A is a set of actors, $E \subseteq A \times A$ is a set of communication channels. Actors represent functions that transform the input data streams into output data streams. The communication channels carry streams of data and work as a FIFO queue with unbounded capacity. An atomic piece of data carried by a channel is also called a *token*. Each channel has an initial status, characterized by its *initial tokens*.

Each actor $a_j \in A$ has a cyclic *execution sequence* of length τ_j, $[f_j(0), \cdots, f_j(\tau_j - 1)]$ which can be understood as follows: The n-th time that actor a_j is fired, it executes the code of function $f_j(n \mod \tau_j)$ and produces (resp. consumes) $x_j^u(n \mod \tau_j)$ (resp. $y_j^u(n \mod \tau_j)$) tokens on its output (input) channel e_u. The firing rule of a cyclo-static actor a_j is evaluated as true for its n-th firing if and only if all input channels contain at least $y_j^u(n \mod \tau_j)$ tokens. The total number of tokens produced (resp. consumed) by actor a_j on channel e_u during the first n invocations, denoted by $X_j^u(n) = \sum_{l=0}^{n-1} x_j^u(l)$ (resp. $Y_j^u(n) = \sum_{l=0}^{n-1} y_j^u(l)$).

One of the most important properties of the CSDF model is the ability to derive at compile-time a schedule for the actors. Compile-time scheduling has been an attractive property of these dataflow models because it removes the need for a run-time scheduler. In order to derive a compile-time schedule for a CSDF graph, it must have a non-trivial repetition vector.

Definition 1 *Given a connected CSDF graph G, a* ***valid static schedule*** *for G is a schedule that can be repeated infinitely on the incoming sample stream and where the amount of data in the buffers remains bounded. A vector $\overrightarrow{q} = [q_1, q_2, ..., q_n]^T$, where*

$q_j > 0$, is a ***repetition vector*** *of G if each q_j represents the number of invocations of an actor a_j in a valid static schedule for G.*

Theorem 1 *In a CSDF graph, a repetition vector $\overrightarrow{q} = [q_1, q_2, ..., q_n]^T$ is given by [19]:*

$$\overrightarrow{q} = P \cdot \overrightarrow{r} \ , with \quad P = P_{jk} = \begin{cases} \tau_j & , if \ \ j = k \\ 0 & , otherwise \end{cases} \tag{2.1}$$

P is a square matrix where the diagonal elements are equal to the number of sub-tasks. And, $\overrightarrow{r} = [r_1, r_2, ..., r_n]^T$, where $r_i \in \mathbb{N}^\star$, is a solution of the balance equation:

$$\Gamma \cdot \overrightarrow{r} = 0, \tag{2.2}$$

and where the topology matrix Γ, which specifies the connections between edges in directed multigraph, is defined by

$$\Gamma_{uj} = \begin{cases} X_j^u(\tau_j) & , if \ task \ a_j \ produces \ on \ edge \ e_u \\ -Y_j^u(\tau_j) & , if \ task \ a_j \ consumes \ from \ edge \ e_u \\ 0 & , otherwise \end{cases} \tag{2.3}$$

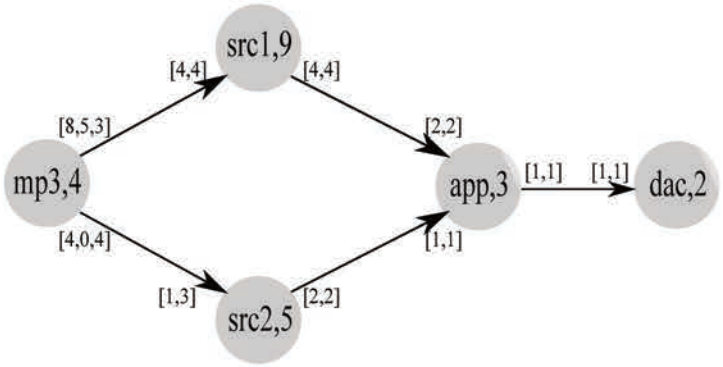


Figure 2.2: A CSDF graph of the MP3 application. Numbers between square brackets are the number of data tokens produced or consumed by the FIFO channel. Numbers next to the name of each actor are its worst-case computation and communication time.

Example 1 *For the CSDF graph shown in Figure 2.2*

$$\Gamma = \begin{bmatrix} 16 & -8 & 0 & 0 & 0 \\ 8 & 0 & -4 & 0 & 0 \\ 0 & 8 & 0 & -4 & 0 \\ 0 & 0 & 4 & -2 & 0 \\ 0 & 0 & 0 & 2 & -2 \end{bmatrix}, \overrightarrow{r} = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 4 \\ 4 \end{bmatrix}, P = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix}, and \quad \overrightarrow{q} = \begin{bmatrix} 3 \\ 4 \\ 4 \\ 8 \\ 8 \end{bmatrix}$$

This CSDF graph is characterized by two repetition vectors $\overrightarrow{r} = [1, 2, 2, 4, 4]$ and $\overrightarrow{q} = [3, 4, 4, 8, 8]$, respectively in the order $mp3, src1, src2, app, dac$. To get q_i, we multiply r_i by the length of the consumption and production rates of a_i. For example, if $r_1 = 1$

then $q_1 = 3$ because actor *mp3* contains 3 sub-tasks. The worst-case computation and communication time of each actor is shown next to its name after a comma (*e.g.*, 4 for actor *mp3*). The graph is an example of an unbalanced graph since the product of actor execution time and repetition is not the same for all actors (*e.g.*, $3 \times 4 \neq 4 \times 9$).

2.2.2 Static Analyses

A key property of CSDF is that it can be statically analyzed for consistency, boundedness and liveness. Analyses of these properties are formally presented in [19].

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 a simple example of an inconsistent graph in Figure 2.3. As shown in the Figure 2.3, for each firing of actor a_1, actor a_2 is enabled once and can produce 2 tokens on e_2. However, actor a_3 is also enabled only once, and cannot consume both tokens that a_2 produces. As a result, tokens will accumulate on edge e_2 and the graph is inconsistent. An CSDF 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.

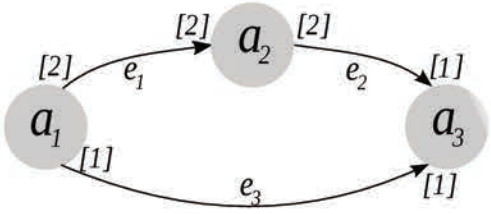


Figure 2.3: An inconsistent CSDF graph.

Definition 2 *(Consistency [73]):* A CSDF graph is *consistent* if and only if it has a non-trivial repetition vector. For a consistent graph, there is a unique smallest non-trivial repetition vector which is designated as the repetition vector of the CSDF graph.

Boundedness

A CSDF graph is bounded if it can execute in finite memory. A consistent CSDF 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 a CSDF graph for boundedness, amounts to check consistency by solving the system of balance equations. Formally:

Definition 3 *(Boundedness):* A CSDF graph is bounded iff its system of balance equations has a non-trivial solution.

Corollary 1 *(From [19]). If a consistent and live CSDF graph G completes k iterations, where $k \in N$, then the net change to the number of tokens in the buffers of G is zero.*

Liveness

A CSDF 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 4 *(Liveness):* A CSDF graph is live if and only if there exists an admissible schedule.

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.

2.2.3 Scheduling Cyclo-Static Dataflow

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).

Self-Timed Schedules (STS)

A self-timed schedule (STS), also known as an as-soon-as-possible schedule, of an CSDF graph is a schedule where each actor firing starts immediately if there are enough tokens in all its input edges. Figure 2.4(a) illustrates the STS schedule for the MP3 application shown in Figure 2.2 with each task/actor mapped in one core. This scheduling policy is considered as the most appropriate for streaming applications modeled as dataflow graphs [89, 90, 99] because it delivers the maximum achievable throughput and the minimum achievable latency if computing resources are sufficient [8]. However, this result can only be true if we ignore synchronization times. Synchronization can be considered as a special form of communication, for which data are control information. Its role is to enforce the correct sequencing of actors firing, and to ensure the mutually exclusive access to shared resources. This synchronization can be made through different methods or using a hierarchical Logical Vector Time (LVT) execution model [42], and the delay it takes should not be negligible [83].

Furthermore, STS does not provide real-time guarantees on the availability of a given result in conformance with time constraints. Due to the complex and irregular dynamics of self-timed operations, in addition to the synchronization overhead, many different hypotheses were suggested, like contention-free communication [89] or considering uniform

costs for communication operations [8, 90]. But neglecting subtle effects of synchronization is not reasonable with regards to real systems and their hard real-time guarantees. In addition, using a predefined schedule of accesses to shared memory [67] makes run-time less flexible. Therefore, analysis and optimization of self-timed systems under real-time constraints remain challenging.

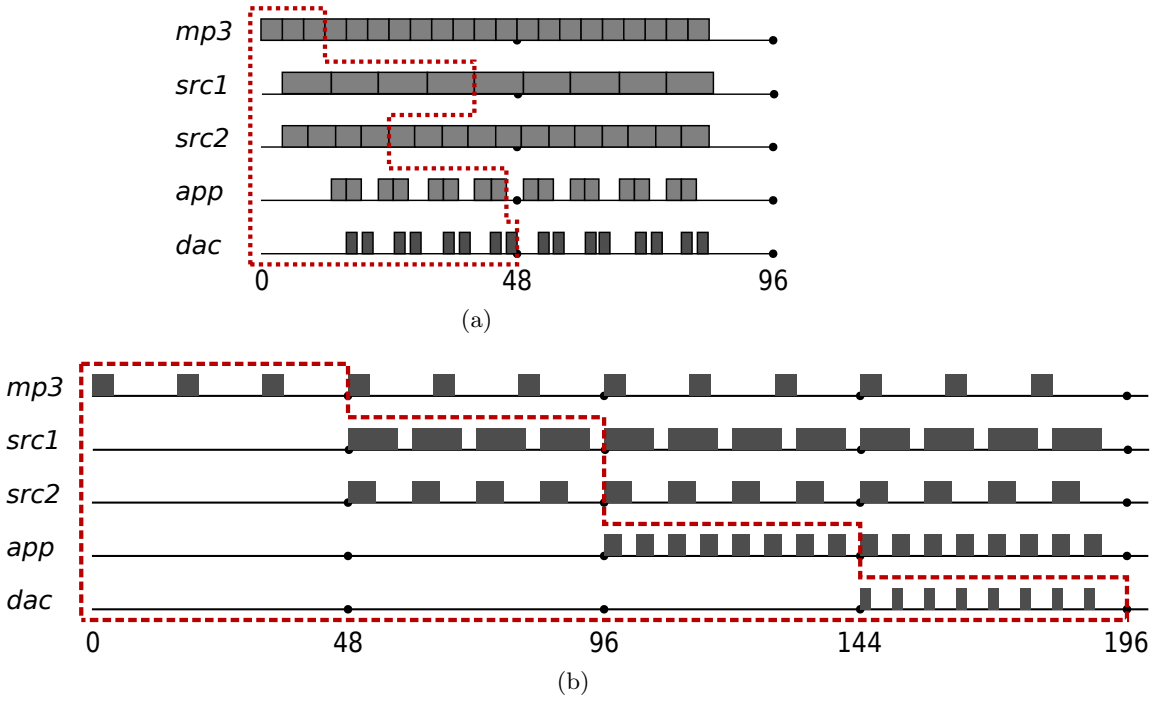


Figure 2.4: Illustration of latency path for the MP3 application shown in Figure 2.2: (a) STS (b) SPS. The dotted line represents a valid static schedule of the graph.

Strictly Periodic Schedule (SPS)

A *Strictly Periodic Schedule* [90] of a Cyclo-Static Dataflow graph is a schedule such that, $\forall a_i \in A$, $\forall k > 0$, $\forall \tau \in [0, \ldots, \tau_i - 1]$:

$$s(i,k) = s(i,0) + \phi \times k, \tag{2.4}$$

and

$$s(i,k,\tau) = s(i,k) + \lambda_i \times \tau, \tag{2.5}$$

where $s(i,k)$ represents the time at which the k-th iteration of actor a_i is fired, $s(i,k,\tau)$ represents the time at which the τ-th phase of the k-th invocation of actor a_i starts execution, λ_i is the period of actor $a_i \in A$, $\overrightarrow{\lambda} = [\lambda_1, \lambda_2, \ldots, \lambda_n]$ is the period vector of actors $a_i \in A$ and ϕ is an equal iteration period for every complete repetition of all the actors.

Theorem 2 *(Period Vector) For a Cyclo-Static consistent and acyclic Dataflow graph G, with cyclically changing consumption and production rates, it is possible to schedule*

actors of this graph as strictly periodic tasks using periods given by a solution to both [8]:

$$\phi = q_1\lambda_1 = q_2\lambda_2 = \cdots = q_{n-1}\lambda_{n-1} = q_n\lambda_n, \tag{2.6}$$

and

$$\overrightarrow{\lambda} - \overrightarrow{\omega} \geq \overrightarrow{0} \tag{2.7}$$

where $\overrightarrow{\omega} = [\omega_1, \omega_2, \ldots, \omega_n]$ is the worst-case execution time (WCET) vector of all actors $a_i \in A$ and $q_i \in \overrightarrow{q}$ is the basic repetition vector of G. It means that in every period ϕ, actor a_i is executed q_i times, for all $a_i \in A$. $\lambda_i \in \mathbb{N}^N$, represents the period measured in time-units of actor $a_i \in A$, is given by:

$$\lambda_i^{min} = \frac{Q}{q_i}\left\lceil\frac{\eta}{Q}\right\rceil \quad for\ \ a_i \in A, \tag{2.8}$$

where $\eta = \max_{a_i \in A}(\omega_i q_i)$ and $Q = lcm(q_1, q_2, \ldots, q_n)$ (lcm denotes the least common multiple operator).

Figure 2.4(b) illustrates the SPS schedule for the MP3 application shown in Figure 2.2. SPS is receiving more attention for streaming applications [8, 38, 90] with its good properties (*i.e.*, timing guarantees, temporal isolation [33] and low complexity of the schedulability test). However, periodic scheduling increases the latency significantly for a class of graphs called unbalanced graphs. A balanced graph is the one where the product of actor execution time and repetition is the same for all actors [7]. In contrast, an unbalanced graph is the one where such products differs between actors and in the real world, unbalanced graphs are the usual cases because execution times of processes can have large variations. Therefore, in this thesis, one of our mission is to find a scheduling policy that provides not only real-time guarantees but improves also the performance metric of the application (e.g., throughput and latency).

2.2.4 Special Cases of CSDF Graphs

Some restrictive classes of CSDF are worth mentioning as they are used in a variety of cases. The Synchronous Dataflow (SDF) [57, 74] graphs are graphs where all execution sequence lengths are equal to 1. Figure 2.3 gives also an example of SDF graphs where the consumption/production rates are fixed and known at compile time. Formally:

Definition 5 *(Synchronous Dataflow):* A CSDF graph is an SDF graph iff

$$\forall a_j \in A, \tau_j = 1 \tag{2.9}$$

where τ_j represents the execution sequence length of actor a_j

Moreover, another convenient class of CSDF is the Homogeneous Synchronous Dataflow (HSDF) graphs, where all execution length and all port rates equal to 1. Any CSDF and SDF graph can be converted to an equivalent HSDF graph. There are many algorithms that convert (C)SDF graphs to HSDF graphs, one widely used can be found in [19]. The main intuition behind the transformation is to replicate each actor as many times as its solution and connect the new actors according to the rates of the original (C)SDF. The resulting graph may have an exponential increase in size. However, this HSDF representation is useful because it exposes all the available task parallelism. It has been successfully used to produce parallel schedules of (C)SDF graphs and evaluate its liveness.

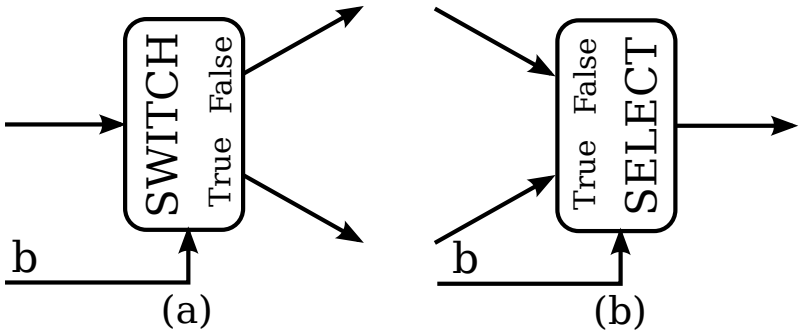


Figure 2.5: BDF special actors: (a) SWITCH actor, (b) SELECT actor. Both actors have a boolean control input that receives boolean tokens. Depending on the value of the boolean tokens, SWITCH (resp. SELECT) selects the output (resp. input) port that is activated.

2.3 Dynamic Extensions of Cyclo-Static Dataflow

This section describes the more prominent of the extensions of (C)SDF, classified in two categories: the ones that allow the graph to change topology at run-time (dynamic topology models, Section 2.3.1) and the ones that allow the amount of data exchanged between actors to change at run-time (dynamic data rate models, 2.3.2). Dynamic topology models like BDF [28] and its natural expansion IDF [26] 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 [17], VRDF [110], SADF [104], SPDF [46] and BPDF [16].

Many models presented below, allow actors to change their internal functionality at run-time. In this thesis, we are interested in dynamic changes of the graph that affect its dataflow 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 modelling of the application. In the CSDF 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 Dynamic Topology Models

In this section, we present two models that focus on altering the graph topology at run-time, Boolean Dataflow (BDF) and Integer Dataflow (IDF). Joseph T. Buck introduced BDF in his thesis [28] as an extension of SDF that provides *if-then-else* functionality. BDF uses two special actors, a SWITCH and a SELECT actor (Figure 2.5). 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

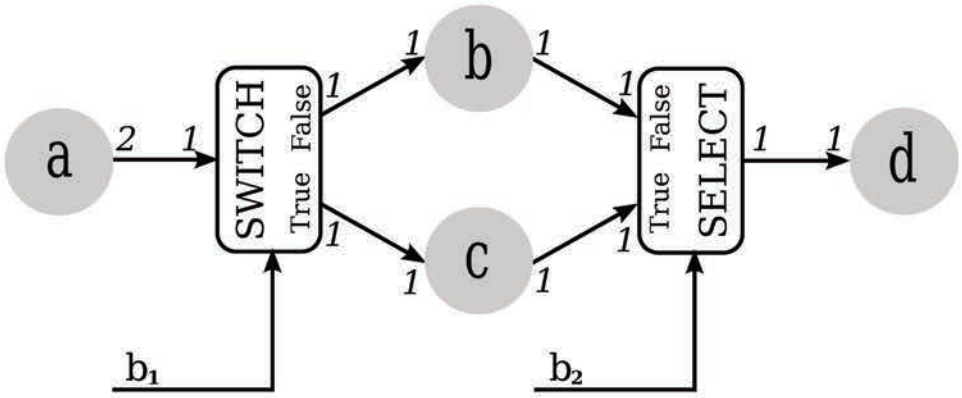


Figure 2.6: A BDF graph. Each boolean stream b_1 and b_2 has a probabilistic rate to choose between $True$ and $False$ ports.

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 $\overrightarrow{p}$. The balance equations of a BDF graph include such probabilistic values. For the graph in Figure 2.6, 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 $\overrightarrow{p}$ as *strongly consistent*. On the contrary, graphs that are consistent only for specific values of 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 streams.

BDF greatly increases SDF expressiveness. In fact, it is shown in [27] 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 [26]. 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.

2.3.2 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.

Parameterized Synchronous Dataflow

Parameterized Synchronous Dataflow (PSDF [17, 18]) allows arbitrary attributes of a dataflow graph to be parameterized, with each parameter characterized by an associated domain of admissible values that the parameter can take on at any given time. Graph attributes that can be parameterized include scalar or vector attributes of individual actors, such as the coefficients of a finite impulse response filter or the block size associated with an FFT; edge attributes, such as the delay of an edge or the data type associated with tokens that are transferred across the edge; and graph attributes, such as those related to numeric precision, which may be passed down to selected subsets of actors and edges within the given graph.

The parameterized dataflow representation of a computation involves three cooperating dataflow graphs, which are referred to as the *body* graph, the *subinit* graph, and the

init graph. The body graph typically represents the functional "core" of the overall computation, while the subinit and init graphs are dedicated to managing the parameters of the body graph. In particular, each output port of the subinit graph is associated with a body graph parameter such that data values produced at the output port are propagated as new parameter values of the associated parameter. Similarly, output ports of the init graph are associated with parameter values in the subinit and body graphs.

Changes to body graph parameters, which occur based on new parameter values computed by the init and subinit graphs, cannot occur at arbitrary points in time. Instead, once the body graph begins execution it continues uninterrupted through a graph iteration, where the specific notion of an iteration in this context can be specified by the user in an application-specific way. For example, in PSDF, the most natural, general definition for a body graph iteration would be a single SDF iteration of the body graph, as defined by the SDF repetitions vector.

However, an iteration of the body graph can also be defined as some constant number of iterations, for example, the number of iterations required to process a fixed-size block of input data samples. Furthermore, parameters that define the body graph iteration can be used to parameterize the body graph or the enclosing PSDF specification at higher levels of the model hierarchy, and in this way, the processing that is defined by a graph iteration can itself be dynamically adapted as the application executes. For example, the duration (or block length) for fixed-parameter processing may be based on the size of a related sequence of contiguous network packets, where the sequence size determines the extent of the associated graph iteration.

Body graph iterations can even be defined to correspond to individual actor invocations. This can be achieved by defining an individual actor as the body graph of a parameterized dataflow specification, or by simply defining the notion of iteration for an arbitrary body graph to correspond to the next actor firing in the graph execution. Thus, when modelling applications with parameterized dataflow, designers have significant flexibility to control the windows of execution that define the boundaries at which graph parameters can be changed.

A combination of cooperating body, init, and subinit graphs is referred to as a PSDF *specification*. PSDF specifications can be abstracted as PSDF actors in higher level PSDF graphs, and in this way, PSDF specifications can be integrated hierarchically.

In Figure 2.7 a PSDF component is shown. The component has two sets of dataflow inputs, one to connected to the *subinit* graph and on 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 *f2* is configured with two parameters g and p. g changes the functionality of *f1* while p sets its output rate. 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.

Scenario-Aware Dataflow

Scenario-Aware Data Flow (SADF) [104] is a modification of the original SDF model inspired by the concept of system scenarios. SADF introduces a special type of actors, called detectors, and enables the use of parameters as port rates. Detectors, detect the current scenario the application operates in and change the port rates of the graph ac-

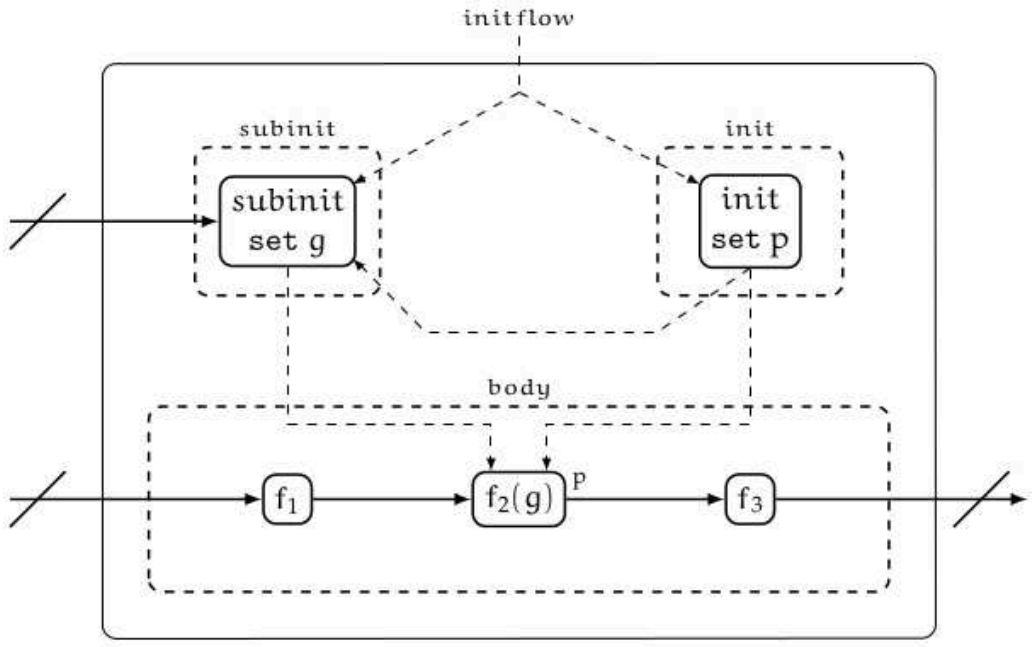


Figure 2.7: A PSDF component with two sets of dataflow inputs, one to connected to the *subinit* graph and on to the *body*. *initflow* carries parameter values from the parent component. The body has three functions *f1*, *f2* and *f3*. When the component is fired, *init* graph is fired and it sets parameter *p*. After, *subinit* fired to set the value for *g* and the rest of the graph executes as in the SDF model.

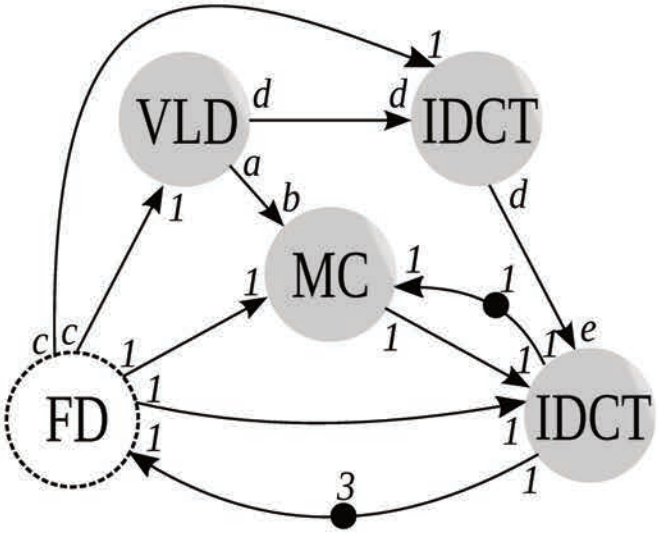
cordingly.

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 dataflow 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.

Figure 2.8 depicts the SADF model of a MPEG-4 decoder. This decoder supports video streams consisting of I and P frames. Such frames consist of a number of macro blocks, each requiring operations like Variable Length Decoding (VLD), Inverse Discrete Cosine Transformation (IDCT), Motion Compensation (MC) and Reconstruction (RC). The VLD and IDCT kernels in this model fire once per macro block that is decoded for a frame, while the MC and RC kernels fire once per frame. The Frame Detector (FD) represents the part of the actual VLD determining the frame type.

When detecting an I frame, all macro blocks must be decoded using VLD and IDCT,



	Scenario		
Rate	I	P_0	P_x
a	0	0	1
b	0	0	x
c	99	1	x
d	1	0	1
e	99	0	x

Figure 2.8: SADF model of an MPEG-4 decoder. Figure reproduced from [104].

while the resulting image is reconstructed by RC straightforwardly. Assuming an image size of 176×144 pixels (QCIF), there are 99 macro blocks to decode for an I frame, which explains the values of the parameterised rates c, d and e in this scenario. Conversely, there are no motion vectors to be taken into account for an I frame and hence, MC does not receive any motion vectors from VLD.

Decoding P frames requires MC to take motion vectors into account for retrieving the correct position of macro blocks from the previous frame. The number of motion vectors and the number of macro blocks to decode may differ for different P frames. The MPEG-4 decoder in Figure 2.8 assumes an equal number (0 or $x \in \{30, 40, 50, 60, 70, 80, 99\}$) of motion vectors and macro blocks to be decoded, each implying a different scenario. The special case of 0 motion vectors may reflect decoding still video (in which case the VLD and IDCT kernels are inactive), where MC simply copies the previously decoded frame.

As all scenarios are known at compile time, SADF is analyzed by considering 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 [104]. Weakly consistent SADF graphs may support changing of 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 Dataflow

Schedulable Parametric Data Flow (SPDF) [46] is a dataflow MoC that was developed to deal with dynamic applications where the rates of an actor may change within an iteration of the graph.

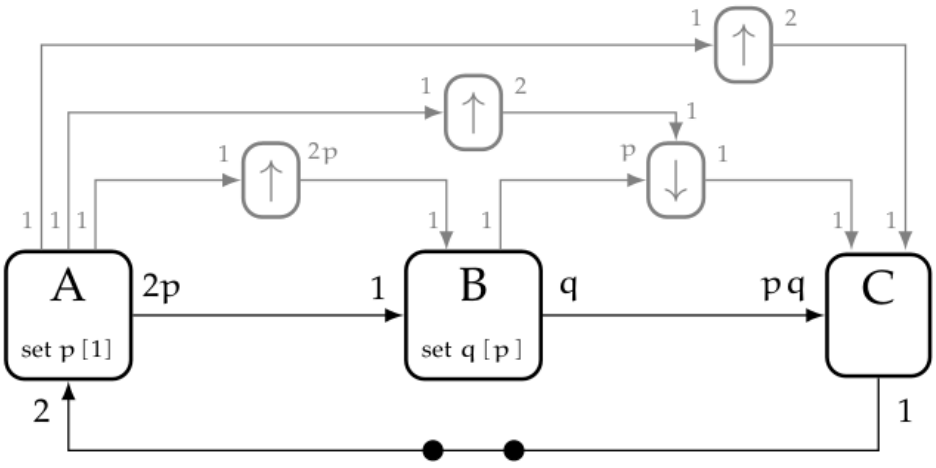


Figure 2.9: An SPDF graph with its parameter propagation network. Figure reproduced from [46].

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.

A sample SPDF graph is shown in Figure 2.9. 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 expressiv-

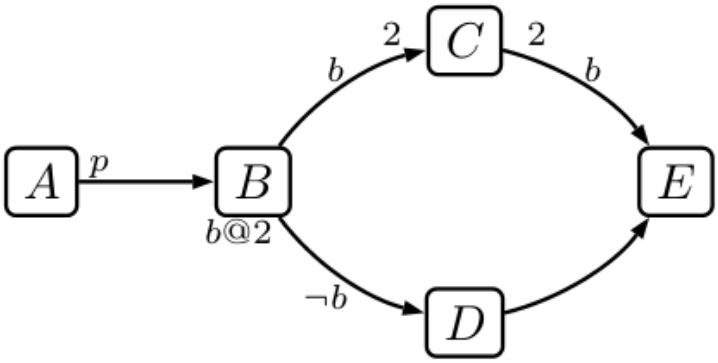


Figure 2.10: A simple BPDF graph with integer parameter p and boolean parameter b.

ity 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.

Boolean Parametric Dataflow

Boolean Parametric Data Flow (BPDF) [16] is a model which combines integer parameters (to express dynamic rates) and boolean parameters (to express the activation and deactivation of communication channels). As in other parametric dataflow models, each BPDF actor has input ports (resp. output) labeled with a production rate (resp. consumption) that can be parametric (a product of integers and symbolic parameters). Integer parameters are allowed to change at runtime, between two iterations of the BPDF graph. Moreover, each BPDF edge can be annotated with a boolean expression defined using boolean parameters that are allowed to change at runtime, even inside an iteration of the graph. When a boolean expression is false, the edge it annotates is considered disabled (absent). Therefore, the topology of the BPDF graph changes according to the values taken by the boolean parameters.

Figure 2.10 shows a simple BPDF graph where actors have constant or parametric rates (e.g., p for the output rate of A). Omitted rates and conditions equal to 1 and *true* respectively. The parametric repetition vector of this graph is $[2, 2p, p, 2p, 2p]$. The edges (B, D), (B, C) and (C, E) are conditional. They are present only when their condition (here b or $\neg b$) is true. A sample execution of the graph is the following: A fires and produces p tokens on edge (A, B). Then B fires and sets the value of boolean parameter b. If b is true, B does not produce tokens on edge (B, D). As the edge is disabled, D fires twice without consuming tokens. 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 false, C is disconnected and it will fire without producing or consuming tokens. 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).

BPDF present also static analyses which ensure statically the liveness and the boundedness of BPDF graphs. However, this is not enough because this model lacks the ability

to impose real-time constraints, a feature that is also required to program modern safety critical applications which will be both highly parallel and time constrained. Moreover, its scheduling policy targets to many-core platforms such as STHORM, which is not developed any more by STMicroelectronics.

2.3.3 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. Because CSDF allows to change its rates within its iteration, this model is a little more expressive than 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 SADF MoC provides both dynamic rates and dynamic topology, however, only in-between iterations. Moreover, this model has limitations not captured by the table, like the requirement of SADF for all scenarios to be known and analyzed at compile time. However, this requirement only work well when the number of scenarios are limited and manageable by a human.

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.

Finally, BPDF allows not only dynamic variations of production and consumption rates between iterations but also dynamic changes of the graph topology. However, like the other models, this model lacks the ability to impose real-time constraints.

Table 2.1: Comparison table of expressiveness and analyzability of dataflow models

Model	Dynamic Rates		Dynamic Topology		Static Analyses
	Between Iterations	Within Iterations	Between Iterations	Within Iterations	
SDF	○	○	○	○	●
CSDF	○	○	○	○	●
BDF	○	○	●	●	○
IDF	○	○	●	●	○
PSDF	●	●	○	○	●
SADF	●	○	●	○	●
SPDF	●	●	○	○	●
BPDF	●	○	●	●	●

2.4 Programming Languages based on Dataflow Models

Because of its good properties (e.g., static guarantees, predictability or amenability to powerful optimization techniques), static models such as SDF or CSDF is used as basis

model for numerous streaming languages, from research languages like StreamIt [105] into more production-ready offerings like ΣC [52].

2.4.1 StreamIt

In this section, we provide a very brief overview of the StreamIt language. See [2] for a more detailed description. In fact, the basis of StreamIt model of computation is synchronous data-flow (SDF). Listing 2.1 shows a snippet of StreamIt code used for transforming a 2D signal from the frequency domain to the signal domain using an inverse Discrete Cosine Transform (DCT). The Figure 2.11 shows a graphical representation of the same code.

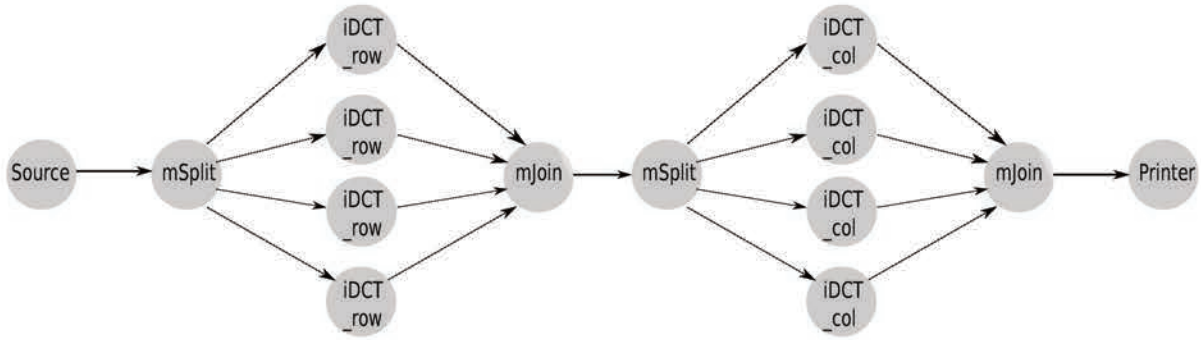


Figure 2.11: Stream graph for the DCT application

Listing 2.1: A snippet of StreamIt code demonstrating the construction of the DCT's stream graph shown in Figure 2.11. Specific keywords of the StreamIt language are underlined.

```
/* Discrete Cosine Transform */
int->int pipeline iDCT4x4_2D_fast_fine() {
    add Source(4);
    add iDCT4x4_1D_X_fast_fine();
    add iDCT4x4_1D_Y_fast_fine();
    add Printer(4);
}
/* Splitjoin filter */
int->int splitjoin iDCT4x4_1D_X_fast_fine() {
    split roundrobin(4);
    for (int i = 0; i < 4; i++) {
        add iDCT4x4_1D_row_fast();
    }
    join roundrobin(4);
}
int->int splitjoin iDCT4x4_1D_Y_fast_fine() {
    /* details elided */
}
/* Operator */
int->int filter iDCT4x4_1D_row_fast() {
    work pop 4 push 4 {
    for (int i = 0; i < 4; i++) pop();
    /* details elided */
    }
}
int->int filter iDCT4x4_1D_col_fast_fine() {
    /* details elided */
}
```

The central abstraction provided by StreamIt is an operator (called a *filter* in the StreamIt literature), which plays the same role as actor (i.e., computation unit) in CSDF

and SDF. Programmers can combine operators into fixed topologies, as shown in Figure 2.11, using three composite operators: *pipeline*, *split-join* and *feedback-loop*. Composite operators can be nested in other composites. The example code in Listing 2.1 shows four principal filters composed in a pipeline. In these filters, there are two split-join filters that connect atomic operators iDCT4x4_1D_row_fast and iDCT4x4_1D_col_fast_fine. Each operator has a *work* function that processes streaming data. To simplify the code presentation, we have elided the bodies of the work functions. When writing a work function, a programmer must specify the *pop* and *push* rates for that function. The pop rate declares how many data items from the input stream are consumed each time an operator executes. The push rate declares how many data items are produced. When all *pop* and *push* rates are known at compilation time, a StreamIt program can be statically scheduled.

2.4.2 ΣC

ΣC is a language designed based on the Cyclo-Static Dataflow model, in order to ensure programmability and efficiency on many-core platforms such as MPPA-256 from Kalray. This language is built as an extension of the C language, with buildings suitable for expression of parallelism by using stream programming concepts. Close and familiar with C, it minimizes the syntactic burden of learning a new language, while making explicit the construction of parallelism [52]. Moreover, ΣC, based on its basis model CSDF, is sufficient to express complex multimedia implementations such as H.264/MPEG-4 Part 10 or AVC (Advanced Video Coding) [52].

The ΣC Underlying Platform

The *MPPA - 256* [34] manycore architecture, from Kalray, consists of 256 user cores (i.e., cores with fully processing power provided to the programmer for computing tasks) organized as 16 (4×4) clusters tied by a Network-on-Chip (NoC) with a torus topology. Each cluster has 16 user processors connected to a shared memory. There are also 2 DMA engines (one in Rx, one out Tx) for communication with the NoC, and one special processor called *Resource manager* which makes the role of orchestra conductor and provides OS-like services. Each processing core PE_i and the RM are fitted with two-way associative instruction and data caches (i.e., each location in main memory can be cached in either of two locations in the cache). In addition to the 16 computing clusters, there are 4 I/O clusters that provide access to external DRAM memory or interfaces, etc. The shared memory of a given compute cluster is a modular memory system. The memory system consists of M memory modules among which physical addresses are distributed cyclically, so that, if i is the address of a memory location, then $j \equiv i \pmod{M} \times l$ is the address of the module containing the location (l is the size of a line, 64 bytes). For the MPPA case, the memory system contains 16 memory modules of $128KB$ so $2MB$ per cluster. Each module has a memory controller connected to each pair of user processors (i.e., via a bus). The memory is implemented as a multi-bus approach [34]: it provides the same functionality as a full crossbar with lower impact on surface occupation and power consumption. A simplified view of this chip can be seen in Figure 2.12.

The ΣC Programming Model

The ΣC programming model builds networks of connected agents. An *agent* is an autonomous entity, with its own address space and thread of control. It has an interface

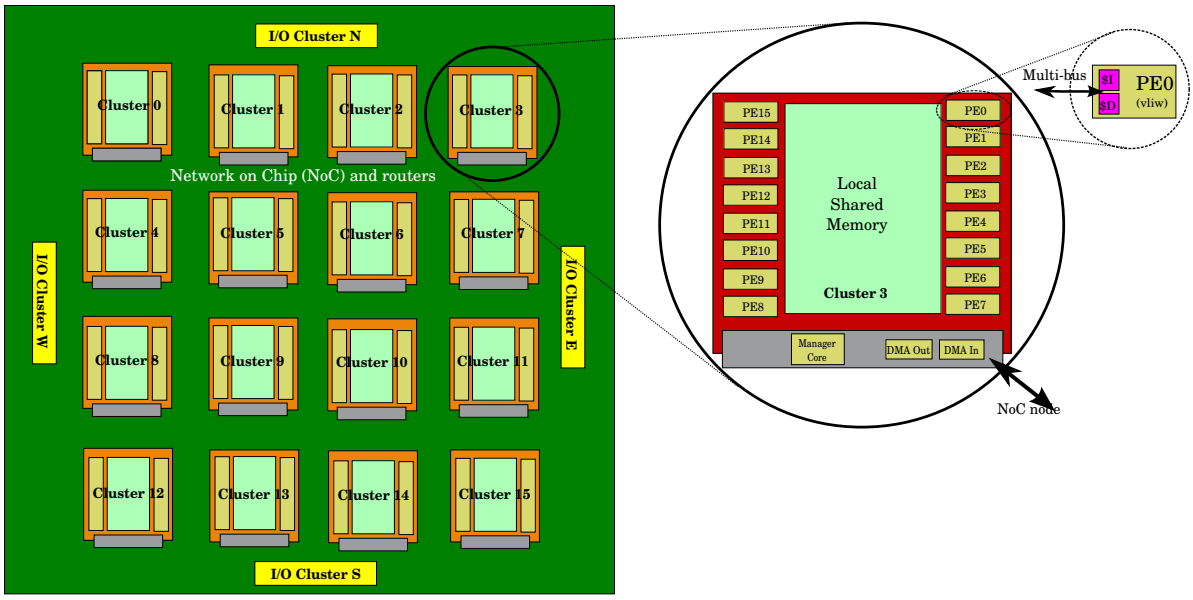


Figure 2.12: A simplified view of the MPPA chip architecture. Cluster 3 is zoomed to see the details of a cluster with its 16 processing elements (PE). Four I/O clusters ensure the communication with the outside. Clusters communicate between each other thank to a NoC.

describing a set of ports, their direction and the type of data accepted; and a behavior specification describing the behavior of the agent as a cyclic sequence of transitions with consumption and production of specified amounts of data on the ports listed in the transition.

