

Modeling, Scheduling, Pipelining and Configuration of Synchronous Dataflow Graphs with Throughput Constraints

Alexandre Honorat

▶ **To cite this version:**

Alexandre Honorat. Modeling, Scheduling, Pipelining and Configuration of Synchronous Dataflow Graphs with Throughput Constraints. Signal and Image processing. INSA de Rennes, 2020. English. NNT : 2020ISAR0010 . tel-03337988

HAL Id: tel-03337988
https://theses.hal.science/tel-03337988

Submitted on 8 Sep 2021

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

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

DOCTORAT BRETAGNE LOIRE / MATHSTIC

INSA RENNES

THÈSE DE DOCTORAT DE

L'INSA RENNES

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

Par

Alexandre HONORAT

Modeling, Scheduling, Pipelining and Configuration of Synchronous Dataflow Graphs with Throughput Constraints

Thèse présentée et soutenue à Rennes (35 000), le 27 novembre 2020
Unité de recherche : IETR – UMR 6164 du CNRS
Thèse N° : 20ISAR 20 / D20 - 20

Rapporteurs avant soutenance :

Claire PAGETTI	HDR, Ingénieur de Recherche à l'ONERA, Toulouse
Frédéric SUTER	HDR, Directeur de Recherche au CNRS (USR 6402 – CCIN2P3), Lyon

Composition du Jury :

Président :	Alain GIRAULT	HDR, Directeur de Recherche à l'INRIA Grenoble
Examinateurs :	Johan LILIUS	Professeur à l'Åbo Akademi, Turku (Finlande)
	Claire PAGETTI	HDR, Ingénieur de Recherche à l'ONERA, Toulouse
	Frédéric SUTER	HDR, Directeur de Recherche au CNRS (USR 6402 – CCIN2P3), Lyon
Dir. de thèse :	Jean-François NEZAN	HDR, Professeur à l'INSA Rennes
Encadr. de thèse :	Karol DESNOS	Maître de Conférences à l'INSA Rennes

Contents

Contents iii

List of Figures v

List of Tables vii

List of Algorithms ix

List of Listings xi

Acknowledgements xiii

Résumé en français xvii

Introduction xxvii

1 Background 1

1.1 Introduction: hardware, software, and parallelism . . . 2
1.2 The SDF dataflow model and its flavors . . . 13
1.3 Scheduling of SDF graphs . . . 22
1.4 The PREESM tool . . . 30
1.5 Conclusion . . . 41

2 Modeling nested `for` loops, with SDF graphs 43

2.1 SIFT keypoints detection application . . . 45
2.2 Modeling of single loops having explicit parallelism . . . 47
2.3 Modeling of nested loops having explicit parallelism . . . 49
2.4 When and how to use SDF iterators? . . . 52
2.5 Evaluation . . . 55
2.6 Related work . . . 57
2.7 Conclusion . . . 59

3 Scheduling partially periodic SDF graphs 61
3.1 Background . . . 64
3.2 Partially periodic constraints . . . 66
3.3 Discussion on the schedulability conditions . . . 75
3.4 Scheduling SDF graphs with partially periodic constraints . . . 77
3.5 Evaluation . . . 84
3.6 Related work . . . 91
3.7 Conclusion . . . 93

4 Pipelining SDF graphs automatically 95
4.1 Admissible graph cuts for pipelining . . . 97
4.2 Automatic pipelining of SDF graphs . . . 103
4.3 Automatic cycle breaking of SDF graphs . . . 108
4.4 Evaluation . . . 111
4.5 Related work . . . 117
4.6 Conclusion . . . 119

5 Configuration of parameterized SDF graphs 121
5.1 PREESM parameters . . . 123
5.2 DSE: entangled problems . . . 128
5.3 An exhaustive DSE method . . . 132
5.4 Improving DSE with automatic delay placement . . . 134
5.5 A naive heuristic for Integer malleable parameters . . . 137
5.6 Evaluation . . . 139
5.7 Related work . . . 144
5.8 Conclusion . . . 147

Conclusion 149

Bibliography 155

List of published contributions 183

Acronyms 185

Edition notice 189
Autorisation de Reproduction . . . 190
Attestation de Corrections . . . 191

List of Figures

1 Exemple de graphe SDF avec son équivalent SRSDF, et deux ordonnancements possibles. . . . xxii
2 Logigramme des contributions telles qu'utilisées dans l'outil PREESM. . . . xxiv

3 Flowchart of the contributions as used in the PREESM framework. . . . xxxi

1.1 Odroid XU3 heterogeneous multi-processor. . . . 3
1.2 SDF graph example with its equivalent SRSDF graph and two possible static schedules with the corresponding buffer usage. . . . 16
1.3 SRSDF graph equivalent representations. . . . 17
1.4 SDF graph example with its equivalent SRSDF graph and a possible static schedule with the corresponding buffer usage. . . . 18
1.5 CSDF graph example with its equivalent SRSDF graph and a possible schedule. . . . 20
1.6 Example of PISDF graph in PREESM. . . . 34
1.7 Example of generated SRSDF graph in PREESM. . . . 40

2.1 SIFT image processing application. . . . 46
2.2 Map and Upscale at a coarse-grain level. . . . 48
2.3 Modeling of iterators with SDF. . . . 52
2.4 Reduce of N elements in SDF, with chunk size $c > 1$. . . . 53
2.5 Modeling of iterators with SDF, with broadcast actor. . . . 55

3.1 SDF graph scheduling examples. . . . 65
3.2 Example of scheduler and graph iterations, with pipelining. . . . 66
3.3 Example and counter-example of Assumption 1. . . . 68
3.4 Periodic actor Π generating an underflow. . . . 70
3.5 Valid schedule example of graph 3.4a. . . . 70
3.6 Counter-example to generalization of Equation (3.6). . . . 72
3.7 Floor function underestimation example, as used in Equation (3.7). . . . 73

3.8 Sample graph for topology ranks example. . . . 76
3.9 A false positive to Algorithm 3.1. . . . 77
3.10 SDF graph G and its corresponding SRSDF version G^*. . . . 78
3.11 Schedule with an actor period smaller than its WCET. . . . 84
3.12 Scheduling bounds on the number of PEs m. . . . 86
3.13 Evaluation of the schedulability gap on small random graphs. . . . 89
3.14 Evaluation of the schedulability gap on large random graphs. . . . 90
3.15 Graph period refinement example. . . . 91
3.16 Evaluation of the graph period computed by Algorithm 3.3 compared to ADFG with global EDF policy. . . . 91

4.1 Topological ordering and schedule example without and with pipeline. . 99
4.2 Delay placement examples, resulting from invalid and admissible graph cuts. . . . 100
4.3 Graph with valid delay placement distributed on the paths. . . . 101
4.4 Split-join graph with four parallel branches. . . . 101
4.5 Schedule example where pipelining does not compensate for the presence of a global barrier. . . . 103
4.6 Graph cut example and related schedule for ALAP topological ordering. 105
4.7 Graph cut example and related schedule for ASAP topological ordering. 105
4.8 Graph cut examples for regular and modified ALAP topological ordering. 105
4.9 Preselected and final cuts computed by the delay placement heuristic on a sample graph. . . . 108
4.10 SDF actor having a self-loop. . . . 109
4.11 Cycle example with an entry actor, an exit one and a normal one. . . . 110

5.1 Example of parameters and their dependencies, here to express image resolution choices. . . . 124
5.2 Example of a parameterized PISDF graph path. . . . 127
5.3 Example to estimate number of cuts in DSE. . . . 137
5.4 Dichotomy bounds example of an Integer malleable parameter. . . . 139
5.5 Duration of II as function of the SIFT application configuration, for different degrees of parallelism p. . . . 142

List of Tables

1.1 Main notations used in this thesis. 42

2.1 Number of scheduled tasks, execution times in ms, and speedup for different number of cores. 57

3.1 Details of the random directed acyclic SDF graphs generated by Turbine, and execution time of the algorithms. 88

4.1 Characteristics and throughput gain with delays (H) of SDF benchmark applications, on four PEs. 114
4.2 Throughput gain with delays (H) of SDF benchmark applications, on four PEs. 115
4.3 Throughput gain with delays (H) of SDF benchmark applications, on sixty-four PEs. 115
4.4 Throughput and memory increases with delays (H), on four PEs, for different parallelism parameters (p). 117
4.5 Throughput increases with delays (H), on four PEs, for different parallelism parameters (p). 118