A *subgraph* is a composition of interconnected agents and it has also an interface and a behavior specification. The contents of the subgraph are entirely hidden and all connections and communications are done with its interface. Recursive composition is possible and encouraged; an application is in fact a single subgraph named root. The directional connection of two ports creates a communication link, through which data is exchanged in a FIFO order with non-blocking write and blocking read operations (the link buffer is considered large enough). An application is a static data-flow graph, which means there is no agent creation or destruction, and no change in the topology during the execution of the application. Entity instantiation, initialization and topology building are performed offline during the compilation process. System agents ensure distribution of data and control, as well as interactions with external devices. Data distribution agents are *Split, Join* (distribute or merge data in round robin fashion over respectively their output ports / their input ports), *Dup* (duplicate input data over all output ports) and *Sink* (consume all data).

The ΣC Programming Language

The ΣC programming language is designed as an extension to C. It adds to C the ability to express and instantiate agents, links, behavior specifications, communication specifications and an API for topology building by using some new keywords like *agent, subgraph, init, map, interface,...* but does not add communication primitives. The communication ports description and the behavior specification are expressed in the *interface* section. Port declaration includes orientation and type information, and may be assigned a default value (if oriented for production) or a sliding window (if oriented for intake).

The construction of the data-flow graph is expressed in the *map* section using extended C syntax, with the possibility to use loops and conditional structures. This construction relies on instantiation of ΣC agents and subgraphs, possibly specialized by parameters passed to an instantiation operator, and on the oriented connection of their communication ports (as in Figure 2.13). All assignments to an agent state in its *map* section during the construction of the application is preserved and integrated in the final executable. Listing 2.2 and Figure 2.13 shows an example of building a new dataflow graph in ΣC.

Exchange functions implement the communicating behavior of the agent. An *exchange* function is a C function with an additional exchange keyword, followed by a list of parameter declarations enclosed by parenthesis. Each parameter declaration creates an exchange variable mapped to a communication port, usable exactly in the same way as any other function parameter. A call to an exchange function is exactly like a standard C function call, the exchange parameters being hidden to the caller. An agent behavior is implemented as in C, as an entry function named *start()*, which is able to call other functions as it sees fit, functions which may be exchange functions or not. Figure 2.14 and Listing 2.3 show an example of an agent declaration in ΣC. The main difference could be seen here is that ΣC could create CSDF graph with cyclically changing firing rules.

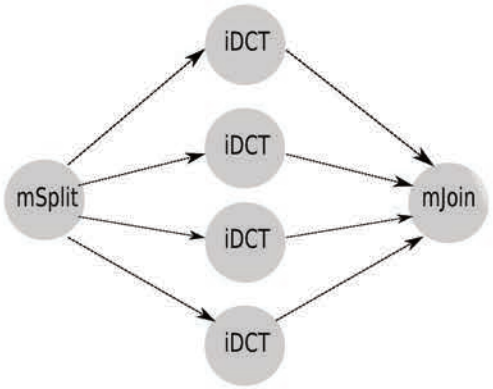


Figure 2.13: Process Network topology of a ΣC subgraph, as shown in Listing 2.2. In this topology, multiple iDCT agents (iDCT) connected to one split (mSplit) and one join (mJoin)

Listing 2.2: Topology building code of the ΣC subgraph shown in Fig. 2.13. Specific keywords of the ΣC language are underlined.

```
subgraph iDCT4x4_1D_X_fast_fine() {
        interface {
        in<int> input;
        out<int> output;
        spec{ {input[4] ; output[4] } };
        }
        map {
                int i;
        agent mSplit= new Split<int>(4,2);
        agent mJoin= new Join<int>(4,2);
        for (i = 0; i < 4; i++) {
                agent iDCT= new iDCT4x4_1D_row_fast();
                connect(mySplit.output[i], iDCT.input);
                connect(iDCT.output, myJoin.input[i]);
        }
        connect(input, mySplit.input);
        connect(myJoin.output, output);
    }
}
```

Listing 2.3: The iDCT agent's ΣC source code. Specific keywords of the ΣC language are underlined.

```
agent iDCT4x4_1D_row_fast() {
        interface {
        in<int> input;
        out<int> output;
        spec{input[2]; input[2] ; output[2] ; output[2] };
        }
    void step(int qin, int qout) exchange (input myIn[qin], output myOut[qout]) {
        /* details elided */
    }
    void start() {
        step(2, 4);
        step(2, 4);
    }
}
```

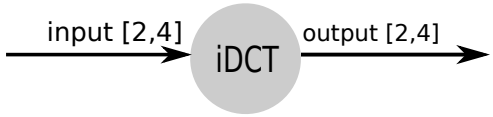


Figure 2.14: The iDCT agent used in Figure 2.13 with one input and one output, and its cyclically changing firing rules

The ΣC compilation toolchain

The compilation toolchain of ΣC includes four principal stages [5], as can be seen in Figure 2.15.

Frontend The frontend of the ΣC toolchain performs syntactic and semantic analysis of the program. It generates per compilation unit a C source file with separate declarations for the offline topology building and for the online execution of agent behavior. The declarations for the online execution of the stream application are a transformation of the ΣC code mainly to turn exchange sections into calls to a generic communication service. The communication service provides a pointer to a production (resp. intake) area, which is used in code transformation to replace the exchange variable. This leaves the management of memory for data exchange to the underlying execution support, and gives the possibility to implement a functional simulator using standard IPC on a POSIX workstation.

Instantiation and Parallelism Reduction The second compilation step of the tool chain aims at building an intuitive representation of the application relies on the dataflow paradigm, where the vertices are instances of agents and the edges are channels. This representation can be used for both compiler internal processings and developer debug interface. Once built, further analyses are applied to check that the graph is well-formed and that the resulting application fits to the targeted host. The internal representation of the application (made of C structures) is designed to ease the implementation and execution of complex graph algorithms.

Instantiating an application is made possible by compiling and running the instantiating program (*skeleton*) generated by the frontend parsing step. In this skeleton program, all the ΣC keywords are rewritten using regular ANSI C code. This code is linked against a library dedicated to the instantiation of agents and communication channels.

The parallelism reduction in the ΣC compilation chain is done in two different ways: graph pattern substitution or generic parallelism reduction based on merging agents [5].

Scheduling, Dimensioning, Placing & Routing, Runtime Generation Once the agents have been instanciated into *tasks*, the resulting dataflow application may pass the scheduling process. The whole scheduling process consists in the following steps. First, one must determine a *canonical period*, which corresponds to the execution of one cycle of the application. Basically, once all task occurrences in the canonical schedule are executed, the application must return to its initial state (list of ready tasks, amount of data present in the FIFOs). This is determined by calculating the repetition vector which is the minimum non-zero integer vector whose components correspond to the number of execution cycles of each task transition, in order to return to the initial state. During the symbolic execution, minimum buffer sizes are generated in order to determine a minimum dimensioning of the FIFOs. For this, the FIFO sizes are considered to be infinite, and we measure the maximum fill size of each FIFO during the symbolic execution. Moreover, the ΣC toolchain computes also the effective buffer sizes for the application to be executed with a certain frequency.

Once satisfying FIFO sizes have been determined, a *working period* is generated. The working period consists in the repetition of several canonical periods, ensuring the allocated buffers for the critical FIFOs may be saturated during the execution, i.e. the produced (and consumed) amount of data in the period corresponds to the allocated buffer size. Tasks are then mapped on the different clusters of the MPPA chip, and routes are determined for communication channels between tasks in different clusters.

Link edition and execution support The final stage in the ΣC compiler is the link edition. It consists in building, per cluster hosting tasks, first a relocatable object file with all the user code, user data and runtime data; then the final binary with the execution support. All this compilation stage was realized using the GNU binutils for MPPA if it targets this architecture.

ΣC applications

Some representative applications have been developed in ΣC in laboratory. We present results about the stability of the execution time of each agent measured on the *MPPA - 256* [34] clustered architecture. Figure 2.11 and 2.17 show two graphs of the DCT application (the graph created by ΣC is the same as the one generated by StreamIt because the source code is automatically translated as presented in the Section 2.4.3) and the Motion Detection application (a process of detecting a change in position of an object relative to its surroundings or the change in the surroundings relative to an object). The execution time of each agent in these streaming programs is described in Figure 2.16 and 2.18 as a function of the number of cores. With a standard error of around 10% of the mean value, these execution times are stable, in accordance with the assumptions of compilation heuristics for CSDF graphs: the execution time of each agent does not depend on the number of cores.

Comparison between StreamIt and ΣC

This section discusses similarities and also differences between ΣC and StreamIt, with the aim of identifying limitations in meeting the growing requirements of modern embedded applications.

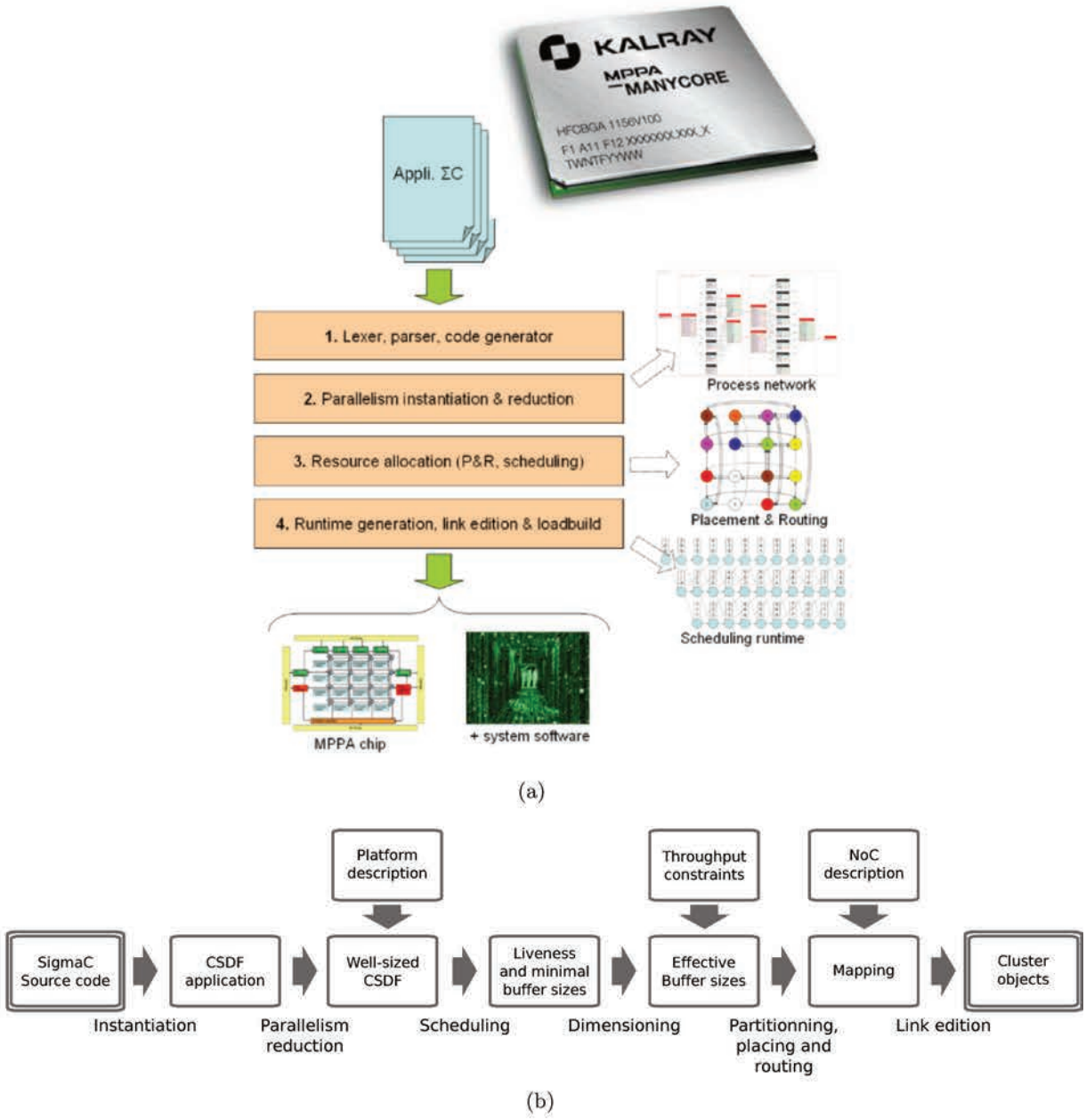


Figure 2.15: (a) Four principal stages of the toolchain and (b) its unfolding. Starting with an application written in ΣC, we obtain an executable for the MPPA architecture.

First, we can say that StreamIt and ΣC have a lot of similarities. The core of these languages are *agent* and *filter*, both are autonomous entity with its own non-shared address space with other tasks of the application. In addition, StreamIt and ΣC programs are interconnected graphs (for ΣC, it is a composition of subgraph and interconnected agents, for StreamIt it is a set of filters connected to each other). Another similarity which is also a weakness, is that both languages support only static software architecture (links between tasks, data exchange amounts are determined at compile time) while new embedded many-core applications require dynamic expressivity (for ΣC) and context-dependent adaptation (the ability to change of the synchronous data-flow graph depending on the

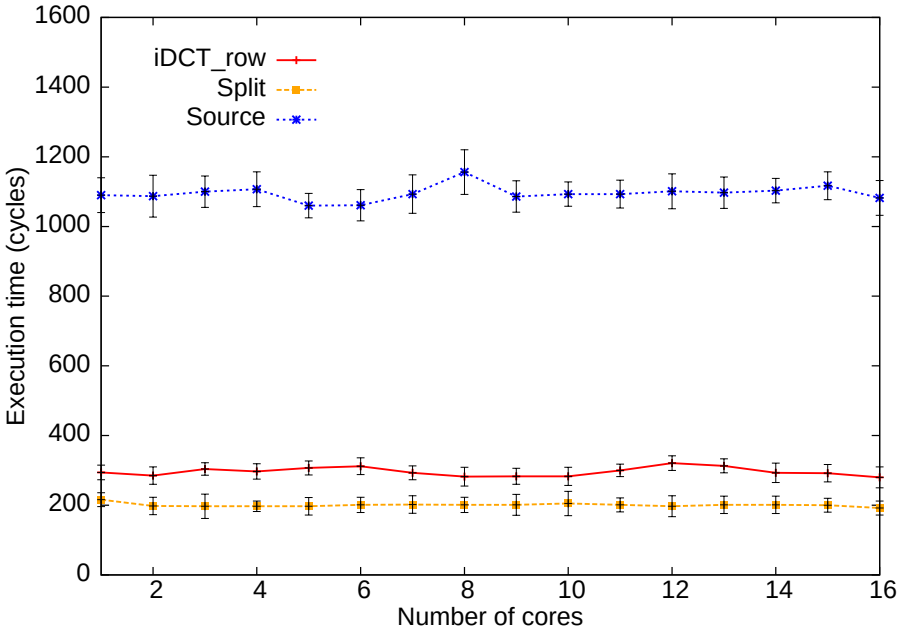


Figure 2.16: The DCT execution time of each agent and its standard error by number of cores

Figure 2.17: The Motion Detector Graph

context). For example, the Motion Detector program, as seen in Fig. 2.17, in some cases need to add some agents before the HandMadeVideoBMPReader to clean up noises if the quality of the video is too low. This requirement therefore need further research.

Other similar aspects between these languages are the real-time requirements which are not often well taken into account. The main reason is that the model of computation of these languages is *data-driven* (actors are fired as soon as there are enough tokens in all of their input edges) and not *time-driven*. Several solutions can be found in [39, 40] and will be discussed further in Section 3 and Section 6.

Besides these similarities, we can recognize a lot of differences between these two languages. While StreamIt tries to create a SDF graph of connected filters, the model of computation of ΣC is CSDF, which is also a special case of data-flow process networks. In SDF, actors have static firing rules: they consume and produce a fixed number of data tokens in each firing. This model is well suited to multirate signal processing applications and lends itself to efficient, static scheduling, avoiding the run-time scheduling overhead incurred by general implementations of process networks. In CSDF, which is a generalization of SDF, actors have cyclically changing firing rules. In some situations, the added generality of CSDF can unnecessarily complicate scheduling. Some higher-order functions can be used to transform a CSDF graph into a SDF graph, simplifying the scheduling problem [92]. To resolve this issue, a new scheduling policy noted Self-Timed Periodic (STP) Schedule, which is a hybrid execution model based on mixing Self-Timed

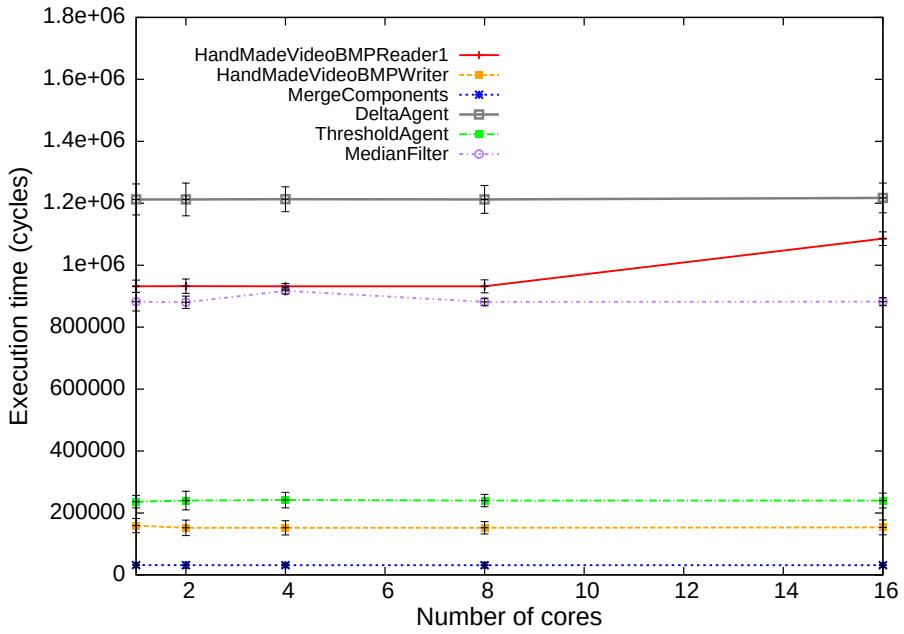


Figure 2.18: The Motion Detection execution time and standard error of each agent by number of cores, *MergeComponents, Delta, Threshold* and *MedianFilter* located in the *MotionDetector* subgraph

schedule and periodic schedule while considering variable Inter-process Communication (IPC) times, could be implemented in ΣC [37]. In other situations, CSDF has a genuine advantage over SDF: simpler precedence constraints. This makes it possible to eliminate unnecessary computations and expose additional parallelism.

Another difference difference is that networks of processes in StreamIt are defined directly through a dedicated coordination language, distinct from the Java or C implementation of StreamIt filters. This limits the topology of the associated Network (StreamIt topology is hierarchical, and is mostly limited to series-parallel graphs with the important addition of feed-back loops. Special features like teleport-messaging are required to overcome this limitation, see [105]). While in ΣC, the networks of processes are built through compilation of a single language: the first step of the compilation symbolically executes the code constructing the network of so-called "agents" (individual tasks in the stream model), and the associated communication interconnect called "subgraph". This approach has the advantage of expressing more general topologies, because it proceeds to an off-line execution of the first-stage compilation to build the process network [84]. Therefore, it is not necessary to describe the concept of *feedbackloop* in ΣC because the task graph model is more flexible than the series-parallel model of StreamIt. In other words, feedback loops are used in StreamIt only to alleviate limitations of the programming model.

As a compiler, ΣC on MPPA can be compared to the StreamIt/RAW compiler, that is the compilation of a high level, streaming oriented, source code with explicit parallelism on a manycore/RAW architecture with limited support for high-level operating system abstractions. However, the execution model supported by the target is different: dynamic tasks scheduling is allowed on MPPA; the communication topology is arbitrary and uses both a Networks on Chip (NoC) and shared memory. Moreover, the average task granu-

larity in ΣC is far larger than the typical StreamIt filter (supposed to be more than 1 μS) because the current implementation does not provide task aggregation like in StreamIt. So a task switch always pay a tribute to the execution support (see [42]). In addition, the current ΣC toolchain does not support paging when the cluster memory size is insufficient. Furthermore there is no way to make the distinction between several states within an application (init, nominal, ...). Lastly, the toolchain does not take into account some other aspects like power consumption, fault management and safety.

2.4.3 Transformation between ΣC and StreamIt

After studying StreamIt and ΣC, a method and tool was developed to convert StreamIt benchmarks in ΣC. This work aims to better understand these two languages and create a library of benchmarks for ΣC. This library allows us to use the many existing StreamIt examples as the number of ΣC applications is still insufficient and requires more programs to test the ability of language. But we have seen that there are some situations where it is undesirable to perform this transformation, as we shall see in the following example.

Rules for transforming

As shown in Section 2.4.2, there are a lot of similarities between StreamIt and ΣC. One of them is the similarity between two models of computation SDF and CSDF. Transforming between these languages is also transformation from the SDF graph of StreamIt to the CSDF graph of ΣC. This leads to a problem that is the loss of the dynamism of the CSDF model because in a cycle, a CSDF agent can have many different firing rules. In this paper, we restrict our discussion to a language transformation from a SDF model of an StreamIt application to the same ΣC model.

In StreamIt, *filter* is the atomic element of programming corresponding to the concept *agent* ΣC. In addition, *Pipeline, SplitJoin*, and *FeedbackLoop*, three constructs for composing filters into a communicating network are able to be replaced by a *subgraph* in ΣC. The first rule to transform between ΣC and StreamIt is to find a way to convert a *filter* of StreamIt in an *agent* in ΣC. To realize this rule, the first thing we have to do is to determine the behavior of the filter which is declared in the *work* part of a filter while this portion of input and output is declared in the *interface* part of an agent. The second rule is to understand how filters of a StreamIt application connect between them. Basically, StreamIt uses *pipeline* as factor to connect between filters. Likewise, ΣC programmers have to create a new instance of agent and *connect* between these instances. Likewise, *splitjoin* filter in StreamIt can be replaced by a subgraph with system agents *Split* and *Join* in ΣC.

An automatic method to transform StreamIt programs in ΣC

The method used here for the transformation between these two languages is to build controller loops that detect StreamIt filters and converts it into ΣC agents. The diagram in Figure 2.19 represents how the method and tool work. 1 and 6 are beginning and ending states, relatively. Firstly, the tool will find in the StreamIt code the main program (state 2), which is a special pipeline filter. After, other elements will be detected and handled by other controllers: normal filter (state 3), Split or Join filter (state 4) and Source or Printer filter (state 5).

The tool will read the code line by line when it encounters the declaration of the main program; it begins to create a main program that starts with ΣC subgraph *root* (the

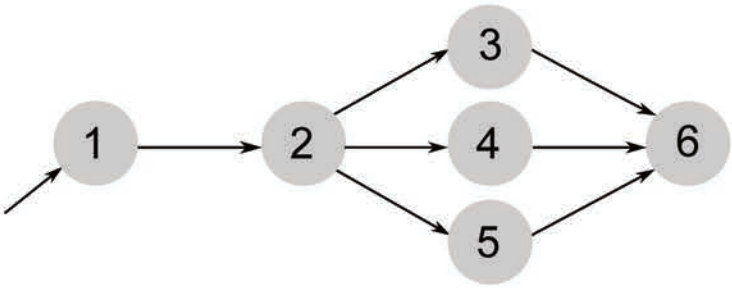


Figure 2.19: Diagram of the Python program

name of the main part in ΣC). Then it use the declarations of the StreamIt code to either prepare to chain agents for **pipeline** declarations, prepare to connect a list of agents for **split** and **dup** declarations, or reconnect the previous gathered list for **join** declarations. The tool is tested with several programs from the StreamIt benchmark suite. The ΣC generated code is compiled and have correct performance, as seen in the Section 2.4.3. However, as it is an automatic generated code, it sometimes looks like generated code, and is not as readable as hand programming. Nevertheless, the two codes play the same role and perform the same task in the program.

Evaluation

We tested the performance of the ΣC applications automatically translated from StreamIt source code by performing an experiment on some applications with several levels of complexity. The streaming applications used in the experiment are chosen among the StreamIt benchmark suite [106]. In total, 12 applications from different domains have been translated as shown in Table 2.2.

Table 2.2: StreamIt benchmark suite used for evaluation

Domain	No.	Application	Source
Audio Processing	1	Audio beam former	[106]
Signal Processing	2	Multi-Channel beamformer	
	3	Discrete cosine Transform (DCT)	
	4	Fast Fourier Transform (FFT) kernel	
	5	Low-pass filter	[1]
	6	Band-pass filter	
	7	FMRadio with equalizer	
Mathematics	8	Minimal program	
	9	Moving Average Filter	
	10	Parallel computing	[98]
	11	Multiply two matrices	[106]
Sorting	12	Bitonic Parallel Sorting	

As can be seen in the Table 2.3, the ΣC code is a little longer than the StreamIt code. This difference between two languages could be explained by the following facts: the declaration of ΣC's *interface* is more complicated but more flexible. Instead of using only one *work* function like in StreamIt, programmers can create CSDF graph while declaring ΣC's *interface*. Another reason is the number of filters of ΣC. StreamIt gives the concept

of *anonymous stream*, a special filter unnamed and is used only once in the program. As there is no concept of anonymous agent in ΣC, the tool will automatically create a new agent in the generated code, which could be reused when needed and also explain why the ΣC code is longer than the StreamIt code.

Table 2.3: StreamIt vs. ΣC

	StreamIt		ΣC	
Benchmark	**Lines**	**Filters**	**Lines**	**Filters**
Audio beam former	219	5	350	8
Multi-channel beamformer	398	9	380	9
Discrete cosine Transform (DCT)	651	19	995	21
Fast Fourier Transform (FFT) kernel	168	8	251	8
Low-pass filter	43	4	96	4
Band-pass filter	61	7	154	7
FMRadio with equalizer	167	12	335	14
Minimal program	12	3	45	3
Moving Average Filter	64	4	97	4
Multiply two matrices	163	9	290	12
Bitonic Parallel Sorting	260	10	412	10

After being translated in ΣC, some benchmarks (with different levels of grain of tasks to understand the impact of extra communication costs on parallelism's efficiency) are tested in the *MPPA - 256* [34] clustered architecture, from Kalray, comprising 256 user cores (i.e., cores with fully processing power provided to the programmer for computing tasks) organized as 16 (4×4) clusters tied by a Network-on-Chip (NoC) with a torus topology. As can be seen in Figure 2.20, the throughput normalized (in comparison with the throughput obtained in the case of mono-core) of the programs augmented when the number of cores used increased with a bottleneck after 12 cores. This result could be explained because of many reasons: there was not sufficient parallelized work to offset the extra communication costs because of the lack of coarse-grained parallelism benchmarks in StreamIt. In addition, the overhead of semi-dynamic scheduling of tasks within a cluster in MPPA is another reason for this result, because it augments as the number of cores to manage augments.

Limitations of the translation tool

A problem we encountered when translating from StreamIt to ΣC is the conversion of anonymous filters. The new agent created automatically results in difficulties when connecting between ports of agents. For example, new agents created in the *Multiply two matrices* application increased significantly the compilation time ($127s$ in comparison with $5s$ when revising the code by hand). In addition, there is some similar concepts in StreamIt and ΣC, such as *Identity* filter. However, if this concept is used automatically in ΣC source code, the application's graph becomes more complicated, resulting in a decrease in performance. This problem could be resolved by making some changes in the source code by hand. For the *Multiply two matrices* application, the number of agents declared could be reduced to 9 by removing the Identity filter used in the StreamIt source code along with improvements in throughput (with an average of 55%).

Another problem can be seen in Fig. 2.21, the throughput decreases when the num-

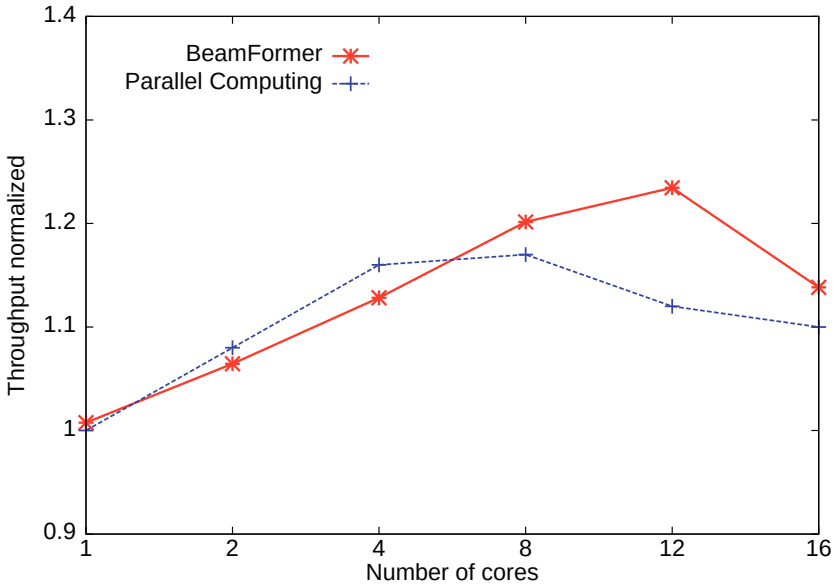


Figure 2.20: Throughput normalized for the *BeamFormer* and *Parallel Computation* application

ber of cores increases. To understand the impact of the results, we use the concept of *synchronization time* (see Section 2.2.3. In the case of this example, the execution time of each agent is relatively small when compared to the synchronization time, so the efficiency that the parallelism gives can not offset the extra synchronization costs, along with the switching time between cores. A few changes in the automatically translated source code could deliver an average improvement of 15% in throughput as can be seen in Fig. 2.21. As mentioned earlier, a slight difficulty is that there is no concept of *feedback loop* in ΣC because the task graph model is more flexible. Therefore, a small number of StreamIt applications using this concept have to be implemented by a *for* loop in ΣC. This translation can also be automated without loss of expressiveness.

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: modelling and its implementation in streaming languages.

Dataflow modelling has been a natural choice for the development of highly parallel streaming applications. The intuitive design and the exposure of the underlying parallelism makes dataflow a very attractive solution. Moreover, dataflow MoCs does not require memory coherence protocols and provide also 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. For this reason, many streaming languages, based on static models of computation, have appeared, from research languages like StreamIt or more production-ready offrerings like ΣC. In Section 2.4.3, a method and tool was introduced to convert StreamIt benchmarks in ΣC. This work aims to create a library of benchmarks for ΣC. This library allows us to use the many existing StreamIt examples as the number of ΣC applications is still insufficient and requires more programs to test the ability of language.

However, programming on manydcore platforms remains very challenging as there are

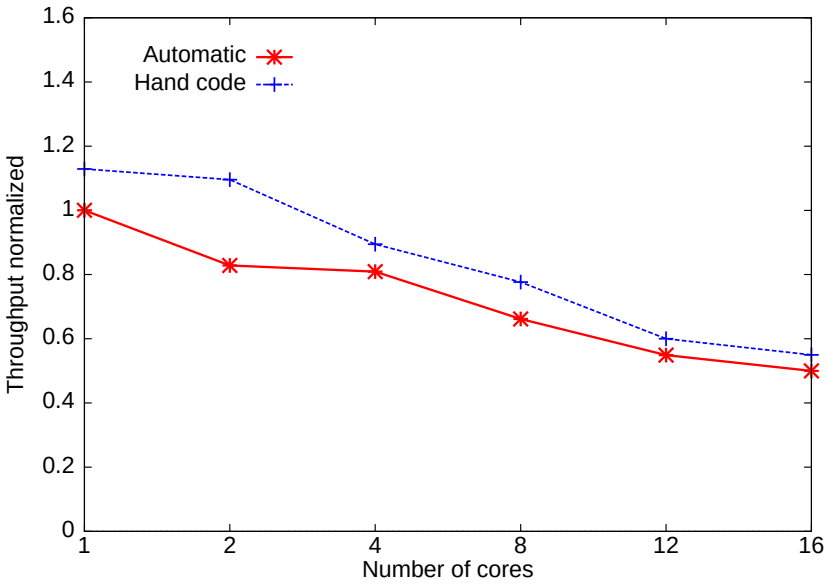


Figure 2.21: Throughput normalized for the DCT application implemented by hand and by the Python program

many different conflicting parameters to take into account. As we saw in Section 2.4.2, StreamIt and ΣC, despite their differences, have a lot of common limitations to meet the demands of emerging embedded applications. For example, many task scheduling heuristics have been developed, and although they can be reused in less expressive models like CSDF or SDF, they quickly become obsolete when developing modern applications because of the requirements for time constraints. In fact, these scheduling techniques (e.g., Self-Timed Scheduling) does not provide real-time guarantees on the availability of a given result in conformance with time constraints. New techniques based on periodic scheduling have been developed. However, they often ignore performance metrics such as latency and throughput or can even have a negative impact on it: the results are quite far from the optimal results obtained under Self-Timed Scheduling (STS).

Still, new dataflow MoCs are needed as current complex applications get even more complex and demand increased expressiveness. Proper combination of both topological, data rate dynamism and real-time constraints has not yet been achieved in any of the existing models. These features are desirable in modern streaming applications that are highly dependent on the context, and often show soft or hard real-time constraints. This context dependency requires a versatile and agile execution environment where the amount of data as well as the configuration of the application that process it may vary, as discussed in Section 1.1.3.

In the following, we propose a scheduling framework, noted Self-Timed Periodic (STP), that can schedule static dataflow MoCs on many-core platforms. The framework relies on combining self-timed scheduling with periodic scheduling. The proposed framework shows that the use of both strategies is possible and that they complement each other; STS improves the performance metrics of the programs, while the periodic model captures the timing aspects. We evaluated the performance of our scheduling policy for a set of 10 real-life streaming applications. We found that in most of the cases, our approach gives a significant improvement in latency compared to the Strictly Periodic Schedule (SPS),

and competes well with STS in terms of performance. The framework is presented in detail and applied to CSDF in Chapter 3 while its extension is introduced in Chapter 6 to consider variable interprocessor communication (IPC) times and real-time constraints imposed by hardware devices or control engineers.

Furthermore, we introduce Transaction Parameterized Dataflow (TPDF), a new model of computation combining integer parameters—to express dynamic rates—and a new type of control actor—to allow topology changes and time constraints enforcement. This new model preserves all the static analyses that make dataflow modelling so attractive, such as liveness, bounded memory usage and evaluation of worst-case throughput. Moreover, its real-time extension makes TPDF available to model task timing requirements in a great variety of situations. We also propose a static scheduling heuristic to map TPDF to massively parallel embedded platforms. We implement these analysis and scheduling methods in a tool and validate the model using not only the benchmarks library developed for ΣC but also a new set of real-life dynamic applications, demonstrating significant buffer size and throughput improvements compared to the state of the art static and dynamic models, including Cyclo-Static Dataflow (CSDF) and Scenario-Aware Dataflow (SADF). Our new data flow MoC, Transaction Parameterized Dataflow (TPDF), and its evaluation are presented in Chapter 5, 6 and 7.

Chapter 3

Self-Timed Periodic Scheduling

I've always felt so grateful that I dropped out of school, that I never had to do a thesis. I wouldn't know how to organise and structure myself to film so that B follows A and C follows B.

— Michael Moore

Contents

3.1 Motivational Example . . . 47
3.2 System Model . . . 48
3.2.1 Timed Graph . . . 48
3.2.2 Graph Levels . . . 48
3.2.3 System's model and Schedulability . . . 49
3.3 Self-Timed Periodic Scheduling . . . 49
3.3.1 Assumptions and Definitions . . . 50
3.3.2 Latency Analysis under STP Schedule . . . 51
3.4 Evaluation Results . . . 54
3.4.1 Benchmarks . . . 54
3.4.2 Experiment: Latency comparison . . . 56
3.4.3 Experiment: Throughput comparison . . . 57
3.4.4 Discussion: Decision tree for real-time scheduling of CSDF applications 58
3.5 Summary . . . 58

Given the scale of the new massively parallel systems (e.g., MPPA-256 chip from Kalray (256 cores) [34], Epiphany from Adapteva (64 cores) or Tegra X1 from NVIDIA (256 GPU-cores and 4 CPU-cores)), programming languages based on the dataflow model of computation have strong assets in the race for productivity and scalability. Nonetheless, as streaming applications must ensure *data-dependency constraints*, scheduling has serious impact on performance. Hence, multiprocessor scheduling for dataflow languages has been an active area and therefore many scheduling and resource management solutions were suggested.

The Self Timed Scheduling (STS) strategy (a.k.a. *as-soon-as-possible*) of a streaming application is a schedule where actors are fired as soon as data-dependency is satisfied. This scheduling policy is considered as the most appropriate for streaming applications modeled as dataflow graphs [89, 90, 99] because it delivers the maximum achievable

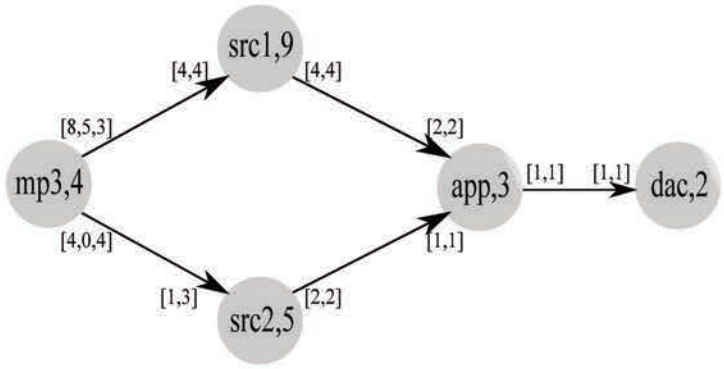
throughput and the minimum achievable latency if computing resources are sufficient [8]. However, this result can only be true if we ignore synchronization times. Synchronization can be considered as a special form of communication, for which data are control information. Its role is to enforce the correct sequencing of actors firing, and to ensure the mutually exclusive access to shared resources, and the time it takes considered as not be negligible.

Furthermore, STS does not provide real-time guarantees on the availability of a given result in conformance with time constraints. Due to the complex and irregular dynamics of self-timed operations, in addition to the synchronization overhead, many different hypotheses were suggested, like contention-free communication [89] or considering uniform costs for communication operations [8, 90]. But neglecting subtle effects of synchronization is not reasonable with regards to real systems and their hard real-time guarantees. In addition, using a predefined schedule of accesses to shared memory [67] makes run-time less flexible. Therefore, analysis and optimization of self-timed systems under real-time constraints remain challenging.

To cope with this challenge, periodic scheduling is receiving more attention for streaming applications [8, 38, 90] because of its good properties (*i.e.*, timing guarantees, temporal isolation [33] and low complexity of the schedulability test). However, periodic scheduling increases the latency significantly for a class of graphs called unbalanced graphs. A balanced graph is the one where the product of actor execution time and repetition is the same for all actors [7]. On the contrary, an unbalanced graph is the one where such products differs between actors and in the real world, as execution times of processes can have large variations, unbalanced graphs are the usual cases.

In this chapter, we propose a new scheduling policy noted Self-Timed Periodic (STP) schedule for Cyclo-Static Dataflow (CSDF) [19] graph, one of the state of the art models for describing applications in the signal processing domain (see Section 2.2). STP is a hybrid execution model based on mixing Self-Timed schedule and periodic schedule. We introduce four classes of STP schedules based on two different granularities and two types of deadline: implicit and constrained. Two first schedules, denoted $STP^{I}_{q_i}$ and $STP^{C}_{q_i}$, are based on the repetition vector q_i, given by resolving Equation (2.1),without including the sub-tasks of actors. Two remaining schedules, denoted $STP^{I}_{r_i}$ and $STP^{C}_{r_i}$, have a finer granularity by including the sub-tasks of actors. It is based on the repetition vector r_i, given by resolving Equation (2.2). For unbalanced graphs, we show that it is possible to significantly decrease the latency and increase the throughput under the STP model for both granularities. We evaluate the proposed STP representation using a set of 10 real-life applications and show that it is capable of achieving significant improvements in term of latency (with a maximum of 96.6%) compared to the SPS schedule and yielding the maximum achievable throughput obtained under the STS schedule for a large set of graphs.

The remainder of this chapter is organized as follows. In Section 3.1, we present a motivational example to illustrate the impact of the STP model on the performance. Section 3.2 introduce the timed graph, system model and schedulability of a CSDF graph which are important points for understanding our scheduling platform in Section 3.3. Section 3.4 present our evaluation of the proposed scheduling policy. Finally, Section 3.5 contains a comparison of STP with other scheduling platforms.



(a)

SPS		STP^{I}_{qi}		STP^{I}_{ri}	
L	Υ	L	Υ	L	Υ
192	1/48	144	1/36	108	1/18

(b)

Figure 3.1: (a) CSDF graph of the MP3 application. Numbers between square brackets are the number of data tokens produced or consumed by the FIFO channel. Numbers next to the name of each actor are its worst-case computation and communication time. (b) Latency (L) and throughput (Υ) metrics for the MP3 application under different scheduling schemes.