5.1 Results of DSE on SIFT video. 144

List of Algorithms

1.1 Modified DFS before computing the repetition vector $\vec{r}$ 37
1.2 Computation of the repetition vector $\vec{r}$ based on the result of Algorithm 1.1 38

3.1 Modified BFS to compute $nblf_{\pi}^{\uparrow}$ and related necessary conditions 74
3.2 Subroutines for partially periodic scheduling of tasks 80
3.3 Partially periodic scheduling of tasks 81

4.1 Selection of buffer breaking cycles 111

List of Listings

1.1 PISDF parameter definitions and use in Figure 1.6. 35
2.1 Simple one dimensional (1-D) `for` loop having explicit parallelism. 47
2.2 Simple 1-D upscale, by interpolation on the element and its successor. . . 48
2.3 Spit SDF actor code. 49
2.4 Upscale SDF actor code. 49
2.5 Three perfectly nested `for` loops having explicit parallelism. 50
2.6 Iteration space simulator for three perfectly nested `for` loops having explicit parallelism. 51
2.7 Non affine 2-D `for` loop having explicit parallelism. 54
2.8 Non affine 2-D `for` loop SDF actor code. 54
2.9 Original non affine 4-D `for` loop in SIFT. 56
5.1 Parameter expression implementing a dictionary. 125
5.2 Parameter expression of `nLocalKptmax` in the SIFT application. 126
5.3 Sample objective input for the DSE algorithm. 133
5.4 Code of the power threshold comparator of DSE points. 134
5.5 Code of the global comparator of DSE points. 135
5.6 Objective input for the DSE evaluation. 140

Acknowledgements

This thesis has received funding from the European Union's (EU) Horizon 2020 research and innovation programme under grant agreement №732105 (project CERBERO) and from the Région Bretagne (France) under grant ARED 2017 ADAMS. First of all, I would like to thank all persons involved in this project. More information about this EU's research program can be found on their website: http://www.cerbero-h2020.eu/.

I would like to thank all the jury members to have reviewed and accepted my thesis: Alain Girault, Johan Lilius, Frédéric Suter and Claire Pagetti. I understood it was quite difficult to read, but thanks to your reviews, I hope that this final version is now readable by anyone interested in Synchronous DataFlow (SDF) graphs. However, I do not guarantee the usefulness of this thesis for the reader!

This thesis has been supervised by Karol Desnos and Jean-François Nezan, and I thank them for all their valuable advice and for their patience. I took a few detours, but finally, we have worked on malleable parameters! I have really liked to work on the topic of this thesis, and I am lucky that you hired me. Actually, I was not the only one that you hired to work on this, I also have to thank Florian Arrestier and Antoine Morvan; the majority of my work relies on their huge SDF and coding skills. Besides, after three years in the same office, Florian has almost convinced me that artificial intelligence and neural networks are useful; I acknowledge my own defeat on this point! During the thesis, I have also been pleased to work with Maxime Pelcat, Mickaël Dardaillon and Shuvra S. Bhattacharyya on different publications. In another collaboration, we have worked with Michael Masin and his team; I thank you and your family for your warm welcome in Haifa!

For sure, I also want to thank all my other colleagues of INSA Rennes and especially the VAADER team. Despite the fact that we are working in a creepy building above a stock of deadly toxic gaz, you manage to make this place enjoyable. And now I understand why everyone likes to work in such a big team, because it means that we have more conference trips, birthdays, and even newborns (!), to celebrate with pastries! I cannot mention all of you here (yes, it is a big team), so I will mention one person who incarnates the most the spirit of the VAADER team, the peaceful and funny guy

who never misses a joke nor a crossword: Pierre-Loup Cabarat! And special thanks to Thomas Amestoy, the only person (except my supervisors) who physically attended my defense. Moreover I thank all the team visitors, especially Claudio Rubattu (the beaver pacifier) and Leonardo Suriano (the beaver imitator)!

Before working at INSA Rennes, I also worked during two years approximately 500 meters from there, at INRIA Rennes. When I arrived in Rennes, I did not know anyone so I am deeply grateful to the welcoming TEA, PANAMA and RAINBOW teams and their visitors. Thanks to you I discovered the spirit of life of Brittany, mainly consisting in salty butter, *galettes*, beers, music, and friends to share all of this with. I was thinking to stay only two years at the beginning ... you clearly changed my mind! In the end, I am not even able to leave Rennes a single week-end by fear of missing the amazing Lices' market (one of the biggest in France, even having one "more organic than organic" producer). Again, it is not possible to mention all of you here, but I will start with Simon Lunel, the most Breton of all persons I met in Brittany! Be careful Simon, as Hai-Nam Tran lives in Brest now, he might dethrones you one day... Moreover, I want to mention two friends of the PANAMA team who have defended their PhD one week before and one week after me, and who have supported, if not worsen, my weird taste in movies (Braguino!): Antoine Chatalic (master of *crêpes* baking) and Diego Di Carlo (master of tiramisu baking, or is Giorgia Cantisani the true master of this recipe??). Regarding my weird taste in music, it has been even more worsen by Corentin Louboutin and Valentin Gillot. However, not only food and music are important, so thank you Cássio Fraga Dantas and Jean-Joseph Marty for all the board games you made me discover! And of course, special thanks to my *belote* and billiards partners: Clément Gaultier, Nicolas Bellot and Lesley-Ann Duflot! During the thesis, I also "played" LaTeX a lot, a very funny sort of escape game, where you have to find your way out of the numerous bugs that you face while solving some challenges; thank you Katharina Boudgoust for the ultimate LaTeX challenge of putting the chapter title in the right margin of this thesis.

This thesis is actually the result of long studies, and by extension the result of all the teachers and supervisors I have had (at ENSEIRB-MATMECA, Pothier, Bertran de Born, ...). I would like to thank them all here, from music teachers (especially French horn and piano) to mathematics teachers. I will only mention Olivier Aumage, Denis Barthou and Jean Roman who have particularly motivated me to finish my engineering diploma, and to continue in the academic field. I do not regret it.

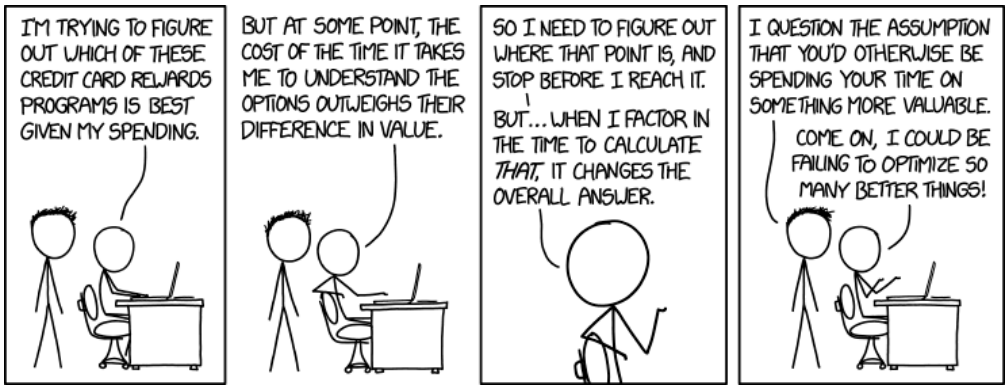
Before Rennes, I have studied in Bordeaux, Orléans and Périgueux. It turns out that almost one third of my friends from there did a PhD. Among them I am currently the last one to have started and defended a PhD, with two or three years of delay, and I

have to say: well, you trapped me! But I have no regrets, the friends I have met during the PhD, and the food I have eaten during trips for the PhD (in Sardegna, Haifa and Samos!) clearly compensate for these three years of severe stress. Yes, food is almost as important as friends, especially because it is the first thing you want to share with them! For brevity, I will mention only four of these friends. First, I thank Nicolas Lafon, whose wedding date forced me to send my PhD manuscript on time! Then, I congratulate Thomas Quezel, who managed to avoid my (yet remote) PhD defense thanks to the birth of his first child on the same day! I also want to thank Anastasia Gkolfi, who has inspired me with her sheerest determination to start a PhD, whatever it takes, even to live in the Nordic cold (for her)! Last but not least, Raphaël Angel is certainly the friend who has unconsciously driven me to this PhD. I am deeply grateful to him, especially for having initiated me to my first programming language (Ruby).