3.1 Motivational Example

In Figure 3.1(a), we show a Cyclo-Static Dataflow (CSDF) [19] graph of an MP3 application implemented using the ΣC language of CEA LIST (see Section 2.4.2). In this application, the compressed audio is decoded by the *mp3* task into an audio sample stream. These samples are converted by the Sample Rate Converter (SRC) task, after which the Audio Post-Processing (APP) task enhances the perceived quality of the audio stream and sends the samples to the Digital to Analog Converter (DAC).

The CSDF graph shown in Figure 3.1(a) is characterized by two repetition vectors $\vec{r} = [1, 2, 2, 4, 4]$ and $\vec{q} = [3, 4, 4, 8, 8]$, respectively in the order $mp3, src1, src2, app, dac$. To get q_i, we multiply r_i by the length of the consumption and production rates of a_i. For example, if $r_1 = 1$ then $q_1 = 3$ because actor *mp3* contains 3 sub-tasks. The worst-case computation and communication time of each actor is shown next to its name after a comma (*e.g.*, 4 for actor *mp3*). The graph is an example of an unbalanced graph since the product of actor execution time and repetition is not the same for all actors (*e.g.*, $3 \times 4 \neq 4 \times 9$). Let Υ and L denote the throughput (*i.e.*, rate) and latency of graph G, respectively, derived in Figure 3.1(b) for the example of Figure 3.1(a). The throughput and latency resulting from scheduling the actors of this graph as strictly periodic tasks is shown under the SPS column in Figure 3.1(b). We see that the SPS model pays a high price in terms of increased latency and decreased throughput for the unbalanced graph. Instead, if the actors are to be scheduled as self-timed periodic tasks as introduced in this paper, then it is possible to achieve 25% to 60% improvement in latency compared to the SPS schedule. For our contribution, we propose two granularities of scheduling. This depends on whether we use q_i or r_i as the basic repetition vector of CSDF. For the proposed example, $STP^{I}_{r_i}$ gives better results and for latency and for throughput, it obtains the maximum throughput, as achieved by the STS model.

3.2 System Model

We introduce in this section the timed graph, system model and schedulability of a CSDF graph which are important points for understanding our contribution in Section 3.3.

3.2.1 Timed Graph

The timed graph is a more accurate representation of the CSDF graph, that associates to each sub-task or instance of an actor a computation time and a communication overhead. We consider the Timed graph $G = \langle A, E, \omega, \varphi \rangle$, where A is a set of actors, $E \subseteq A \times A$ is a set of communication channels, $\overrightarrow{\omega} \in \mathbb{N}^N$ is the *execution time vector* of G, such that $\omega_i \in \overrightarrow{\omega}$ is the worst-case execution time (WCET) of actor $a_i \in A$. Similarly, $\overrightarrow{\varphi} \in \mathbb{N}^N$ is the *communication time vector* of G, such that $\varphi_i \in \overrightarrow{\varphi}$ is the communication cost of actor $a_i \in A$ (*i.e.* worst-case time needed for reading and writing data tokens, *etc.*).

Example 2 *Figure 3.1(a) represents also a Timed graph of the MP3 application with execution vector* $\overrightarrow{\omega} = [4, 9, 5, 3, 2]^T$ *and communication vector* $\overrightarrow{\varphi}$ *approximately equal to* $\overrightarrow{0}$.

3.2.2 Graph Levels

In this chapter, we restrict our attention to acyclic CSDF graphs which can be used to model most of the static dataflow applications. An acyclic graph G has a number of levels, denoted by α. Different graph traversals types exist like topological, breadth-first, *etc.* Actors will be assigned to a set of levels $V = \{V_1, V_2, ..., V_\alpha\}$. Authors in [8], proposed a method, presented in Algorithm 1, based on assigning the actors in the graph according to precedence constraints. An actor a_i that belongs to level V_j in Algorithm 1 has a level index $\sigma_i = j$. Each actor $a_i \in A$ is associated with two sets of actors. The sets of actors are the successors set, denoted by $\mathbf{succ}(a_i)$, and the predecessors set, denoted by $\mathbf{prec}(a_i)$.

$$\begin{aligned} \mathbf{succ}(a_i) &= \{a_j \in A : \exists e_u = (a_i, a_j) \in E\} \\ \mathbf{prec}(a_i) &= \{a_j \in A : \exists e_u = (a_j, a_i) \in E\} \end{aligned} \tag{3.1}$$

Algorithm 1 TIMED-GRAPH-LEVELS(G)

Require: Timed graph $G = \langle A, E, \omega, \varphi \rangle$
1: $i \leftarrow 1$
2: **while** $A \neq \emptyset$ **do**
3: $V_i \leftarrow \{a_j \in A : \mathbf{prec}(a_j) = \emptyset\}$
4: $E_i \leftarrow \{e_u \in E$: $\exists a_k \in V_i$ that is the source of $e_u\}$
5: $A \leftarrow A \setminus V_i$
6: $E \leftarrow E \setminus E_i$
7: $i = i + 1$
8: **end while**
9: $\alpha \leftarrow i$ - 1
10: **return** α disjoint sets $V_1, V_2, \ldots, V_\alpha$ where $\bigcup_{i=1}^{\alpha} V_i = A$

3.2.3 System's model and Schedulability

This section presents the system's model and its schedulability analysis.

System's Model

A system Π consists of a set $\pi = \{\pi_1, \pi_2, ..., \pi_m\}$ of m homogeneous processors. The processors execute a level set $V = \{V_1, V_2, ..., V_\alpha\}$ of α periodic levels. A periodic level $V_i \in V$ is defined by a 4-tuple $V_i = (S_i, \hat{\omega}_i, \hat{\varphi}_i, D_i)$, where $S_i \geq 0$ is the start time of V_i, $\hat{\omega}_i$ is the worst-case computation time (where $\hat{\omega}_i = \max_{k=1\rightarrow\beta_i} \omega_k$ with β_i representing the number of actors in level V_i), $\hat{\varphi}_i \geq 0$ is the worst-case communication time of V_i under STP schedule and D_i is the relative deadline of V_i where $D_i = \max_{k=1\rightarrow\beta_i} D_k$. A periodic level V_i is invoked at time $t = S_i + k\phi$, where $\phi \geq \hat{\omega}_i + \hat{\varphi}_i$ is the level period, and has to finish execution before time $t = S_i + k\phi + D_i$. If $D_i = \phi$, then V_i is said to have *implicit-deadline*. If $D_i < \phi$, then V_i is said to have *constrained-deadline*.

Schedulability Analysis

Actors in the Timed graph G are scheduled as *implicit-deadline* or *constrained-deadline* periodic tasks (depending on the STP approach being used) and assigned to levels. At run-time, they are executed in a *self-timed* manner. This is possible because actors of level $k+1$ consume the data produced in level k. A necessary condition for scheduling an asynchronous set of implicit-deadline periodic tasks $\Gamma \subseteq A$ on m processors is $U_{sum} \leq m$, where U_{sum} is the total utilization of Γ as proof in [33]. In this work, we consider only consistent and live CSDF graphs. A static schedule [99] of a consistent and live CSDF graph is valid if it satisfies the precedence constraints specified by the edges. Authors in [89] introduced a theorem that states the sufficient and necessary conditions for a *valid schedule*. However, this result was established for Synchronous Dataflow graphs where actors have constant execution times. In this context, our research uses the test introduced in [37] which allows the timing of firing to respect the firing rules of actors in a CSDF graph.

3.3 Self-Timed Periodic Scheduling

The effect of Self-timed Periodic (STP) scheduling can be modeled by replacing the period of the actor in each level by its worst-case execution time under periodic scheduling. The worst-case execution time is the total time of computation and communication parts of each actor. The period of each level i is the maximum time it needs to fire each actor $a_j \in V_i$, when resource arbitration and synchronization effects are taken into account. This is counted from the moment the actor meets its enabling conditions to the moment the firing is completed. There are 4 types of STP scheduling that we are interested in modeling as depicted in Table 3.1.

Table 3.1: Proposed STP Schedules

Type/Repetition vector	q_i	r_i
$\phi = D_i$ (Implicit-Deadline)	$STP^I_{q_i}$	$STP^I_{r_i}$
$\phi > D_i$ (Constrained-Deadline)	$STP^C_{q_i}$	$STP^C_{r_i}$

STP_{X_i} refers to scheduling decisions using the different granularities offered by CSDF model:

- *Coarse-Grained Schedule*: coarse-grained description of STP schedule regards instances of actors by using $\overrightarrow{q}$ as the repetition vector. Each actor a_i is viewed as executing through a periodically repeating sequence of q_i instances of sub-tasks.
- *Fine-Grained Schedule*: fine-grained description of STP schedule regards smaller components (*i.e.*, sub-tasks of actors) of which the actors are composed by using $\overrightarrow{r}$ as the repetition vector.

3.3.1 Assumptions and Definitions

A graph G refers to an acyclic consistent CSDF graph. A consistent graph can be executed with bounded memory buffers and no deadlock. We base our analysis on the following assumptions:

A1. External sources in data-flow: The model is accomplished with interfaces to the outside world in order to explicitly model inputs and outputs (I/Os). A graph G has a set of input streams $I = \{I_1, I_2, ..., I_\Delta\}$ connected to the input actors of G, and a set of output streams $O = \{O_1, O_2, ..., O_\Lambda\}$ processed from the output actors of G. An actor $a_i \in A$ is defined, inter alia, with $E_i = (E_i^{in}, E_i^{out})$ such that E_i^{in} and E_i^{out} represent the sets of input and output edges of a_i. A source and a sink nodes can be integrated as closures since they define limits of an application. These special nodes are defined as follows: $src \in A$, $E_{src}^{in} = \emptyset$ and $E_{src}^{out} = \{I_1, I_2, ..., I_\Delta\}$, $snk \in A$, $E_{snk}^{in} = \{O_1, O_2, ..., O_\Lambda\}$ and $E_{snk}^{out} = \emptyset$.

Definition 6 *For a graph G under periodic schedule, the worst-case communication overhead $\hat{\varphi}_j$ of any level $V_j \in V$ depends on the maximum number of accesses to memory m_{β_j} processed in the time interval $[(j-1) \times \phi, j \times \phi[$. In [38], the authors proved that $\hat{\varphi}_j$ is a monotonic increasing function of the number of conflicting memory accesses:*

$$\hat{\varphi}_j = \uparrow f(m_{\beta_j}), \quad \forall V_j \in V \tag{3.2}$$

A2. For periodic schedules, synchronization cost is constant, because periodic behavior guarantees that an actor $a_i \in V_j$, $\forall i \in [1, ..., \beta_j]$, will consume tokens produced at level $(j-1)$ [38]. This implies that actors of the same level can start firing immediately in the beginning of a given period because all the necessary tokens have already been produced.

Definition 7 *A graph G is said to be **matched input/output (I/O) rates graph** if and only if:*

$$\eta \mod Q = 0 \tag{3.3}$$

*If Formula (3.3) does not hold, then G is a **mismatched I/O rates graph**.*

Definition 8 *A graph G is called **balanced** if and only if:*

$$q_1\omega_1 = q_2\omega_2 = \cdots = q_n\omega_n, \tag{3.4}$$

*where $q_i \in \overrightarrow{q}$ is the repetition of actor $a_i \in A$ and ω_i is its worst-case computation time. If Equation (3.4) does not hold, then the graph is called **unbalanced**.*

Definition 9 *An **actor workload** is defined as:*

$$W_i = v_i \times \omega_i, \tag{3.5}$$

where v_i is the ith component of the repetition vector used for STP schedule. For STP_{q_i}, $v_i = q_i$ and for STP_{r_i}, $v_i = r_i$. The maximum workload of level V_j is $\hat{W}_j = \max_{a_i \in V_j}\{W_i\}$.

Definition 10 *Let $p_{a \rightsquigarrow z} = \{(a_a, a_b), \ldots, (a_y, a_z)\}$ be an output path in a graph G. The latency of $p_{a \rightsquigarrow z}$ under periodic input streams, denoted by $L(p_{a \rightsquigarrow z})$, is the elapsed time between the start of the first firing of a_a which produces data to (a_a, a_b) and the finish of the first firing of a_z which consumes data from (a_y, a_z).*

Consequently, we define the maximum latency of G as follows:

Definition 11 *For a graph G, the maximum latency of G under periodic input streams, denoted by $L(G)$, is given by:*

$$L(G) = \max_{p_{i \rightsquigarrow j} \in P} L(p_{i \rightsquigarrow j}), \tag{3.6}$$

*where P denotes the set of all output paths in G. A path $p_{i \rightsquigarrow j} = (a_i, a_j)$ is called **output path** if a_i is a source node which receives an **input stream** of the application and a_j is a sink node which produces an **output stream**.*

Example 3 *The CSDF graph shown in Figure 3.1(a) has two output paths given by $P = \{p_1 = \{(mp3, src1), (src1, app), (app, dac), p_2 = \{(mp3, src2), (src2, app), (app, dac)\}$. This graph is also an example of an unbalanced graph since the product of actor execution time and repetition is not the same for all actors (*e.g., $3 \times 4 \neq 4 \times 9$*).*

3.3.2 Latency Analysis under STP Schedule

A self-timed schedule does not impose any extra latency on the actors. This leads us to the following result:

Definition 12 *(Periods of Levels in STP_{q_i}) For a graph G, a period ϕ, where $\phi \in \mathbb{Z}^+$, represents the period, measured in time-units, of the levels in G. If we consider $\vec{q}$ as the basic repetition vector of G in Definition 9, then ϕ is given by the solution to:*

$$\phi \geq \max_{j=1 \rightarrow \alpha} (\hat{W}_j + \hat{\varphi}_j) \tag{3.7}$$

Definition 12 defines the level period ϕ as the maximum execution time of all levels. ϕ can be chosen as this value or greater. Similarly, we define the schedule function for the finer granularity of CSDF characterized by the repetition vector $\vec{r}$ if we consider $\vec{r}$ as the basic repetition vector of G in Equation (3.7).

For STP_{q_i}, we use Algorithm 1 to find the levels of G. For STP_{r_i}, Algorithm 2 is used because this scheduling policy has a finer granularity and requires an algorithm which depends also on the precedence constraints and firing rules of actors. In this case, each

Algorithm 2 GRAPH-LEVELS-STP-Ri(G)

Require: Timed graph $G = \langle A, E, \omega, \varphi \rangle$
1: $count_i \leftarrow 0$
2: $j \leftarrow 1$
3: $S \leftarrow \{a_1\}$ ▷ a_1 assumed to be the source actor;
4: **while** $\exists a_i \in A\ count_i < q_i$ **do**
5: $V_j \leftarrow \{a_i \in S : \textit{there are enough tokens in all input edges to fire } a_i \textit{ for } r_i \textit{ times}\}$
6: **for all** $a_i \in V_j$ **do**
7: $count_i \leftarrow count_i + r_i$
8: **if** $count_i < q_i$ **then**
9: $S = S \bigcup \mathbf{succ}(a_i)$
10: **else**
11: **if** $count_i = q_i$ **then**
12: $S \leftarrow S \setminus \{a_i\}$
13: **end if**
14: **end if**
15: **end for**
16: $j \leftarrow j + 1$
17: **end while**
18: $\alpha' \leftarrow j$ - 1
19: **return** α' disjoint sets $V_1, V_2, \ldots, V_{\alpha'}$

actor a_i could only be fired for r_i times if there are enough tokens in all of their input edges.

An actor $a_i \in V_j$ is said to be a level-j actor. For STP_{q_i}, let ϕ denote the level period as defined in Definition 12, and let a_1 denote the level-1 actor. a_1 will complete one iteration when it fires q_1 times. Assume that a_1 starts executing at time $t = 0$. Then, by time $t = \phi \geq q_1\omega_1$ as defined in Definition 12, a_1 is guaranteed to finish one iteration in a self-timed mode (start the next sub-task immediately after the end of the precedent). According to Theorem 1, a_1 will also generate enough data such that every actor $a_k \in V_2$ can execute q_k times (*i.e.* one iteration). According to Definition 12, firing a_k for q_k times in a self-timed mode takes $q_k\omega_k$ time-units. Thus, starting level-2 actors at time $t = \phi$ guarantees that they can finish one iteration. Similarly, by time $t = 2\phi$, level-3 actors will have enough data to execute for one iteration. By repeating this over all the α levels, a schedule S_α (shown in Figure 3.2) is constructed in which all actors $a_i \in V_j$ are started at *start time*, denoted $s_{i,j}$, given by:

$$s_{i,j} = (j - 1)\phi \tag{3.8}$$

$V_j(k)$ denotes level-j actors executing their k-th iteration. For example, $V_2(1)$ denotes level-2 actors executing their first iteration. At time $t = \alpha\phi$, G completes one iteration. It is trivial to observe from S_α that as soon as a_1 finishes one iteration, it can immediately start executing the next iteration since its input stream arrives periodically. If a_1 starts its second iteration at time $t = \phi$, its execution will overlap with the execution of the level-2 actors. By doing so, level-2 actors can start immediately their second iteration after finishing their first iteration since they will have all the needed data to execute one

time	$[0,\phi)$	$[\phi,2\phi)$	$[2\phi,3\phi)$	...	$[(\alpha-1)\phi,\alpha\phi)$
level	$V_1(1)$	$V_2(1)$	$V_3(1)$	...	$V_\alpha(1)$
		$V_1(2)$	$V_2(2)$	...	$V_{\alpha-1}(2)$
			$V_1(3)$	...	$V_{\alpha-2}(3)$
				...	$V_{\alpha-3}(4)$
				...	...
					$V_1(\alpha)$

Figure 3.2: Initial phase of schedule S_α

iteration in a self-timed mode at time $t = 2\phi$. Now, the overlapping can be applied α times to yield a schedule S_α as shown in Figure 3.2. Starting from $t = \alpha\phi$, a schedule S_∞ can be constructed by pipelining the S_α schedule, as can be seen in Figure 3.3. The start time defined in Equation (3.8) guarantees that actors at a given level will execute only when they have enough data to execute. Thus, schedule S_∞ shows the existence of a self-timed periodic schedule of G where every actor $a_j \in A$ is self-timed periodic with a period level equal to ϕ.

time	$[0,\phi)$	$[\phi,2\phi)$	...	$[(\alpha-1)\phi,\alpha\phi)$	$[\alpha\phi,(\alpha+1)\phi)$
level	$V_1(1)$	$V_2(1)$	...	$V_\alpha(1)$	$V_\alpha(2)$
		$V_1(2)$	...	$V_{\alpha-1}(2)$	$V_{\alpha-1}(3)$
			...	$V_{\alpha-2}(3)$	$V_{\alpha-2}(4)$
			...	$V_{\alpha-3}(4)$	$V_{\alpha-3}(5)$
			...	...	...
				$V_1(\alpha)$	$V_1(\alpha+1)$

Figure 3.3: Schedule S_∞ by pipelining the steady state S_α

According to Definition 10 and 11, latency is defined as the maximum time elapsed between the first firing of *src* actor in level V_1 and the finish of the first firing of *snk* actor in level V_α. Then, the graph latency $L(G)$ is given by:

$$L(G) = \max_{p_{i \rightsquigarrow j} \in P} (s_{snk,\alpha} + \hat{y}{}^{u}_{snk}\phi + D_\alpha - (s_{src,1} + \hat{x}{}^{r}_{src}\phi)) \tag{3.9}$$

where $s_{snk,\alpha}$ and $s_{src,1}$ are the earliest start times of the *snk* actor and the *src* actor, respectively, D_α is the deadline of *snk* and V_α, and $\hat{x}{}^{r}_{src}$ and $\hat{y}{}^{u}_{snk}$ represent the first non-zero production (consumption) sub-task of the *src* (*snk*) actor, such that for an output path $p_{src \rightsquigarrow snk}$ in which e_r is the first channel and e_u is the last channel, $\hat{x}{}^{r}_{src}$ and $\hat{y}{}^{u}_{snk}$ are given by:

$$\hat{x}{}^{r}_{src} = \min\{k \in N : x^r_{src}(k) > 0\} - 1 \tag{3.10}$$

$$\hat{y}{}^{u}_{snk} = \min\{k \in N : y^u_{snk}(k) > 0\} - 1 \tag{3.11}$$

Under the *implicit-deadline* model, $D_\alpha = \phi$ and under the *constrained-deadline* scheduling, $D_\alpha < \phi$. Using Equations (3.7) and (3.8), it is possible to obtain a simple version of

Table 3.2: Benchmarks used for evaluation

Domain	No.	Application	N	Q	max($q_i\omega_i$)	Source
Signal Processing	1	Discrete cosine Transform (DCT)	4	12	1800	CEA LIST
	2	Fast Fourier Transform (FFT) kernel	4	6	900	
	3	Multi-Channel beamformer	4	12	7800	
Audio Processing	4	Filter bank for multirate signal processing	17	600	113430	MIT [106]
	5	MP3 audio decoder	5	24	36	CEA LIST
	6	Sample-rate converter used in CDs	6	23520	960	SDF^3 [102]
Video Processing	7	H.263 video encoder	5	33	382000	
	8	H.263 video decoder	4	2376	10000	
Mathematics	9	Bipartite graph	4	144	252	
Communication	10	Satellite receiver	22	5280	1056	

Equation (3.9) under the *implicit-deadline* model for acyclic CSDF graphs where production of *src* actor and consumption of *snk* actor taking place from the first firing of each node ($\hat{x}^r_{src} = \hat{y}^u_{snk} = 0$):

$$L_{STP^I_{q_i/r_i}} = (s_{snk,\alpha} + \phi) - s_{src,1} = \alpha \times \phi \tag{3.12}$$

Example 4 *We illustrate in Figure 3.4 different scheduling policies applied for the MP3 application shown in Figure 3.1(a). This application has an execution vector* $\overrightarrow{\omega} = [4, 9, 5, 3, 2]^T$ *and a communication vector* $\overrightarrow{\varphi}$ *approximately equal to* $\overrightarrow{0}$*. The mp3 node is the src actor and dac is the snk actor. Applying Algorithm 1, the number of levels for* $STP^I_{q_i}$ *is* $\alpha = 4$ *and we have 4 sets:* $V_1 = \{mp3\}$, $V_2 = \{src1, src2\}$, $V_3 = \{app\}$, $V_4 = \{dac\}$.

Applying Algorithm 2, the number of levels for $STP^I_{r_i}$ *is* $\alpha = 6$ *and we have 6 sets:* $V_1 = \{mp3\}$, $V_2 = \{mp3, src1, src2\}$, $V_3 = \{mp3, app\}$, $V_4 = \{src1, src2, dac\}$, $V_5 = \{app\}$, $V_6 = \{dac\}$. *Given* $\overrightarrow{q} = [3, 4, 4, 8, 8]^T$ *and* $\overrightarrow{r} = [1, 2, 2, 4, 4]^T$*, we use Equation (3.5) and (3.7) to find the period of levels* $\phi = 36$ *for* $STP^I_{q_i}$ *and* $\phi = 18$ *for* $STP^I_{r_i}$.

This graph has two output paths given by $P = \{p_1 = \{(mp3, src1), (src1, app), (app, dac), p_2 = \{(mp3, src2), (src2, app), (app, dac)\}$. *Finally, using Equation (3.12), we have* $L_{STP^I_{q_i}}(p_1) = L_{STP^I_{q_i}}(p_2) = 144$ *and* $L_{STP^I_{r_i}}(p_1) = L_{STP^I_{r_i}}(p_2) = 108$ *as depicted in Figure 3.4.*

3.4 Evaluation Results

We evaluate our proposed scheduling policy in Section 3.3 by performing an experiment on a set of 10 real-life streaming applications. The objective of the experiment is to compare the efficiency of our STP approach to their maximum achievable performance obtained via self-timed scheduling and the results achieved under strictly periodic scheduling.

3.4.1 Benchmarks

The streaming applications used in the experiment are real-life applications which come from different domains (*e.g.*, signal processing, video processing, mathematics, *etc.*) and from different sources to check the efficiency of this scheduling in different architectures. The first source is the ΣC benchmark which contributes 4 streaming applications. The

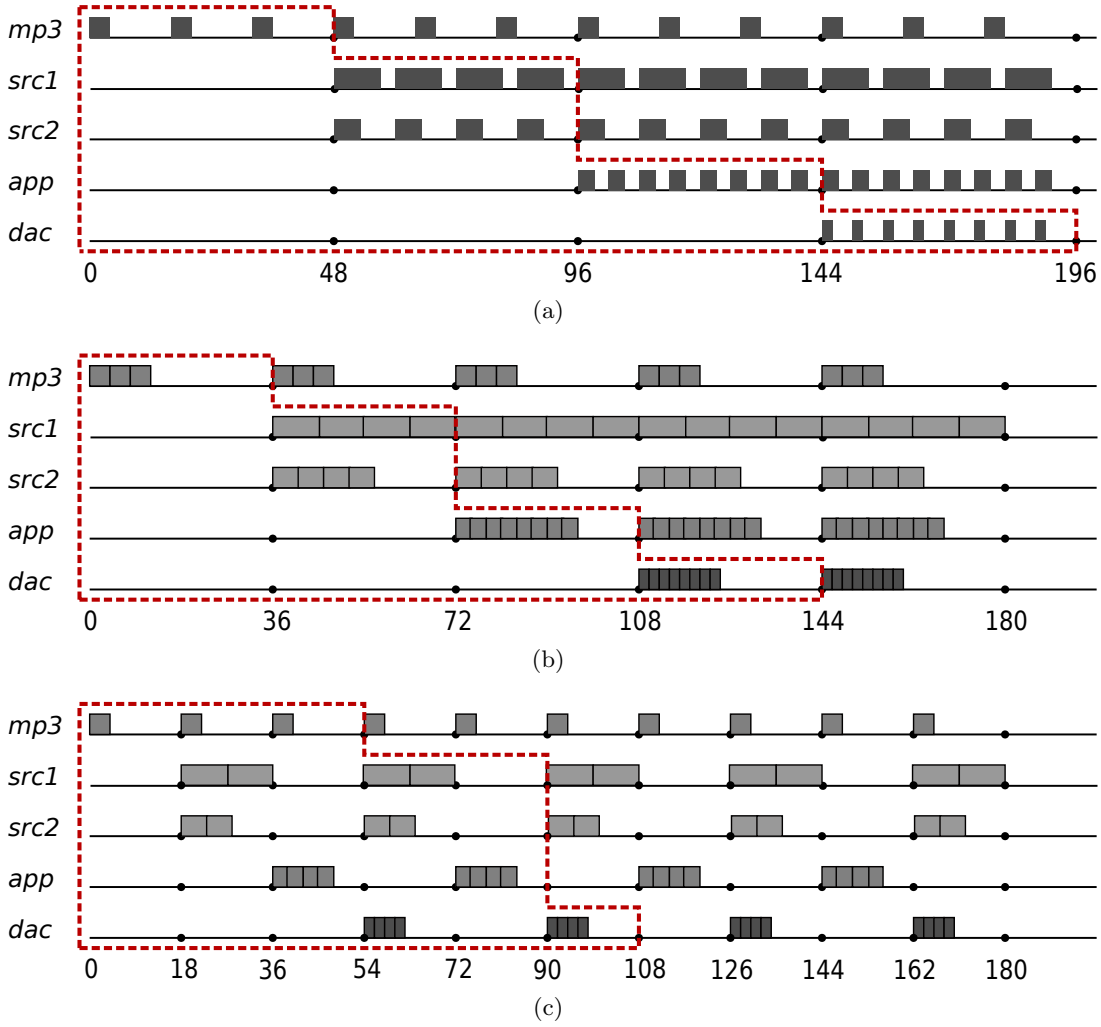


Figure 3.4: Illustration of latency path for the MP3 application shown in Figure 3.1(a): (a) SPS (b) $STP^I_{q_i}$ (c) $STP^I_{r_i}$. The dotted line represents a valid static schedule of the graph. An improvement of 25% to 60% in latency could be achieved by the $STP^I_{q_i}$ and $STP^I_{r_i}$ schedules compared to the SPS schedule.

second source is the SDF^3 benchmark which contributes 5 streaming applications [102]. The last source is the StreamIt benchmark [106]. In total, 10 applications are considered as shown in Table 3.2. The graphs are a mixture of CSDF (ΣC's applications) and SDF (StreamIt and SDF^3 benchmark) graphs. The use of synchronous dataflow (SDF) models does not affect our scheduling policy because SDF, with static firing rules of actors, is a special case of CSDF model [19, 74]. The fourth column (N) shows the number of actors in each application, the fifth column (Q) shows the least-common-multiple of the repetition vector elements (*i.e.*, $Q = lcm(q_1, q_2, \ldots, q_n)$) and the sixth column is the maximum of the product $q_i\omega_i$ used to calculate the end-to-end latency by Formula (3.5), (3.7) and (3.12).

The actors execution times of the ΣC benchmark are measured in clock cycles on the MPPA 256 cores, while the actors execution times of the SDF^3 benchmark are specified by its authors for ARM architecture. For the StreamIt benchmark, the actors execution times are specified in clock cycles measured on MIT RAW architecture.

We use SDF^3 tool-set for several purposes during the experiments. SDF^3 is a powerful analysis tool-set which is capable of analyzing CSDF and SDF graphs to check for consistency errors, compute the repetition vector, compute the maximum achievable throughput, *etc.* In this experiment, we use SDF^3 to compute the minimum achievable latency of the graph and use it as a reference point for comparing the latency under the SPS and STP models. For StreamIt benchmark, the graph exported by this language is converted in the XML required by SDF^3. For ΣC applications, the ΣC compiler is capable of checking consistency errors and computing the repetition vector during its 4 stages of compilation [53]. The latency of its applications is calculated by using the execution times measured on the MPPA platform.

3.4.2 Experiment: Latency comparison

In this experiment, we compare the end-to-end latency resulting from our STP approach to the minimum achievable latency of a streaming application obtained via self-timed scheduling and the one achieved under strictly periodic scheduling. The STS latency is computed using the latency algorithm of the *sdf3analysis* tool from SDF^3 with auto-concurrency disabled and unbounded FIFO channel sizes.

Table 3.3 shows the latency obtained under STS, SPS, $STP^I_{q_i}$, $STP^I_{r_i}$, $STP^C_{q_i}$ and $STP^C_{r_i}$ schedules as well as the improvement of these policies compared to the SPS model. We report the graph maximum latency according to Formula (3.12). For SPS schedule, we used the minimum period given by Equation (2.6). For STP schedule, we used the level period given by Definition 12. We see that the calculation of the STP schedule is not complicated because the graph is consistent and an automatic tool could be implemented to find this schedule.

Table 3.3: Results of Latency comparison

Application	STS	SPS	$STP^I_{q_i}$	$Eff_{STP^I_{q_i}}$	$STP^I_{r_i}$	$Eff_{STP^I_{r_i}}$	$STP^C_{q_i}$	$Eff_{STP^C_{q_i}}$	$STP^C_{r_i}$	$Eff_{STP^C_{r_i}}$
DCT	2500	7200	5400	38.3	4500	57.5	3500	78.7	3200	85.1
FFT	23000	36000	27000	69.2	32000	30.8	23000	100.0	23000	100.0
Beamformer	9500	25200	23400	11.5	30000	-30.6	12100	83.4	13700	73.3
Filterbank	124792	1254000	1247730	0.6	1247730	0.6	309033	83.7	309033	83.7
MP3	48	192	144	33.3	108	58.3	88	72.2	72	83.3
Sample-rate	1000	141120	5760	96.6	5760	96.6	2439	99.0	2439	99.0
Encoder	664000	1584000	1528000	6.1	1528000	6.1	799000	85.3	799000	85.3
Decoder	23506	47520	40000	31.3	40000	31.3	25880	90.1	25880	90.1
Bipartite	293	576	504	25.4	504	25.4	369	73.2	369	73.2
Satellite	1314	58080	11616	81.9	11616	81.9	2377	98.1	2377	98.1

For the $STP^I_{q_i}$, we see that it delivers an average improvement of 39.4% (with a maximum of 96.6%) compared to the SPS model for all the applications. In addition, we clearly see that our $STP^I_{q_i}$ provides at least 25% of improvement for 7 out of 10 applications. Only three applications (Filterbank, Beamformer and H.263 Encoder) have lower performance under our $STP^I_{q_i}$. To understand the impact of the results, we use the concept of *balanced* graph (see Definition 8). According to [7], periodic models increase the latency significantly for *unbalanced* graphs. For our approach, Definition 12 and Formula (3.12) indicate that if the product $q_i\omega_i$ is too different between actors, so the period of levels ϕ and the latency L become higher. For actors where this product is much smaller, wasted time in each level increases the final value of latency. This is the main reason why we

reduce these bad effects by using the constrained-deadline self-timed periodic schedule $STP^C_{q_i}$ and $STP^C_{r_i}$.

We also see that the *mismatched* I/O rates applications (*i.e.* with large Q such as Sample-rate, Satellite and Filterbank in Table 3.2) have higher latency under strictly periodic scheduling. This result could be explained using an interesting finding reported in [106]: *Neighboring actors often have matched I/O rates. This reduces the opportunity and impact of advanced scheduling strategies proposed in the literature.* This issue can be resolved by using our approach. In fact, for nearly balanced graphs (*i.e.*, graphs where the product $q_i\omega_i$ is not too different between actors) such as Sample-rate and Satellite, we have an improvement of 96.6% and 81.9%, relatively, for the end-to-end latency of each benchmark. For the remaining applications, the SPS model increases the latency on average by 2.5× compared to the STS latency while this rate for $STP^I_{q_i}$ is 2×.

For the $STP^I_{r_i}$ approach, we have an average improvement of 35.8% compared to the SPS model for all the applications. For 8 out of 10 benchmarks, this scheduling policy give at least the result given by $STP^I_{q_i}$. Only two applications (Beamformer and FFT) have lower performance when using this scheduling policy. The main reason is that the $STP^I_{r_i}$ give a finer granularity based on the repetition vector r_i. This means that if $\vec{r}$ is too close to $\vec{1}$, the sum of wasted time in each level will significantly increases the end-to-end latency.

For this reason, we extend our result by using 2 other constrained deadline approaches: $STP^C_{q_i}$ and $STP^C_{r_i}$. The constrained deadline model assigns for each task a deadline $D < \phi$, where ϕ is the period of levels. Figure 3.5 shows the ratio of the latency of 5 scheduling policies (including $STP^C_{q_i}$ and $STP^C_{r_i}$) to the minimum achievable latency (*i.e.*, STS latency). A ratio equal to 1.0 means that the $STP^C_{q_i}$ and $STP^C_{r_i}$ latency are equal to the STS latency. We see that the $STP^C_{r_i}$ model achieves nearly the minimum achievable latency for 7 graphs. In addition, it is worth noting that these approaches have, on average, 86.4% of improvement for $STP^C_{q_i}$ and 87.1% for $STP^C_{r_i}$ compared to the SPS latency; it means that we have only 13.6% and 12.9%, respectively, degradation of latency compared to STS. However, this degradation is negligible for a schedule that guarantees periodic properties.

3.4.3 Experiment: Throughput comparison

In this experiment, we compare the throughput resulting from our STP approach to the maximum achievable throughput of a streaming application obtained via self-timed scheduling. Computing the throughput of the STS using SDF^3 is done using the algorithm *throughput* of the $sdf3analysis - (c)sdf$ tool.

The last column of Table 3.4 shows the ratio of the STS schedule throughput to the $STP^I_{q_i}$ schedule throughput ($\Upsilon_{STS}/\ \Upsilon_{STP^I_{q_i}}$). Notice that the unit for the results in Table 3.4 is $\frac{1}{clock\ cycle}$. A result in seconds could be obtained by dividing these results by the value of one cycle (*e.g.* $2.5 \times 10^{-9} s$ for MPPA 256 cores). We clearly see that our STP delivers the same throughput as STS for 9 out of 10 applications. The only application have lower throughput is H.263 Encoder but overall, we demonstrated good results while ensuring high level of time determinism. This fact show one more advantage of using our STP framework.

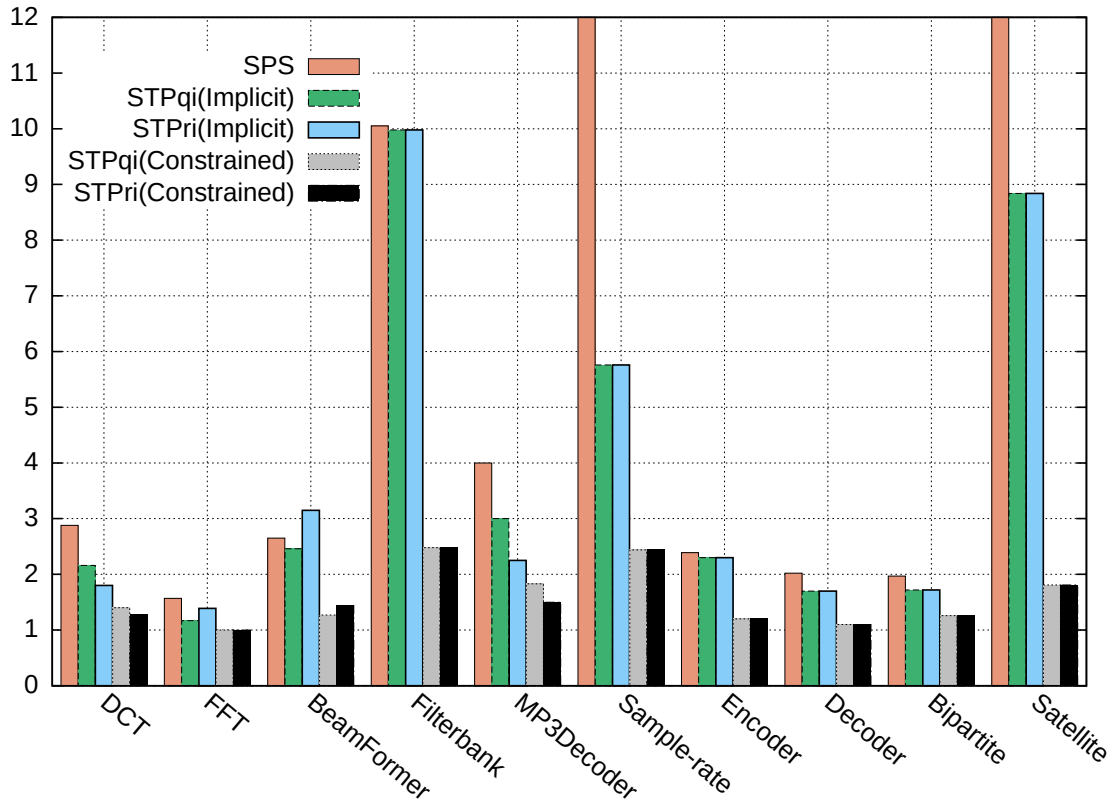


Figure 3.5: Ratios of the latency under SPS, $STP^I_{q_i}$, $STP^I_{r_i}$, $STP^C_{q_i}$ and $STP^C_{r_i}$ to the STS latency. It must be noted that the Sample-Rate and Satellite programs have a ratio for SPS much larger than 12, but the graph is zoomed to display accurately the results for most of the programs.

3.4.4 Discussion: Decision tree for real-time scheduling of CSDF applications

Based on the evaluation results in Section 3.4.2 and 3.4.3, we present a decision tree for selecting between different real-time scheduling policies that we propose for CSDF graphs in this chapter. The decision tree is illustrated in Figure 3.6. The first decision is to determine whether the application is safety-critical or not. If the application is safety-critical, then the SPS model, with its better temporal isolation property, is chosen to guarantee time constraints. If the application have simpler real-time constraints, $STP^I_{q_i}$ and $STP^I_{r_i}$ could be chosen, based on which granularity gives better result, to reduce the end-to-end latency while ensuring the maximum throughput obtained under the Self-Timed Scheduling. In the case of unbalanced graphs, $STP^C_{q_i}$ and $STP^C_{r_i}$ helps in further reducing latency if the constrained deadline model is possible.

3.5 Summary

In this chapter, we prove that the actors of a streaming application modeled as CSDF graph, can be scheduled as self-timed periodic tasks. As a result, we conserve the properties of a periodic scheduling and in the same time improve its performance. We also show how the different granularities offered by CSDF model can be explored to decrease latency. We present an analytical framework for computing the periodic task parameters while taking into account inter-processor communication and synchronization overhead.

Table 3.4: Results of Throughput comparison

Application	$\Upsilon_{\mathbf{STS}}$	$\Upsilon_{\mathbf{STP^I_{q_i}}}$	$\Upsilon_{\mathbf{STS}}/\Upsilon_{\mathbf{STP^I_{q_i}}}$
DCT	2.22×10^{-3}	4/1800	1.0
FFT	3.33×10^{-3}	3/900	1.0
Beamformer	5.13×10^{-4}	4/7800	1.0
Filterbank	8.81×10^{-6}	1/113430	1.0
MP3	2.22×10^{-1}	8/36	1.0
Sample-rate	1.04×10^{-3}	1/960	1.0
Encoder	4.73×10^{-6}	1/382000	1.8
Decoder	1.0×10^{-4}	1/10000	1.0
Bipartite	3.96×10^{-3}	1/252	1.0
Satellite	9.46×10^{-4}	1/1056	1.0

Based on empirical evaluations, we show that our STP approach reduces significantly the latency compared to the SPS model and delivers the maximum throughput achieved under the STS model. We summarize our results in the form of a decision tree to assist the designer in choosing the appropriate scheduling policy for acyclic CSDF graphs.

In comparison with other scheduling frameworks, Ghamarian *et al.* propose a heuristic for optimizing latency under a throughput constraint [51]. It gives optimal latency and throughput results under a constraint of maximal throughput for all DSP and multimedia models. However, this approach uses Synchronous Data-flow (SDF) graphs which are less expressive than CSDF graphs in that SDF supports only a constant production/consumption rate on their edges, whereas CSDF supports varying (but predefined) production/consumption rates. As a result, the analysis result in [51] is not applicable to CSDF graphs. In [20], Bodin *et al.* present a characterization of feasible periodic schedules associated with a CSDF. Two algorithms are deduced to approximately solve the evaluation of the maximum throughput of a CSDF and the buffer sizing with a throughput constraint. However, the throughput computed for instances with bounded buffers is quite far from the optimal achieved under self-timed schedule.

In [70, 111], the authors present a buffer sizing approach and its extension which exploits that FIFO buffers bound interference between tasks on shared processors. The approach combines temporal analysis using a cyclic dataflow model with computation of buffer capacities in an iterative manner and thereby enables higher throughput guarantees at smaller buffer capacities. In [4], Ali *et al.* propose an algorithm for extracting the real-time properties of dataflow applications with timing constraints. Our approach differs from [4, 70, 111] in: these papers use Homegeneous Synchronous Data-flow (HSDF) as analysis model, which is less expressive than CSDF and transformation of (C)SDF graphs into equivalent HSDF graphs use an unfolding process that replicates each actor possibly an exponential number of times.

Bouakaz *et al.* [22] propose a model of computation in which the activation clocks of actors are related by affine functions. This model, named Affine Dataflow (ADF), extends the CSDF model and proposed an analysis framework to schedule the actors in an ADF graph as periodic tasks. A major advantage of their approach is the enhanced expressiveness of the ADF model. For most benchmarks, both CSDF and ADF achieve the same throughput and latency while requiring the same buffer sizes. However, in few cases,

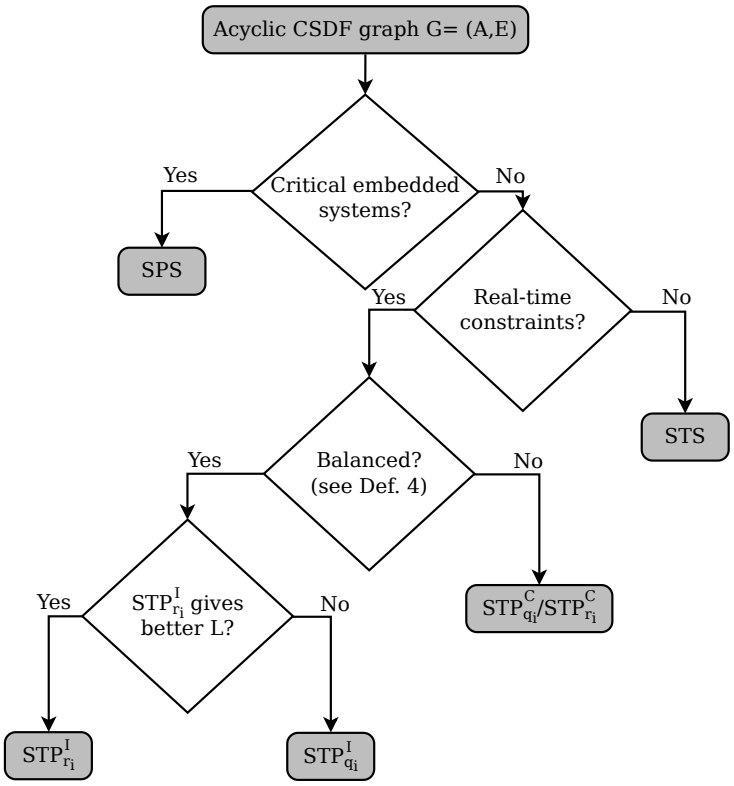


Figure 3.6: Decision tree for real-time scheduling of CSDF applications. The STP scheduling could be used to reduce the end-to-end latency of real-time CSDF applications. In the case of unbalanced graphs, $STP^C_{q_i}$ and $STP^C_{r_i}$ give better results if the constrained deadline model is possible.

ADF results in reduced buffer sizes compared to CSDF [22]. In [23], the authors provide another symbolic expression of the maximal throughput of acyclic synchronous dataflow graphs. Based on these investigations, they define symbolic analyses that approximate the minimum buffer sizes needed to achieve maximal throughput for acyclic graphs. The same approach can be assumed to be applied for exact and approximate symbolic evaluations of the latency of parametric graphs. However, in this paper only graphs with a single parametric edge dataflow are studied and symbolic analysis of cyclic dataflow graphs is still to solve.

In [68], Klikpo *et al.* propose an approach to model formally the synchronous semantic of multi-periodic Simulink systems by Synchronous Dataflow Graph (SDF).This model is constructed on a formal equivalence between the data dependencies imposed by the communication mechanisms in Simulink and the precedence constraints of a synchronous dataflow graph. In [8], Bamakhrama and Stefanov present another complete framework for computing the periodic task parameters using an estimation of worst-case execution time. They assume that each write or read has constant execution time which is often not true. Our approach is somewhat similar to [8] in using the periodic task model which allows to apply a variety of proven hard-real-time scheduling algorithms for multiprocessors. However, it is different from [8] in: 1) in our model, actors will no longer be strictly periodic but self-timed assigned to periodic levels, and 2) we treat the case variable execution time of actors due to synchronization and contention in shared resources.

Nevertheless, the STP technique does not consider variable interprocessor communication (IPC) overhead and real-time constraints imposed by hardware devices or control engineers. In the next section, we introduce an improvement of the STP platform to satisfy all the system and user requirements, latency could be evaluated between the initiation times of any two dependent actors. As a result, a latency-based approach for fault-tolerance could be implemented in a manycore architecture to guarantee real-time services.

Chapter 4

Latency-based approach for fault-tolerance

When Larry and Sergey founded Google Search, one of the things that struck me is that it was available for everyone to use. We deeply desire our services to work for everyone. And that inherently means we have to work with partners. That is the thesis underlying everything we do.

— Sundar Pichai

Contents

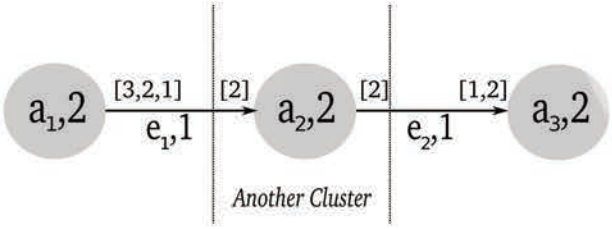
4.1 Motivational Example . . . 64
4.2 Hard-Real-Time Scheduling of CSDF . . . 65
4.3 Actor Dependence Function . . . 67
4.3.1 Definition . . . 67
4.3.2 Calculating ADF . . . 68
4.3.3 Illustrative example . . . 69
4.4 Latency Analysis . . . 70
4.4.1 Definition . . . 70
4.4.2 Latency Analysis under a hard-real-time scheduling . . . 70
4.5 Fault-Tolerance . . . 71
4.5.1 Data Model . . . 71
4.5.2 Support for fault-tolerance . . . 72
4.6 Evaluation results . . . 73
4.6.1 Benchmarks . . . 73
4.6.2 Experiment: Throughput comparison . . . 73
4.7 Summary . . . 75

In the last section, we demonstrate that it is possible to apply periodic scheduling for applications modelled as CSDF graphs and improve its latency and throughput performance by using the STP platform. However, this scheduling technique can still violate timing constraints and safety requirements of critical real-time embedded systems (*e.g.* avionics). Such violations are usually caused by delays that are not accounted for, due to resource sharing (*e.g.* the communication medium). In this chapter, we improve the

proposed scheduling technique while considering variable interprocessor communication (IPC) times, and real-time constraints imposed by hardware devices or control engineers. From this scheduling platform which satisfies all user constraints, we establish a latency constraint on the initiation times of predecessor actors on which a given actor is dependent. The purpose of this latency constraint is different from Chapter 3, which aims to optimize the latency of a dataflow application. In this Chapter, we want to find a way to guarantee real-time constraints for streaming applications implemented in distributed systems. Based on this constraint, we show how to reduce inconsistencies in a CSDF application by introducing a fault-tolerant procedure. We evaluate the performance of our scheduling policy for a set of 12 real-life streaming applications. We find that in most of the cases, our approach yields also the maximum achievable throughput as obtained under Self-Timed scheduling (STS).

The remainder of this chapter is organized as follows. In Section 4.1, we present a motivational example to illustrate the impact of our approach. Section 4.2 introduces an improved hard-real-time scheduling for applications modeled as CSDF graphs. Section 4.3 defines the actor dependence function, and Section 4.4 evaluates the latency between initiation times of 2 dependent actors. Based on this evaluation, a fault-tolerant procedure for real-time streaming applications is introduced in Section 4.5 and Section 4.6 presents our evaluation of the proposed hard-real-time scheduling policy.

4.1 Motivational Example



(a)

a_1	a_2	a_3
1	1	1
2	2	2
		3
3	3	4

(b)

Figure 4.1: (a) Distributed stream graph. Numbers between square brackets are the number of data tokens produced or consumed by the FIFO channel. The worst-case computation of each actor is shown next to its name after a comma. The communication time of each channel is shown next to its name and is attached to the production actor. (b) Dependence between actor's executions.

In Figure 4.1(a), we show a pipeline CSDF graph. In CSDF graphs, each actor in the graph is executed through a periodically repeated sequence of sub-tasks. Any CSDF graph is characterized by two repetition vectors $\vec{q}$ and $\vec{r}$. $\vec{q}$ is the minimal set of sub-tasks firings returning the data-flow graph to its initial state (all inputs are consumed). For the example depicted in Figure 4.1(a), $\vec{r} = [1, 3, 2]$ and $\vec{q} = [3, 3, 4]$, respectively in the order a_1, a_2, a_3. To get q_i, we multiply r_i by the length of the consumption and production rates of a_i. For example, if $r_1 = 1$ then $q_1 = 3$ because actor a_1 contains 3 sub-tasks. The worst-case computation and communication time of each actor is shown next to its name after a comma. In this example, all actors have an execution time of 2, actor a_2 is executed in another cluster with a communication overhead between a_1 and a_2, a_2 and a_3 is 1. Actor 3 has a period of 3 imposed by the control engineer. A Static (Strictly)

Periodic Scheduling (see Section 4.2) which takes into account all these conditions could be found in Figure 4.2 with a vector of minimum period for each actor $\vec{T^{min}} = [4, 4, 3]$.

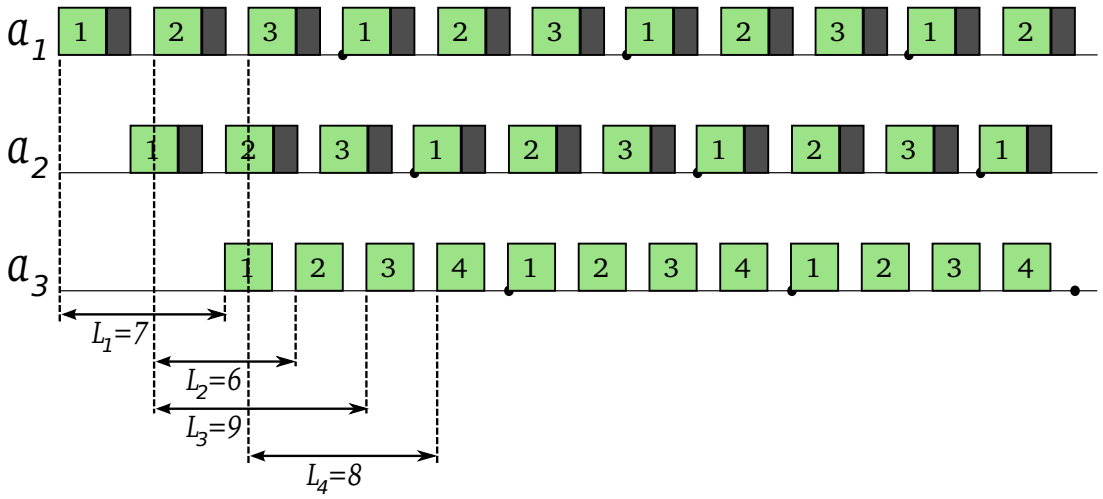


Figure 4.2: Example of hard-real-time scheduling for the distributed stream graph in Figure 4.1(a).

In this work, some communication channels will be considered as fragile. This can result in time constraints violated or inconsistency of data because of communication failures. Therefore, it is necessary to have a method to check the consistency of data in a real-time streaming application. We introduced an actor dependence function, *ADF*, that describes the dependence between the executions of 2 actors connected by a directed path in a stream graph. As a result, a latency constraint could be evaluated between the executions of any two dependent actors in a CSDF graph and the consistency of data could be checked automatically by the dependence between actors' executions and latency from the source to the destination actor. In the case of the dataflow graph in Figure 4.1(a), Figure 4.1(b) shows the dependence between actors. This table can be read as follows: The first execution of a_1 produces the last token required for the first execution of a_2 and the first execution of a_2 produces the last token required for the first execution of a_3. Similarly, the second execution of a_1 produces the last token required for the second execution of a_2 and the second execution of a_2 produces the last token required for the third execution of a_3. From this dependency, the latency between a_1 and a_3 could be calculated as in Figure 4.2. The consistency of data between actors could be verified when at a latency check-point, if a destination actor did not receive a packet of data from the actor on which this destination actor is dependent. In this case, a fault-tolerance procedure (see Section 4.5) is necessary to guarantee real-time services.

4.2 Hard-Real-Time Scheduling of CSDF

We present in this section an extended hard-real-time scheduling (RTS) algorithm for the timed graph, which is proven to meet the precedence constraints [8, 37] introduced by the Late Schedule in Section 4.3 and could obtain the maximum throughput achieved under self-timed scheduling, as evaluated in Section 4.6.

As presented in Chapter 3, a *Static Periodic Schedule* [90] of a Cyclo-Static Dataflow graph is a schedule such that, $\forall a_i \in A$:

$$s(i,k) = s(i,0) + \alpha \times k, \tag{4.1}$$

where $s(i,k)$ represents the time at which the k-th iteration of actor a_i is fired and α is an equal iteration period for every complete repetition of all the actors. The authors in [8] proved that it is possible to schedule a graph G actors as static periodic tasks using periods given by the following equation:

$$\alpha = q_1\lambda_1 = q_2\lambda_2 = \cdots = q_{n-1}\lambda_{n-1} = q_n\lambda_n, \tag{4.2}$$

where $q_i \in \overrightarrow{q}$ (The basic repetition vector of G) and $\lambda_i \in \overrightarrow{\lambda}$ (The minimum period vector of G), given by:

$$\lambda_i^{min} = \frac{Q}{q_i}\left\lceil \frac{\eta}{Q} \right\rceil \quad for \;\; a_i \in A, \tag{4.3}$$

where $\eta = \max_{a_i \in A}(\omega_i q_i)$ and $Q = lcm(q_1, q_2, \ldots, q_n)$ (lcm denotes the least common multiple operator).

However, in a real-time applications, temporal constraints are usually imposed by the control engineers or by electronic devices. For instance, an audio output sink should not experience any hiccups due to the aperiodic behavior caused by either the initial transition phase of the STS or by the variation of execution times from iteration to iteration. In this case, a throughput constraint could be imposed for the sink node (*i.e.* terminal actor) by the programmer. This constraint could be converted into a periodic constraint for the sink node. Moreover, the control engineer in the domain of avionics and automotive sector usually impose real-time constraints on actors for safety requirements or hardware features. In this case, we take care of these real-time constraints by defining:

$$\eta = \max_{a_i \in A}(\omega_i^* q_i, T_i^* q_i) \tag{4.4}$$

where T_i^* is the period imposed by the control engineer and $\omega_i^* = \omega_i + \varphi_i + \Delta_{clock}$ where φ_i is the communication time from a_i to its successors and Δ_{clock} is the sum of the maximum clock offset between 2 consecutive distributed nodes. In fact, the scheduler and latency has to take into account that the local clocks are not perfectly synchronized and that communication between distributed nodes can take a notable amount of time. For the communication cost, the mean delay-time that can be experienced when accessing on-chip shared memory used for interprocessor communication in a MPPA's cluster is evaluated in [83]. It is quite a general result that can also fit the STHORM chip with a little adaptation. As a result, the minimum period of each actor is given by:

$$T_i^{min} = \frac{\eta}{q_i} \quad for \;\; a_i \in A \tag{4.5}$$

where $q_i \in \overrightarrow{q}$ is the repetition of a_i.

However, this period does not mean that all actors in a CSDF graph would be executed periodically. For a long time, self-timed scheduling was considered the most appropriate policy for streaming applications modeled as dataflow graphs. Our approach allows subgraph of a CSDF graph, without real-time constraints, could be executed in a self-timed mode. This subgraph could be seen as a single CSDF actor, with only a time constraint for all actors in the subgraph.

The earliest start time under the strictly periodic schedule is given by:

$$s_j = \begin{cases} 0 & ,if \ \mathbf{prec}(a_j) = \emptyset \\ \max\limits_{a_i \in \mathbf{prec}(a_j)} (s_{i\to j}) & ,if \ \mathbf{prec}(a_j) \neq \emptyset \end{cases} \tag{4.6}$$

where $\mathbf{prec}(a_i)$ is the predecessors set of a_i

$$\mathbf{prec}(a_i) = \{a_j \in A : \exists e_u = (a_j, a_i) \in E\} \tag{4.7}$$

and

$$s_{i\to j} = \min_{t \in [0, s_i+\alpha]} \{t : \underset{[s_i, \max(s_i,t)+\alpha)}{\mathbf{prd}} (a_i, e_u) \geq \underset{[t, \max(s_i,t)+k]}{\mathbf{cns}} (a_j, e_u) \ \forall k = 0, 1, \ldots, \alpha\} \tag{4.8}$$

where $\underset{[t_s,t_e)}{\mathbf{prd}}(a_i, e_u)$ (or $\underset{[t_s,t_e]}{\mathbf{cns}}(a_i, e_u)$) is the sum of the number of tokens produced (consumed) by an actor a_i into a channel e_u during the interval $[t_s, t_e)$

4.3 Actor Dependence Function

This section defines an Actor Dependence Function (*ADF*) that describes the dependence between the execution of 2 actors connected by a directed path in a stream graph. Our approach is to construct a *Late Schedule*, which represents the execution order between actors in a CSDF graph. The hard-real-time scheduling introduced in Section 4.2 is proven to meet these precedence constraints [8, 37].

4.3.1 Definition

We say that the *upstream* actor is at the start of the path, while the *downstream* actor is at the end. Dependences between parallel actors fall outside the scope of this report but could be discussed in future work.

Definition 13 *Let $a_i, a_j \in A$ be two actors of a Timed graph $G = \langle A, E, \omega, \varphi \rangle$ on a path $p_{i \rightsquigarrow j}$ connecting a_i to a_j. We say that the $k-$th firing of a_j is dependent on the $n-$th firing of a_i if n is the last firing of a_i which produces at least one token for the $k-$th firing of a_j.*

Informally, $ADF_{A \leftarrow B}(n)$ gives the list of B's execution which depends on the nth execution of A. This dependence is meaningful only if A is upstream of B, otherwise ADF assumes an empty set. A formal definition of *ADF* using the notations introduced above is as follows:

Definition 14 $ADF_{A \leftarrow B}(n) = \underset{\phi \in \Phi}{maxList}(|\phi \wedge A(n)| \wedge \{B\})$

where ϕ is an ordered sequence of actor firings of a dataflow graph and Φ denotes the set of all legal schedules. Each firing represents the execution of a single *sub-task* of the actor. $\phi[i]$ is the ith actor appearing in sequence ϕ. Let $|\phi \wedge A(n)|$ denotes the list of actors between the nth and $(n+1)$th execution of A in ϕ and $|\phi \wedge A(n)| \wedge \{B\}$ denotes the list of B's firings in this list.

The Definition 14 reads: over all legal execution in which A fire $(n+1)$ times, $ADF_{A \leftarrow B}(n)$ is the list of execution of B between the *nth* and the $(n+1)$th firing of A.

4.3.2 Calculating ADF

Our approach is to construct an execution ϕ that provides the maximum execution of the downstream actor, which also means that the downstream actor have to use the "best" of its source or fires its sources only when it is necessary. We construct ϕ by using a *Late Schedule* with respect to actor B as can be seen in Algorithm 3.

Algorithm 3 LATE-SCHEDULE(X,n)

```
1: //Returns a Late Schedule for n executions of X where its predecessors are fired as late as
   possible
2: lateSchedule(X,n) {
3: φ = {}
4: for i = 1 to n do
5:     //execute predecessors of X only when X can not execute and until X can fire
6:     for all input channels c_i of X do
7:         while X need more tokens on c_i in order to fire do
8:             //extend schedule (⊕ denotes concatenation)
9:             φ = φ ⊕ lateSchedule(source(c_i, 1)
10:        end while
11:    end for
12:    //add X to schedule
13:    φ = φ ⊕ X
14:    //update number of tokens on I/O channels of X
15:    simulateExecution(X)
16: end for
17: return φ }
```

This schedule is obtained by calculating the demand for data items on the input channels of X, and firing its predecessors only when X can not fire. This schedule is then propagated back through the stream graph via scheduling actors connected to X. Some stream graphs admit multiple *Late Schedules*, as actors might be connected to multiple inputs that can be scheduled in any orders. However, the set of dependent actors remains constant even when the order changes.

The following theorems allow us to use the *Late Schedule* to calculate the ADF function:

Theorem 3 $ADF_{A \leftarrow B}(n) = |lateSchedule(B, q_B) \wedge A(n)| \wedge \{B\}$ *with* $1 \leqslant n \leqslant q_A$

where q_A and q_B are the number of times A and B have to fire to return the graph to the initial state (the number of items on each channel after the execution is the same as it was before the execution). In other words, q_A and q_B are elements of the repetition vector of the Timed graph.

Proof 1 *$lateSchedule(B, q_B)$ gives a steady state of the stream graph and in this steady schedule, each predecessor of B is fired only when it is necessary or as few times as possible to fire B as many times as possible. In addition, we have to analyze the dependence only in this steady state because all other schedules are repetitions of this steady state.*

4.3.3 Illustrative example

Figure 4.4 illustrates an example of *ADF* calculation for the CSDF version of an MP3 application depicted in Figure 4.3. In this application, the compressed audio is decoded by the *mp3* task into an audio sample stream. These samples are converted by the Sample Rate Converter (SRC) task, after which the Audio Post-Processing (APP) task enhances the perceived quality of the audio stream and sends the samples to the Digital to Analogue Converter (DAC). This CSDF graph is characterized by two repetition vectors $\vec{r} = [1, 2, 2, 4]$ and $\vec{q} = [3, 4, 4, 8]$. $\vec{q}$ is also the number of executions to obtain a steady state of this graph: *mp3* has to fire 3 times, *src1* and *src2* 4 times and *app* 8 times.

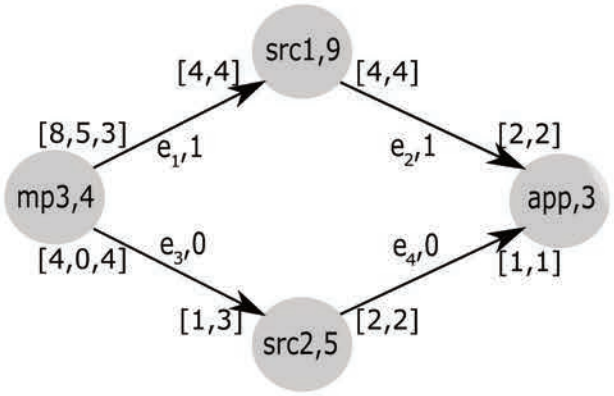


Figure 4.3: Example MP3 application. Nodes are annotated with their I/O rates. For example, node *mp3* has 3 sub-tasks: during the first phase, it produces 8 tokens on its upper channel and 4 tokens on its lower channel. During the second phase, its produces only 5 tokens on its upper channel and so on. The worst-case computation of each actor is shown next to its name after a comma. The communication time of each channel is shown next to its name and is attached to the production actor.

The function $ADF_{mp3 \leftarrow app}(n)$ tries to find the dependence between each firing of *mp3* and *app* in a steady state of this CSDF application. The results of Late scheduling and *ADF* calculations appear in Figure 4.4 and are summarized in Table 4.1. This table can be interpreted as follows: the 1st firing of *app* depends on the 1st of its predecessors: *mp3*, *src1*, *src2*. The 7th firing of this agent depends on the 3rd firing of *mp3* and the 4th firing of *src1* and *src2*.

Table 4.1: Dependence between actor's executions in the MP3 application

mp3	**src1**	**src2**	**app**
1	1	1	1
			2
	2	2	3
			4
2	3		
3		3	5
			6
	4	4	7
			8

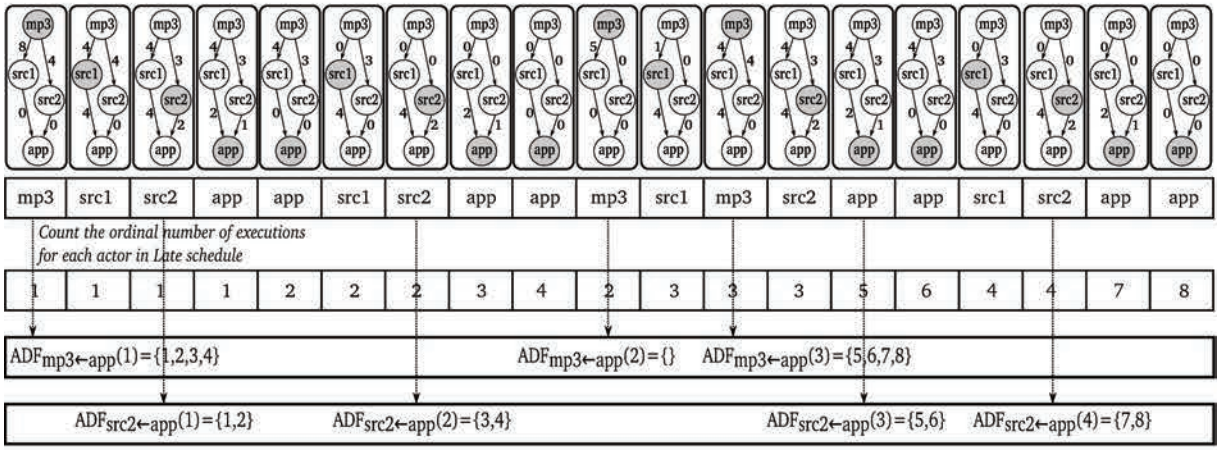


Figure 4.4: Example of Late scheduling and *ADF* calculation for the stream graph in Figure 4.3. The stream graphs illustrate a steady state cycles of a Late schedule; execution proceeds from left to right, and channels are annotated with the number of tokens present. The second line lists the actors that fire in a Late schedule for *app*. The third line represents the ordinal numbers of each actor in a steady state. The maximum ordinal number of each actor is its corresponding element in the repetition vector. The fourth line illustrates the computation of $ADF_{mp3 \leftarrow app}(n)$: the list of *app*'s execution which depends on the nth execution of $mp3$. The last line illustrates the computation of $ADF_{src2 \leftarrow app}(n)$.

4.4 Latency Analysis

Latency is the time delay between the moment when a stimulus occurs and the moment when its effect begins or ends. In timed CSDF, stimuli are actor firings and their effects are the consumptions of produced tokens by some other actors.

4.4.1 Definition

Definition 15 *Let $a_i, a_j \in A$ be two actors of a Timed graph G. The $k-th$ latency of a_i and a_j for an execution ϕ is defined as the time interval between the $k-th$ firing of a_j and its corresponding firing of a_i in ϕ, and is denoted by $L_k^{\phi}(a_i, a_j, k)$:*

$$L^{\phi}(a_i, a_j, k) = s^{\phi}_{a_j,k} - s^{\phi}_{a_i, ADF^{-1}_{a_i \leftarrow a_j}(k)} \tag{4.9}$$

where $s^{\phi}_{a_j,k}$ represents the start time of the $k-th$ firing of actor a_j in execution ϕ and $ADF^{-1}_{a_i \leftarrow a_j}(k)$ gives the execution of a_i on which the $k-th$ execution of a_j depends (i.e. the last firing of a_i which produces at least one token for the $k-th$ firing of a_j). This definition is consistent with and generalizes the definition of latency given for HSDF in [99] and SDF in [51]. It takes into account the fault-tolerant requirements of industrial control applications (e.g. the DO-178 standard for avionics software systems).

4.4.2 Latency Analysis under a hard-real-time scheduling

Let s_i be the earliest start time of an actor $a_i \in A$. According to Definition 15, the latency between two actors, while considering the communication time between distributed nodes, is given by:

$$L(a_i, a_j, k) = s_j + (k-1)T_j - (s_i + (ADF^{-1}_{a_i \leftarrow a_j}(k) - 1)T_i) \tag{4.10}$$

Example 5 *We illustrate in Figure 4.5 the hard-real-time scheduling for the MP3 application shown in Figure 4.3 with time constraints* $T_1 = 24$ *imposed by the control engineer. This application has an execution vector* $\overrightarrow{\omega} = [4, 9, 5, 3]^T$ *and a communication vector* $\overrightarrow{\varphi}$ *approximately equal to* $\overrightarrow{\varphi} = [1, 1, 0, 0]^T$*. Given* $\overrightarrow{q} = [3, 4, 4, 8]^T$ *and* $\overrightarrow{r} = [1, 2, 2, 4]^T$*, we have a period of all levels* $\alpha = 72$ *and the minimum period vector* $\overrightarrow{T} = [24, 18, 18, 9]^T$*. Using Equation (4.6), we have the earliest start time vector* $\overrightarrow{S} = [0, 29, 28, 39]^T$*.*

Finally, using Equation (4.10), we have $L(mp3, app, 1) = 39$*,* $L(mp3, app, 2) = 48$ *and* $L(mp3, app, 5) = 27$*, etc. as depicted in Figure 4.5.*

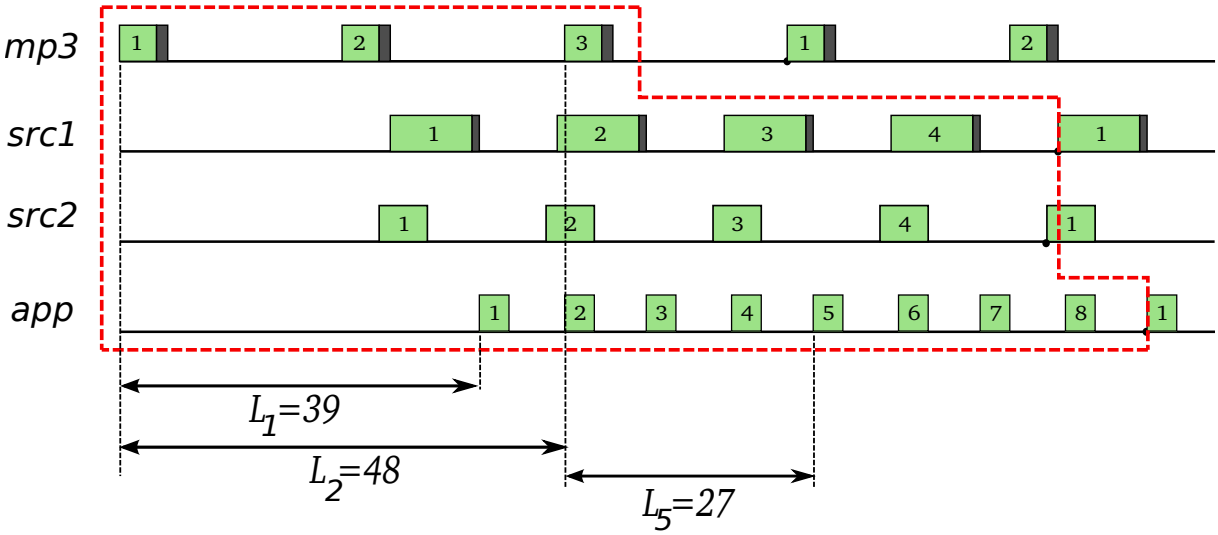


Figure 4.5: Example of hard-real-time scheduling for the MP3 application with time constraints imposed by the control engineer or hardware devices

4.5 Fault-Tolerance

In this section, we present a latency-based approach for fault-tolerant stream processing modeled as a CSDF graph in the face of node failures or network failures. Our approach aims to reduce the degree of inconsistency in the system while guaranteeing that available inputs capable of being processed are processed within a specified latency constraint.

4.5.1 Data Model

To accommodate our new token semantics, we adopt and extend the CSDF data model. In CSDF, an atomic piece of data carried out by a channel is called a *token*. CSDF allows the number of tokens to vary from one execution of the actor to the other, in a cyclic way, *i.e.* after a given number of firing each channel produce the same amount of tokens again. These tokens form, for each firing, a *tuple*. We extend the notion of tuple by adding the *tuple_id*, which is the ID of the source node between two dependent actors. This *tuple_id* will be repeat after a steady state. In our fault-tolerant model, a tuple between actors takes now the following form:

$$(\text{tuple\_id}, a_1, a_2, \ldots, a_m)$$

4.5.2 Support for fault-tolerance

The aerospace and automotive industries offer many examples of systems whose failure may have unacceptable costs (financial, human or both). In the future when many-core chips will be mainstream, the design of these hard real-time systems is made even more challenging by the requirement of making them resilient to faults. Our designed goal is to ensure, for each node, that any tuple on an input stream is processed within a specified time bound. In the case of a latency violation or non-receipt of the exact number of expected tokens, two solutions are possible: 1) firing a *redundancy* of the source node or 2) *suspend* processing until the tuple with the expected ID is received. Each node implements the state machine shown in Figure 4.6 that has three states: NORMAL, FAILURE and STABILIZATION.

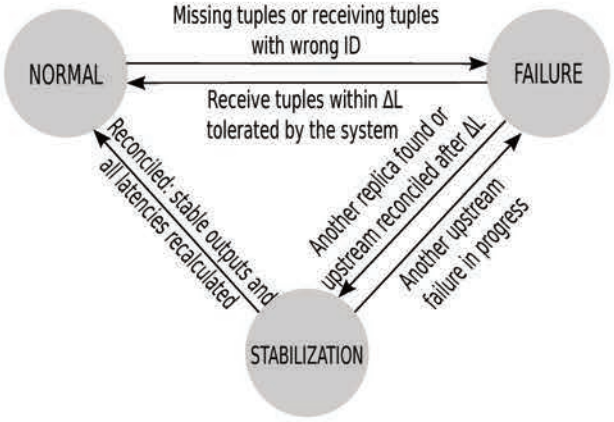


Figure 4.6: State Machine of the fault-tolerant procedure

As long as all predecessor neighbors of a node are producing their expected number of tokens, the node is in the NORMAL state. In this state, it processes tokens as they arrive and passes stable results to downstream neighbors.

If one input is missing after a latency constraint derived in Section 4.4.2, or when receiving tuples without the expected ID calculated by using the ADF function, a node goes into the FAILURE state, where it tries to find a redundancy for the input stream. Hence, we introduce the concept of *dormant* actor: unlike active actors, dormant actors do not fire right away when enabled but wait until they detect the violation of a latency constraint. It is up to the designer to select the relevant nodes to replicate, using dormant actors to implement redundancy.

If no such source is available, the node suspends processing until receiving the exact token from the neighbor predecessors. The distinguishing feature of actors in terms of fault tolerance is their idempotent and deterministic nature: an actor presented with the same tokens will always produce the same result. Thus, fault-tolerance, in the case of missing token (*e.g.* because of network failure and no redundancy), can be achieved by replaying the predecessor nodes in the steady state of a CSDF graph. This approach works for latency constraint violations and transient faults. For permanent faults, the designer is expected to rely on redundancy.

Once a node succeeds to find a redundancy or have the expected token from the replayed nodes, it transitions into the STABILIZATION state. In this state, if there is a change in the CSDF because of a *dormant* actor or replayed nodes, all actor dependence

and latency constraints between affected actors have to be recalculated.

4.6 Evaluation results

We evaluate our proposed scheduling policy in Section 4.2 by performing an experiment on a set of 12 real-life streaming applications. The objective of the experiment is to compare the efficiency of our approach to the maximum achievable throughput obtained via self-timed scheduling.

4.6.1 Benchmarks

We used benchmarks from different domains (*e.g.* signal processing, video processing, mathematics, *etc.*) and from different sources to check the efficiency of this scheduling in different architectures. The first source is the ΣC benchmark [52] which contributes 5 streaming applications. The second source is the SDF^3 benchmark which contributes 6 streaming applications [102]. The last source is the StreamIt benchmark [106]. In total, 12 applications are considered as shown in Table 4.2. The graphs are a mixture of CSDF (ΣC's applications) and SDF (StreamIt and SDF^3 benchmark) graphs. The use of synchronous dataflow (SDF) models does not affect our scheduling policy because SDF, with static firing rules of actors, is a special case of CSDF model [19, 74]. The fourth column (N) shows the number of actors in each application, the fifth column (Q) shows the least-common-multiple of the repetition vector elements (*i.e.* $Q = lcm(q_1, q_2, \ldots, q_n)$) and the sixth column is $\eta = \max_{a_i \in A}(\omega_i^* q_i, T_i^* q_i)$ used to calculate the minimum period according to Formula 4.5.

Table 4.2: Benchmarks used for evaluation

Domain	No.	Application	N	Q	η	Source
Signal Processing	1	Discrete cosine Transform (DCT)	4	12	1800	CEA LIST
	2	Fast Fourier Transform (FFT) kernel	4	6	900	
	3	Multi-Channel beamformer	4	12	7800	
Audio Processing	4	Filter bank for multirate signal processing	17	600	113430	MIT [106]
	5	Sample-rate converter used in CDs	6	23520	960	*SDF*^{3} [102]
	6	MP3 audio decoder	6	24	36	CEA LIST
Video Processing	7	Motion detection	9	1	57232627	
	8	H.263 video encoder	5	33	382000	*SDF*^{3} [102]
	9	H.263 video decoder	4	2376	10000	
Mathematics	10	Bipartite graph	4	144	252	
Communication	11	Satellite receiver	22	5280	1056	
	12	Modem	16	16	16	

The actors execution times of the ΣC benchmark are measured in clock cycles on the MPPA 256 cores, while the actors execution times of the SDF^3 benchmark are specified by its authors for ARM architecture. For the StreamIt benchmark, the actors execution times are specified in clock cycles measured on MIT RAW architecture. This fact shows clearly the portability of our scheduling policy to be applied for different architectures.

4.6.2 Experiment: Throughput comparison

We use SDF^3 tool-set for several purposes during the experiments. SDF^3 is a powerful analysis tool-set which is capable of analyzing (C)SDF graphs to check for consistency errors, compute the repetition vector, compute the maximum achievable throughput, *etc.*

In this experiment, we compare the throughput resulting from our scheduling approach to the maximum achievable throughput of a streaming application obtained via self-timed scheduling. Computing the throughput of the STS using SDF^3 is done using the algorithm *throughput* of the $sdf3analysis - (c)sdf$ tool. For StreamIt benchmark, the graph exported by this language is converted in the XML required by SDF^3. For ΣC applications, the ΣC compiler is capable of checking consistency errors and computing the repetition vector during its 4 stages of compilation.

We measure the throughput of the actors which produces the output stream, *i.e.* the sink actor under the hard-real-time sheduling (RTS), given by:

$$\Upsilon_{snk}^{RTS} = \frac{1}{T_{snk}^{\min}} \tag{4.11}$$

The throughput of the sink actor in the self-timed scheduling:

$$\Upsilon_{snk}^{STS} = q_{snk} \times \Upsilon_{G}^{STS} \tag{4.12}$$

where Υ_{G}^{STS} is the graph throughput under STS, measured by using SDF^3. It should be noted that the unit for the results in Table 4.3 is $\frac{1}{clock\ cycle}$. A result in seconds could be obtained by dividing these results by the value of one cycle (*e.g.* $2.5 \times 10^{-9}s$ for MPPA Andey 256 cores or $1.67 \times 10^{-9}s$ and $1.25 \times 10^{-9}s$ for its Bostan version).