Finally, I thank my family, especially my parents and my sister, who know better than anyone at what point this three years were stressful. Thank you for your support, during the thesis, and of course, during all my life. I remember that, as a child, you subscribed me to the *Research EU* magazine which describes the research projects funded by the EU; well, more than fifteen years later, I have been funded by the EU too, it might have had a greater influence than what you think.

To conclude these acknowledgements, I would like to thank *XKCD* to have perfectly described the entanglement problem of the configuration challenge studied in this PhD:

Credit Card Rewards / XKCD



To put you in a good mood before reading this thesis, I suggest you to listen to nice music first, as "La frite équatoriale" composed by François de Roubaix. Playing a good old "Carcassonne" board game might also help, but only if you win at the end!

À Georges, Brigitte et Élisabeth.

Résumé en français

Au fur et à mesure que les processeurs sont devenus de plus en plus petits et de moins en moins coûteux, ils ont été utilisés dans de plus en plus d'appareils facilitant la vie de tous les jours. L'un des exemples majeurs est le *smartphone*, qui est presque aussi puissant qu'un ordinateur personnel. Les smartphones contiennent même plus de composants qu'un ordinateur : au moins une caméra et une antenne. En 2019, 77% des Français possédaient un smartphone, et 76% avaient accès à un ordinateur personnel[1].

Chaque composant d'un smartphone peut en réalité contenir son propre processeur. Par exemple, la dernière *smartcamera* Sony MX 500 intègre sur une même puce un processeur situé en dessous du capteur d'image, pour une largeur n'excédant pas $1,5$ centimètre. Les Systèmes Multi-Processeurs Intégrés sur Puce (MPSoC)[2] sont l'un des principaux ingrédients de ce succès de miniaturisation. Comme les MPSoCs contiennent plusieurs processeurs sur une même puce, ils abritent efficacement une forte capacité de calcul. Cette capacité de calcul peut servir par exemple à stabiliser une vidéo en temps-réel, en utilisant uniquement des algorithmes plutôt que de lourds stabilisateurs mécaniques.

Un smartphone ne contient pas un, mais plusieurs processeurs hétérogènes : des processeurs génériques pour les applications standards, et des processeurs spécifiques à certains types de calcul, tels que pour le traitement du signal. Les calculs se composent généralement de plusieurs tâches à exécuter, formant un graphe. Pour accélérer les calculs, les tâches sont distribuées aux différents processeurs selon leur capacité à exécuter tel type de calcul. La sélection de l'emplacement et de l'ordre des tâches à exécuter sur les processeurs s'appelle l'*ordonnancement*. Le problème d'ordonnancement est complexe à résoudre, et est aussi lié aux moyens de communication reliant les composants.

La création de panorama à partir d'images prises depuis différents points de vue est un autre exemple d'application pour les smartphones et smartcameras. Plusieurs algorithmes peuvent être utilisés pour ces applications logicielles, chacun offrant un dif-

[1] Ces statistiques sont extraites du rapport officiel Baromètre du numérique 2019 .

[2] Les acronymes sont gardés en langue anglaise afin de pouvoir les repérer dans le reste du document, rédigé en anglais. Seules leurs formes étendues sont traduites.

férent niveau de qualité et de rapidité. En sus, l'utilisateur spécifie généralement une Qualité de Service (QoS) minimale définissant des exigences logicielles telle qu'une résolution d'image ou une cadence de prises de vue minimales. Cependant, les applications logicielles de stabilisation d'image ou de création de panorama doivent aussi respecter des contraintes matérielles : cadence et résolution maximales du capteur d'image, puissance maximale de la batterie, ou fréquence maximale du MPSoC. Satisfaire les exigences logicielles en même temps que respecter les contraintes matérielles est également un problème complexe. Ce problème de *configuration* est en général résolu en testant différentes configurations de l'application et du matériel jusqu'à ce que l'une des configurations satisfasse toutes les exigences et contraintes. Le test des configurations possibles correspond à l'Exploration de l'Espace des Designs (DSE). Les configurations diffèrent les unes des autres par la fréquence du MPSoC ou par la résolution d'image supportée par l'application. Si aucune configuration ne satisfait toutes les exigences et contraintes, un compromis est nécessaire et certaines d'entre elles doivent être assouplies.

Au bout du compte, l'adaptation d'une application logicielle à une architecture matérielle telle qu'un smartphone n'est pas triviale. Cette adaptation nécessite de résoudre plusieurs problèmes à la fois, au moins l'ordonnancement et la configuration. Pour ce faire, l'application ainsi que l'architecture sont toutes les deux *modélisées*. Le but du modèle est d'abstraire les propriétés essentielles qui participent aux problèmes d'ordonnancement et de configuration.

Ces problèmes de modélisation, d'ordonnancement et de configuration existent depuis quelques dizaines d'années et sont des sujets actuels de recherche. Par exemple, le Modèle de Calcul (MoC) Synchrone de Flux de Données (SDF) [LM87b] est dédié aux applications de traitement du signal, pour les images ou les antennes. Dans le modèle SDF, les tâches à exécuter et leurs données sont toutes deux représentées. Les tâches et les échanges de données sont prédéterminés par le concepteur, ce qui permet une exécution déterministe de l'application. Le modèle SDF peut être paramétré avec le Méta-Modèle Interfacé et Paramétré (PiMM) [Des+13] par exemple, qui introduit le choix entre plusieurs valeurs de paramètres et le rend donc éligible au problème de configuration. SDF appartient à la grande famille des MoCs orientés flux de données, qui se concentrent sur la représentation des échanges et des traitements de données.

De nombreux outils existent pour automatiquement générer et optimiser le code informatique d'une application logicielle sur une architecture matérielle donnée, tel que SynDEx [GLS99] qui repose sur la méthodologie *Adéquation Architecture Algorithme*. SynDEx optimise l'exécution d'une application de flux de données sur une architecture MPSoC ou autre, en déterminant l'ordonnancement offrant la meilleur cadence par

exemple. Cependant, il ne *configure* pas lui-même l'application pour satisfaire des exigences et contraintes. Autrement dit, SynDEx ne va pas automatiquement sélectionner la meilleure résolution d'image pour des contraintes matérielles et des exigences logicielles données. D'autres modèles et outils, comme AADL[3] [Hug+08] et Ptolemy[4] [Guo+14], permettent au concepteur de représenter plus d'informations : l'application, l'architecture, et respectivement les exigences et les contraintes associées. Cette approche aide grandement la vérification de la satisfaction des exigences et contraintes ; mais de même qu'avec SynDEx, le concepteur a toujours besoin de tester lui-même plusieurs configurations avant de choisir la meilleure. L'utilisation d'algorithmes de DSE accélère ce processus, mais ne le résout pas complètement lorsqu'il faut configurer les exigences de QoS. En effet ces outils n'ont pas connaissance des compromis acceptables pour le concepteur.

Finalement, la sélection de la meilleure configuration satisfaisant contraintes matérielles et exigences logicielles sur une architecture matérielle hétérogène est toujours un défi à l'heure actuelle. Ce problème de configuration est composé de plusieurs autres problèmes, dont certains sont introduits dans la section suivante. Les contributions proposées dans cette thèse, concernent trois de ces problèmes en sus de la configuration elle-même. Les sections restantes présentent brièvement le modèle SDF, puis les contributions exposées dans cette thèse, et leurs possibles extensions.

Définition du problème

Cette thèse aborde le problème de configuration d'applications SDF paramétrées ayant des exigences logicielles et étant exécutées sur des architectures de multi-processeurs ayant des contraintes matérielles. Nos contributions ne concernent malheureusement que le cas des multi-processeurs homogènes. Le problème de configuration se pose également pour les architectures hétérogènes et est alors d'autant plus difficile à résoudre.

Exécuter un graphe SDF nécessite de résoudre plusieurs problèmes : l'ordonnancement de l'exécution des tâches et celui des échanges de données, l'allocation de la mémoire pour l'exécution des tâches et les échanges de données, tout en prenant en compte les contraintes telles que le nombre de processeurs dans l'architecture et la périodicité de certains tâches. Chacun de ces problèmes est complexe, NP-complet en général. Alors que de nombreux algorithmes existent déjà pour résoudre chacun de ces problèmes, nous nous concentrons sur ceux qui sont rapides et peuvent traiter de grands nombres de

[3] C.f. AADL ↗ et OpenAADL ↗ websites.
[4] C.f. Ptolemy ↗ website.

tâches à la fois, de sorte que toutes les solutions possibles peuvent être testées dans un court laps de temps, par exemple moins d'une heure.

La rapidité est particulièrement importante pour les graphes SDF paramétrés. Les paramètres sont toujours fixés lors de l'exécution des tâches du graphe les utilisant, mais ils permettent d'explorer automatiquement toutes les configurations possibles, et de réagir dynamiquement à certaines données en entrée qui les modifieraient. En effet, pour l'exploration des configurations, le nombre de configurations peut exploser de manière exponentielle en fonction du nombre de paramètres. Pour l'exécution dynamique, la rapidité est aussi nécessaire pour limiter la surcharge de calculs créée par la résolution dynamique des problèmes d'ordonnancement et d'allocation. Dans cette thèse, nous utilisons le modèle PISDF, qui est l'extension du modèle SDF obtenue en y appliquant le méta-modèle PIMM. Les propriétés dynamiques de PISDF ne sont pas utilisées, et notre travail se restreint à sa sémantique statique afin de trouver à l'avance une bonne configuration de l'application satisfaisant au mieux l'ensemble des contraintes. Pour information, les paramètres matériels tels que la fréquence du processeur, peuvent aussi être représentés par des paramètres de graphes PISDF.

Dans cette thèse, le mot *contrainte* peut autant signifier contrainte matérielle qu'exigence logicielle. Bien que nous ayons distingué les deux notions jusqu'à présent, toutes deux peuvent être liées à la même métrique. Par exemple, la cadence d'images est contrainte par une valeur minimale du côté logiciel, et par une valeur maximale du côté matériel. Les contraintes que nous considérons peuvent être : la périodicité de certaines tâches, la minimisation de l'énergie ou de la puissance à chaque exécution du graphe, la minimisation de la latence ou la maximisation de la cadence des exécutions, la minimisation ou la maximisation de n'importe lequel des paramètres apparaissant dans le graphe PISDF. La sélection de la meilleure configuration s'appuie sur une DSE pour évaluer les plus intéressantes d'entre elles. Ce problème principal de configuration peut être divisé en plusieurs problèmes, dont :

- paramétrisation d'une application statique dans le modèle PISDF ;
- modélisation de tâches périodiques dans le modèle PISDF ;
- ordonnancement rapide de graphes PISDF ayant des tâches périodiques ou non ;
- modélisation des contraintes de cadence, latence, énergie, puissance ou QoS.

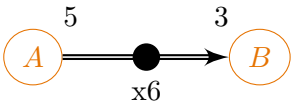
Cette liste est largement incomplète. Nous détaillons maintenant le modèle (PI)SDF, qui est utilisé par toutes les contributions.

Modèle de calcul synchrone de flux de données SDF

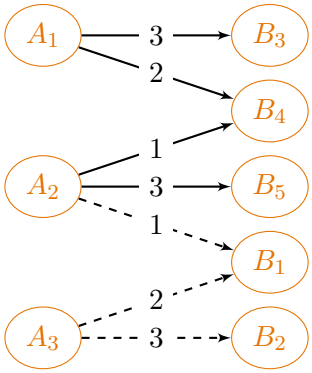
Cette thèse utilise le modèle de calcul PISDF, une extension paramétrée du modèle SDF [LM87b]. La Figure 1a présente un exemple de graphe SDF. Les tâches à exécuter, A et B dans la figure, y sont représentées par les nœuds du graphe. Les échanges de données, de A vers B dans la figure, y sont représentés par les arcs dirigés du graphe. Ces arcs possèdent deux propriétés : un nombre entier précisant la quantité de données produites lors de l'exécution de la tâche à l'extrémité de départ, et symétriquement un nombre entier précisant la quantité de données consommées à l'extrémité d'arrivée. Les arcs correspondent à des files d'attente stockant les données d'une tâche vers l'autre. Chaque tâche peut être le départ ou l'arrivée de plusieurs arcs à la fois. Les arcs peuvent aussi contenir des données présentes avant l'exécution de chacune des extrémités ; ces données sont alors appelées *délais*.

Un des avantages du modèle SDF est son déterminisme. Un effet, grâce aux deux propriétés de consommation et de production de chaque arc, il est possible de déterminer le nombre d'exécution de chaque tâche de sorte que la taille des files d'attente reste bornée au cours du temps. Ce nombre d'exécutions de chaque tâche s'appelle le *vecteur de répétition* du graphe. Un tel vecteur n'existe pas toujours, et dans cette thèse nous nous intéressons uniquement au cas des graphes SDF *cohérents*, c'est-à-dire lorsque le vecteur de répétition existe. En somme le vecteur de répétition garantit qu'en exécutant autant de fois toutes les tâches, les files d'attente contiendront toutes autant de données qu'initialement. Dans la Figure 1a, la tâche A est ainsi exécutée trois fois et produit 15 données sur la file d'attente. À l'autre extrémité de l'arc, la tâche B est exécutée cinq fois et consomme donc en tout 15 données également.

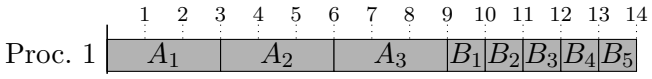
Le problème d'ordonnancement correspond au choix de l'ordre d'exécution des tâches. Cet ordre doit garantir que chaque tâche a suffisamment de données disponibles dans les files d'attente y arrivant, afin de pouvoir consommer autant de données que spécifié au début de son exécution. Quant à la production des données, elle n'est effective qu'à la fin de l'exécution. Pour garantir cet ordre, le graphe SDF est souvent déroulé en un graphe Single Rate SDF (SRSDF) afin d'expliciter les dépendances de données entre les tâches. Un exemple est donné dans la Figure 1b. Les arcs y sont représentés par de simples traits car la quantité de donnée envoyée et reçue sur ces arcs y est fixée et égale ; elle est spécifiée au milieu de chaque arc. Par ailleurs, le graphe SDF original contenait un délai de 6 données, qui supprime donc les premières dépendances de données représentées en pointillé. En effet, grâce au délai, les deux premières exécutions de la tâche B peuvent démarrer avant même que A n'ait commencé à être exécutée. Les 6 données du délai se



(a) Exemple de graphe SDF, ayant pour vecteur de répétition $[3, 5]^T$. Un délai est présent sur l'unique canal de transmission de données, ce qui supprime la dépendance de données depuis les tâches A_1 et A_2 vers B_1 et B_2.



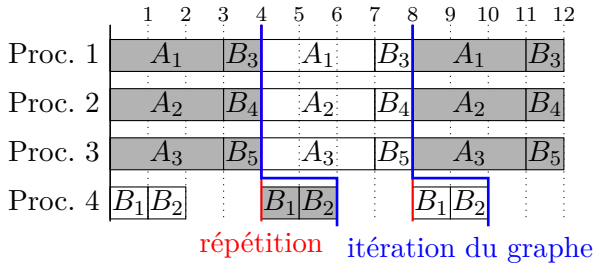
(b) Graphe SRSDF de 1a. Les dépendances de données cassées par le délai sont représentées par une ligne en pointillé.



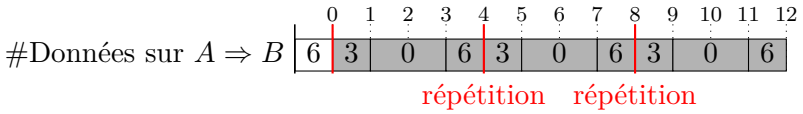
(c) Exemple d'ordonnancement du graphe 1a sur un processeur.



(d) Remplissage de la liste d'attente lors de l'ordonnancement sur1c. Les nombres sont la quantité de données présente à chaque unité de temps sur la liste d'attente entre les acteurs A et B.