Table 4.3: Results of Throughput comparison

Application	$\mathbf{\Upsilon_{snk}^{STS}}$	$\mathbf{\Upsilon_{snk}^{RTS}}$	$\mathbf{\Upsilon_{snk}^{STS} / \Upsilon_{snk}^{RTS}}$
DCT	2.22×10^{-3}	4/1800	1.0
FFT	3.33×10^{-3}	3/900	1.0
Beamformer	5.13×10^{-4}	4/7800	1.0
Filterbank	2.64×10^{-5}	3/113430	1.0
Sample-rate	1.67×10^{-1}	1/6	1.0
MP3	2.22×10^{-1}	8/36	1.0
Motion detection	1.74726×10^{-8}	1/57232627	1.0
Encoder	4.73×10^{-6}	1/382000	1.8
Decoder	1.0×10^{-4}	1/10000	1.0
Bipartite	6.35×10^{-2}	16/252	1.0
Satellite	9.46×10^{-4}	1/1056	1.0
Modem	6.25×10^{-2}	1/16	1.0

The last column of Table 4.3 shows the ratio of the STS schedule throughput to our scheduling approach throughput ($\Upsilon_{snk}^{STS} / \Upsilon_{snk}^{RTS}$). We clearly see that our approach delivers the same throughput as STS for 11 out of 12 applications. The only application have lower throughput is H.263 Encoder but overall, we demonstrated good results while ensuring high level of time determinism, as required in hard real-time systems. In comparison with the results in Chapter 3, our improved scheduling policy delivers also the same throughput and succeeds to introduce periodic constraint for the MP3 and Motion detection applications

4.7 Summary

In this chapter, we present an analytical framework for computing the periodic task parameters for the actors of a streaming application, modeled as an acyclic CSDF graph such that a strictly periodic schedule exists. As a result, a variety of hard real-time scheduling algorithms for periodic tasks can be applied to schedule such applications while considering variable interprocessor communication and real-time constraints imposed by hardware devices or control engineers. Based on empirical evaluations, we show that our real-time scheduling approach delivers the maximum throughput achieved under the STS model. Based on this, we evaluate the latency between initiation times of any two dependent actors, and we introduce also a latency-based approach for fault-tolerant stream processing modeled as a CSDF graph, addressing the problem of node or network failures. We view this work as an important step to provide a failure-handling strategy for distributed real-time streaming applications based on static decidable dataflow models (e.g., CSDF or SDF). However, complex signal and media processing applications, such as cognitive radio or modern video codecs, often display dynamic behaviors that do not fit the classical static models' restrictions. As a result, in the next chapter, we introduce an extension of the CSDF model and demonstrate how this new approach tackles the limitations of static models while always allowing time constraints enforcement, failure-handling strategy and static analyses (i.e., consistency, liveness and boundedness) as in decidable models.

Chapter 5

Transaction Parameterized Dataflow

Obviously, simulation and testing may pinpoint some errors of this kind. It is well known, however, that testing is efficient only in the first steps of a design, and that formal methods are necessary to find the last bugs.

— Paul Feautrier

Contents

5.1 Model of Computation . . . 78
5.2 (max, +) Algebraic Semantics of TPDF . . . 80
5.3 Static Analyses . . . 82
5.3.1 Rate consistency . . . 82
5.3.2 Boundedness . . . 83
5.3.3 Liveness . . . 84
5.3.4 Throughput Analysis . . . 86
5.3.5 Scheduling . . . 88
5.4 Summary . . . 89

In the last chapter, we demonstrate that it is possible to schedule decidable dataflow graphs (e.g., SDF or CSDF) as periodic tasks while considering variable interprocessor communication and real-time constraints imposed by devices or control engineers. These dataflow models are also useful for their predictability, formal abstraction, and amenability to powerful optimization techniques. However, for signal processing applications, it is not always possible to represent all of the functionality in terms of purely decidable dataflow representations; typical challenges include variable data rate processing, multi-standard or multi-mode signal processing operation, and data-dependent forms of adaptive signal processing behavior. For this reason, numerous dynamic dataflow modeling techniques—whose behavior is characterized by dynamic variations in resource requirements—have been proposed. In many of these, in exchange for the increased modeling flexibility (high expressive power) provided by the underlying techniques, one must give up guarantees on compile-time buffer underflow (deadlock), overflow validation (boundedness) or performance analysis (e.g., throughput).

In this chapter, we introduce a new dynamic MoC, called *Transaction Parameterized Dataflow* (TPDF), allowing variable production/consumption rates and dynamic changes

of the graph topology. TPDF is designed to be statically analyzable regarding the essential deadlock and boundedness properties, while avoiding the aforementioned restrictions of usual decidable dataflow models.

The remainder of this chapter is organised as follows. In Section Section 5.1, we introduce our new dynamic model TPDF, as a parametric extension of Cyclo-Static Dataflow (CSDF). Section 5.3 presents the static analyses for liveness, boundedness, worst-case throughput and a scheduling heuristic for this model, which is illustrated and evaluated in Chapter 7 by different real-life case studies.

5.1 Model of Computation

In this work, we choose Cyclo-Static Dataflow (CSDF) [19] as the base model for TPDF because it is deterministic and allows for checking conditions such as deadlocks and bounded memory execution at compile/design time, which is usually not possible for Dynamic Dataflow (DDF).

We extend CSDF by allowing rates to be *parametric* and a new type of *control* actor, channel and port. For a compact formal notations, we assume that kernels, which play the same role as computation units (actors) as in CSDF, have at most *one* control port. Kernels without control ports are considered to always operate in a *dataflow* way, i.e., a kernel starts its firings only when there is enough data tokens on all of its data input ports.

Definition 16 *A TPDF graph $\mathcal{G}$ is defined by a tuple (K, G, E, P, R_k, R_g, α, ϕ^*) where:*

- *K is a non-empty finite set of kernels and G is a finite set of control actors such that $K \cap G = \varnothing$. For each kernel $k \in K$, M_k denotes the set of modes indicated by the control node connected to its unique control port c. The following modes are available within a TPDF graph:*
  - *Mode 1: Select one of the data inputs (outputs)*
  - *Mode 2: Select more than one data input (output)*
  - *Mode 3: Select available data input with the highest priority*
  - *Mode 4: Wait until all data inputs are available*

  In this context, the effect of control tokens can be also described as selecting data input and output ports besides choosing modes. Indeed, at a given time, the input and output ports of an edge may be in a different state. However, it does not affect the firings of kernels or control actors, only the data tokens that are chosen or rejected.
- *$E \in O \times (I \cup C)$ is a set of directed channels, where I, C, O is the union of all input, control and output port sets respectively. $E_c = E \setminus (O \times I)$ is the set of all control channels. A control channel can start only from a control actor and is connected to a control port.*
- *P is a set of integer parameters.*
- *$R_k : M_k \times (I_k \cup C_k \cup O_k) \times \mathbb{N} \longrightarrow \mathbb{N}$ assigns the rate to the ports of the n-th firing of k for each mode. The rate $R_k(m, c, n) = \{0, 1\}$ for all modes $m \in M_k$ and for all firings of k.*
- *$R_g : (I_g \cup C_g \cup O_g) \times \mathbb{N} \longrightarrow \mathbb{N}$ assigns the rate to each port of the n-th firing of a control actor g.*
- *$\alpha : (I \cup C \cup O) \longrightarrow \mathbb{N}$ returns for each port its priority.*
- *$\phi^* : E \longrightarrow \mathbb{N}$ is the initial channel status.*

In this paper, we assume that a kernel $k \in K$ waits until its control port becomes available to be fired by reading one token from this port. This token defines in which mode $m \in M_k$ in which k will operate. One of the most important property of TPDF, which is different from SDF and CSDF, it is that a kernel or a control actor does not have to wait until sufficient tokens are available at every data input port. This new property allows the capacity of reconfiguration of graphs depending on context and time.

The n-th firing of a control actor $g \in G$ starts by waiting until $R_g(i,n)$ and $R_g(c,n)$ tokens are available at every input $i \in I_g$ and $c \in C_g$, where $R_g(c,n) = \{0,1\}$. After performing its actions, the n-th firing of g ends by removing the $R_g(i,n)$ and $R_g(c,n)$ tokens from its input data and control ports and writing $R_g(o,n)$ tokens to each output control port $o \in O_g$.

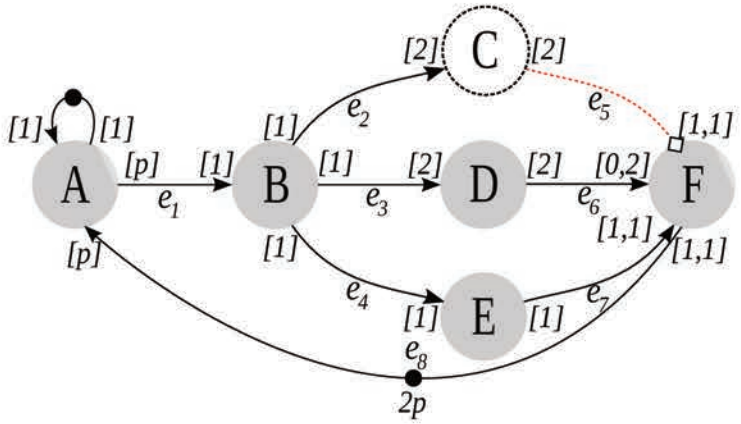


Figure 5.1: Example of a TPDF graph with integer parameter p, control actor C and control channel e_5.

Example 6 *Figure 5.1 shows a simple TPDF graph where actors have constant or parametric rates (e.g., p for the output rate of A). The repetition vector is* $[2, 2p, p, p, 2p, 2p]$. *C is a control actor and* e_5 *is a control channel. A sample execution of the graph is the following: A fires and produces p tokens on* e_1. *Then B fires and produces one token on edge* e_2, e_3, e_4. *Only E can fire because there are enough tokens on its input edge and produce one token on edge* e_7. *B (and A if necessary) will fire a second time and produce another token on edge* e_2, e_3, e_4. *Then C, D and E will fire and produce* 2, 2 *and* 1 *token, respectively, on edge* e_5, e_6, e_7. *Finally, F fires two times, each time it consumes one token from its control port. This token determines in which mode F will be fired. In this case, F can choose two tokens from* e_6 *or one from* e_7 *and remove remaining tokens. This continues until each actor has fired a number of times equal to its repetition count.*

In TPDF, we define 2 new data distribution kernels *Select-duplicate*, *Transaction Box* [85] and a new type of control clock in a dataflow way.

Select-duplicate kernels with *one* entry and n outputs (n is a maximum number for automatic sizing). At a given time any combination of the n outputs can be enabled. Each time a data token is read on the input line, it is copied on this combination of the n output channels.

Transaction symmetric processes with n inputs and *one* output. Its role is to atomically select a predefined number of tokens from one or several of its input to its output. By using special modes predefined by TPDF and combining with a control actor, the Transaction process implements important actions not available in usual dataflow MoC: Speculation, Redundancy with vote, Highest priority at a given deadline, Selection of an active data-path among several [85].

Clock can be considered as a watchdog timer with control tokens sent each time there is a timing out. The kernel which receives this time token will be awakened and fired immediately. In this way, TPDF can be applied to model streaming applications with time constraints, as can be seen in Section 6.4.2 and 7.1.1.

5.2 (max, +) Algebraic Semantics of TPDF

We use (max, +) algebra [6, 48] to capture the semantics of modes introduced by TPDF graphs. In fact, this MoC can be considered as a dynamic switching between cases (each case is one graph iteration and can consist of different modes), each of which is captured by a CSDF graph with two fundamental characteristics of its self-timed execution: *synchronisation* (the *max* operator), i.e. when the graph (in a specific mode of TPDF) waits for sufficient input tokens to start its execution, and *delay* (the + operator), i.e. when an actor starts firing it takes a fixed amount of time before it completes and produces its output tokens.

We briefly introduce some basic concepts of (max, +) (see [6] for background on (max, +) algebra, linear system theory of the (max, +) semiring). (max, +) algebra defines the operations of the operations of the maximum of numbers and addition over the set $\mathbb{R}_{\max} = \mathbb{R} \cup \{-\infty\}$, where $\mathbb{R}$ is the set of real numbers. Let $a \oplus b = \max(a, b)$ and $a \otimes b = a + b$ for $a, b \in \mathbb{R}_{\max}$. For scalars x and y, $x \cdot y$ (with short hand xy denotes ordinary multiplication), not the (max, +) $\otimes$ operator. For $a \in R_{\max}$, $-\infty \oplus a = a \oplus -\infty = a$ and $a \otimes -\infty = -\infty \otimes a = -\infty$. By (max, +) algebra we understand the analogue of linear algebra developed for the pair of operations $(\oplus, \otimes)$ extended to matrices and vectors. The set of n dimensional (max, +) vectors is denoted $\mathbb{R}^n_{\max}$ while $\mathbb{R}^{n\times n}_{\max}$ denotes the set of $n \times n$ (max, +) matrices. The sum of matrices $A, B \in \mathbb{R}^{n\times n}_{\max}$, denoted by A $\oplus$ B is defined by $[A \oplus B]_{ij} = a_{ij} \oplus b_{ij}$ while the matrix product $A \otimes B$ is defined by $[A \otimes B]_{ij} = \bigoplus_{k=1}^{n} a_{ik} \otimes b_{kj}$. For $a \in \mathbb{R}^n_{\max}$, $||a||$ denotes the vector norm, defined as $||a|| = \bigoplus_{i=1}^{n} a_i = \max_i a_i$ (i.e., the maximum element). For a vector a with $||a|| > -\infty$, we use a^{norm} to denote $[a_i - ||a||]$. With $A \in \mathbb{R}^{n\times n}_{\max}$ and $c \in \mathbb{R}$, we use denotations $A \oplus c$ or $c \oplus A$ for $[a_{ij} + c]$. The $\otimes$ symbol in the exponent indicates a matrix power in (max, +) algebra (i.e., $c^{\otimes n} = c \cdot n$).

Within an iteration, numerous modes can be invoked. Each mode corresponds to a member of the execution sequence of the kernel which receives the control token (e.g., for the TPDF graph in Figure 5.1, with $p = 1$, F can be fired in two different modes within an iteration). Each combination of these modes forms a *case* of the TPDF graph, which can be represented by a CSDF graph. We record the production times of initial tokens after the $i-$th case CSDF iteration using the *time-stamp* vector λ_i consisting of as many entries as there are initial tokens in the graph (e.g., $(2p + 1)$ initial tokens in Figure 5.1). The relationship between the $i-$th and the $(i + 1)$-st case iteration is given by (5.1):

$$\lambda_{i+1} = M_{i+1} \otimes \lambda_i \tag{5.1}$$

M_i is the characteristic (max, +) matrix of case i. This matrix can be obtained by symbolic simulation of one iteration of the CSDF graph in case i. To illustrate, we use case 1 of Example 1 in Figure 5.1, where $p = 1$. The execution time vector of this graph is $[2, 2, 1, 5, 2, 1.5]$, respectively in the order A, B, C, D, E, F and two tokens from the control actor set the kernel F in mode 4 (i.e., wait until all data inputs are available). This graph has three initial tokens so $\lambda_i = [t_1, t_2, t_3]^T$. Entry t_k represents the time stamp of initial token k after the $i-$th case iteration. Initially, time-stamp t_1 corresponds to the symbolic time-stamp vector $[0, -\infty, -\infty]^T$, t_2 corresponds to the symbolic time-stamp vector $[-\infty, 0, -\infty]^T$ and finally t_3 to $[-\infty, -\infty, 0]^T$. We start by firing actor A consuming two tokens, one from the self edge and one from the edge from actor F, labelled t_1 and t_2 respectively. The tokens produced by A carry the symbolic time-stamp:

$$max([0, -\infty, -\infty]^T, [-\infty, 0, -\infty]^T) + 2 = [2, 2, -\infty]^T$$

which corresponds to the expression $max(t_1 + 2, t_2 + 2)$. The subsequent first firing of actor B with a duration of 2 consumes this token and produced output tokens labelled as:

$$max([2, 2, -\infty]^T) + 2 = [4, 4, -\infty]^T$$

If we continue the symbolic execution till the completion of the iteration, we obtain the symbolic time-stamp for the second firing of A $[4, 4, 2]^T$ which reproduces the token in the self edge for the next iteration. The tokens produced by the first and second firings of F in the back edge and reused by the next iteration has the time-stamp $[8.5, 8.5, 6.5]^T$ (by consuming t_2) and $[12.5, 12.5, 10.5]^T$ (by consuming t_3). If we collect the symbolic time-stamp vector of these new tokens into a new vector $\lambda'_i = [t'_1, t'_2, t'_3]^T$, we obtain the following (max, +) equation:

$$\begin{bmatrix} t'_1 \\ t'_2 \\ t'_3 \end{bmatrix} = \begin{bmatrix} 4 & 4 & 2 \\ 8.5 & 8.5 & 6.5 \\ 12.5 & 12.5 & 10.5 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \tag{5.2}$$

If we assume that all initial tokens are available from time 0, the first time-stamp is $\lambda_0 = [0, 0, 0]^T$. After one iteration in the case where both the first and second tokens from C set F in mode 4, the time-stamp of initial tokens becomes $\lambda_1 = [4, 8.5, 12.5]^T$.

If this case followed by another case where the TPDF graph works in the mode 4 (i.e., wait until all data available) for the first token of C and mode 1 (i.e., select input E) for the second token of C, we obtain the following matrix:

$$\begin{bmatrix} t'_1 \\ t'_2 \\ t'_3 \end{bmatrix} = \begin{bmatrix} 4 & 4 & 2 \\ 8.5 & 8.5 & 6.5 \\ 9.5 & 9.5 & 7.5 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \tag{5.3}$$

With the initial tokens $\lambda_1 = [4, 8.5, 12.5]^T$, after this case, the time-stamp becomes $\lambda_2 = [14.5, 19, 20]^T$.

After analyzing all cases individually, the theory of (max, +) automata is used to capture the dynamic semantics of modes introduced by TPDF graphs. The completion time of a sequence of cases $cs_1 cs_2 cs_3 \ldots cs_n$ is given by:

$$\lambda_{i+1} = M_{i+1} \otimes M_i \otimes \ldots M_1 \otimes \lambda_0 \quad \forall i \in [0, n-1] \tag{5.4}$$

A careful reader might have noticed that for the example in Figure 5.1, the initial token vectors between the cases where $p = 1$ and $p = 2$ are not in the same dimensions and therefore the matrix multiplication of (5.4) will not be well-defined. However, case matrices can be extended with entries 0 and $-\infty$ to accommodate the initial tokens of the entire case sequence. For example, if p has a maximum value of 2 in the case sequence, the case iteration has five initial tokens so $\lambda_i = [t_1, t_2, t_3, t_4, t_5]^T$ we can extend the matrix of case cs_1 (Equation (5.2)) to accommodate this case sequence and yield the following matrix:

$$M_1^{ext} = \begin{bmatrix} 4 & 4 & 2 & -\infty & -\infty \\ 8.5 & 8.5 & 6.5 & -\infty & -\infty \\ 12.5 & 12.5 & 10.5 & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix} \tag{5.5}$$

The synchronization data between two different cases is made through the common initial tokens between two consecutive iterations. For the synchronization between different values of parameters, we use the initial token labelling to model inter-case synchronization. Initial tokens are explicitly defined by their identifier (e.g., t_1). Two initial tokens of two different cases are common only if they share the same identifier. With this approach, if a case has five initial tokens ($p = 2$) is followed by a case which has only three initial tokens ($p = 1$), only time-stamp value of initial tokens with the same identifier will be selected by using the extended case matrix. In the opposite case, the missing values for the *(i+1)*-th are replaced by $||\lambda_i||$ (the production time of the last token produced by the last iteration).

5.3 Static Analyses

This section presents the three static analyses needed to ensure consistency, boundedness and liveness of TPDF graphs. In Section 5.3.1, we check rate consistency by adapting the analysis of CSDF to TPDF. Conditions for rate consistency and solutions of balanced equations are computed in terms of symbolic expressions. In Section 5.3.2, we check that, along with rate consistency, the TPDF MoC with control actor and parameter setting ensures boundedness. Section 5.3.3 completes the analysis chain by checking for liveness.

5.3.1 Rate consistency

As in SDF and CSDF, we check the rate consistency of a TPDF graph by generating the associated system of balance equations expressed in matrix-form as in Equation (5.6) and this system must be shown to have a non null solution for all possible values of parameters, all possible reconfigurations of the graph and all modes of the kernels.

$$\Gamma \cdot \overrightarrow{r} = 0 \tag{5.6}$$

and where the topology matrix Γ is defined by

$$\Gamma_{uj} = \begin{cases} X_j^u(\tau_j) & , \textit{if task } a_j \textit{ produces on edge } e_u \\ -Y_j^u(\tau_j) & , \textit{if task } a_j \textit{ consumes from edge } e_u \\ 0 & , \textit{otherwise} \end{cases} \tag{5.7}$$

The matrix is generated by considering the parametric rates and by ignoring all possible configurations of the graph. If the system is rate consistent when all edges are present, then it is also consistent when one or more several edges are removed. Indeed, when removing edges (i.e., tokens produced in this edge are not used and will be rejected), the resulting system of balance equations will be a subset of the system of equations of the fully connected graph. Checking rate consistency of all edges maybe considered too strict because it does not take into account the fact that some input edges may not be active in the same mode. However, it simplifies the understanding and implementation since the graph has a unique iteration vector.

The minimal solutions for all actors when solving the system of equations (5.6) are found by eliminating all the coefficients or parametric factors common to all solutions. Then, we arbitrarily set one of the solutions to 1 and recursively find other solutions. Finally, we normalize the solutions to integers. If the system of equations (5.6) has a non-trivial solution, the TPDF graph also satisfies the necessary and sufficient conditions for the existence of parametric solutions, introduced for parametric models such as SPDF [46] and BPDF [16].

Example 7 *For the graph of Figure 5.1, by setting* $r_A = 1$*, we consecutively get:*

$$r_B = p, r_C = p/2, r_D = p/2, r_E = p, r_F = p/2 \tag{5.8}$$

To normalize the fractional solutions we multiply by 2*. Finally, we get the vector* $\overrightarrow{r}$ *and also the repetition vector* $\overrightarrow{q}$ *by multiplying with the matrix* P*:*

$$\overrightarrow{r} = [2, 2p, p, p, 2p, p]\,, \overrightarrow{q} = [2, 2p, p, p, 2p, 2p] \tag{5.9}$$

and a possible valid static schedule for this graph is $A^2B^{2p}C^pD^pE^{2p}F^{2p} = A^2(B^2CDE^2F^2)^p$*.*

5.3.2 Boundedness

Without dynamic topology changes, rate consistency is sufficient to ensure that a graph returns to its initial state after each iteration and boundedness is guaranteed. However, in TPDF, a graph can change its topology within a valid static schedule by using the control actor. Yet, not all static schedules are safe and their boundedness must be checked. The criterion ensuring that reconfiguration by using control actor and several modes for one kernel are safe relies on the notion of *control area*. Intuitively, the criterion states that each control actor will be fired once per local iteration of the area it reconfigures.

Definition 17 *(Control area): The area of a control actor* $g \in G$*, noted* $Area(g)$*, is defined as:*

$$Area(g) = \boldsymbol{prec}(g) \cup \boldsymbol{succ}(g) \cup \boldsymbol{infl}(g) \tag{5.10}$$

where

$$\begin{aligned} \boldsymbol{succ}(g) &= \{a_i \in (K \cup G) : \exists e_u = (g, a_i) \in E\} \\ \boldsymbol{prec}(g) &= \{a_i \in (K \cup G) : \exists e_u = (a_i, g) \in E\} \end{aligned} \tag{5.11}$$

and $\boldsymbol{infl}(g) = (\boldsymbol{succ}(\boldsymbol{prec}(g)) \cap \boldsymbol{prec}(\boldsymbol{succ}(g))) \setminus \{g\}$ *is the list of actors between* $\boldsymbol{prec}(g)$ *and* $\boldsymbol{succ}(g)$*, influenced by* g*.*

The *control area* of g is the set containing its sources, kernels or controls that receive its control tokens and all other influenced actors between these actors.

Definition 18 *(Local Solution) The local solution of an actor (kernel or control actor) a_i in a subset of actors $Z = \{a_1, \ldots, a_n\}$, written $q^L_{a_i}$, is defined as:*

$$q^L_{a_i} = \frac{q_{a_i}}{q_{\mathcal{G}}(Z)} \tag{5.12}$$

where $q_{\mathcal{G}}(Z) = gcd(q_{a_i}/\tau_i)\ \forall a_i \in Z$ (gcd denotes the greatest common divisor). Local solutions can be considered as a repetition vector for a subset of actors.

Definition 19 *(Rate Safety): A TPDF graph is rate safe if and only if, for each control actor $g \in G$ and each actor $a \in Area(g)$, the consumption and production rates of these actors ensure that, during a local iteration of its area, a control actor will be fired only one time. This condition is guaranteed, if and only if, for each control actor $g \in G$ and for each actor $a_i \in \boldsymbol{prec}(g) \cup \boldsymbol{succ}(g)$, connected with g by the channel e_u:*

$$\begin{cases} X^u_g(1) = Y^u_i(q^L_{a_i}) & \text{if } g \text{ is the production actor} \\ Y^u_g(1) = X^u_i(q^L_{a_i}) & \text{if } g \text{ is the consumption actor} \end{cases} \tag{5.13}$$

Rate safety ensures that, during a local iteration of an area of a given control actor, the total number of tokens consumed (produced) on any edge connected with the control actor is sufficient for this actor to fire exactly one time. It is ensured by a simple syntactic check on TPDF graphs.

Example 8 *In Figure 5.1, $Area(C) = \{B, D, E, F\}$ and possible static schedules for this graph are $A^2B^{2p}C^pD^p\ E^{2p}F^{2p}$ and $A^2(B^2CDE^2F^2)^p$, where $B^2CDE^2F^2$ is a local solution for the $Area(C)$. C will be fired only one time for each iteration of this local area.*

Theorem 4 *(Boundedness) A rate consistent, safe and live TPDF graph returns to its initial state at the end of its iteration. Hence it can be scheduled in bounded memory.*

Proof 2 *There are several modes defined by a control actor as defined in Section 5.1. In fact, the Rate safety ensures that, during a local iteration of a control area, its influenced kernels receive synchronous control tokens, calculated by only one firing of the control actor and define in which mode this kernel will fire. For the modes that choose between the data inputs (e.g., Transaction), the dependence with kernels which produce these input tokens is not broken here because unchosen data input are considered only as not to be used. For the modes which choose between data outputs (e.g., Select-duplicate), we assume that there is a virtual control actor and a virtual kernel which chooses between data inputs, as in Figure 5.2.*

Rate consistency and safety were crucial to ensure that the graph returns to its initial state after an iteration. However, we assumed that actors can be fired in the right order to respect dataflow constraints. This holds only when the graph is live and the next section shows how it is checked.

5.3.3 Liveness

In SDF and CSDF, 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. For TPDF, the situation is more complex for two reasons:

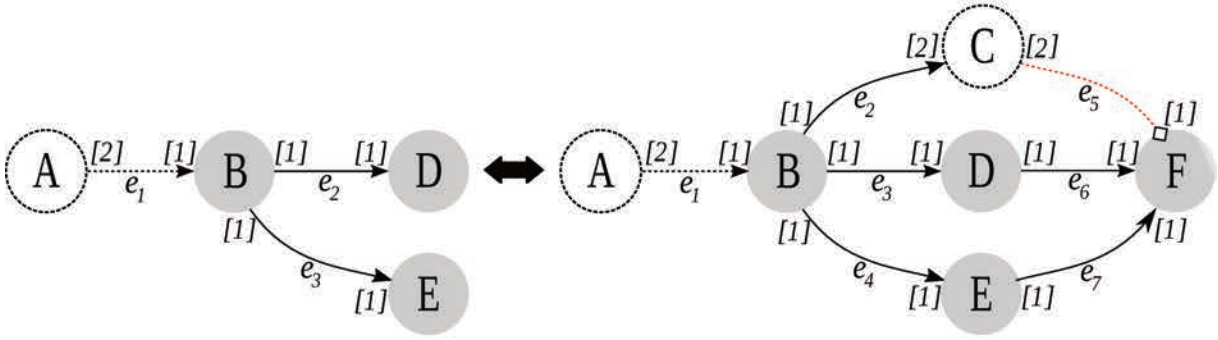


Figure 5.2: B is a Select-duplicate to choose between D and E. This graph is equivalent with the second graph by adding two virtual actors: C and F, which consume data outputs from C, D and E. When B must choose between its data outputs, it sends a signal token to C, this control actor will send a control token to F, which choose only the data-paths chosen by B. In this case, boundedness of the graph is guaranteed.

- First, actors may have to be fired a parametric number of times during an iteration. Finding a schedule may, in general, involve some inductive reasoning.
- Second, control tokens change the topology of the graph by the selection or removal of data tokens within the iteration. However, this selection does not introduce new constraints among firings of control and kernel actors. Then, the topology change of the graph is not a reason which may introduce deadlocks.

For the first challenge, a (C)SDF and TPDF graph deadlocks only if it contains at least one cycle. Equivalently, acyclic (C)SDF and TPDF are always live. In such case, the topological order of the DAG defines a single appearance schedule [14]. For general graph, we start with cutting all cycles, which contain a saturated edge. An edge $e_u = (a_i, a_j)$ is said to be saturated if it contains enough initial tokens to fire a_j q_j times or $\phi^*(e_u) \geq Y_j^u(q_j)$. Since this edge has at least the total number of tokens consumed by a_j, it does not introduce any constraint and can be ignored. If each cycle has a saturated edge, then the graph can be considered as acyclic and therefore live.

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 ordering of firings can be tested. We adapt this approach to TPDF by testing all schedules where each consecutive firings of an actor $(a_i)^k$ represents a non-parametric (integer) fractional of its number of firings in the iteration *i.e.*, $\exists c \in \mathbb{N}, q_{a_i} = c \cdot k$. By considering only occurrences of the form $(a_i)^k$ with $q_{a_i} = c \cdot k$, we bound the number of such occurrences. For instance, if $q_{a_i} = 2 \cdot p$, then at most 2 occurrences of $(a_i)^p$ will be considered for the abstract execution. With this constraint, the liveness algorithm of SDF can be reused. For instance, the graph of Figure 5.3(a) (with single appearance schedule $A^2B^{2p}C^{2p}$ has a cycle without any saturated edge. By using our algorithm, we can find the schedule $A^2(B^pC^p)^2$ and this graph is live only when p is even.

Nevertheless, there are cases for which this approach is not sufficient. Consider, for example, the graph in Figure 5.3(b) when p is odd or with only 2 tokens for the delay in the cycle. Its single appearance schedule is also $A^2B^{2p}C^{2p}$ and it is clearly live with the schedule $A^2(B^2C^2)^p$. However, our algorithm cannot find it. In this case, we continue by refining the SDF algorithm.

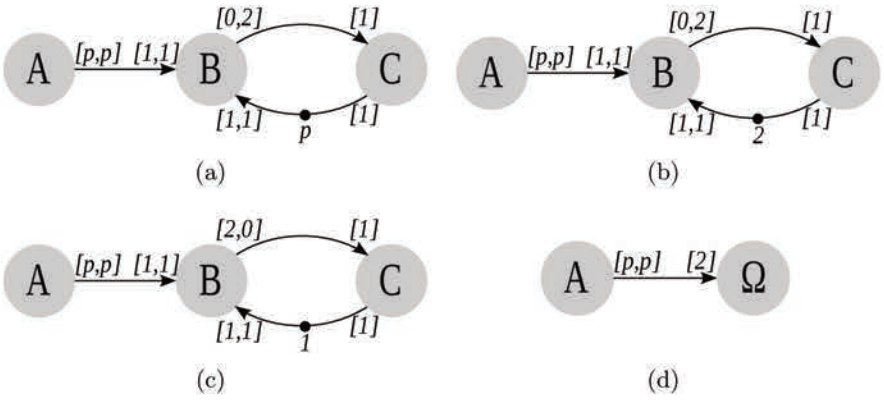


Figure 5.3: (a), (b), (c): Live TPDF graphs; (d): New graph obtained by clustering the cycle $Z = (B, C)$ into a single new actor Ω

To deal with such problematic cycles, we use the standard clustering described in [14]. Given a connected, consistent TPDF graph $\mathcal{G}$, a subset $Z \subseteq (K \cup G)$, clustering Z involves replacing Z by a single actor Ω. The new actor Ω is connected to the same external ports as Z was, but the port rate and execution sequence must be adjusted. The rate sequence of each port of an actor $a_i \in Z$ connected to the rest of the graph is replaced by $r = Y_i^u(q_{a_i}^L)$ if e_u connected to an input port of a_i or $r = X_i^u(q_{a_i}^L)$ if e_u connected to an output port of a_i. It follows that a firing of Z corresponds to $q_{a_i}^L$ firings of its actors. Here, by clustering only cycles, we resolve the problem of liveness by checking the local solution of the clustered cycles. If this graph is live so is the original graph.

For the example in Figure 5.3(b), by clustering the cycle $Z = (B, C)$ into a new actor Ω to get the new graph in Figure 5.3(d). By using the previous abstract algorithm, we find the schedule $A^2\Omega^p$ which corresponds to $A^2(B^2C^2)^p$ for the original graph (with $q_{\mathcal{G}}(Z) = p$, $q_B^L = 2$ and $q_C^L = 2$), which is therefore correctly found to be live.

A final refinement is needed to take into account the case in Figure 5.3(c). For instance, the previous refined algorithm would fail to find a schedule for the cycle with only one initial token. However, it is clearly live with the schedule $A^2(BC)^{2p}$ or $A^2(BCCB)^p$. In this case, we construct a *Late schedule*, presented in Chapter 4 , which introduces the execution order between actors in CSDF and TPDF graphs. For instance, the cycle in Figure 5.3(c) has a local Late schedule $(BCCB)$ by applying the Algorithm presented in [41] for q_C^L firings of C. Then, the original graph is live with the global schedule A^2Z^p or $A^2(BCCB)^p$.

5.3.4 Throughput Analysis

The performance analysis of TPDF is challenging because TPDF is a dynamic dataflow model and the graph behavior is control-dependent. However, its dynamic behaviours can be viewed upon as a collection of different behaviours, called *cases*, occurring in certain unknown patterns. Each case of a TPDF graph is characterized by a value of its parameter and a graph topology. For this reason, each case is by itself fairly static and predictable in performance and resource usage and can be dealt with by traditional methods. In this chapter, we introduce an analysis technique based on the work in [48]. We characterized the TPDF graphs by the possible orders in which certain cases may occur. It is also possible by stochastically specifying each mode sequence (or graph case) by a Markov

Chain. For worst-case analysis, we can abstract from these transition probabilities and obtain a Finite State Machine (FSM). Every state is labelled with a graph case and different states can be labelled with the same graph case.

Definition 20 *Given a set U of cases. A finite state machine F on U is a tuple $(S, s_0, \sigma, \varphi)$ consisting of a finite set S of states, an initial state $s_0 \in S$, a set of transitions between two states $\sigma \subseteq S \times S$ and a case labelling $\varphi : S \to U$.*

Each case of the graph is characterized by a (max, +) matrix, as can be seen in Section 5.2. Since TPDF is designed to be well adapted with streaming applications, we consider here infinite executions of the FSM to characterize the mode sequences that may occur. With this FSM and an initial time-stamp λ_0, we can associate a time-stamp sequence $\lambda_0\lambda_1\lambda_2 \ldots \lambda_n$ with $\lambda_{i+1} = M_{i+1} \otimes \lambda_i \ \forall i \in [0, n-1]$. According to the theory of CSDF [19], each case is guaranteed to have an upper bound on the self-timed execution of the dataflow's execution, so we can derive straightforwardly that this case sequence has also an upper bound value.

To compute throughput, we have to check all possible case sequences. From the FSM of a TPDF graph, we define a state-space of case sequence executions as follows.

Definition 21 *Given a TPDF graph $\mathcal{G}$ characterized by the set of (max, +) matrices $\{M_i | i \in U\}$ and a FSM $F = (S, s_0, \sigma, \varphi)$, a state-space of $\mathcal{G}$ on F is a tuple (Q, q_0, θ), consisting of:*

- *A set $Q = S \times \mathbb{R}^N_{\max}$ of configurations (s, λ) with a state $s \in S$ of F and a time-stamp vector λ.*
- *q_0 is the initial configuration of the state-space.*
- *θ is the set of transitions between two configurations $\theta \subseteq Q \times \mathbb{R} \times Q$ consisting of the following transitions: $\{((s, \lambda), ||\lambda'||, (s', \lambda'^{norm})) | (s, \lambda) \in Q, (q, q') \in \sigma, \lambda' = M_{\Sigma(i)}\lambda\}$, where $M_{\Sigma(i)}$ is a multiplication of all case matrices between λ and λ', as defined in Equation (5.4).*

A state in the state-space is a pair consisting of a state in the FSM and a normalized vector representing the relative distance in time of the time-stamp of the common initial tokens. An edge in this state-space $((s, \lambda), ||\lambda'||, (s', \lambda'^{norm}))$ represents the execution of a single iteration in the case of the destination of the edge. If we start with the initial tokens having time-stamp vector λ and we execute a single iteration in case s' then the new time-stamp vector of initial tokens are produced at $\lambda' = ||\lambda'|| + \lambda'^{norm}$ (or earlier). The state-space of a TPDF graph can be constructed in depth-first-search (DFS) or breadth-first-search (BFS) (or any other) manner incrementally [48]. For a self-timed bounded graph with rational execution times, the state-space is finite. However, it can be large in some cases because of the number of modes used in each case and because of the complexity of the FSM. Further techniques (e.g., (max, +) automaton [47, 48]) can be applied to reduce the size of the state-space and analyse the worst-case throughput in a faster way.

From Definition 21, according to [99], the throughput of a TPDF graph is equal to the inverse of the Maximum Cycle Mean (MCM) attained in any reachable simple cycles of the state-space. For example, Figure 5.4 represents the reachable state-space of example in Figure 5.1 and its Finite State Machine. The blue box and bold arrows highlights the cycle with maximum cycle mean or lowest throughput (i.e., 10.5 for the MCM and 1/10.5 for the throughput).

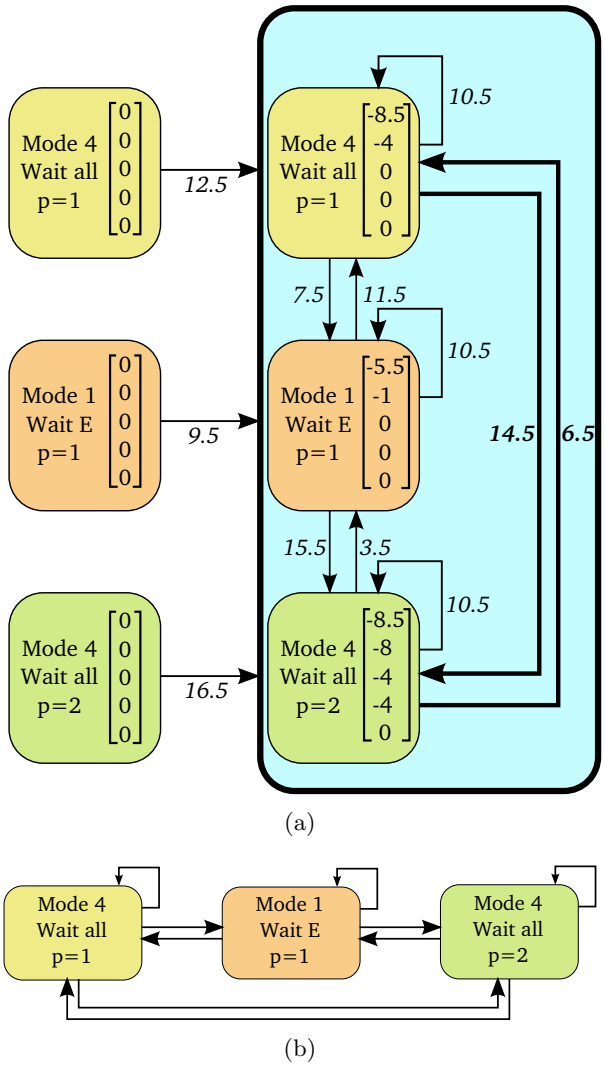


Figure 5.4: (a) State-space of the example in Figure 5.1 and (b) its Finite State Machine. The blue box and bold arrows highlight the cycle with the maximum cycle mean which determines the throughput. Each edge is labelled with the worst-case time dependencies between the individual tokens in the specific locations and corresponding cases. For example, the channel connected between two cases using mode 4 is annotated by $||\lambda'||$, where $\lambda' = M \otimes \lambda$ and M is the matrix characterizing mode 4 as in Equation (5.5).

5.3.5 Scheduling

In Section 5.3.3, we described a way to find a sequential schedule for TPDF applications and Section 5.3.4 introduced an analysis method based on the dynamic change between cases. Each case represents an entire iteration of the graph. This mechanism is well-suited to the approach by which programmers implement streaming applications on many-core platforms with highly parallel schedules such as the MPPA-256 [34] clustered architecture, from Kalray, comprising 256 cores. The native dataflow programming model developed in ΣC (Sigma-C) for MPPA-256 uses the notion of *canonical period* [5], which represents the partial order corresponding to the execution of one iteration of the application. For TPDF

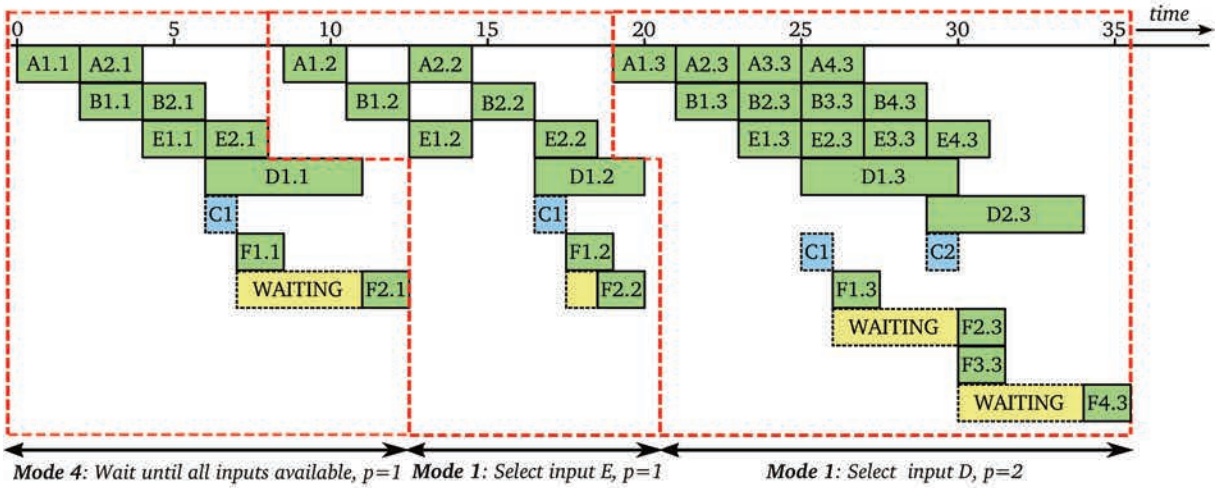


Figure 5.5: Scheduling of the example in Figure 5.1 for $p = 1$ and $p = 2$. Green and blue boxes represent the firings of kernels and control actors, respectively. Dashed red lines represent valid *canonical periods* of this graph. Number next to the name of each actor is its ordinal number of execution, followed by the ordinal number of iteration.

graphs, we reuse this notion of canonical period with several changes in the scheduling techniques:

- The control actor is scheduled for execution with the highest priority (i.e., if there are several kernels and a control actor available concurrently, the control actor is ensured to have a processing unit available before the others). Message passing time must be accounted for within the scheduling so that the system acts as if it was instantaneous.
- The kernel which receives the control token is fired immediately after receiving the control token. If this kernel has to wait until its input data tokens are available, it passes into a sleeping queue and wakes up when there is enough tokens in its inputs ports, as requested by the control token. If this kernel is fired in a mode where several of its input ports are rejected, the scheduler uses the Actor Dependence Function [41] which defines the dependency between actors' executions to stop unnecessary firings of kernels whose outputs are not chosen by the transaction process.