(e) Exemple d'ordonnancement du graphe 1a sur quatre processeurs. Trois répétitions de l'ordonnancement statique sont représentées, séparées par une ligne verticale rouge. Une exécution du graphe de bout en bout est délimitée par les lignes bleues.



(f) Remplissage de la liste d'attente lors de l'ordonnancement sur1e. Les nombres sont la quantité de données présente à chaque unité de temps sur la liste d'attente entre les acteurs A et B.

FIGURE 1 – Exemple de graphe SDF avec son équivalent SRSDF, et deux ordonnancements possibles avec le remplissage de liste d'attente correspondant.

retrouvent initialement dans la file d'attente dont l'évolution est décrite en Figure 1d.

Grâce au graphe de dépendances de données il est possible de réaliser l'ordonnancement, dont un exemple est représenté sur la Figure 1c pour un processeur. L'ordonnancement correspond au placement et à l'ordre d'exécution d'autant de tâches que spécifiées dans le vecteur de répétition. Cet ordonnancement est ensuite répété indéfiniment (une fois dans la figure). L'écoulement du temps est représenté par l'indexation au-dessus de l'ordonnancement. La durée d'exécution de chaque tâche est fixée a priori et ne varie pas en fonction du temps. L'ordonnancement de la Figure 1c dure 14 unités de temps et la cadence du graphe est donc 1/14. L'ajout de délais permet parfois de réduire ce temps d'exécution lorsque plusieurs processeurs sont disponibles. L'ordonnancement représenté en Figure 1e pour quatre processeurs ne dure que 4 unités de temps. Dans ce cas, comme des dépendances de données sont supprimées grâce au délai du graphe initial, on parle de *pipelinage*. En effet, certaines exécutions de B chevauchent dans le temps les exécutions de A correspondant à une précédente exécution du graphe de bout en bout, ce qui ne serait pas possible sans délai. Une exécution du graphe de bout en bout, aussi appelée itération du graphe, est représentée en Figure 1e sans couleur de fond et délimitée par les lignes bleues.

Finalement, la paramétrisation des graphes SDF revient simplement à remplacer les nombres entiers utilisés pour les tailles de délais, les propriétés de consommation et de production, et pour les durées d'exécution des tâches, par des expressions arithmétiques utilisant des variables mathématiques. Ces expressions sont évaluées avant la transformation du graphe SDF vers le graphe de dépendance de données SRSDF correspondant.

Contributions

Dans cette thèse, nous présentons quatre contributions, dont la principale est la configuration automatique de graphes PISDF grâce à une DSE. Les trois autres contributions aident toutes à réaliser cette DSE, par le biais de la modélisation de contraintes ainsi que de l'ordonnancement respectant ces contraintes. Les contributions sont brièvement résumées dans les paragraphes suivants ; chaque contribution correspond à un chapitre de la présente thèse. L'application de traitement d'image SIFT [Low04] est notre principal cas d'utilisation et sert à réaliser les évaluations expérimentales des contributions présentées dans les chapitres 2, 4 et 5.

La Figure 2 représente schématiquement comment nos contributions sont intégrées au l'outil PREESM [Pel+14] qui permet de développer des applications dans le modèle PISDF. Comme SynDEx, PREESM sépare la modélisation de l'architecture de celle de

l'application. La phase *Scénario* dans la Figure 2 permet de définir les informations spécifiques à la fois à une application et à la fois à une architecture, telles que les durées d'exécution de chaque tâche. Nos contributions correspondent aux phases vertes* : modélisation de l'application, pipelinage, ordonnancement et DSE. La configuration est réalisée lors de la DSE.

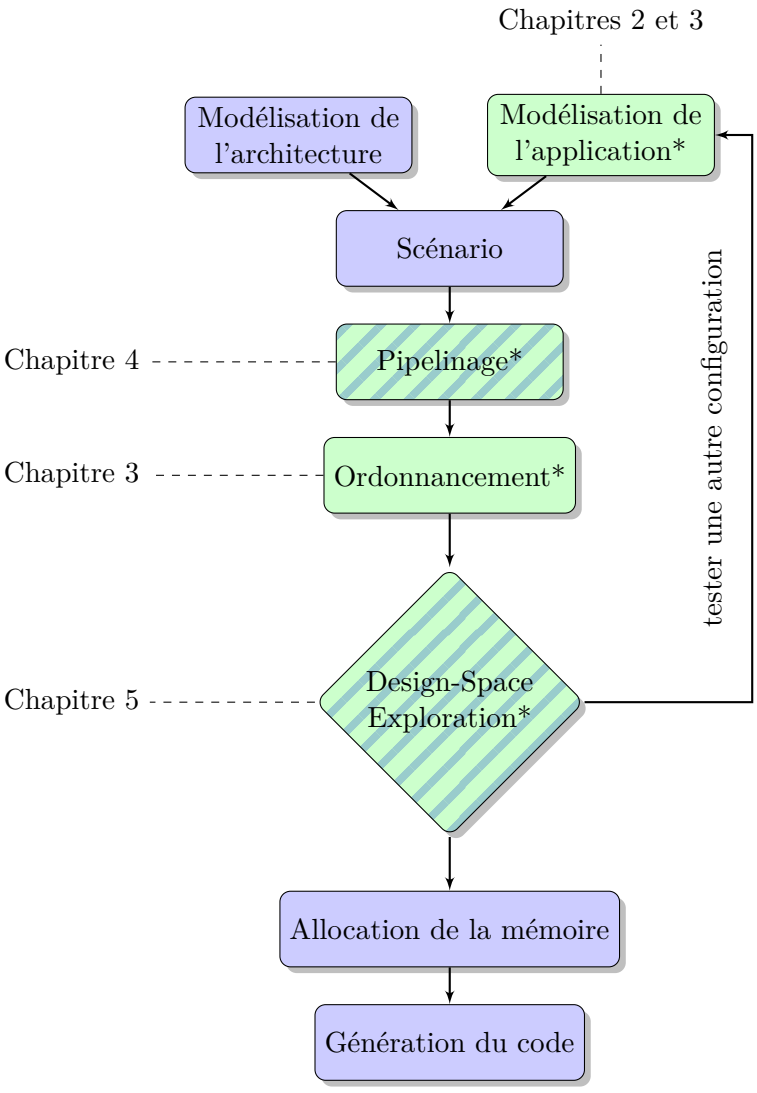


FIGURE 2 – Logigramme des contributions telles qu'utilisées dans l'outil PREESM. Les boîtes vertes* correspondent aux contributions. Le motif hachuré indique les contributions qui sont optionnelles lors de l'utilisation de PREESM.

Chapitre 2 : modélisation des boucles `for` imbriquées dans le modèle SDF. Le point de départ de la conception d'une application est sa modélisation. Cependant, il n'est pas commun de commencer directement par le modèle SDF, et beaucoup de concepteurs développent d'abord un prototype utilisant les langages de programmations impératifs les plus répandus comme C ou Java. Dans cette contribution, nous étudions la modélisation des boucles `for`, l'une des plus importantes structures de contrôle des langages impératifs. Plus spécifiquement nous introduisons les *itérateurs SDF*, qui permettent la modélisation des boucles `for` parfaitement imbriquées avec du parallélisme explicite. Les itérateurs SDF sont particulièrement utiles lorsqu'ils sont paramétrés. Dans ce cas, nous montrons comment adapter la parallélisation des boucles aux nombres de processeurs disponibles dans l'architecture. La parallélisation est le fait de répartir les calculs des tâches sur plusieurs processeurs de sorte que le temps d'exécution de l'ordonnancement est plus court.

Chapitre 3 : ordonnancement partiellement périodique des graphes SDF. Dans cette contribution, nous étudions la modélisation de contraintes de périodicité de certaines tâches, par exemple exécutées toutes les 10 unités de temps. Dans le contexte des graphes SDF, ces contraintes entraînent l'existence d'une période du graphe. Nous détaillons un algorithme permettant d'ordonnancer le graphe en respectant les contraintes de périodicité de ses tâches. Cet algorithme est rapide car il appartient à la famille des ordonnanceurs par liste de priorité, statiques et non préemptifs. Non préemptif signifie que lorsqu'une tâche commence son exécution, il n'est pas possible de l'arrêter avant sa complétion.

Chapitre 4 : pipelinage automatique de graphes SDF. Dans cette contribution, nous présentons un algorithme calculant automatiquement les tailles de délais nécessaires au pipelinage. Cette contribution n'est pas mentionnée dans le problème de départ mais elle aide à réduire le temps d'exécution d'un ordonnancement. De ce fait, cette contribution aide à réaliser la configuration sous contrainte de cadence.

Chapitre 5 : configuration de graphes SDF paramétrés. Cette dernière contribution aborde le problème principal de configuration automatique de graphes PISDF. Plusieurs algorithmes de DSE sont proposés, exhaustifs ou non, afin de sélectionner la meilleure configuration de graphes PISDF en fonction des contraintes spécifiées. Les contraintes peuvent concerner l'énergie, la puissance, la latence, la cadence ou la QoS par le biais de n'importe quel paramètre du graphe, tel que la résolution d'image. Cette

contribution repose sur les trois précédentes : elle paramétrise les itérateurs SDF avec le niveau de parallélisme adéquat, elle évalue chaque configuration grâce à l'ordonnanceur de tâches avec périodicité, et finalement elle ajoute des délais pour améliorer la cadence lorsque cela est possible.

Conclusion et extensions

Dans cette thèse, nous avons étudié le problème de configuration des graphes SDF paramétrés ayant des contraintes de QoS et étant exécutés sur des architectures multi-processeurs hétérogènes ayant des contraintes matérielles. Pour résoudre ce problème, nous avons proposé quatre contributions concernant la modélisation, l'ordonnancement, le pipelinage et la configuration elle-même. Malheureusement ces contributions sont limitées au cas des architectures multi-processeurs homogènes. Néanmoins, les évaluations expérimentales ont montré que le problème de configuration peut être résolu pour le cas homogène en un temps raisonnable de quelques minutes, avec des contraintes sur la résolution d'image ou la fréquence du MPSoC. Les contraintes d'énergie, de puissance, de latence et de cadence sont aussi considérées. En particulier nous avons pu configurer l'application de traitement d'image SIFT pour une utilisation en temps-réel, de sorte que la qualité d'image est maximisée tout en respectant une contrainte de cadence de 30 images par seconde. Toutes nos contributions sont implantées dans l'outil libre PREESM.

Parmi les extensions possibles, il est primordial de résoudre le problème de configuration pour les architectures hétérogènes, qui sont la norme des MPSoCs modernes. Par ailleurs, nos contributions ne prennent pas en compte les temps de communications ni les limites de la mémoire des systèmes intégrant ces MPSoCs. Pour attaquer ce cas plus général, nous pensons que deux domaines sont particulièrement importants : (1) l'analyse symbolique des paramètres et (2) le raffinement de solution par mesures réelles. Le domaine (1) permettrait de rapidement connaître l'évolution des objectifs en fonction des paramètres sans avoir à tester toutes les configurations. Le domaine (2) permettrait d'évaluer la configuration choisie dans son environnement réel, car un modèle ne modélise jamais tout. Enfin, l'extension de nos travaux par les domaines (1) et (2) rendrait possible d'effectuer les configurations de manière dynamique, par exemple pour pallier en temps-réel la panne d'un des processeurs.

Introduction

As computer processors went smaller and cheaper, they have been used in more and more devices easing the human life. One of the most iconic example is the smartphone, which is now almost as powerful as a personal computer. Smartphones embed even more components than a computer: at least one camera and an antenna. In 2019, 77% of French people owned a smartphone, and 76% had a personal computer[5].

Each component of a smartphone may actually contain its own processor. For example, the latest *smartcamera* Sony MX 500 integrates a processor directly on the chip embedding the image sensor, not larger than 1.5 centimeter. Multi-Processor System-on-Chips (MPSoCs) are one of the main ingredients for this miniaturization success. Since MPSoCs contain multiple Processing Elements (PEs) on a single chip, they host on-site efficient computation capacities which for example, allow us to stabilize a video in real-time on our smartphone using algorithms instead of heavy mechanical stabilizers.

A smartphone contains not one, but multiple heterogeneous processors: multiple generic processors for regular applications, and specific processors dedicated to a kind of computations such as signal processing. Computations are generally composed of multiple tasks to execute, forming a graph. To accelerate the computations, their tasks are shared over the processors and the specific processors execute the tasks they are dedicated to. To select where and when to execute the tasks, that is *scheduling*, is a complex problem to solve, also related to means of communication between the components.

Another smartphone and smartcamera application example is the creation of panorama pictures thanks to regular pictures from different point of views. Multiple algorithms may be used in such software applications, each offering a different degree of quality and rapidity. Moreover, the user generally sets software requirements defining its minimal Quality of Service (QoS), such as the minimum image resolution or the minimum frame rate. Yet, stabilization or panorama software applications require to be adapted to the smartphone hardware constraints: maximum image resolution and frame rate of the camera, maximum available power of the battery, maximum frequency of the MPSoC. Meeting software requirements while respecting hardware constraints is also a

[5]Statistics extracted from the official French report Baromètre du numérique 2019 .

challenging problem. This *configuration* problem is generally solved by testing different configurations of the application and the hardware, that is Design Space Exploration (DSE), until one matches all constraints and requirements. For example, configurations may differ by the MPSoC frequency, or by the image resolution supported by the application. If no configuration matches all constraints and requirements, a trade-off is necessary. In such case, a few constraints or requirements have to be relaxed.

In the end, the adaptation of a software application to a target hardware architecture, such as a smartphone application, is not trivial. This adaptation requires to solve multiple problems at the same time, at least scheduling and configuration. To solve these two problems, both the application and the architecture are *modeled*. The goal of the model is to abstract the key properties which are involved in the scheduling and configuration problems.

These modeling, scheduling, and configuration problems are all active research fields existing for a few decades. For example, the Synchronous Data Flow (SDF) [LM87b] Model of Computation (MoC) is dedicated to signal processing applications, including processing tasks on image and antenna signals. In the SDF model, both processing tasks and their data are represented. Processing tasks and data exchanges are fixed by the designer and the SDF model allows for a deterministic execution of the application. SDF may be *parameterized* with Parameterized Interfaced Meta-Model (PIMM) [Des+13] for example, giving choice between multiple possible values and making it eligible for the configuration problem. SDF belongs to the wide family of *dataflow* MoCs, which focus on the representation of data exchanges and data processing.

Many tools exist to automatically generate and optimize the code of a software application on a specific hardware architecture, as SynDEx [GLS99] implementing the *Algorithm Architecture Adequation* methodology. SynDEx optimizes the execution of a dataflow application on an architecture as an MPSoC, by selecting the shortest possible schedule for example. However, it does not *configure* the application in order to meet both hardware constraints and software requirements. For instance, SynDEx will not select automatically the best parameter value of image resolution given hardware constraints and software requirements. Other models and related tools, such as Architecture Analysis and Design Language (AADL)[6] [Hug+08] and Ptolemy[7] [Guo+14], enable the designer to represent more information: the application, the architecture, and respectively some of their requirements and constraints. This approach greatly helps to verify if requirements and constraints are met, but as with SynDEx, designers still need to test

[6] See AADL and OpenAADL websites.
[7] See Ptolemy website.

different configurations before selecting the best one. Using DSE algorithms accelerates this selection process, but does not completely solve it yet, especially when configuring software requirements or software QoS.

Finally, selecting the best configuration to meet hardware constraints and software requirements on an heterogeneous hardware architecture is still a modern challenge. This configuration problem is composed of multiple other problems, some of which are introduced in the next section. The contributions proposed in this thesis concern three of theses problems plus the configuration itself. The configuration and related problems are introduced in the next section. Our contributions are listed in the second section of this introduction. Then, chapter 1 presents the background of this thesis. The contributions are detailed in Chapters 2 to 5 and each contribution has its own specific related work. As the last contribution uses the three previous ones, we advise the reader to read this thesis in order.

Problem statement

This thesis tackles the configuration problem of parameterized SDF applications having software requirements and running on multi-processor architectures having hardware constraints. The contributions proposed in this thesis consider only homogeneous multi-processor architectures containing multiple identical PEs. However, the same configuration problem applies for heterogeneous architectures, and is even more difficult to solve in such case.