The self-timed execution of this graph and its canonical periods are illustrated in Figure 5.5. Time is depicted on the horizontal axis and the actors vertically. Green boxes represent the firings of kernels, labelled by the kernel's name and its ordinal number of firing within an iteration. Blue and dashed boxes represent the firings of control actors. Each dashed red line region represent a canonical period corresponding to a parameter value. The initial tokens after the first iteration is generated when all firings of A and F finish (i.e, the 1st initial token generated at time 4, the 2nd and 3rd at 8.5 and 12.5, respectively). We obtain the same result as calculated by the (max, +) algebra in Section 5.2.

5.4 Summary

In this chapter, we presented TPDF, a novel parametric dataflow model of computation for embedded streaming applications. TPDF extends CSDF with parametric rates and a new type of control actor, channel and port, which expresses dynamic changes of the

graph topology and time-triggered semantics. So far, several *parametric* dataflow models have been proposed, for instance PSDF [17], VRDF [110], SPDF [46]. In contrast to CSDF, these models allow a dynamic variation of the production and consumption rates of actors to change at runtime according to the manipulated data. However, none of these models provide any of the static guarantees that TPDF does (rate consistency, boundedness and liveness) or propose parametric rates without dynamic topology changes. In [69], Koek *et al.* introduces another extension of CSDF: the CSDF^a model with Auto-concurrency. In CSDF^a, tokens have indices and the consumption order of tokens is static and time-independent. This allows expressing and trading off pipeline and coarse-grained data parallelism in a single, unified model. Another related model is the Scenario-Aware Data-Flow (SADF) MoC [104], which exploits also the mode-based approach where the dynamic behavior of an application is viewed as a collection of different scenarios; yet, our model is more generic as it provides a unified view of manycore systems, which is entirely composable.

Our approach is somewhat similar to Boolean Parametric Dataflow (BPDF) [16] which allows not only dynamic variations of rates but also dynamic changes of the graph topology. Each BPDF edge can be annotated with a boolean parameter which changes the topology of the graph. When a boolean parameter is false, the edge annotated by this parameter is considered absent. Still, this is not enough because this model lacks the ability to impose real-time constraints, a feature also required to program modern safety critical applications which will be both highly parallel and time constrained. Our model already solves this problem by using control actors of type clock. Moreover, all SPDF and BPDF case studies (e.g., the VC-1 Video Decoder), presented in [46], [16] and [15], can be replicated using our approach, as presented in Section 7.1, without introducing parameter communication and synchronization between firings of modifiers and users, which would have complicated scheduling significantly because of the additional actors, edges and ports. We can also improve the quality of the AVC Encoder, a much more complex application, by using a quality threshold, as can be seen in the Edge Detection benchmark in Section 7.1, for the motion vector detection, implemented with a Transaction kernel, to choose dynamically the highest quality video available within real-time constraints.

Moreover, in this chapter, the problem of throughput calculation of parametric dataflow graphs was also 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 static models, such as SDF and CSDF [41, 99, 102]. For this reason, we used the (max, +) algebra to capture the dynamic semantics of modes introduced by TPDF. From this theory, we introduced a technique to automatically analyse its worst-case throughput. In the next chapter, we investigate the applicability of TPDF to model hard-real-time applications and an implementation/validation of the model by using a set of real-life dynamic applications is introduced in Chapter 7.

Chapter 6

Real-Time Extension for TPDF

Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time.

— Thomas A. Edison

Contents

6.1 Time-Constrained Automata 91
6.1.1 Chains 92
6.1.2 Time-constrained trees 93
6.1.3 Automata 93
6.1.4 The visibility principle 93
6.2 Systematic translation from TCA to TPDF 94
6.3 Example 96
6.4 Application 100
6.4.1 QDS design 100
6.4.2 TPDF design 102
6.5 Summary 102

The advent of embedded manycores opens perspectives for new applications, which will be both highly parallel and time constrained. In this section, we bring real-time support for embedded high performance applications by using control actor of type clock and parametric extension. We demonstrate that TPDF can be used to accurately model task timing requirements in a great variety of situations, described by using the Time Constrained Automata (TCA) [79, 80] model.

6.1 Time-Constrained Automata

TCA was initially designed for the expression of multi-task critical real-time Instrumentation and Control (I&C) systems. It offers deterministic parallel execution and communications, with periodic dataflows. The expression of the time constraints is quite simple, consisting in simple boundaries around functions, which enables tools to perform feasibility checks and schedule the tasks on multi-core systems.

To present TCA, we can start from the model given for chains and trees before explaining the time-constrained automata.

6.1.1 Chains

In TCA, a task (as a kernel in TPDF or an actor in CSDF) is modelized as a sequence of indivisible code *blocks*. Each block, represented by *arcs* and are separated by *nodes*. The node from which the arc starts *immediately precedes* the arc, and the node to which it leads *immediately succeeds* the arc. A *chain* is a sequence of blocks, executing one after the other. When blocks a and b are consecutive, instructions of a have to be executed before those of b.

TCA supports only two kinds of temporal constraints, represented by *nodes*: *Before* nodes express a deadline constraint (denoted by $\triangleleft$) and *After* nodes express the earliest time for an execution (denoted by $\triangleright$). When a block bears both a before and an after constraint at the same date, then it becomes a synchronization point (denoted by $\diamondsuit$). Except for synchronization points, nodes cannot bear more than one time constraint. Nodes with no temporal constraint are represented by $\bigcirc$. Figure 6.1 gives an example of a TCA chain of blocks. The constraint nodes are labelled by the absolute date that they represent.

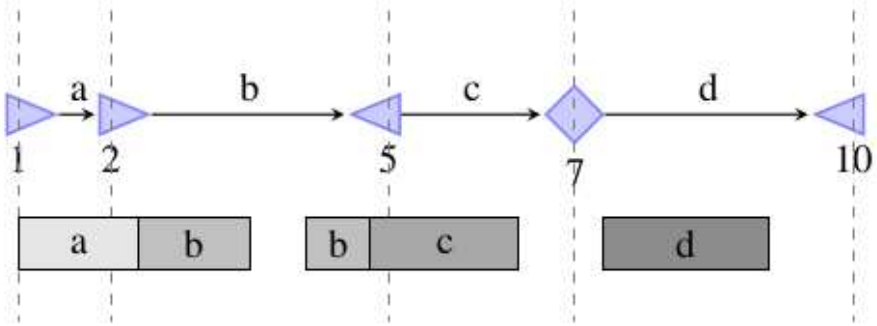


Figure 6.1: Constrained chain: Block A must start after date 1, B must execute between 2 and 5, C must end before 7, and D must execute between 7 and 10.

In TCA, an *after* node implicitly constrains all the succeeding blocks to start after its date, and a *before* node constrains all the preceding blocks to end before its date. Thus the following labelling convention can be adopted without modifying the semantics of our model: *all node dates can be labeled using the relative date from the previous after node* (including synchronization points). This convention will allow expression of loops in automata and the relative labeling, as shown in Figure 6.2, is denoted by putting underscores below dates.

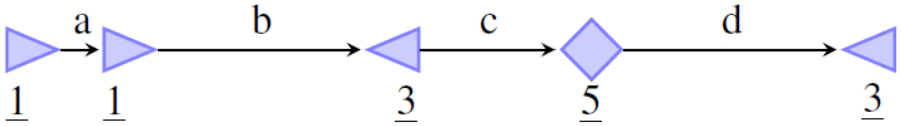


Figure 6.2: Chain of Figure 6.1 with relative labeling. The relative date is calculated from the previous after node.

6.1.2 Time-constrained trees

The previous concept of chains is extended to *trees*, which requires to handle "choices". Several blocks can start from a node (such a node is called a *choice* node). This expresses the fact that different execution paths may be taken. The choice of which path to take is made when finishing executing the immediately preceding block. Figure 6.3 gives an example.

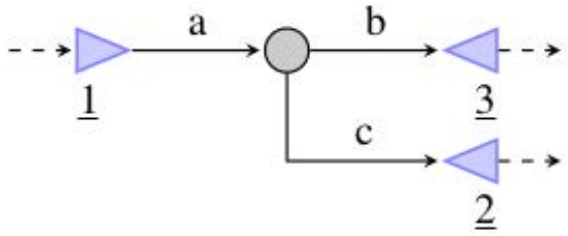


Figure 6.3: Depending on the execution of a, either b or c will be executed. The path a → b has 2 units of time to complete, but the path a → c only has 1.

6.1.3 Automata

To represent infinite computations with finite objects, TCA uses *automata*. Automata differ from trees in two aspects: first they allow several arcs to finish on the same node; second they allow cycles in the graph. These differences change the precedes relation on blocks, and affects negatively the semantics, which depends on this relation. TCA addresses these problems by defining the semantics of automaton as such: the semantics of a time-constrained automaton is equivalent to that of its unfolded tree, which is the infinite tree representing all possible traversals of the automaton. Thus, the precedes relation, and the semantics, are preserved. Figure 6.4 shows example of such an unfolding.

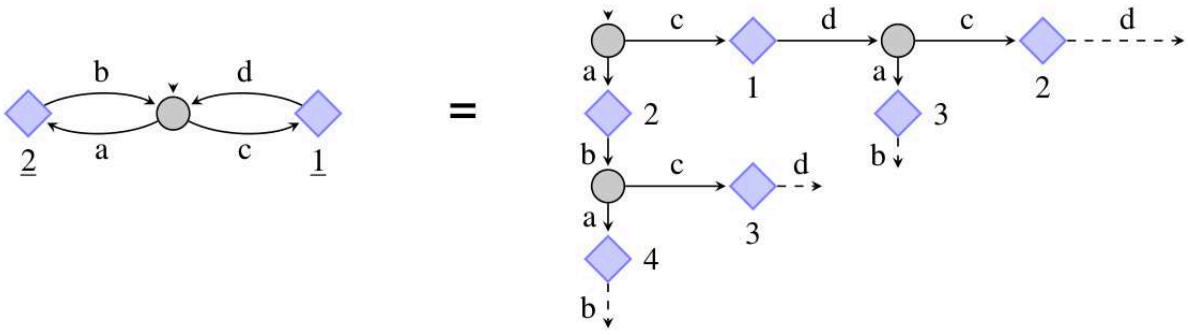


Figure 6.4: Unfolding of an automaton

6.1.4 The visibility principle

This section explains how determinism can be preserved in a real-time system modeled by TCA and composed of parallel tasks that communicate using the visibility principles, by controlling when the received messages are made accessible to the tasks.

In [79], the visibility principle is defined as: *a task A shall be able to see a received message at time t only if the message would have been received at t in every possible*

execution. From this definition, there are two modes of communication between tasks. The first mode is implicit with temporal variable which corresponds to real-time data flows: past values, available for each agent that deals with them, are stored and updated by the unique writer, the owner agent, at a predetermined temporal rhythm. The system layer is in charge of updating (i.e., copying) the available values of these variables. As can be seen in the example in Figure 6.5, if we consider a temporal variable X owned by the task a, and a second task b that want to access the last value of X, then it will observe the past value $X(t0) = X(sta)$. The rationale is quite simple: during processing the first firing of b (i.e., $b1$), only the value at the formal start date is observable, and at $t0$, processing $a2$ may be not terminated, so the last coherent observable value of X is $X(sta)$.

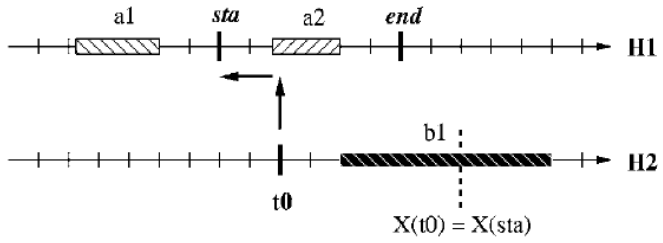


Figure 6.5: Observable values of a temporal variable

The second communication mechanism which is explicit, is the sending of messages. All message attributes (content, recipient id, type, ...) of the sending messages are formally checked by the compiler. A visibility date specifies the date after which the message can be observed and read by the recipient. This date allows to give a validity duration to the receiving message. The system also verifies if the recipient consumes the message before it expires, else a specific exception is raised. To achieve determinism, the order of messages induced by the visibility dates is made to be total. For example, if a message M is sent by the second task c with visibility date $t1$, it could not be observed (and therefore consumed) before the end date, because $t1 > sta$, as can be seen in Figure 6.6.

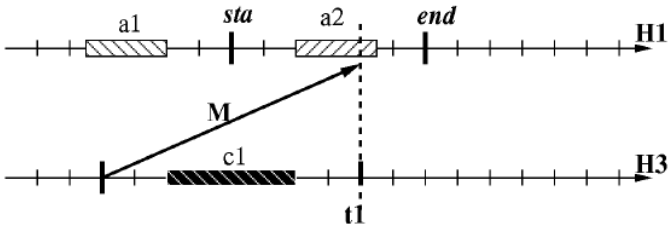


Figure 6.6: Send of message

6.2 Systematic translation from TCA to TPDF

To reuse existing analysis techniques for real-time applications modelized by using TCA, we find a way to translate systematically from TCA to TPDF by representing temporal nodes *Before* and *After* in a dataflow way. Once this straightforward translation

is done, the time constraints can be added to a TPDF graph without affecting its static analysis.

An *After* constraint on actor a_i simply can be modeled by using a control clock connected directly to this actor, as can be seen in Figure 6.7(a). The condition for the execution of A then becomes that it must receive the control token from the $3ms$ clock before executing in a mode defined by this token (e.g., if the mode is "Wait until all data input are available", A will be fired immediately if there are enough data in all of its input edges or wait until this condition is satisfied. However, in any case, it will be fire only after $3ms$, so the after constraint is guaranteed).

As can be seen in Figure 6.7(b), a Before constraint requires a check (if actor terminated on time), and a decision depending on the result of this check (normal execution or overrun management). To do so, we introduce deadline transactions, denoted DTr. These deadline transactions have a null execution time and will be used mainly as an indication in the analysis of the application for schedulability checks and buffer dimensioning.

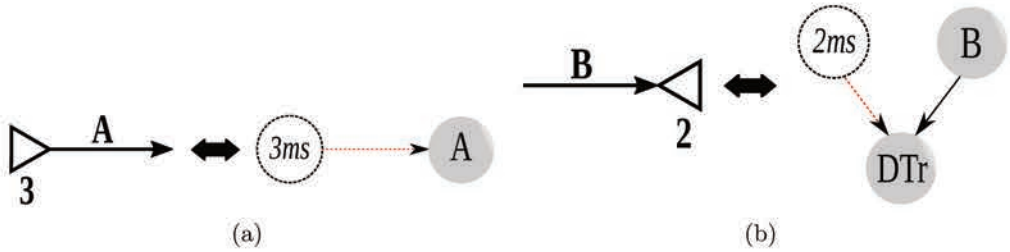


Figure 6.7: Modeling of TCA node as a TPDF graph; (a): Actor A can only be fired after receiving a control token from the $3ms$ clock; (b): The Before constraint requires a check in actor DTr (if actor B terminated on time), and a decision depending on the result of this check (normal execution or overrun management).

Regarding communication, the blocks in TCA cannot share data outside the scope of communication devices with a clearly defined temporal behavior. There are no synchronization primitives in TCA, all communications are asynchronous, with an ordering controlled by the time constraints. A message sent through a TCA communication device has a *visibility date*, which has to be at least the deadline of the block that sends the message, or any later date. In the implementation of TCA [79], the visibility date is given as a Before node for the *send* block, this ensure that this block will always send the message before the visibility date. This primitive is also well-suited to handle sporadic communications.

For example, Figure 6.8 gives an example of a sender and a receiver, allowing to define communication primitives for safe interactions between tasks. The *relative visibility date* of $3ms$ can be implemented by using a deadline transaction. In this case, the control actor and its special mode are well-suited to synchronize tasks that must communicate.

It is possible that a TCA application has more than one connected component, as in Figure 6.8. Once the final TPDF for these component are built, the static analysis for the whole application can be applied.

From this basic translation, any TCA application can be remodeled by using TPDF. We demonstrate in the next section several popular examples in the real-time domain.

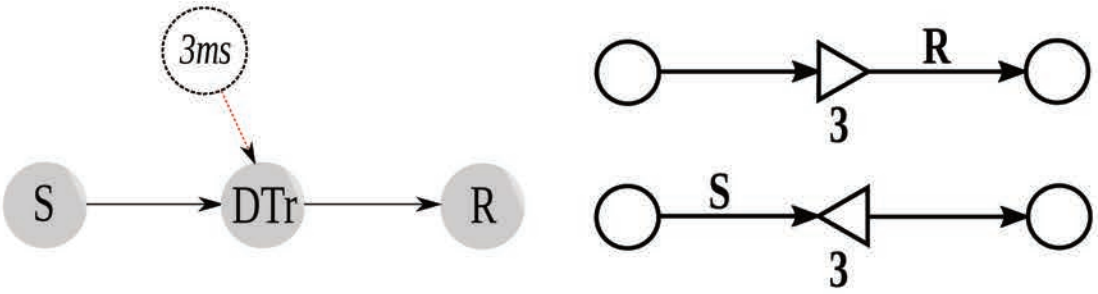


Figure 6.8: Synchronisation using control clock: S must send a message before time 3, R receives it after time 3. This guarantees that the message will always be received.

6.3 Example

The basic example is modeling periodic tasks with deadline equal to their periods as in Figure 6.9(a). This graph starts by firing kernel A and the control actor of type clock $2ms$ in the same time because of initial tokens in its data channel input. After 2 ms, the control clock sends a control token to fire immediately the Transaction box in the "Select one of the data inputs" mode, which plays the role of a deadline checker. If this time constraint is violated by A, a fault-tolerance procedure or a simple user warning is necessary. In the opposite case, the system can continue because the time constraint is satisfied. More evolved usage are general periodic tasks, whose deadlines can be smaller or even larger than periods (Figure 6.9(b)).

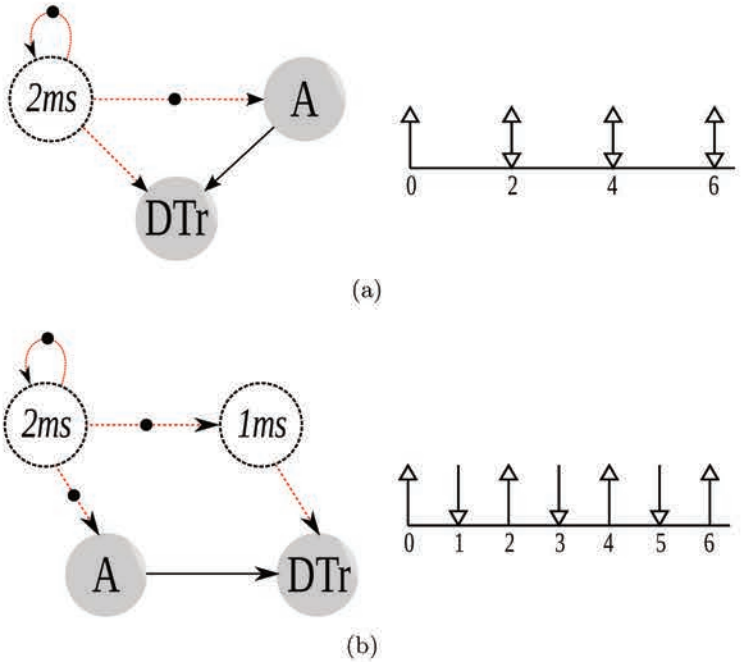


Figure 6.9: (a) Periodic task of period 2 with relative deadline equal to the period. Omitted rate equal to 1. (b) Periodic tasks with period 2 and relative deadline 1.

A periodic task might require an initialization stage (e.g., for sensor calibration), which may impose time constraints. TPDF also allows to phase different periodic tasks, as in Figure 6.10 by using parametric rates. In this case, the parameter p is set up to be 1 in

the first time the control actor 1 ms is fired and 0 for all subsequent times. By this way, we eliminate the channel connected with the first 1 ms control clock and continue to fire the nested loop as a periodic task.

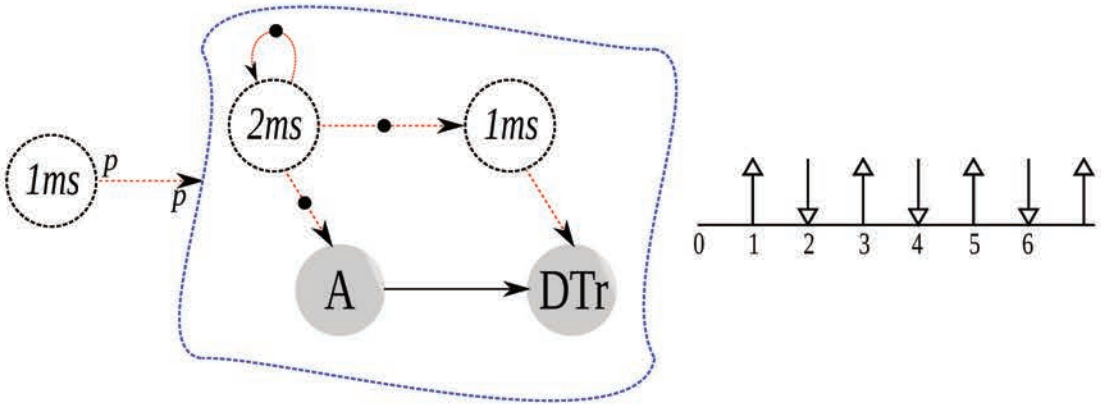


Figure 6.10: Periodic tasks with period 2, relative deadline 1 and phase 1. The blue dotted line is used to cluster the region with a phase equal to 1.

Another example is modeling jitter, which is the variation between successive executions of a periodic task. Current real-time methodologies consist of analyzing jitter once the design is done and the execution time of the task are known. Thus, the whole design has to be modified if the jitter of a task is too high. Another time-constraied task models (e.g., Time-Constrained Automata (TCA) [80]) try to express the maximum jitter directly in the model, making it a constraint that the scheduling algorithm has to enforce. Our model also allows to specify this type of time constraints by remodeling the Before node, expressing a deadline constraint, and After node, expressing the earliest time for an execution.

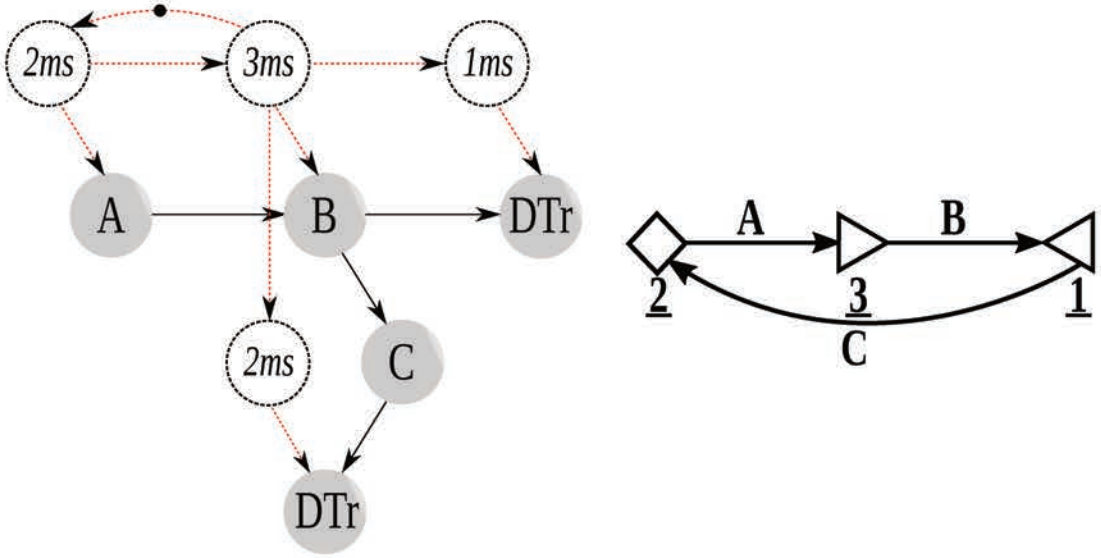


Figure 6.11: A periodic task with period 5. B is constrained as a fine-grained jitter with maximum value 1.

It is possible that a TCA application has different blocks executing in different periods.

Algorithm 4 RATE-SEQUENCE(A,B)

Require: Source kernel A with period T_A and phase p_A, destination kernel B with period T_B and phase p_B

1: $k = \text{lcm}(T_A, T_B)$ ▷ lcm denotes the least common multiple of T_A and T_B;
2: $k_A = \frac{k}{T_A}$
3: $k_B = \frac{k}{T_B}$
4: $\tau_{DTr} = k_A$ ▷ τ_{DTr} denotes the length of the rate sequence of DTr;
5: $i = 0$
6: $j = 0$
7: $l = 0$
8: **for** $j < k_B$ **do**
9: $max = 0$
10: $s_B(j) = p_B + j \times T_B$
11: **for** $i < k_A$ **do**
12: $v_A(i) = p_A + (i+1) \times T_A$ ▷ $v_A(i)$ denotes visibility date of the token produced by the i-th firing of A;
13: **if** $i > max$ and $v_A(i) \leq s_B(j)$ **then**
14: $max = i$
15: **end if**
16: $i = i + 1$
17: **end for**
18: $count(max) = count(max) + 1$ ▷ $count(max)$ denotes the number of execution of B depends on the $max-$th execution of A;
19: $j = j + 1$
20: **end for**
21: **for** $l < \tau_{DTr}$ **do**
22: $cons(l) = 1$
23: $prod(l) = count(l)$
24: **end for**

One of this actor can have a phase, as in Figure 6.12 where two connected kernels A and B have two different periods of $25ms$ and $15ms$, respectively. The dependency between these kernels, by using the visibility principle of TCA model, can be described as follows: the two first executions of B receive the token produced by the first execution of A with the visibility date of $25ms$, the third execution of B consumes the token from the second execution of A with a visibility date of $50ms$. Finally for the first iteration of this graph, the two last firings of B depend on the the third firing of A with a visibility date of $75ms$. This dependent relationship is repeated for each iteration, represented by one dashed red line region in Figure 6.12. Algorithm 4 illustrates a method to determine the production/consumption rate sequences of the DTr used to connect between two kernels of different periods.

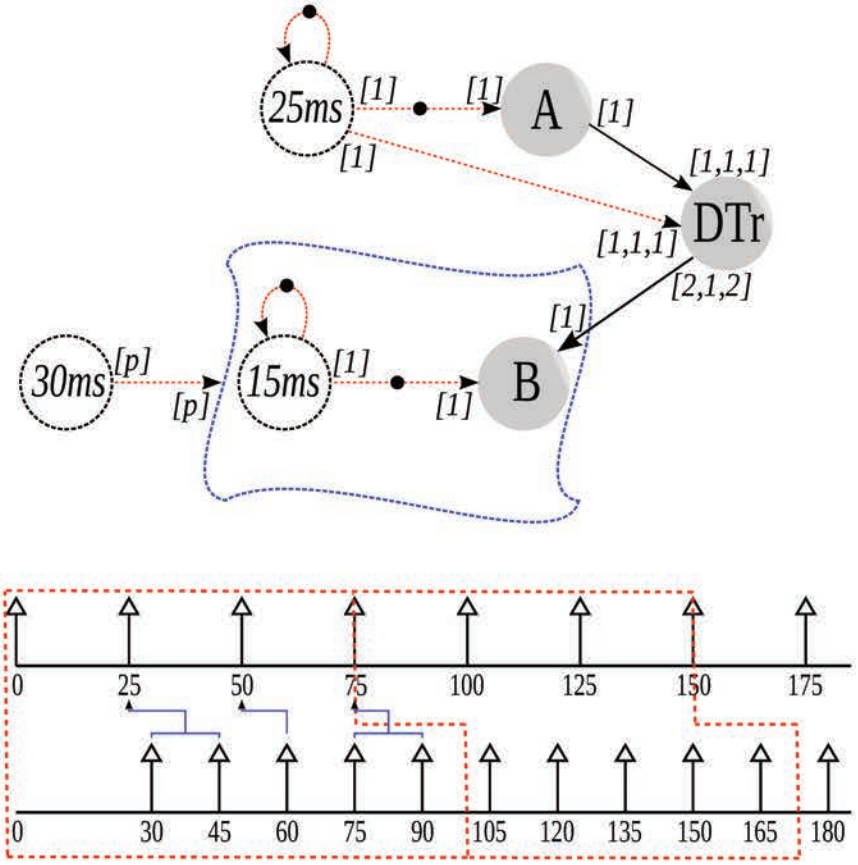


Figure 6.12: Two periodic tasks of period $25ms$ and $15ms$. B has a phase of $30ms$. This task is represented by the region surrounded by the blue dotted line. Message sent by A has also a *visibility date* of $25ms$. The two first executions of B receive the token produced by the first execution of A with the visibility date of $25ms$, the third execution of B consumes the token from the second execution of A with a visibility date of $50ms$. Finally for the first iteration of this graph, the two last firings of B depend on the the third firing of A with a visibility date of $75ms$. This dependent relationship is repeated for each iteration, represented by one dashed red line region in the figure.

Figure 6.13 gives an example of constrained chain of tasks with numerous phase and deadline constraints.

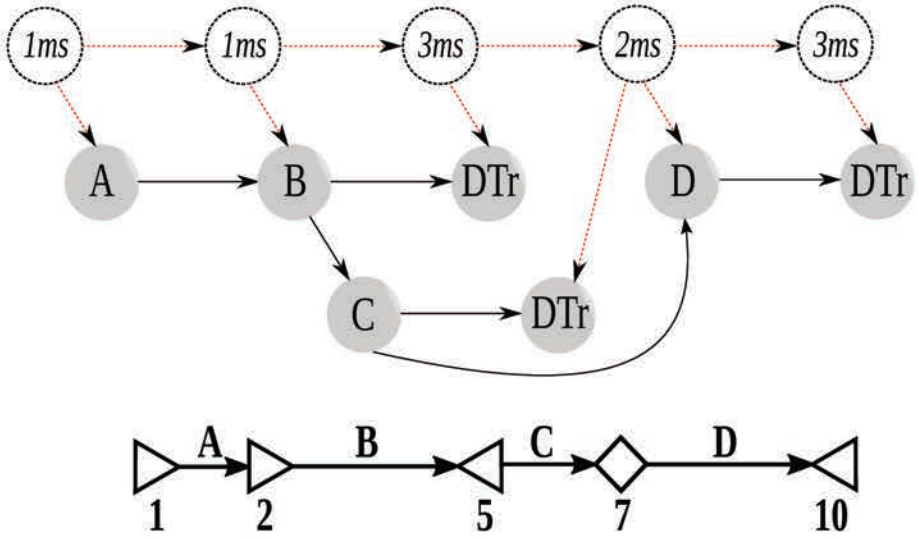


Figure 6.13: Constrained chain: Kernel A must start after date 1, B must execute between 2 and 5, C must end before 7, and D must execute between 7 and 10. After this iteration, the graph continues a new one by waiting 1 ms before firing A.

6.4 Application

We illustrate the possibility to mix timed dataflow actors with hard real-time I&C tasks with the safety-classified Qualified Display System (QDS) application [32] implemented by CEA and Framatome-ANP for nuclear activities. To be more specific, the QDS is used to:

- acquire data transmitted by one or several Teleperm XS^{TM} units,
- process these input data (e.g. for taking into account redundant signals)
- display the elaborated information to users through a project specific provided HMI
- process operator inputs such as setpoints or other parameters.

This application requires strong safety features dealing with the detection and confinement of faults. This is achieved by TCA concepts and methodology supported by its system software package and tools (e.g., OASIS [86]), and by additional hardware self-tests.

The main requirement is to achieve HMI functionalities with high dependability. Real time performances deal with the data storage capacities and rhythms, the response times of the display and the validity of displayed information. Hence, up to few thousands analog or binary values can be recorded each second during 30 minutes or 24 hours. The time between the acquisition of data and its subsequent display on the screen shall not exceed one second and the HMI response times have to be less than 200 milliseconds. All information on a screen must be time consistent. Finally, design tools have to be embedded in an user friendly development environment.

6.4.1 QDS design

Figure 6.14 gives the description of the QDS software in terms of OASIS agents and their interfaces. The software compounds the following agents:

- *appliNAg* is the applicative agent that implement the functional treatments on the acquired data;

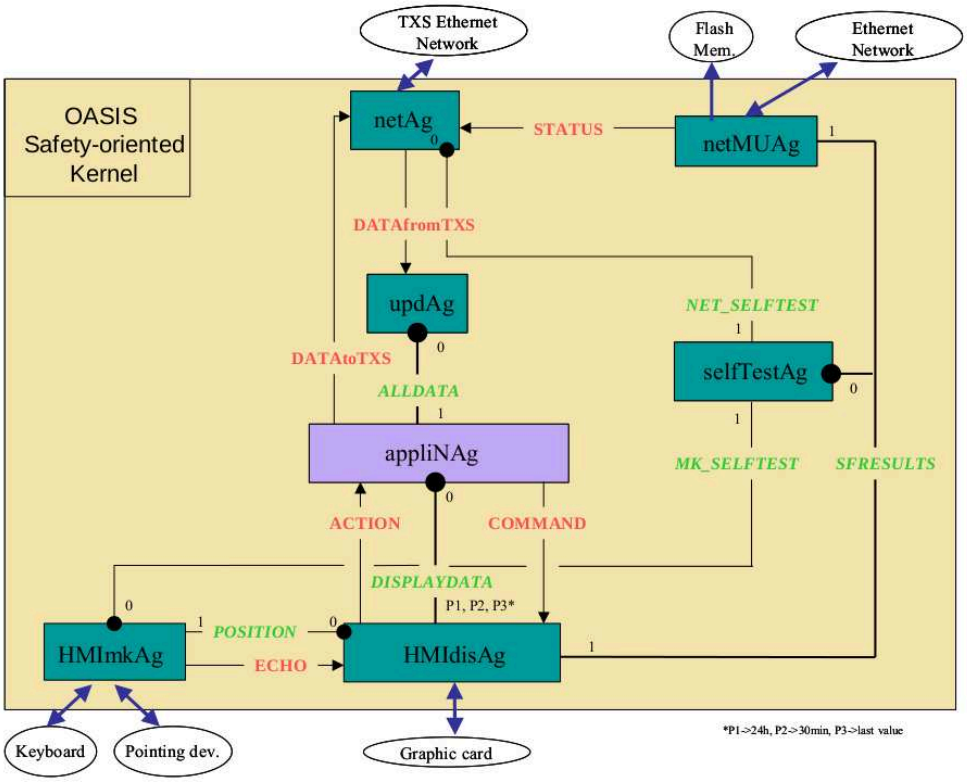


Figure 6.14: System architecture with multi Ethernet port configuration of a QDS software, implemented in terms of OASIS agents.

- *HMIdisAg* performs the screen display management that prepare and display all defined screen pages;
- *HMImkAg* for the keyboard and pointing devices management;
- *netAg* for the low-level network management, including Ethernet chip management and protocol implementation (to receive data from the Teleperm XS^{TM} (TXS) units and to send them specific applicative data and QDS self monitoring results). In a real QDS architecture, there are 4 similar *netAg* agents, named as net$i$Ag (i from 1 to 4) and connected to the 4 TXS units: each agent controls the communication port with one TXS unit.
- *updAg* for the high-level network management for building functional real-time data streams;
- *netMUAg* for maintenance management;
- *selfTestAg* for additional hardware self-tests and collection of those performed by agents that manage hardware components.

Table 6.1 shows the list of these agents' period parameters with their typical values. In Figure 6.14, there are two kinds of exchanges: data "continuously" or cyclically needed (such as values of information to be displayed) represented by plain lines beginning with a dot, and with their names in italic; and data that are exchanged when they are available (such as HMI actions and commands), represented by plain lines ending with an arrow. Bold-arrowed lines represent the interactions with hardware components.

Table 6.1: Period parameter of the QDS and their typical values

Agent	Period parameter	Typical value (unit)
net$i$Ag	QDS_NET_SCAN_PERIOD	10 or 20 (ms)
netMUAg	QDS_MU_MAIN_PERIOD	1000 (ms)
selfTestAg	QDS_SELFTEST_MAIN_PERIOD	20 (ms)
updAg	QDS_UPD_PERIOD	20 (ms)
HMImkAg	QDS_IMK_PERIOD	10 (ms)
HMIdisAg	QDS_DISPLAY_MAIN_PERIOD	200 (ms)
appliNAg	APPLI_AG	20 (ms)

6.4.2 TPDF design

Figure 6.15 gives a TPDF model of the QDS software. Each periodic kernel has a control of type clock to define its period. Each time these kernels receive a control token from its clock, it starts by changing its status to be ready to fire. After having enough tokens on its input, it changes the status to be ready to execute and starts its computation. Each control clock has a self-loop to be restarted after a period of time equal to its value. This kind of control actor plays the same roles as the temporal After node in TCA. For the Before node, we use the DTr kernel which plays the role as a deadline checker for the continuous data needed by a kernel when its period arrives. In QDS, we assume that the visibility date of a token is also the deadline of its source. For example, the *netMuAg* kernel stores the self-test results, provided by the *selfTestAg* kernel. From this self-test results, it computes the QDS status and send it to each *netiAg* of the graph. However, because these kernels are executed in two different periods, the faster kernel (i.e., *netiAg*) uses DTr kernels as deadline checkers to send the token to the *netiAg* kernel in time. Each $1000ms$, the DTr kernel consumes one token from the $netMu$ kernel and produces 100 similar tokens, enough for the next 100 firings of $net1Ag$ dependent on only one firing of $netMu$. In this way, the data dependency between kernels is guaranteed and the rate consistency of the graph can be checked by generating the system of balance equations expressed in matrix-form as in Equation (5.6).

The QDS is designed to ensure that a data coming from the network kernel $net1Ag$ is taken into account and the result displayed on the screen by the $HMIdisAg$ kernel within $1s$ in the worst case. By following the blue line, representing the way of a coming data token from a network kernel to the screen display management, and by applying the scheduling algorithm proposed for TPDF in Section 5.3.5, we have the worst-case latency for the coming data to be displayed is $260ms$, satisfying the display condition required by the nuclear power plant.

6.5 Summary

From the TPDF model, it is possible to introduce timing constraints to allow integration of dataflow high-performance computation graphs to hard real-time I&C tasks. By demonstrating that all time constraints and visibility principles of TCA can be systematically translated to TPDF, we reused existing analysis techniques for real-time applications modelized by using TCA to guarantee strong safety features required by safety-critical applications while preserving its determinism and its static analysis with quadratic complexity algorithms in the worst case. The product of this analysis for TPDF gives all the

required information for placing and routing tasks, dimensioning buffers and scheduling hard real-time jobs. Thanks to the safety-classified QDS case-study for nuclear activities, we demonstrate that the TPDF implementation of this application satisfies the worst-case latency required for the display of incoming data. In the next section, we present a way to validate the performance of TPDF model by evaluating its throughput, another performance metric which is as important as the latency.

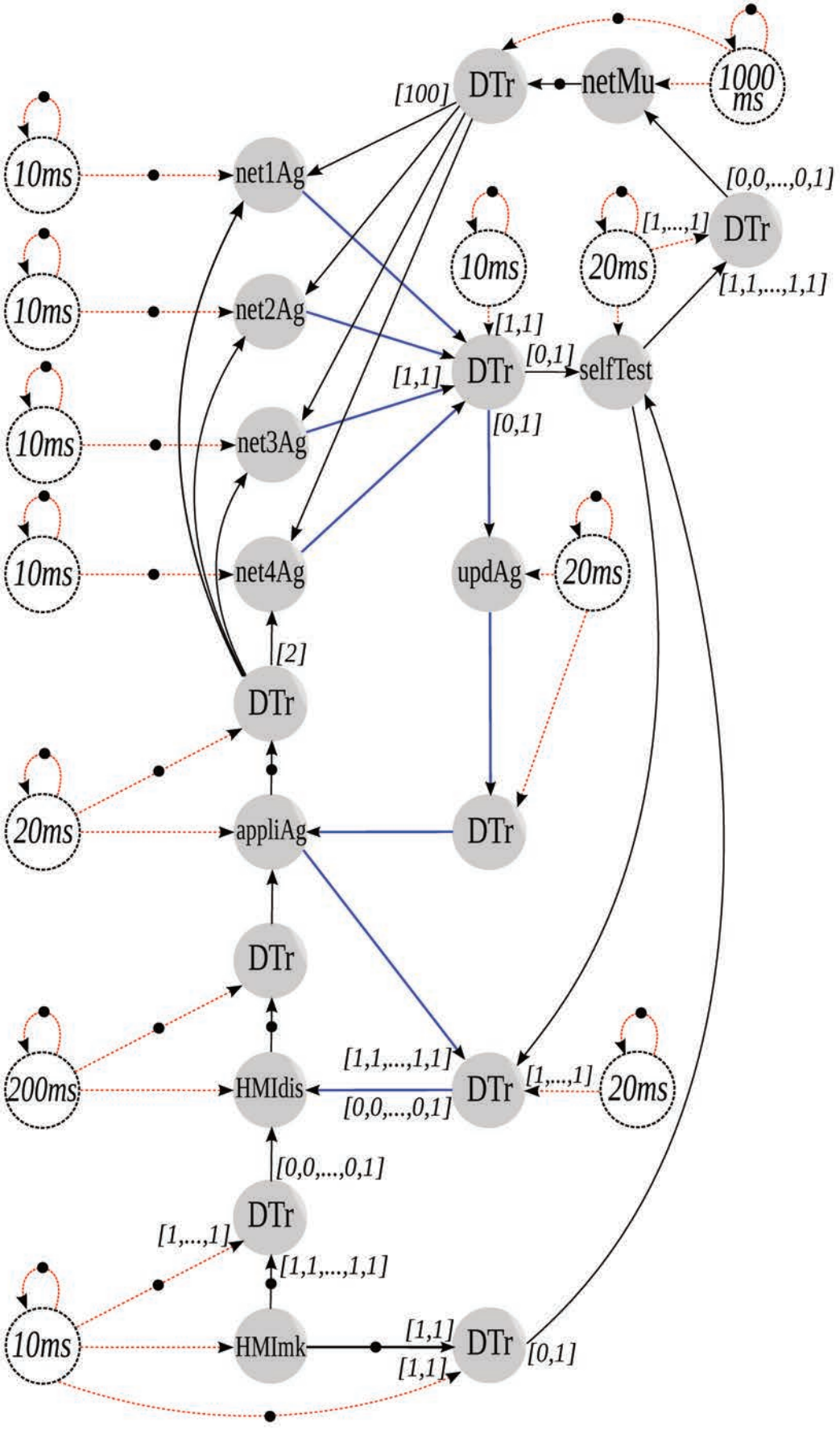


Figure 6.15: A TPDF model of the QDS software

Chapter 7

Experimental Results

Experiments were not attempted at that time, we did not believe in the usefulness of the concept anyway, and I finished my thesis in 1962 with a feeling like an artist balancing on a high rope without any interested spectators.

— Richard Ernst

Contents

7.1 Benchmarks 105
7.1.1 Case-study on Edge Detection 106
7.1.2 Case-study on Viola & Jones 108
7.1.3 Case study on Satellite positioning 109
7.1.4 Case-Study on Cognitive Radio 111
7.1.5 Case-Study on VC-1 Decoder 112
7.2 Analysis Tool 113
7.3 Experimental Results 115
7.4 Summary 116

Digital systems design consists in fitting one or several applications onto a given platform with limited resources while satisfying predefined criteria. In this chapter, by focusing on many-core platforms such as Kalray MPPA-256, we introduce a tool to automatically analyse and validate a TPDF application for its static properties (i.e., consistency, liveness and boundedness) and its worst-case performance (i.e., throughput). This chapter starts with the overview of several benchmarks and case studies used to evaluate our TPDF model and its analysis tool introduced in Section 7.2. Then, the experimental results are presented and discussed in Section 7.3.

7.1 Benchmarks

We have done experiments with a set of cognitive radio applications (e.g., OFDM, Adaptive Coding Transceiver and Receiver model), on the well-known video codec VC-1 model, on the time-constrained application Edge Detection as presented in Section 7.1.1 and on a large collection of randomly generated graphs, on a standard Intel Core i3@2.53GHz based PC. Table 7.1 shows these applications with its number of kernels (N) in the third column and its number of states in the fourth column. These benchmarks are

selected from different sources to check the expressiveness of TPDF and its performance results. The first source is several case studies of other dynamic dataflow models (e.g., VC-1 Decoder and OFDM). These applications are also official benchmarks of industrial tools such as LabView or Simulink. Another source is to build TPDF graph from real-life application programmed in C (e.g., Edge Detection). From this approach, dynamic code analysis tool, as the method introduced in [72], can be developed in the near future to transform automatically legacy code to TPDF graph.

Table 7.1: Benchmarks from different sources to check the expressiveness of TPDF and its performance results

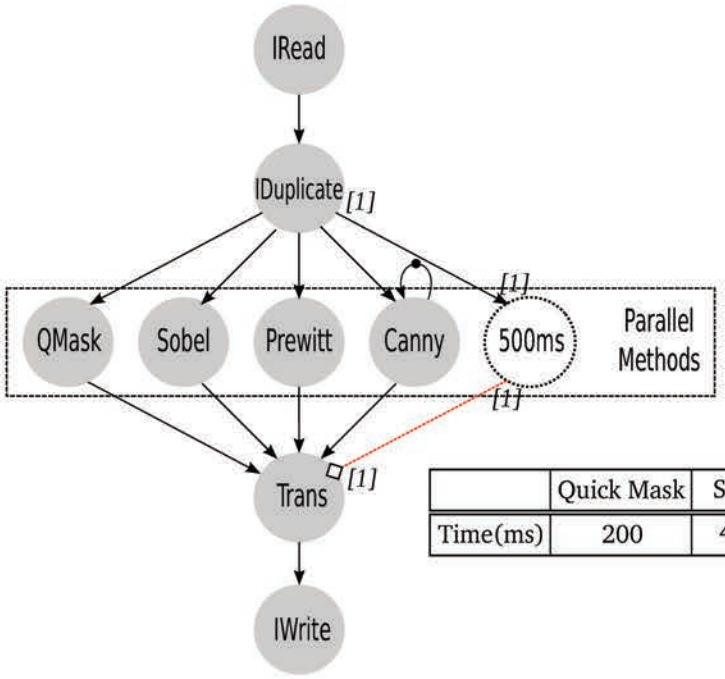
Application	**Source**	**N**	**#States**
OFDM	[108]	8	15
Adaptive Coding	[82]	14	6
VC-1 Decoder	[16]	12	12
Edge Detection	[93]	8	18
Viola & Jones	[109]	10	68
Satellite	[9]	9	55
Random Graphs	_	10-150	5-30

7.1.1 Case-study on Edge Detection

Edge detection is one of the most significant tasks in image processing systems. For the last few decades a lot of researches has been done in this field. Various edge detection techniques are proposed: Quick Mask, Sobel, Prewitt, Kirsch [93], Canny [29] or Canny with loop [95]. Gradient based edge detectors like Prewitt and Sobel are relatively simple and easy to implement, but are very sensitive to noise. The Canny algorithm is an optimal solution to problem of edge detection which gives better detection specially in presence of noise, but it is time consuming and require a lot of parameter setting. As can be seen in Figure 7.1, the execution time of Quick Mask is the shorter and Canny is the longest (tested on a standard Intel Core i3@2.53GHz).

In an ideal world, a programmer would use the best algorithm and be done with it, but possible real-time constraints can mitigate this idyllic view. When dealing with timing constraint, an average quality result at the right time is far better than an excellent result, later. The Canny filter may be the best algorithm for edge detections, but the execution time depends on the input image. In contrast, Quick Mask or Sobel have image-independent execution time (*i.e.* depending only on the size of the input image, not on its contents). So we can use a control actor of type clock and a Transaction kernel set to be fired in mode 3 "Select available data input with the highest priority" to implement this time constraint, as can be seen in Figure 7.1. If a deadline arrive soon Quick Mask will be chosen. In the other cases, the best current result with Canny will be delivered. This kind of time-dependent decision is not available in usual CSDF. By contrast, our model fits well the case by using a transaction filter. The exercise is now to find how our new proposal could satisfy the real-time constraints of this type of application. A ΣC implementation of this method to use the transaction filter could be seen in Figure 7.1. The *IRead* kernel reads images from source and duplicates this image to be tested by different Edge Detection methods: Quick Mask, Sobel, Prewitt, Canny,... and its output is connected directly to the Transaction kernel. This box will select the best results in

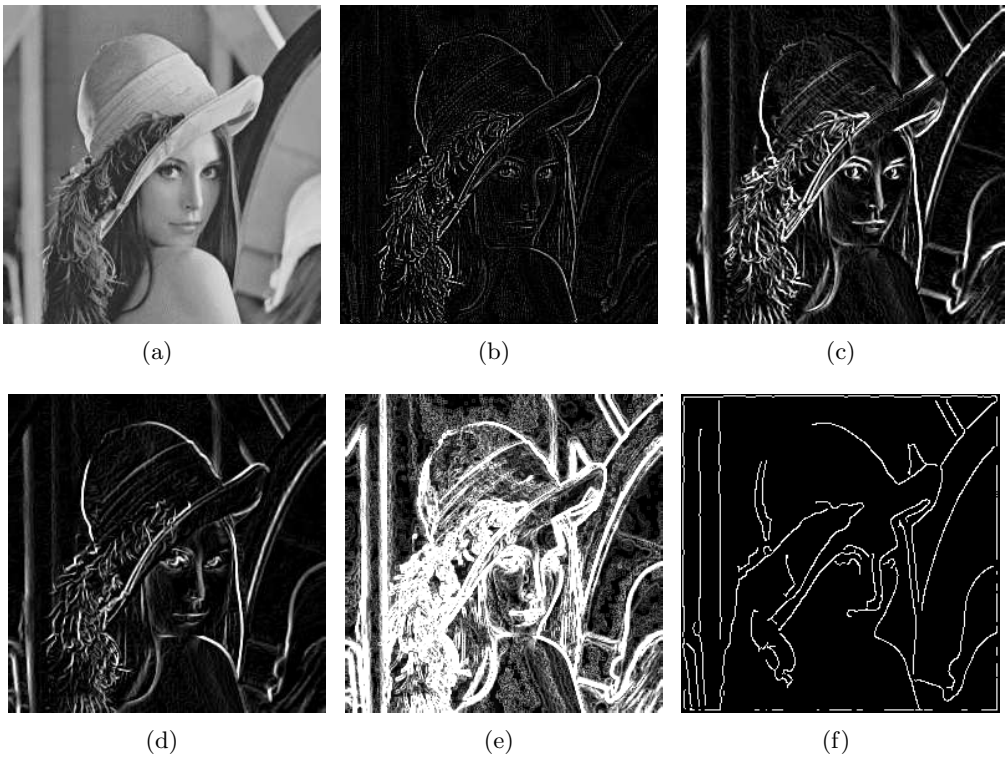
accordance with the deadline, implemented by using a control token received from the control actor every $500ms$. As can be seen in Figure 7.2, Quick Mask gives an acceptable result, Sobel and Prewitt is a little better while Canny give the best result with an execution time which is about 400% higher than Quick Mask.



	Quick Mask	Sobel	Prewitt	Canny
Time(ms)	200	473	522	1040

Figure 7.1: TPDF graph of the Edge Detection application. Omitted rates equal to the image size $p \times q$. Execution time of different methods are measured for a 1024×1024 image. At the deadline, the best result will be chosen by the Transaction kernel, according to the order: Canny > Prewitt > Sobel > Quick Mask.

A more elaborate case could want to improve the quality of result by using incremental feedback loop. For example, the curvature scale space (CSS), which detects corners using the intuitive notion of locating when the contour of an object makes a sharp turn in the image, is a state-of-the-art corner detector. However, the success of the CSS is strongly dependent on the quality of the input boundary segmented image, which is obtained by Canny edge detection [95]. With a too high threshold used as default edge detection parameters, the segmentation of a low contrast image of a ship corner, taken in welding preparation phase, is poor, as shown in Figure 7.3(a). However, a too low threshold yields an over-segmented image, as shown in Figure 7.3(b). Neither an under- nor an over-segmented image is a good input to the CSS corner detector. The first one allows detection of only few corners, while the latter one results in detection of too many false positives. It is therefore decisive that a threshold value that will yield reasonable and efficient segmentation, as shown in Figure 7.3(c), is defined [95]. The suggested feedback loop structure, shown in Figure 7.1, aims to do this automatically before the segmentation results are passed to higher level of the CSS system. The control objective of the closed-loop is to provide a binary segmented image which is free of noise. It is accomplished by changing the threshold of the Canny edge detector, so that the measured variable, two-dimensional entropy S of edge segmented pixels, is minimized. We may not know



(a) (b) (c)

(d) (e) (f)

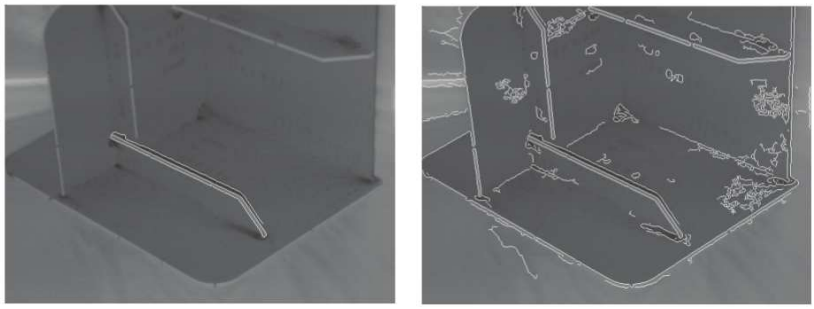
Figure 7.2: Results of different Edge Detection methods: (a) The image source (b) Quick Mask (c) Sobel (d) Prewitt (e) Kirsch (f) Canny.

how many loops can be done before the deadline occurs. As can be seen in Figure 7.1, if the incremental path is still running when the deadline occurs, the latest available corner detected (with a two-dimensional entropy minimized) is chosen. Again, usual CSDF cannot express this kind of time- and context-dependent decision.

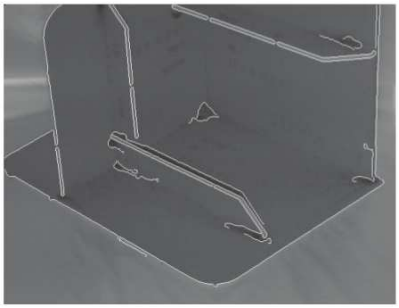
7.1.2 Case-study on Viola & Jones

The Viola & Jones object detection framework is the first object detection framework to provide competitive object, face or person detection rates in real-time proposed by Paul Viola and Michael Jones [109]. There are three components in the framework. The first is the introduction of a representation called the Integral Image which allows the features used by the detector to be computed very quickly. The second is a learning algorithm, based on AdaBoost, which selects a small number of critical visual features from a larger set and yields extremely efficient classifiers. The third is a method for combining increasingly more complex classifiers in a Cascade which allows background regions of the image to be quickly discarded while spending more computation on promising object-like regions. The cascade can be viewed as an object specific focus-of-attention mechanism which unlike previous approaches provides statistical guarantees that discarded regions are unlikely to contain the object of interest.

However, in most applications of person or object detection (*e.g.* in avionics, military and video surveillance), a high level of accuracy is not sufficient, as real-time constraints are equally critical. For example, with a fighter, all persons or objects detected must be synthesized after a firm deadline. The exercise is now to find how our new data



(a) (b)



(c)

Figure 7.3: Original object image overlaid with the edge-contours detected using "too high" (a), "too low" (b) and optimal (c) threshold (Source: [95])

distribution filters could satisfy the real-time constraints of this application. A TPDF implementation of this method to use the transaction filter could be seen in Figure 7.4. An image will be analysed and tested in different scales and only persons, objects or interest point detected before a deadline will be selected by a *transaction* box. As can see in Figure 7.5, the transaction filter selects only results from *GoodPoint* which arrive before the deadline and the number of persons detected on the image will also change. In the first case with a longer deadline, 2 persons and 1 object-like person are detected. In the second case, only 2 persons are detected and in the third case with a deadline which arrives too soon, no person or object is detected. This kind of time-dependent decision is not available with usual programming models based on SDF or CSDF MoCs, contrary to our model which is more dynamic and context-dependent by integrating transaction filter as a new data distribution filter.

7.1.3 Case study on Satellite positioning

Our example to study is the case of satellite positioning thanks to data-fusion. The concept of data-fusion is to use several sources from different inputs to infer a value that matches all the relevant inputs. A strong constraint we want to impose is resilience to a failure of a part of the input sources but also to be able to use them again easily when they become relevant again.

Potentially, relevant sources for satellite [1] localization, as can be seen in Figure 7.6, are

1. This example is based on a real-life application of a low-cost "satellite" attached to a weather balloon.

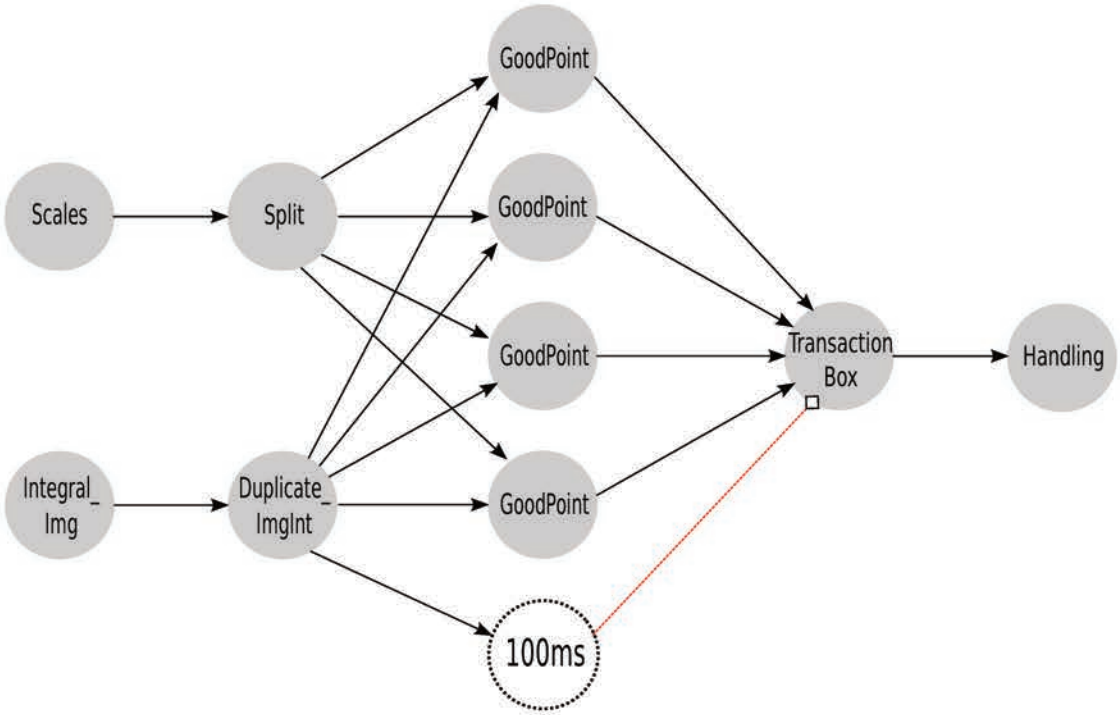
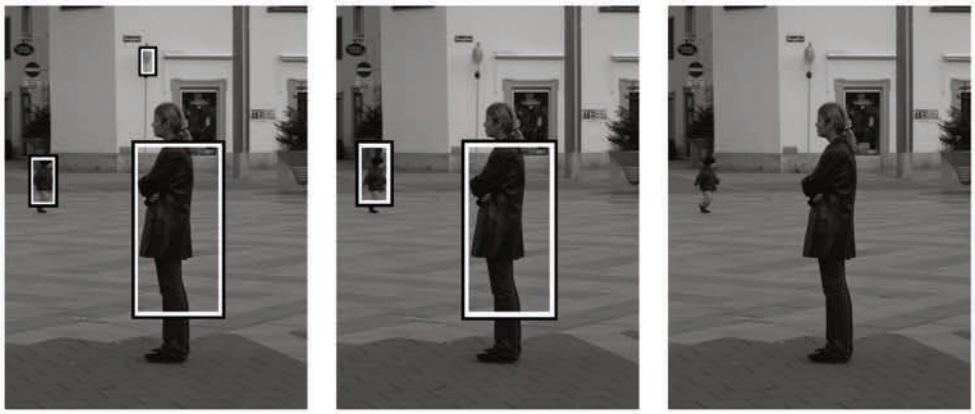


Figure 7.4: TPDF graph of the Viola & Jones application. The *Scales* is to calculate scales on which the image will be tested, *Integral_Img*: calculate the integral of the image, *GoodPoint*: search positions where there may be a person or an object, *TransactionBox*: a *transaction* filter fired in mode 3 "Select available data input with the highest priority", *Handling*: clear positions that are too close and draw rectangles in these positions.



(a) (b) (c)

Figure 7.5: Person detection with the Viola Jones algorithm according to the arrival of the deadline (measured on Intel core i3@2.53GHz): (a) 2 persons and 1 object-like person are detected (b) 2 persons (c) nothing is detected with a deadline of 200, 100 and $50ms$, respectively.

the following: (1) 2 GPS units (base for localization, but may be not always available), (2) Captors (*e.g.* accelerometers, pressure/temperature sensors) and data from satellite

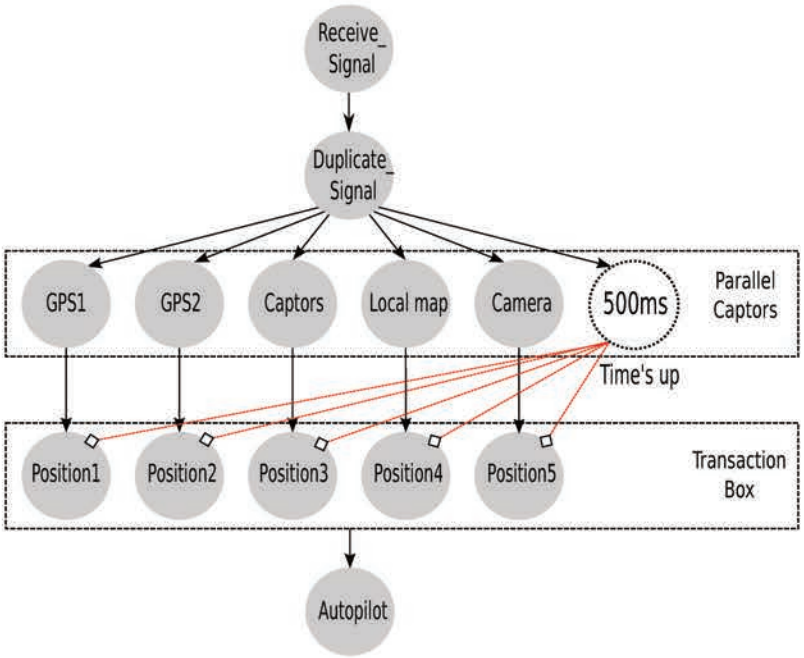


Figure 7.6: TPDF graph of the Satellite application with different sources of localization. At the end of the deadline, the best information is chosen, according to an order setup by the programmer.

(speed, angle of motors), with an interpolation of position (*e.g.* altitude from pressure) and redundancy, (3) A local map regularly updated from an online-service, (4) Camera capture data processing (planet trajectory or kind of topography, evaluation with regards to local map data) with higher compute time, and less accuracy, and obviously more power consumption. The process for data-fusion and localization of the satellite by using the 5 considered sources must take into account the fact that most of the sources are sporadic and may not be available each time a decision must be made. Such process is not available with usual CSDF derivatives. This use case illustrates the needs of creating a MoC whith the interesting properties of CSDF (execution determinism, time boundedness and execution in bounded memory) without its limitation. In this context, we have applied the TPDF model to treat the case of uncertain arrival of data. As can be seen in Figure 7.6, the transaction kernel, with n inputs and 1 output, implements important actions not available in usual dataflow MoC: Redundancy with vote and Best-of at a given deadline. In fact, this filter votes to choose the best source which arrives before a deadline, resolves the problem of redundancy in hard real-time systems (*e.g.* in avionics) by choosing between 2 different GPS and other sources of positions.

7.1.4 Case-Study on Cognitive Radio

We have applied the TPDF approach to an OFDM demodulator from the domain of cognitive radio, which is one of the fundamental subsystems of LTE and WiMAX wireless communication systems. Figure 7.7 illustrates a runtime-reconfigurable OFDM demodulator that is modeled as a TPDF graph. Here, actor *SRC* represents a data source that generates random values to simulate a sampler. In a wideband OFDM system, information is encoded on a large number of carrier frequencies, forming an OFDM symbol

stream. In baseband processing, a symbol stream can be viewed in terms of consecutive vectors of length N. The symbol is usually padded with a cyclic prefix (CP) of length L to reduce inter-symbol inference (ISI) [108]. In Figure 7.7, the CP is removed by actor *RCP*. Then, actor *FFT* performs a fast Fourier transform (FFT) to convert the symbol stream to the frequency domain. This kernel is connected to a M-ary *QAM* demodulation, with a configurable *QPSK* configuration ($M = 2$ or $M = 4$). Finally, the output bits are collected by the data sink *SNK*.

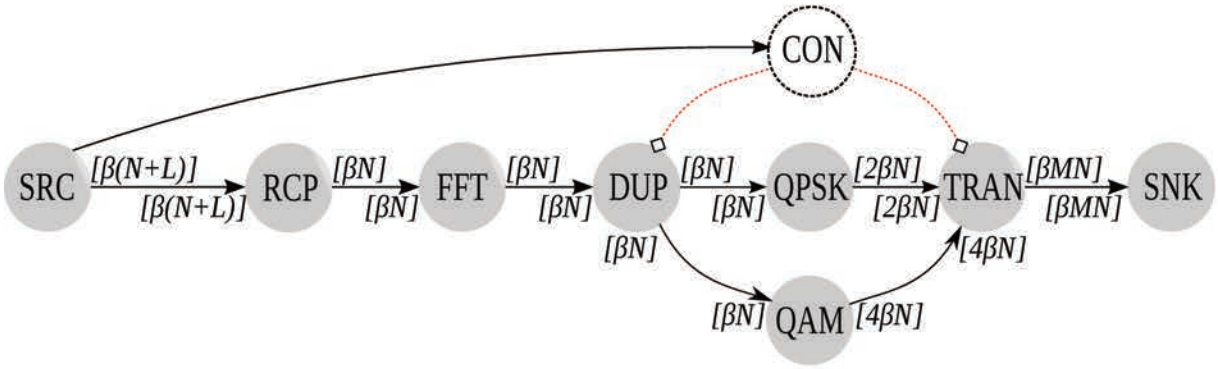


Figure 7.7: TPDF model of a OFDM demodulator with a configurable *QPSK* ($M = 2$) or *QAM* ($M = 4$) configuration. Omitted rates equal to 1.

In summary, there are four principal parameters: β, M, N and L, where L depends on the cyclic prefix, N is the OFDM symbol length ($N = 512$ or $N = 1024$) and β, which varies between 1 and 100, is the number of OFDM symbols to be processed in a single activation of the actor. For example, if $M = 4$ and $\beta = 10$, this means that the system is operating in a mode that uses *QAM* as the demapping scheme, and executes actors in blocks of 10 firings each. A possible scheduling for this application: *SRC* [*CON RCP FFT DUP QPSK QAM*] *TRAN SNK*. When *SRC* fired, it send a data token to the control node, which decides to choose between *QPSK* or *QAM*, depending on the value of M. The square bracket can be used to inform the compiler about the region influenced by the control token, starting from the firing of the control node and terminating with the firing of all nodes which receive this control token. Figure 7.8 presents the minimum buffer size required by the application, depending on the vectorization degree β and the symbol length N ($L = 1$ and M is chosen by the control node). We find out that the buffer size increases proportionally to the vectorization degree and we have an improvement of 29% in comparison with the implementation by using CSDF. This result can be explained by the fact that the dynamic topology obtained using TPDF is more flexible than the static topology of CSDF, allowing to remove unused edges and decrease the minimum buffer size required by one iteration of the TPDF graph. In a similar way, several StreamIt benchmarks (e.g., FM Radio [106]) must perform redundant calculations that are not needed with models allowing dynamic topology changes such as TPDF.

7.1.5 Case-Study on VC-1 Decoder

The VC-1 decoder [16, 66] is a good example of a demanding codec. Its resemblance with the more recent and widely used H.264 [78] or HEVC (also known as H.265 [103]), makes it especially relevant.

The decoder is composed of two main pipelines, the inter and the intra. The inter

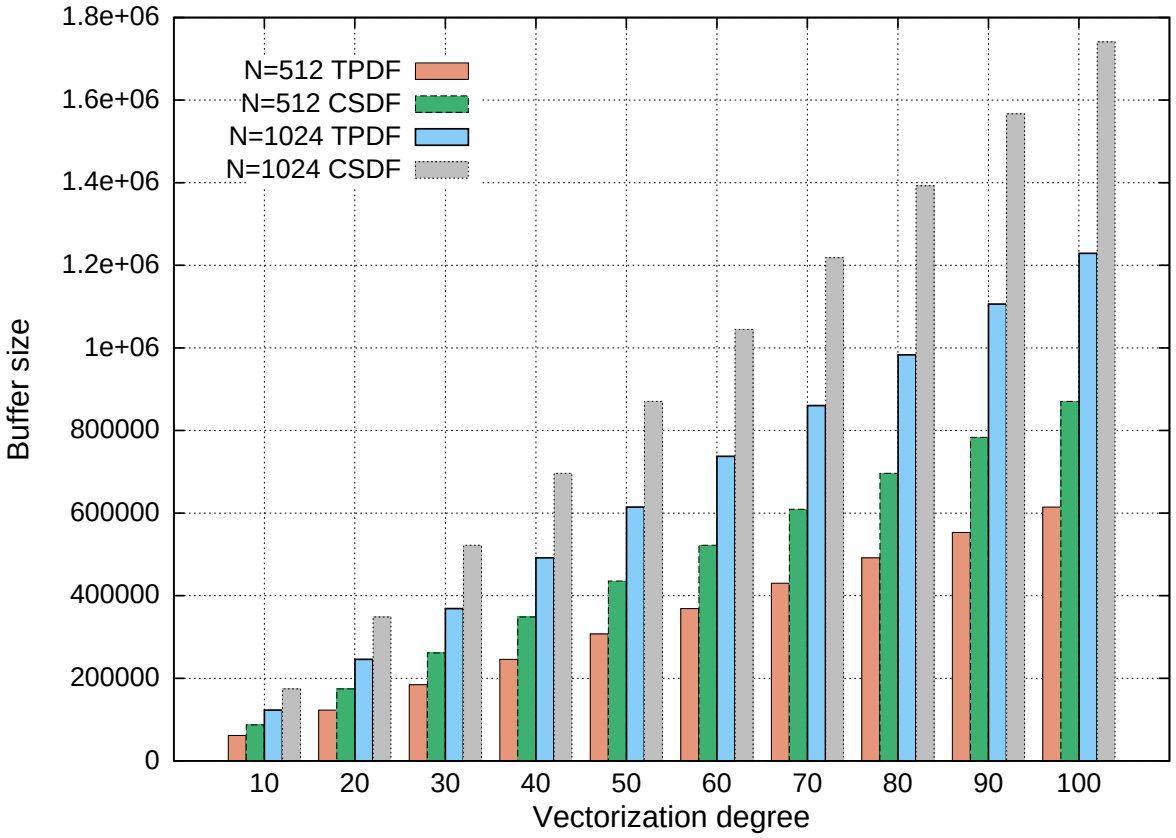


Figure 7.8: Minimum buffer size increased proportionally to β, given by $Buff = 3 + \beta \times (12 \times N + L)$ for TPDF and $Buff = \beta \times (17 \times N + L)$ for CSDF.

pipeline is composed of actors MV PRED, PREF, and MCOMP, while the intra pipeline is composed of actors PRED, IZZ, ACDC, IQIT, and SMOOTH. In the VC-1 decoder, three possible modes of operation can be distinguished: Intra only , Inter only, Intra and Inter. 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 but in the intra part the PRED actor is bypassed. The TPDF implementation of VC-1 is shown in Fig. 7.9. The decoder makes use of two integers and a control actor. The integer parameters are p, which encodes the slice size in macroblocks, and q, which encodes the macroblock size in blocks. The control actor captures whether a block is using intra, inter or both of these two modes.

7.2 Analysis Tool

We have implemented our method to check the static guarantees, compute the execution state-space and analyse the worst-case throughput of TPDF graphs in the publicly available SDF^3 software library for SDF, CSDF and SADF analysis [102]. In this new TPDF analysis tool written in C++, we reuses the native scheduling technique *canonical period* proposed by the native dataflow programming model of the MPPA-256 platform, as presented in Section 2.4.2 and 5.3.5. We also use a basic breadth-first-search to constructs

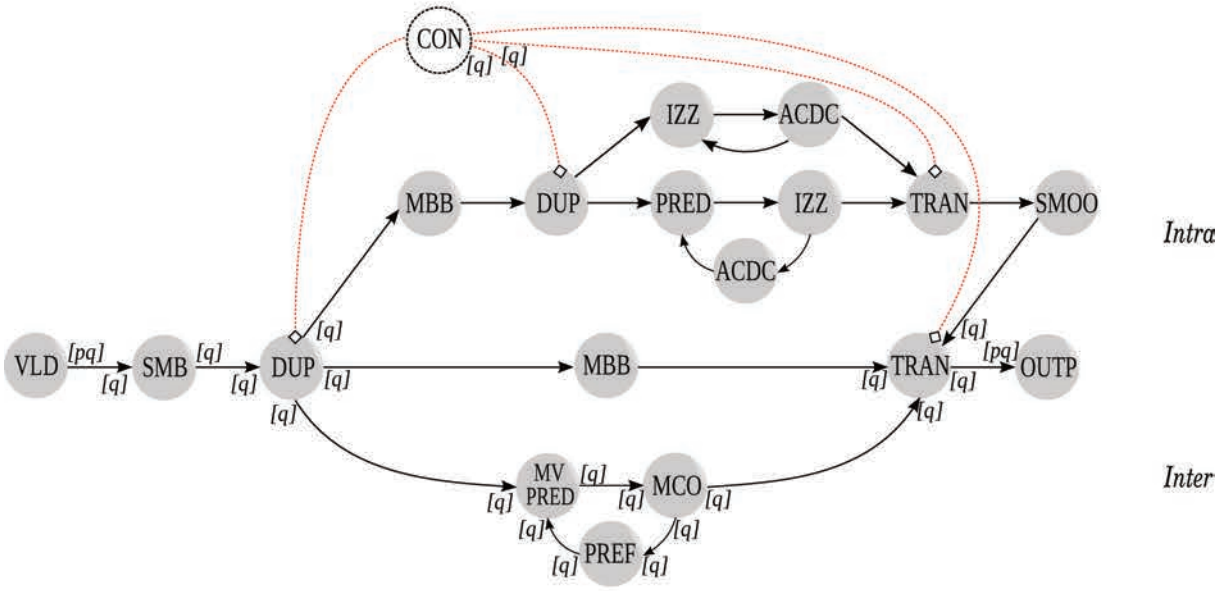


Figure 7.9: The VC-1 decoder captured in TPDF. Three possible modes of operation can be distinguished: *Intra only*, *Inter only*, *Intra and Inter*.

the state-space straightforwardly according to the definition of Section 5.3.4. During the exploration procedure, we enumerate the state-space and check timing constraints. Exploration continues until we reach a state that we visited before, we back-track and stop exploration in that direction.

This tool takes as input an XML file, describing a TPDF application with its kernels, control actors and channel. For example, Figure 7.10 illustrates the XML representation of the TPDF example graph in Figure 5.1. In this case, five kernels and one control actor is defined using keywords *kernel* and *control*, respectively. Similarly, five data channel and a control channel is defined by using the *channel* keyword, as seen in Figure 7.10.


```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="tpdf" version="1.0" xmlns:xsi="http://www.w3.org/2001/
<tpdf name="example1tpdf">

<structure>
  <kernel name="A"/>
  <kernel name="B"/>
  <kernel name="D"/>
  <kernel name="E"/>
  <kernel name="F"/>

  <control name="C"/>

  <channel name="A2B" source="A" destination="B" type="data"/>
  <channel name="B2C" source="B" destination="C" type="data"/>
  <channel name="B2D" source="B" destination="D" type="data"/>
  <channel name="B2E" source="B" destination="E" type="data"/>
  <channel name="D2F" source="D" destination="F" type="data"/>
  <channel name="E2F" source="E" destination="F" type="data"/>

  <channel name="C2F" source="C" destination="F" type="control"/>
</structure>
```

Figure 7.10: An XML representation of the TPDF example graph in Figure 5.1.

The TPDF features are implemented in several different options of the **tpdfanalyze** tool:

– **tpdfanalyze –graph graphfile –algo 'consistency'**: Check if the TPDF graph

is consistent or not.
- **tpdfanalyze –graph graphfile –algo 'liveness'**: Check if the TPDF graph is live or not.
- **tpdfanalyze –graph graphfile –compute 'buffersize'**: Computes the maximum buffer size for all channels.
- **tpdfanalyze –graph graphfile –compute 'throughput[(kernel—control actor)]'**: Computes the worst-case throughput for the specified kernel/control actor or for the graph.

7.3 Experimental Results

We implemented these applications, by using TPDF, SADF and a conservative CSDF/SDF model, without modes. Table 7.2 shows the throughput obtained using these models as well as the improvement of the results using TPDF compared to the SADF and CSDF/SDF model. For the OFDM application, a conservative SDF model, without modes, can guarantee a throughput of 1.58×10^{-2} iterations per cycle. This result by using SADF is 1.6×10^{-2}. Experiments with the TPDF graph show that the OFDM graph can achieve a guaranteed throughput of 2×10^{-2} iterations per processors cycle, 25% and 26.58% higher than SADF and SDF implementations, respectively. This result can be explained as follows: TPDF maximizes the degree of parallelism and driving around some sequential limitations by using speculation, a technique often used in hardware design to reduce the critical data-path, all kernels and cases which receive enough data tokens will be fired immediately and in parallel. Moreover, the kernel which receives the control token will also be fired instantly and choose the input or output data channel depending on the control token. All unnecessary kernels will be stopped or superseded, then computation will be accelerated and parallelized.

Table 7.2: Throughput obtained and the improvement of TPDF compared to the SADF and (C)SDF model ($\mathbf{Eff_{SADF}}$ and $\mathbf{Eff_{(C)SDF}}$, respectively). The last column represents the analysis time by using our tool set.

Application	TPDF	SADF	$\mathbf{Eff_{SADF}}$	(C)SDF	$\mathbf{Eff_{(C)SDF}}$	N	#States	Time(ms)
OFDM	2×10^{-2}	1.6×10^{-2}	25%	1.58×10^{-2}	26.58%	8	15	4
Adaptive Coding	10×10^{-2}	8.5×10^{-2}	17.65%	4.6×10^{-2}	117.39%	14	6	16
VC-1 Decoder	4×10^{-1}	2.5×10^{-1}	60%	2.2×10^{-1}	81.81%	12	12	8
Edge Detection	7.86×10^{-1}	-	-	-	-	8	18	12
Viola & Jones	1.48×10^{-1}	-	-	-	-	10	68	22
Satellite	5.5×10^{-1}	-	-	-	-	9	55	20
Random Graphs	1.5×10^{-1}	1.29×10^{-1}	15.51%	1.09×10^{-1}	36.6%	10-150	5-30	-

VC-1 Decoder is another application modelled by BPDF, a recent dynamic dataflow model [16]. The existing analysis method of this model takes only into account the maximum throughput. Our technique focuses on the worst-case throughput (i.e., a guaranteed lower bound on the application throughput), which is a more interested performance metric. Experiments with the TPDF model shows that we have an improvement of 60% compared to the SADF model and 81% to the conservative SDF model. For the Edge Detection case study, as discussed in Section 7.1.1, our tool succeed to analyse the throughput of the graph with a state-space of 18 states and the analysis took only 12ms. This type of time constraint is complicated, even impossible by using SADF or SDF model.