Executing an SDF graph requires to solve multiple problems: the scheduling of the tasks and the one of data exchanges, the memory allocation for tasks execution and the one for data exchanges, while taking into account constraints such as the number of PEs in the architecture and the periodicity of some tasks. Each of this problem is complex, NP-complete in the general cases. While many algorithms already exist to solve each problem, we focus on the fast and scalable ones, so that all solutions can be tested in a short time, for example, in less than one hour.

Rapidity is especially needed when considering parameterized SDF graphs. The parameters have still to be fixed when executing the processing tasks of the SDF graph using them, but they allow algorithms to automatically explore different configurations, or to react dynamically to inputs. Indeed for configuration exploration, the number of configurations may explode exponentially in the number of parameters. And in the dynamic case, rapidity is needed to limit the overhead created by online solving of the scheduling and allocation problems. In this thesis, we use the Parameterized Interfaced

Synchronous Data Flow (PISDF) MoC, an extension of SDF parameterized with PIMM. The dynamic properties of PISDF are not used, and instead we restrict our work to its static semantics to find offline a suitable configuration of the modeled application, fitting all its constraints when possible.

In this thesis, the word *constraint* may refer to both hardware constraint and software requirement. Although we have distinguished software requirement from hardware constraint in the previous paragraphs, both may be related to the same metric. For instance, the frame rate is constrained by a minimum value the software side and a maximum value on the hardware side. Constraints may be the real-time periodicity of some tasks, the minimization of energy or power consumption per execution, the minimization of latency and the maximization of throughput of the executions, the minimization or maximization of any graph parameter to ensure QoS. The choice among multiple configurations results from a DSE to evaluate the interesting ones. This main configuration problem may be split into multiple problems, a non-exhaustive list follows:

- parameterization of static applications in the PISDF MoC;
- modeling of periodic tasks with the PISDF MoC;
- rapid offline scheduling of PISDF graphs having periodic tasks;
- modeling of the throughput, latency, energy, power and QoS constraints.

We now detail our contributions with regard to the main problem and the list above.

Contributions

In this thesis we present four contributions, one of which is the configuration of PISDF graphs via automatic DSE. The three other contributions all help to perform this DSE, by the modeling of constraints and the scheduling of tasks respecting them. The contributions are briefly summarized in the next paragraphs; each contribution corresponds to a chapter of this thesis. The Scale Invariant Feature Transform (SIFT) [Low04] image processing application provides our main use-case; it is used in the experiments all along the thesis, in Chapters 2, 4 and 5.

Figure 3 illustrates how our contributions are related in the design process. All have been implemented in the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) [Pel+14] tool supporting the PISDF model. Note that as SynDEx, PREESM separates architecture modeling from application modeling, and the *scenario*

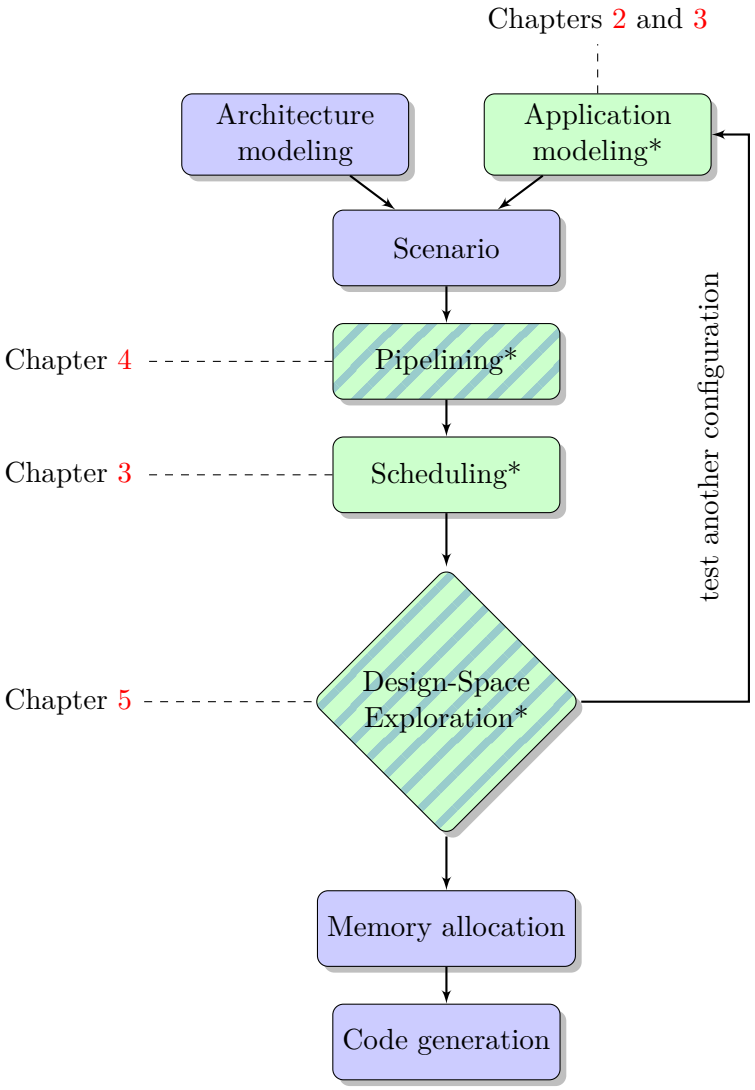


Figure 3 – Flowchart of the contributions as used in the PREESM framework. Green* boxes correspond to contributions. The hatch pattern indicates the contributions which are optional in the PREESM workflow.

node in Figure 3 stores information specific to one architecture and one application, as the Execution Times (ETs) of each task. Our contributions concern the green* steps: application modeling, pipelining, scheduling and DSE, which performs configuration.

Chapter 2: Modeling nested `for` loops, with SDF graphs. The starting point of the design of an application is its modeling. However, it is not common to start with an SDF model, and many designers first develop a prototype using widespread imperative

programming languages as C or Java. In this contribution we study the modeling of `for` loops, one of the most important control structure in imperative languages. More specifically we introduce *SDF iterators*, which support the modeling of perfectly nested `for` loops having explicit parallelism. SDF iterators are especially useful if parameterized. In such case, we show how to adapt the parallelization of loops to the number of PEs in the target architecture. Parallelization is the process of splitting computations of a task on multiple PEs, so that its execution time is shorter.

Chapter 3: Scheduling partially periodic SDF graphs. In this contribution, we study the modeling of periodicity constraints, expressed on some tasks but not all. In the context of SDF graphs, we show that any task period enforces a common *graph period*. We develop an algorithm to schedule the graph while respecting the periodicity constraints. This algorithm belongs to the family of offline static non-preemptive *list schedulers* and thus is rapid and scalable. Non-preemptive means that when starting the execution of a task, it is not possible to stop it.

Chapter 4: Pipelining SDF graphs automatically. SDF graphs model the exchange of data between tasks, and it is possible to feed these exchanges with initial data, present before the execution. These initial data are called *delays*. They break the data dependencies and thus they may improve the scheduling result, giving a greater execution throughput. In this contribution, we present an algorithm to automatically add such delays on an SDF graph. This process is called *pipelining*. While this contribution is not mentioned in the problem statement, it helps to solve the main problem by providing more possible configurations.

Chapter 5: Configuration of parameterized SDF graphs. This contribution tackles the main problem, that is the automatic configuration of PISDF graphs. It provides multiple DSE algorithms, exhaustive or not, to select the best PISDF graph configuration with regard to multiple constraints. Constraints may be on energy, power, throughput, latency, or QoS via any parameter, as the image resolution. This contribution uses all the three others; it parameterizes the SDF iterators with the correct amount of parallelism, it evaluates each configuration thanks to our fast list scheduler and finally, it adds delays if the throughput can be improved.