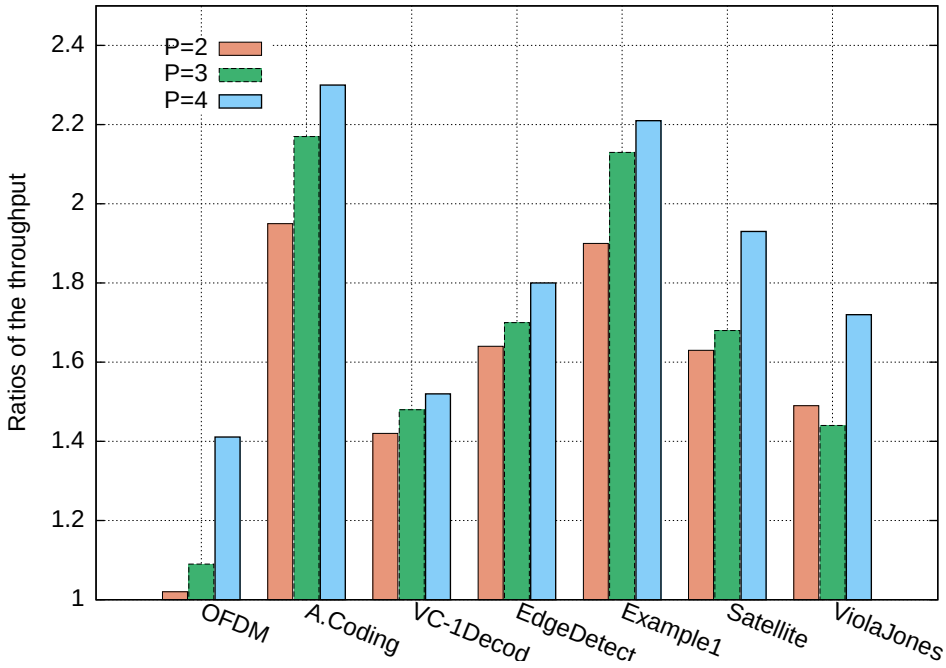


Figure 7.11: Ratios of the throughput under different degrees of pipelining, compared to the case when only one canonical period is used.

We have also extended these results by testing the capacity of pipelining canonical periods of TPDF graphs. This experiment is to test the ability of TPDF to adapt the compile tool chain used for MPPA-256, which constructs for each CSDF application a canonical period per iteration and optimise the parallelism by pipelining these canonical periods. Figure 7.11 shows the ratios of the throughput by using a degree of pipelining of 2, 3 and 4 compared to the case when only one canonical period is used. We can see that a higher level of pipelining (under 4) gives always a higher throughput. In this way, we can conclude that TPDF is well adapted to the existing parallelism method used for ΣC and its present real-world many-core platform, the Kalray's MPPA. We have also analysed a collection of more than 200 random TPDF graphs between 10 and 150 actors with an entry in the repetition vector between 1 and 10 and between 5 and 30 modes to use, giving an average of 15.51% and 36.6% higher throughput guarantee than from a SADF and a conservative CSDF model of the same graph, respectively. The execution times for state-space analysis are summarised in Figure 7.12, averaged for graphs with a particular number of kernels. The graph shows that the average analysis times scale roughly linear with the complexity of the graph and the number of kernels. The number of kernels can be analysed is approximately 1400 kernels/s. This fact shows the ability of TPDF and its analysis tool to be expanded for real-time streaming applications with millions of execution units, which should be available in the near future.

7.4 Summary

In this chapter, the TPDF model and its associated methods to check the consistency, liveness, boundedness and worst-case throughput is validated using a set of real-life dynamic applications. We used benchmarks from different sources to check the expres-

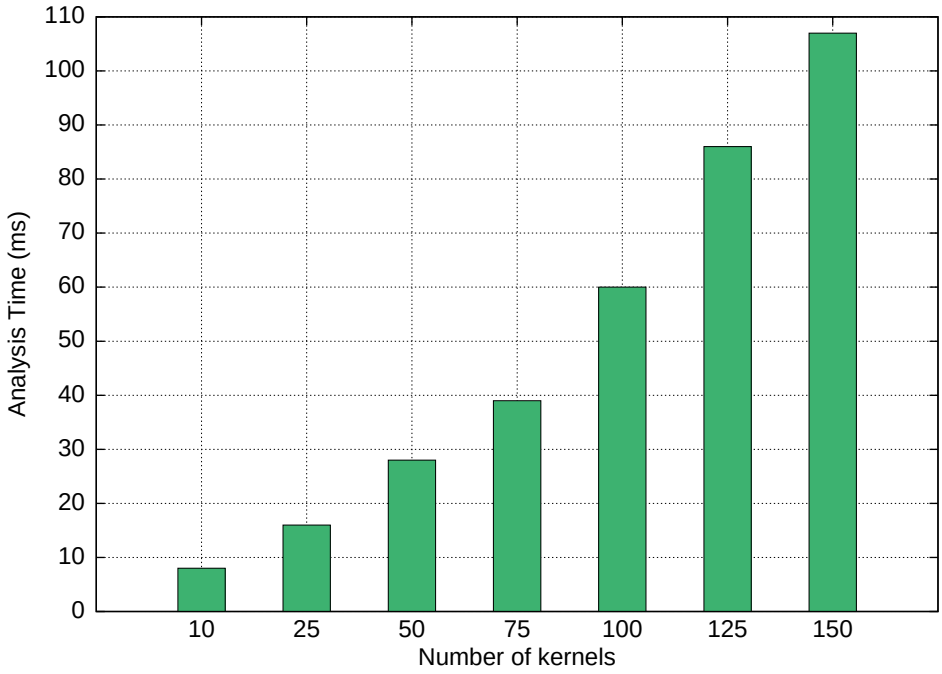


Figure 7.12: Average analysis times in ms for random graphs.

siveness of TPDF and its efficiency when implemented in different architectures. The experimental results demonstrates significant buffer size and performance improvements compared to the state of the art models, including Cyclo-Static Dataflow (CSDF) and Scenario-Aware Dataflow (SADF). Moreover, by combining two data distribution kernels (Select-duplicate and Transaction Box) and a new type of control clock with the basic model CSDF, in valid combinations that are intuitively easy to figure out and that can be checked automatically by a compiler, we demonstrated that the main properties of CSDF graphs are preserved in TPDF, while overcoming a range of limitations that use to be associated with CSDF models. Lifting these restrictions is useful on real-time and cyber-physical systems with computational requirements: Speculation, Redundancy with vote, Best-of at a given deadline, Select an active data-path among several, and so on. The real-life case-studies tested in this section demonstrated also the importance of TPDF to define a new set of applications for embedded computing, which can be encompassed in the coming years.

Chapter 8

Conclusions

I have strengths, and I have weaknesses. I don't pretend to be able to write a great thesis or doctorate - I have no pretensions in that direction.

— Bob Ainsworth

Contents

8.1 Conclusions 119
8.2 Open Problems and Future Research 120
8.2.1 The STP scheduling policy 120
8.2.2 The TPDF Model of Computation 121
8.2.3 Compilation toolchain 123

8.1 Conclusions

In this thesis, we resolve two existing challenges of dataflow models and streaming languages to meet the needs of emerging complex signal and media processing applications: 1) How to provide guaranteed services against unavoidable interferences which can affect real-time performance?, and 2) How these streaming languages which are often too static can meet the needs of emerging embedded applications, such as context- and data-dependent dynamic adaptation? For the first challenge, in Chapter 3, we proved that the actors of a streaming application modeled as CSDF graph, can be scheduled as self-timed periodic tasks. As a result, we proposed four classes of STP schedules based on two different granularities and two types of deadline: implicit and constrained. Two first schedules, denoted $STP^I_{q_i}$ and $STP^C_{q_i}$, are based on the repetition vector q_i without including the sub-tasks of actors. Two remaining schedules, denoted $STP^I_{r_i}$ and $STP^C_{r_i}$, have a finer granularity by including the sub-tasks of actors. It is based on the repetition vector r_i. Based on empirical evaluations, we showed that our STP approach reduces significantly the latency compared to the SPS model and often delivers the maximum throughput achieved under the STS model. We summarize our results in the form of a decision tree to assist the designer in choosing the appropriate scheduling policy for acyclic CSDF graphs.

We presented also in Chapter 4 an analytical framework for computing the the periodic task parameters for the actors of CSDF graphs while taking into account inter-processor

communication and real-time constraints imposed by hardware devices or control engineers. Based on this, we evaluate the latency between starting times of any two dependent actors, and we introduce a latency-based approach for fault-tolerant stream processing modeled as a CSDF graph, addressing the problem of node or network failures. We view this work as an important first step to provide a failure-handling strategy for distributed real-time streaming applications.

For the second challenge, we developed and presented TPDF in Chapter 5, a novel parametric data flow Model of Computation (MoC) that allows dynamic changes of the graph topology, variable production/consumption rates and time constraints enforcement. Despite the increase in expressiveness, TPDF 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 TPDF 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.

A difference of TPDF from other dynamic dataflow models is that our model allows time constraints enforcement. In Chapter 6, we introduced a method to systematically translate from TCA to TPDF, a time-constrained model widely used to model multi-task critical real-time Instrumentation and control (I&C) systems. In this way, we demonstrated that all properties (before and after constraints, visibility principle) of TCA are preserved while modeling its applications by using the TPDF model. We illustrated the possibility to mix timed dataflow actors with hard real-time I&C by using the safety-classified Qualified Display System (QDS) application and demonstrated that the TPDF implementation guarantees the worst-case latency required for the display of incoming data.

Finally, the proposed model and its associated methods to check the consistency, liveness, boundedness and worst-case throughput is validated in Chapter 7 using a set of real-life dynamic applications. For the worst-case throughput, we use the (max, +) algebra to capture the dynamic semantics of modes introduced by TPDF graphs by considering this MoC as a dynamic switching between cases (each case can consist of different modes). An implementation of the analysis method was tested and demonstrated significant buffer size and throughput improvements compared to the state of the art models, including CSDF and SADF. Moreover, thanks to several real-life case-study, we showed that TPDF overcomes a range of expressive limitations often associated with dataflow models (e.g., dynamic variations of consumption/production rates, time constraints or dynamic reconfiguration of the graph).

8.2 Open Problems and Future Research

Static dataflow models enable powerful analyses for parallel embedded applications while dynamic dataflow MoC (e.g., TPDF) hepls to tackle its limitations to express dynamic behaviors of new emerging embedded streaming applications. Both of these two directions to implement embedded streaming applications can be further extended to offer a better system-level design flow for streaming programmers.

8.2.1 The STP scheduling policy

Improving the WCET by Considering the Effect of Mapping In Chapter 3, we assume that the WCET of an actor is computed assuming the worst-case latency of com-

munication operations. This worst-case latency occurs when the underlying interconnect is fully congested. However, such assumption overestimates the WCET value. In a real system, many communication streams are isolated from the others. Therefore, communication operations occur without congestion and they do not take their worst-case latency. Therefore, it is possible to reduce the WCET values if the mapping is taken into account. A first step towards "communication-aware" allocation in hard real-time systems realized on MPSoCs is the work presented in [114]. Zimmer and Mueller in [114] presented a framework for deriving low-contention mapping of real-time programs mapped onto NoC-based MPSoCs. They devised two solvers: one based on exhaustive search and another based on a heuristic. The resulting mapping tries to reduce the communication contention and, hence, reduce the communication latency. This, in turn, leads to a tighter WCET estimates of the tasks.

Support for Hierarchical Scheduling The scheduling policy and mapping derivation explained in Chapter 3 does not support hierarchical scheduling. Hierarchical scheduling is becoming more popular in modern hard real-time systems since it allows different programs to be scheduled using different scheduling policies. Furthermore, in some application domains such as avionics, it is mandatory to use two-levels of scheduling in order to provide complete partitioning in time and space as mandated by industry standards (such as ARINC 653 Specification [61]). Therefore, it is interesting to investigate how such hierarchical scheduling schemes affect the derivation of the architecture and mapping specifications.

Optimal granularity The study of the whole CSDF granularity, throughput and latency trade-off space is a very interesting open issue. In this thesis, we consider two degrees of granularities (i.e., STP_{q_i} and STP_{r_i}) naturally offered by the CSDF model. Finding the optimal granularity for all the actors in the CSDF graph while considering the constrained deadline and achieving maximum throughput or/and minimum latency is a very interesting direction.

Towards More Accurate IPC estimation In Chapter 3 and Chapter 4, we assume that the inter-processor communication of actors assigned to the periodic level is computed assuming that all the memory accesses are done on the same memory bank. However, such assumption overestimates the IPC value. In a real system, many communication streams are isolated from the others to ensure that a memory bank would not be a hot-spot for memory accesses. Therefore, it is possible to reduce the IPC values if this assumption is taken into account. A first step towards "random access memory-modules" allocation is the work presented in [83], in which the author dealt with probabilistic contention issues in uniformly accessing the shared memory modules of the MPPA-256 architecture.

8.2.2 The TPDF Model of Computation

TPDF 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 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 and/or polynomial. Fractional rates can be used to avoid duplication of the same value. Moreover, it provides a more intuitive representation of some applications. An example is given in Figure 8.1.

Actor a_2 needs to produce only a single token on edge e_2. However, for the graph to be consistent, a_2 fires p times producing p tokens (Figure 8.1(a)). Data duplication can be avoided with a fractional rate of $\frac{1}{p}$ (Figure 8.1(b)). In this way, a_2 fires p times but produces only a single token on edge e_2.

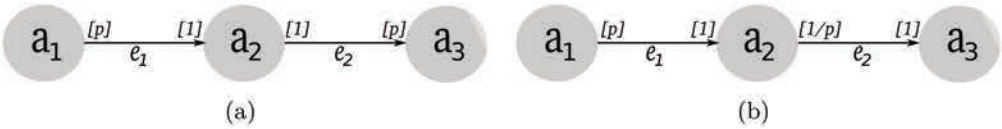


Figure 8.1: (a) A TPDF graph with data duplication on edge e_2 (b) Equivalent TPDF graph with fractional rates to suppress data duplication; Actor a_2 needs to produce a single token. In Figure 8.1(a), the token is duplicated p times because of consistency.

Polynomial rates are implicitly used in TPDF when a multi-rate graph is used, as shown in Figure 8.2(a). When the multiple edges are reduced to one (Figure 8.2(b)), the graph has parametric rates. However, graphs like the on in Figure 8.2(c) 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.

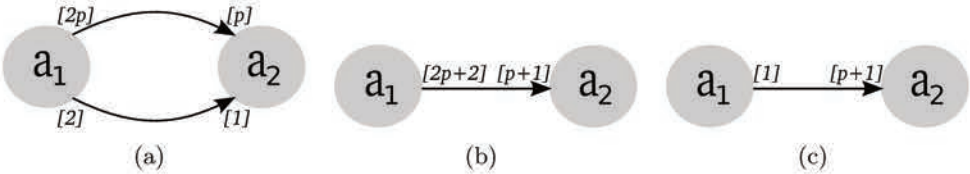


Figure 8.2: (a) A TPDF multi-graph (b) Equivalent TPDF graph with the multiple edges reduced to one and polynominal rates (c) Graph not supported by TPDF.

In Chapter 7, we have done experiments by building TPDF graph from real-life application programmed in C (e.g., Edge Detection). From this approach, dynamic code analysis tool, as the method introduced in [72], can be developed in the near future to transform automatically legacy code to TPDF graph. The principle of this tool is to transform each function into a TPDF kernel and connect these kernels by its common data. Moreover, this tool has to detect automatically time constraints and transaction process to build control actor and set the mode in which this actor will be fired. Another possible benchmark source is to build TPDF graphs from other dynamic models, like the transformation tool to convert between ΣC and StreamIt, as presented in Section 2.4.3.

Moreover, the actor merging approach [24], i.e. automatic composition of high granularity actors from a set of low granularity actors, can be used as a solution for the performance portability issue of TPDF applications in different manycore platforms. In practice, the specification granularity of dataflow applications remains an arbitrary choice of the designer. Dataflow specifications of the same application with equivalent I/O behaviour can range from a single dataflow actor to a very fine grained network composed of elementary processing operations. A very fine grained dataflow specification might result into a high performance implementation when synthesized as hardware, but might

perform poorly when executed on a programmable processor. In this context, the actor merging approach introduced in [24] can be used as a methodology to increase performance metrics for not only static but also dynamic dataflow model (e.g., TPDF).

Finally, the TPDF analysis tool has been implemented as a standalone model in C++. Such an implementation makes comparison with other existing dataflow MoCs difficult. TPDF needs to be integrated in larger frameworks such as Ptolemy [94] and its extensions PeaCE [56], Open RVC-CAL [112] and TURNUS [30] as well as Dataflow Intechange Format (DIF) [60], to make such a comparison possible.

Dataflow 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.

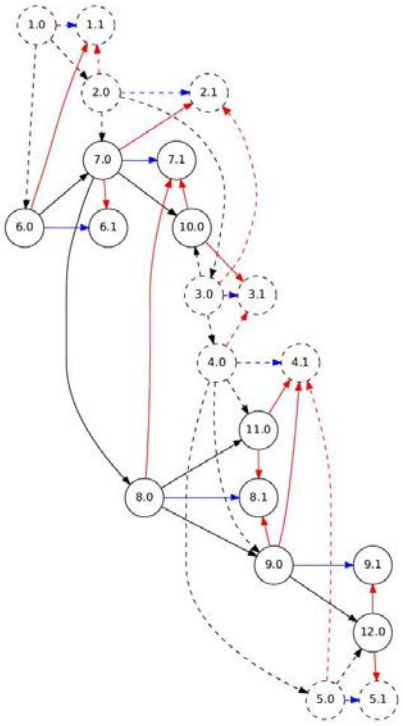
8.2.3 Compilation toolchain

A TPDF compilation toolchain has been developed to map real-time streaming applications on the MPPA chip architecture. This toolchain is based on the ΣC programming tool, as presented in Section 2.4.2. However, the biggest difference between two compilation toolchain is the static model CSDF of ΣC is replaced by our dynamic model TPDF. As a result, several changes have to made for the fours stages of the compilation toolchain: from the frontend, which performs syntactic and semantic analysis, to the runtime generation, which builds the final binary for the MPPA chip. Until now, 3/4 steps of the new compilation toolchain is developed to use the TPDF as the model of programmation and expand the compilation toolchain to not only homogeneous architectures such as MPPA-256 but also new emerging heterogeneous architectures:

- For the frontend with lexer, parser and code generator, new syntaxes are added to well define a control actor, its ports and its communication channels, as can be seen in Figure 8.3.
- The second step of the compilation toolchain is to build an intuitive graph representation of the application, as can be seen in Figure 8.4. This representation can be used for both compiler internal processing and developer debug interface. Once built, further analyses are applied to check that the graph is well-formed and that the resulting application fits to the targeted host. The internal representation of the application (made of C structures) is also designed to ease the implementation and execution of complex graph algorithms.
- For the Resource allocation stage for scheduling, dimensioning, placing & routing, the scheduling algorithm has been developed to adapt the TPDF scheduling policy as presented in 5.3.5. Moreover, new task mapping method needs to be developed for heterogeneous architecture.
- The fourth step is under development and will be presented in the near future.

```
58  control Averager(int n) {
59
60  interface
61      {
62          in<int> input;
63          controlout<int> output;
64
65          spec{input ; output };
66      }
67  int sum;
68  void start () exchange ( input myIn, output myOut)
69      {
70      if(myIn == 0) {
71          myOut = WAIT_UNTIL_ALL;
72      } else if(myIn == 1) {
73          myOut = SELECT_FIRST_INPUT;
74      } else {
75          myOut = SELECT_SECOND_INPUT;
76      }
77  }
78  }
79
80  agent BlockTransaction(int size)
81  {
82      interface
83      {
84      controlin<int> input1;
85      in<int,1> input2;
86      in<int,2> input3;
87          spec{input1; input2; input3};
88      }
89
90      void start () exchange (input1 val1, input2 val2,input3 val3)
91      {
```

Figure 8.3: TPDF code of a simple application. Compared with ΣC programming language, the *control* keyword is added to define a control actor, the *controlin* and *controlout* is also used to define control in and out ports.



(a)

```
−<DB>
 −<application>
  −<dataflows>
   −<dataflow id="0" top_level="0" label="TaskTime">
     <timestamp seconds="0" useconds="0"/>
    −<types>
      <type id="0" name="root" family="subgraph" unit="TaskTime"/>
      <type id="1" name="Clock1" family="N_A" unit="TaskTime"/>
      <type id="2" name="Clock2" family="N_A" unit="TaskTime"/>
      <type id="3" name="Clock3" family="N_A" unit="TaskTime"/>
      <type id="4" name="Clock4" family="N_A" unit="TaskTime"/>
      <type id="5" name="Clock5" family="N_A" unit="TaskTime"/>
      <type id="6" name="TaskA" family="agent" unit="TaskTime"/>
      <type id="7" name="TaskB" family="agent" unit="TaskTime"/>
      <type id="8" name="TaskC" family="agent" unit="TaskTime"/>
      <type id="9" name="TaskD" family="agent" unit="TaskTime"/>
      <type id="10" name="BlockTransaction" family="agent" unit="TaskTime"/>
      <type id="11" name="BlockTransaction2" family="agent" unit="TaskTime"/>
      <type id="12" name="BlockTransaction3" family="agent" unit="TaskTime"/>
     </types>
    −<components>
     −<component name="root_0" id="0" type_id="0">
       <interface/>
      −<topology>
       −<nodes>
          <node id="1"/>
          <node id="2"/>
          <node id="3"/>
          <node id="4"/>
          <node id="5"/>
          <node id="6"/>
          <node id="7"/>
          <node id="8"/>
          <node id="9"/>
          <node id="10"/>
          <node id="11"/>
          <node id="12"/>
        </nodes>
       −<edges>
          <edge id="0" link_id="10" source_component="6" source_port="15" sink_component="7" sink_port="17"/>
          <edge id="1" link_id="11" source_component="7" source_port="18" sink_component="10" sink_port="27"/>
          <edge id="2" link_id="12" source_component="7" source_port="19" sink_component="8" sink_port="20"/>
          <edge id="3" link_id="13" source_component="8" source_port="21" sink_component="11" sink_port="29"/>
          <edge id="4" link_id="14" source_component="8" source_port="22" sink_component="9" sink_port="24"/>
          <edge id="5" link_id="15" source_component="9" source_port="25" sink_component="12" sink_port="31"/>
        </edges>
       </topology>
      </component>
```

(b)

Figure 8.4: (a) The graph representation and (b) its XML file of a simple TPDF application; the dashed boxes and lines represents different instances of the control actors and channels.

Acronyms

(C)SDF (Cyclo-Static) Synchronous Dataflow 20, 58, 71, 83, 106

ADC Analog to Digital Converter 7

ADF Affine Dataflow xi, 63, 65–68, 70

BDF Boolean Dataflow xi, 11, 21, 22, 28

BPDF Boolean Parametric Dataflow xi, 21, 26–28, 81, 88, 114

CSDF Cyclo-Static Dataflow v–viii, xi, 1, 2, 5, 9–11, 13, 15–18, 20, 21, 25, 27–29, 33, 36–38, 41–49, 52, 54, 57–59, 61–65, 67–73, 75–80, 82, 84, 85, 87, 88, 90, 104–107, 109, 111–114, 116–119, 121

DAC Digital to Analog Converter 7, 45, 67

DCT Discrete Cosine Transform xi, 28, 29, 33, 35, 38–40, 54, 71

DDF Dynamic Dataflow 76

DSP Digital Signal Processing 15, 57

FFT Fast Fourier Transform 22, 38, 39, 54, 55, 71, 111

FIFO First-In First-Out 15, 16, 30, 33, 45, 54, 62, 105

FSM Finite State Machine 85

FSMs Finite State Machines 14

HMI Human-Machine Interface 98, 99

HSDF Homogeneous Synchronous Dataflow 20, 58, 68

IDF Integer Dataflow 21, 22, 28

ILP Instruction Level Parallelism 4

IPC Interprocessor Communication 36, 42, 59, 62, 119

KPN Kahn Process Network 14

KPNs Kahn Process Networks 14, 15

lcm Least Common Multiple 20, 64, 96

MCM Maximum Cycle Mean 85

MdC Modèle de Calcul 1, 2

MoC Model of Computation v, 9, 10, 15, 21, 25, 28, 75, 76, 78, 88, 111, 118

MoCs Models of Computation v, 6, 9, 10, 12, 14, 27, 28, 41, 109, 120

MPSoC Multiprocessor Systems-on-Chip vii, 3, 4

NoC Network-on-Chip 4, 36, 39, 104, 105

OFDM Orthogonal Frequency-Division Multiplexing 105, 106, 111–113

PE Processing Element 104

PSDF Parameterized Synchronous Dataflow xi, 21–24, 28, 87

QDS Qualified Display System xiii, 97–100, 118

RTS Hard-Real-time Scheduling 63, 72

SADF Scenario-Aware Dataflow vi, xi, 2, 10, 21, 23–25, 28, 42, 88, 105, 106, 113, 114, 116, 118

SDF Synchronous Dataflow 5, 9–11, 15, 20–25, 27–29, 36, 37, 41, 54, 57–59, 71, 73, 75, 77, 80, 82, 83, 88, 105, 109, 113, 114

SoC Systems-on-Chip vii, 3, 4

SPDF Schedulable Parametric Dataflow xi, 21, 25, 26, 28, 81, 87, 88, 119

SPS Strictly Periodic Scheduling v, 1, 9, 19, 20, 41, 44, 45, 53–57, 117

STP Self-Timed Periodic v, viii, ix, xiii, 1, 9, 11, 36, 41, 43, 44, 47–49, 53, 54, 56, 57, 59, 61, 117, 118

STS Self-Timed Scheduling v, 1, 9, 11, 18, 19, 41, 43–45, 54–57, 62, 64, 72, 73, 117

TCA Time Constrained Automata viii, xii, 89–93, 95, 98, 100, 118

TPDF Transaction Parameterized Dataflow v, viii, ix, xii, xiii, 2, 10–12, 42, 75–90, 92–94, 99–101, 103, 105, 106, 108, 110–123

VRDF Variable-Rate Dataflow 21, 87

WCET Worst-Case Execution Time 20, 46, 118, 119

Personal Publications

Published papers

Xuan Khanh Do, Stéphane Louise, and Albert Cohen. Transaction parameterized dataflow: A model for context-dependent streaming applications. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 960–965, March 2016.

Xuan Khanh Do, Amira Dkhil, and Stéphane Louise. Self-timed periodic scheduling of data-dependent tasks in embedded streaming applications. In *Algorithms and Architectures for Parallel Processing*, volume 9529 of *Lecture Notes in Computer Science*, pages 458–478. Springer International Publishing, 2015.

Xuan Khanh Do, Stéphane Louise, and Albert Cohen. Managing the latency of data-dependent tasks in embedded streaming applications. In *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 9–16, 2015.

Xuan Khanh Do, Stéphane Louise, Albert Cohen, Thierry Goubier, Paul Dubrulle and Philippe Doré. An empirical evaluation of a programming model for context-dependent real-time streaming applications. In *International Conference On Computational Science, ICCS 2015*, volume 51 of *Procedia Computer Science*, pages 1423 – 1432, 2015.

Xuan Khanh Do, Stéphane Louise, and Albert Cohen. Comparing the streamit and ΣC languages for manycore processors. In *Proceedings of the 2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, DFM '14, pages 17–25, Washington, DC, USA, 2014. IEEE Computer Society.

Amira Dkhil, Xuan Khanh Do, Stéphane Louise, and Christine Rochange. A hybrid scheduling algorithm based on self-timed and periodic scheduling for embedded streaming applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 711–715, March 2015.

Amira Dkhil, Xuan Khanh Do, Stéphane Louise, and Christine Rochange. Self-timed periodic scheduling for cyclo-static dataflow model. In *International Conference On Computational Science, ICCS 2014*, volume 29 of *Procedia Computer Science*, pages 1134 – 1145, 2014.

Submitted paper

Xuan Khanh Do, Stéphane Louise, and Albert Cohen. Design and performance analysis of real-time dynamic streaming applications. *ACM Trans. Architec. Code Optim. 0, 0, Article 0*, August 2016.

Bibliography

[1] Streamit cookbook. Technical report, MIT, September 2006.

[2] Streamit language specification. version 2.1. Technical report, MIT, September 2006.

[3] Andrzej Abramowski. Towards H.265 video coding standard, 2011.

[4] H.I. Ali, B. Akesson, and L.M. Pinho. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *Proceedings of PDP*, 2015.

[5] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Dore, Paul Dubrulle, Benoît Dupont de Dinechin, François Galea, Thierry Goubier, Michel Harrand, Samuel Jones, Jean-Denis Lesage, Stéphane Louise, and Renaud Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In *ICCS*, volume 18, pages 1624–1633, 2013.

[6] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat. *Synchronization and Linearity*. John Wiley & Sons Ltd, 1992.

[7] Mohamed A. Bamakhrama and Todor Stefanov. Managing latency in embedded streaming applications under hard-real-time scheduling. In *CODES+ISSS*, pages 83–92, 2012.

[8] Mohamed A. Bamakhrama and Todor Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 2012.

[9] Sébastien Bardot, Camille Barnier, Pascal Bouda, Xuan Khanh Do, Sylvain Hochede, Eric Li, Trong Thuc Nguyen, Ismael Paqueriaud, Adrien Polidano, and Revyll-Jones Ratanga. Red dragons : The origins by l'ensmacansat. In *Planete Science*, 2013.

[10] 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.

[11] S. Baruah. The non-cyclic recurring real-time task model. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 173–182, Nov 2010.

[12] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[13] Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.

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

[15] Vagelis Bebelis, Pascal Fradet, and Alain Girault. A framework to schedule parametric dataflow applications on many-core platforms. In *LCTES*, pages 125–134, 2014.

[16] Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *EMSOFT*, pages 3:1–3:10, 2013.

[17] B. Bhattacharya and S.S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 2001.

[18] Shuvra S. Bhattacharyya, Ed F. Deprettere, and Bart D. Theelen. *Dynamic Dataflow Graphs*, pages 905–944. Springer New York, New York, NY, 2013.

[19] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *ICASSP*, volume 5, pages 3255–3258, May 1995.

[20] Bruno Bodin, Alix Munier Kordon, and Benoît Dupont de Dinechin. Periodic schedules for cyclo-static dataflow. In *ESTImedia*, pages 105–114, 2013.

[21] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[22] A. Bouakaz, J. Talpin, and J. Vitek. Affine data-flow graphs for the synthesis of hard real-time applications. In *Proceedings of ACSD*, pages 183–192, 2012.

[23] Adnan Bouakaz, Pascal Fradet, and Alain Girault. Symbolic Buffer Sizing for Throughput-Optimal Scheduling of Dataflow Graphs. In *RTAS 2016 - 22nd IEEE Real-Time Embedded Technology & Applications Symposium*, Vienne, Austria, April 2016.

[24] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén. Actor merging for dataflow process networks. *IEEE Transactions on Signal Processing*, 63(10):2496–2508, May 2015.

[25] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, USA, 2004. ACM.

[26] J. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 508–513 vol.1, Oct 1994.

[27] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, 1993. AAI9431898.

[28] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432 vol.1, April 1993.

[29] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, Nov 1986.

[30] S. Casale-Brunet, M. Wiszniewska, E. Bezati, M. Mattavelli, J. W. Janneck, and M. Canale. Turnus: An open-source design space exploration framework for dynamic stream programs. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pages 1–2, Oct 2014.

[31] NVIDIA Corp. NVIDIA CUDA: Compute unified device architecture, 2007.

[32] V. David, C. Aussaguès, S. Louise, Ph, B. Ortolo, and C. Hessler. The OASIS Based Qualified Display System. In *Lecture Notes in Computer Science, 17th International Conf. on Computer Safety, Reliability and Security Fourth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT 2004), Columbus, Ohio. September, 2004.*, September 2004.

[33] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 2011.

[34] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the Kalray MPPA-256 integrated manycore processor. *Procedia Computer Science*, 2013.

[35] J. 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.

[36] A. Dkhil, Xuan Khanh Do, S. Louise, and C. Rochange. A hybrid scheduling algorithm based on self-timed and periodic scheduling for embedded streaming applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 711–715, March 2015.

[37] Amira Dkhil, Xuan Khanh Do, Stéphane Louise, and Christine Rochange. Self-timed periodic scheduling for cyclo-static dataflow model. In *International Conference On Computational Science, ICCS 2014*, volume 29 of *Procedia Computer Science*, pages 1134 – 1145, 2014.

[38] Amira Dkhil, Stéphane Louise, and Christine Rochange. Worst-Case Communication Overhead in a Many-Core based Shared-Memory Model. In *Junior Researcher Workshop on Real-Time Computing, Nice*, 2013.

[39] Xuan Khanh Do, Amira Dkhil, and Stéphane Louise. Self-timed periodic scheduling of data-dependent tasks in embedded streaming applications. In *Algorithms and Architectures for Parallel Processing*, volume 9529 of *Lecture Notes in Computer Science*, pages 458–478. Springer International Publishing, 2015.

[40] Xuan Khanh Do, Stéphane Louise, and Albert Cohen. Transaction parameterized dataflow: A model for context-dependent streaming applications. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 960–965, March 2016.

[41] Xuan Khanh Do, Stéphane Louise, and Albert Cohen. Managing the latency of data-dependent tasks in embedded streaming applications. In *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 9–16, 2015.

[42] Paul Dubrulle, Stéphane Louise, Renaud Sirdey, and Vincent David. A low-overhead dedicated execution support for stream applications on shared-memory cmp. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 143–152, New York, NY, USA, 2012. ACM.

[43] Christian Fabre, Iuliana Bacivarov, Ananda Basu, Martino Ruggiero, David Atienza, Éric Flamand, Jean-Pierre Krimm, Julien Mottin, Lars Schor, Pratyush Kumar, Hoeseok Yang, DeveshB. Chokshi, Lothar Thiele, Saddek Bensalem, Marius Bozga, Luca Benini, MohamedM. Sabry, Yusuf Leblebici, Giovanni De Micheli, and Diego Melpignano. Pro3d, programming for future 3d manycore architectures: Project's interim status. In *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 277–293. Springer Berlin Heidelberg, 2013.

[44] S. Feiner, Blair MacIntyre, T. Hollerer, and A. Webster. A touring machine: prototyping 3d mobile augmented reality systems for exploring the urban environment. In *Wearable Computers, 1997. Digest of Papers., First International Symposium on*, pages 74–81, Oct 1997.

[45] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149 – 1172, 2007.

[46] P. Fradet, A. Girault, and P. Poplavko. SPDF: A schedulable parametric data-flow MoC. In *DATE*, pages 769–774, March 2012.

[47] S. Gaubert. Performance evaluation of (max,+) automata. *IEEE Transactions on Automatic Control*, 40(12):2014–2025, Dec 1995.

[48] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *CODES+ISSS*, pages 125–134, Oct 2010.

[49] Marc Geilen and Twan Basten. Requirements on the execution of kahn process networks. In *Proceedings of the 12th European Conference on Programming*, ESOP'03, pages 319–334, Berlin, Heidelberg, 2003. Springer-Verlag.

[50] Marc Geilen and Twan Basten. Reactive process networks. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 137–146, New York, NY, USA, 2004. ACM.

[51] A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. Latency minimization for synchronous data flow graphs. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, DSD '07, pages 189–196, Washington, DC, USA, 2007. IEEE Computer Society.

[52] T. Goubier, R. Sirdey, S. Louise, and V. David. ΣC: A programming model and language for embedded manycores. In *Proceedings of ICA3PP*, pages 385–394, 2011.

[53] T. Goubier, R. Sirdey, S. Louise, and V. David. ΣC: A programming model and language for embedded manycores. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP'11, pages 385–394, 2011.

[54] Khronos OpenCl Working Group. The OpenCL specification, 2008.

[55] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[56] 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 Trans. Des. Autom. Electron. Syst.*, 12(3):24:1–24:25, May 2008.

[57] Soonhoi Ha and Hyunok Oh. *Decidable Dataflow Models for Signal Processing: Synchronous Dataflow and Its Extensions*, pages 1083–1109. Springer New York, New York, NY, 2013.

[58] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[59] Marnix Heskamp, Roel Schiphorst, and Kees Slump. Public safety and cognitive radio. In Alexander M. Wyglinsk, Maziar Nekovee, and Y. Thomas Hou, editors, *Cognitive radio communications and networks*, pages 467–488. Elsevier, November 2009.

[60] 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, New York, NY, USA, 2005. ACM.

[61] ARINC Incorporated. *653P1-3 Avionics Application Software Standard Interface, Part 1, Required Services*. 2013.

[62] Gary W. Johnson. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*. McGraw-Hill School Education Group, 1997.

[63] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[64] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[65] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[66] Hari Kalva and Jae-Beom Lee. The vc-1 video coding standard. *IEEE Multimedia*, 14(4):88–91, 2007.

[67] M. Khandelia, N.K. Bambha, and S.S. Bhattacharyya. Contention-conscious transaction ordering in multiprocessor dsp systems. *IEEE Transactions on Signal Processing*, 2006.

[68] E. C. Klikpo, J. Khatib, and A. Munier-Kordon. Modeling multi-periodic simulink systems by synchronous dataflow graphs. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–10, April 2016.

[69] Peter Koek, Stefan J. Geuns, Joost P.H.M. Hausmans, Henk Corporaal, and Marco J.G. Bekooij. Csdfa: A model for exploiting the trade-off between data and pipeline parallelism. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, pages 30–39, New York, NY, USA, 2016. ACM.

[70] P. S. Kurtin, J. P. H. M. Hausmans, and M. J. G. Bekooij. Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[71] Alex Kushleyev, Daniel Mellinger, Caitlin Powers, and Vijay Kumar. Towards a swarm of agile micro quadrotors. *Autonomous Robots*, 35(4):287–300, 2013.

[72] Mihai T. Lazarescu and Luciano Lavagno. Interactive trace-based analysis toolset for manual parallelization of C programs. *ACM Trans. Embed. Comput. Syst.*, 14(1):13:1–13:20, January 2015.

[73] E. A. Lee. Consistency in dataflow graphs. *IEEE Trans. Parallel Distrib. Syst.*, 1991.

[74] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE, vol. 75, no. 9,*, pages 1235–1245, 1987.

[75] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

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

[77] Edward A. Lee. Embedded software. In *Advances in Computers*, page 2002. Academic Press, 2002.

[78] Jae-Beom Lee and Hari Kalva. *The VC-1 and H.264 Video Compression Standards for Broadband Video Services*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[79] M. Lemerre and E. Ohayon. A model of parallel deterministic real-time computation. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 273–282, 2012.

[80] Matthieu Lemerre, Vincent David, Christophe Aussaguès, and Guy Vidal-Naquet. An introduction to time-constrained automata. In *Proceedings of ICE*, 2010.

[81] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[82] J. Lotze, S.A. Fahmy, J. Noguera, and L.E. Doyle. A model-based approach to cognitive radio design. *IEEE J-SAC*, 29(2):455–468, February 2011.

[83] S. Louise. A formal evaluation of mean-time access latencies for interleaved on-chip shared banked-memory in manycores. In *Proceedings of MCSoC*, pages 19–24, 2013.

[84] S. Louise. Toward a model of computation for time-constrained applications on manycores. In *ENASE 2015 - Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 45–50, 2015.

[85] S. Louise, P. Dubrulle, and T. Goubier. A model of computation for real-time applications on embedded manycores. In *MCSoC*, Sept 2014.

[86] S. Louise, M. Lemerre, C. Aussagues, and V. David. The OASIS kernel: A framework for high dependability real-time systems. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pages 95–103, Nov 2011.

[87] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, 1995.

[88] J. Markoff. Google cars drive themselves in traffic. *The New York Times*, Oct 2010.

[89] Orlando Moreira. Temporal analysis and scheduling of hard real-time radios running on a multi-processor. PHD Thesis, Technische Universiteit Eindhoven, 2012.

[90] Orlando M Moreira and Marco JG Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.

[91] A.C. Murillo, D. Gutierrez-Gomez, A. Rituerto, L. Puig, and J.J. Guerrero. Wearable omnidirectional vision system for personal localization and guidance. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on*, pages 8–14, June 2012.

[92] T.M. Parks, J.L. Pino, and E.A. Lee. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 204–210 vol.1, Oct 1995.

[93] Dwayne Phillips. Image processing in C, Part 5: Basic edge detection. pages 47–56, 1994.

[94] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[95] Danijela Ristić-Durrant and Axel Gräser. Closed-loop control of segmented image quality for improvement of digital image processing. *Automatic Control and Robotics*, 7(1):27–34, 2004.

[96] Y. Sato, M. Nakamoto, Y. Tamaki, T. Sasama, I. Sakita, Y. Nakajima, M. Monden, and S. Tamura. Image guidance of breast cancer surgery using 3-d ultrasound images and augmented reality visualization. *Medical Imaging, IEEE Transactions on*, 17(5):681–693, Oct 1998.

[97] Stefania Sesia, Issam Toufik, and Matthew Baker. *LTE, The UMTS Long Term Evolution: From Theory to Practice*. Wiley Publishing, 2009.

[98] R. Soulé, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel. Dynamic expressivity with static optimization for streaming languages. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 159–170, New York, NY, USA, 2013. ACM.

[99] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2nd edition, 2009.

[100] M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, April 2011.

[101] M. Stigge, P. Ekberg, N. Guan, and W. Yi. On the tractability of digraph-based task models. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 162–171, July 2011.

[102] Sander Stuijk, Marc Geilen, and Twan Basten. Sdf^3: Sdf for free. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 276–278, 2006.

[103] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, Dec 2012.

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

[105] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 224–235, 2005.

[106] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of PACT*, 2010.

[107] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, 23(9):661–692, 2006.

[108] J.-J. van de Beek, M. Sandell, M. Isaksson, and P. Ola Borjesson. Low-complex frame synchronization in OFDM systems. In *ICUPC*, 1995.

[109] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511–I–518 vol.1, 2001.

[110] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *RTAS*, pages 183–194, April 2008.

[111] P. S. Wilmanns, S. J. Geuns, J. P. H. M. Hausmans, and M. J. G. Bekooij. Buffer sizing to reduce interference and increase throughput of real-time stream processing applications. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 9–18, April 2015.

[112] 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, 2009.

[113] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. *SIGPLAN Not.*, 46(3):369–380, March 2011.

[114] C. Zimmer and F. Mueller. Low contention mapping of real-time tasks onto TilePro 64 core processors. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 131–140, April 2012.