Chapter 1

Background

Contents

1.1 Introduction: hardware, software, and parallelism 2
1.1.1 Common tools and hardware 3
1.1.2 Software side 6
1.1.3 Hardware side 9
1.2 The SDF dataflow model and its flavors 13
1.2.1 Original SDF 14
1.2.2 Static SDF flavors 19
1.2.3 Dynamic SDF flavors 21
1.3 Scheduling of SDF graphs 22
1.3.1 Scheduling techniques 22
1.3.2 Tools and benchmarks 26
1.3.3 Analysis for scheduling efficiency 29
1.4 The PREESM tool 30
1.4.1 Architecture of a PREESM project 31
1.4.2 Definition of PISDF graphs and parameters 33
1.4.3 Computation of the repetition vector 36
1.4.4 Single-Rate graph and special actors 39
1.5 Conclusion 41
1.5.1 Model used in this thesis 41
1.5.2 Main notations 41

In this thesis, we study the configuration of a software application with Quality of Service (QoS) constraints which has to be executed on a specific hardware architecture. In this chapter, we briefly introduce the main concepts related to this configuration problem, especially *parallelism*, on the hardware and software sides. In Section 1.2, we introduce the main model used in this thesis, the Synchronous Data Flow (SDF) Model of Computation (MoC). For each contribution, a specific related work is presented at the end of the corresponding chapter. Thus, the Section 1.3 presenting the scheduling and the analysis of Synchronous Data Flow (SDF) graphs only gives an overview of these concepts. The Parallel and Real-time Embedded Executives Scheduling Method (PREESM) tool used to implement the contributions of this thesis is described in Section 1.4. In Section 1.5, we briefly recall the characteristics of the model used in this thesis and give notations which are common to all chapters.

1.1 Introduction: hardware, software, and parallelism

Parallelism is the simple fact to execute two tasks in parallel. In the everyday life, cooking is an example of activity where we often parallelize tasks. The parallelism is clearly written in good recipes, in general stating: "while this thing is cooking, do something else". In the french recipe of "la tourte aux blettes ", it is possible to cook the pears while preparing the grapes and pine nuts. This kind of parallelism is called *task parallelism*. To perform multiple different and independent tasks in parallel accelerates the cooking process. Another way to accelerate the process is to have multiple cookers performing the same task, for example each cooker peels one pear. This is *data parallelism*, where the same task is performed on multiple inputs. The metaphor between cooking and computer science can also be used for the software and hardware sides. The software side is the recipe itself, that is the list of tasks to execute and extra instructions on how to perform those tasks. The hardware side corresponds to the equipment and the cookers. Cookers are the Processing Elements (PEs) processing the food. When the food is not used, you place it in the fridge, that is the memory of a computer. Last but not least, cooking comes with QoS constraints: the food must be good and served before it becomes cold. In any restaurant, the main ingredient to respect the QoS actually is *parallelism*. The chef is not only here to create the recipes but also to *schedule* the tasks of the cookers.

In computer science, parallelism helps to accelerate the computations when multiple Processing Elements (PEs) are available. Then, heavy computations such as weather forecast, can be executed in a reasonable time. In the following subsections, we list

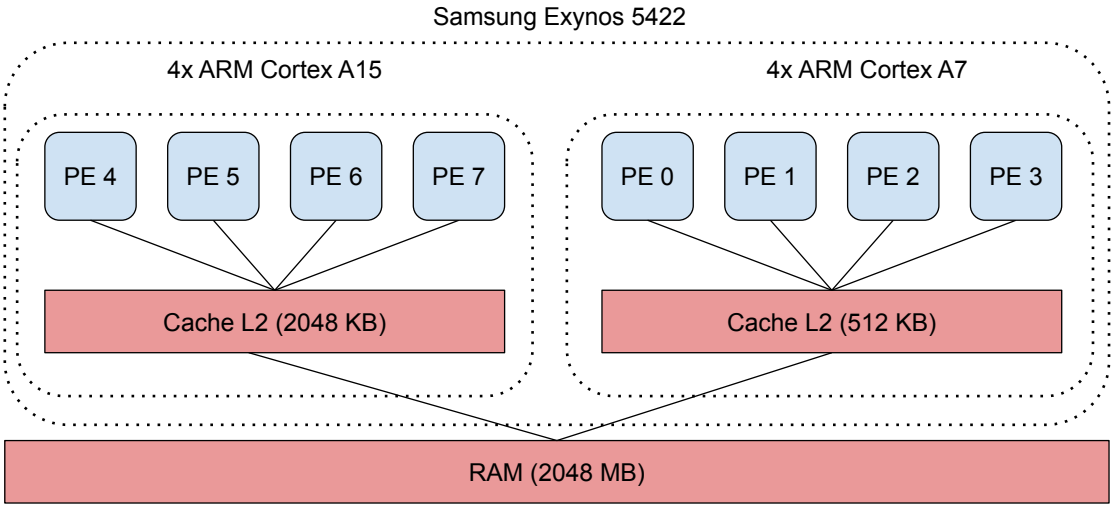


Figure 1.1 – Samsung Exynos 5422 MPSoC architecture: ARM big.LITTLE design with 4x Cortex A15 (@ 2.1 GHz) and 4x Cortex A7 (@ 1.5 GHz) CPUs. It also embeds a GPU, not depicted here. Odroid XU3 board contains one Samsung Exynos 5422 and 2 Gbytes of RAM. Only LLC size is specified.

some of the main tools to perform parallelism. In Section 1.1.1, we list programming languages and libraries dedicated to parallelism on regular Multi-Processor System-on-Chips (MPSoCs). Programming languages are a common way to represent the recipes in computer science, and we precise their main concepts in Section 1.1.2 dedicated to the software side. Multiple kinds of hardware are listed in the last Section 1.1.3.

1.1.1 Common tools and hardware

Parallelism is made possible on the hardware side by the possibility to use multiple PEs of a multi-processor architecture. Figure 1.1 depicts a MPSoC architecture having 8 PEs of two different types. A main Random Access Memory (RAM) memory is accessible from all PEs. To continue the cooking metaphor, the architecture is the map and the elements of the kitchen, that is the position of the fridge compared to the gas stoves, and the number of gas stoves.

The most common way to use all PEs at the same time is to create *threads* defined by the Operating System (OS) running on the MPSoC. Threads are a Model of Computation (MoC). Each thread contains a list of tasks to execute successively. There might be more threads than PEs. In such case, extra threads are not executed until a currently executed one finishes or is suspended. The OS may suspend the execution of any thread to execute

another one; this is called *preemption*. Suspended threads can continue their execution later on any PE, not necessarily the same as previously. One role of the OS is to ensure that threads will be completed sooner or later.

Threads can express both data and task parallelism. When there is only one *shared* memory, threads can exchange data between them without specific communication *library*. A library is a set of predefined functions made available to the designer. However, if multiple threads have to work on the same data, a lock system may be needed to ensure that multiple data writes do not occur at the same time for example. Semaphores between threads are the main lock system to temporarily limit parallelism. Parallelism limitations are called *bottlenecks* and reduce the execution time of an application. The extreme case of parallelism limitation is *mutual exclusion*, which enforces that only one thread is executed at a time, even with multiple available PEs.

Parallelism through programming languages

Programming languages help to express the computations of an application. The text files written in any programming language are called the *code* of the application. They correspond to the language in which the cooking recipe is written. Their main advantage is to be readable by human programmers, and by either *compilers* or *interpreters* which translate the code into binary instructions for the OS or directly for the PEs.

Most of the modern programming languages support the thread MoC through native libraries: C/C++, Python, Java, etc. Some languages can even represent parallel computation directly in the language without exposing a thread library: ADA, Cilk, Chapel, Erlang, LARA [Car+12], ABS [Joh+12], etc. Programming languages supporting threads generally offer at least two functions: one function to create and launch a thread, that is to make the thread executable by the OS, and one function to wait for the completion of a thread. This is the *fork-join* model. Waiting for the completion of multiple threads is called a *synchronisation*. It is also possible to synchronize threads before their completion, usually with semaphores.

Parallelism through compiler

Mastering programming languages is difficult, especially when using threads. Human programmers have to find which tasks may be executed parallel, which data have to be protected for writing, which threads to wait and when, etc. The compiler does not only translate the code for the PEs, but it can also perform parallelism automatically in a few cases. For example, C/C++ compilers can control fine grain parallelism available

on vector PEs implementing the SSE/AVX binary instruction sets. Such instruction sets offer data parallelism at the instruction level: the same instruction is executed on multiple data at the same time by only one PE.

However, both fine grain parallelism controlled by the compiler and coarse grain parallelism controlled by the programmer are difficult to use properly. Thus, it is common to perform automatic code analysis and source-to-source transformations, such as in the PIPS [IJT91] tool, to discover and improve parallelism. An important class of static analysis is the polyhedral analysis [FL11], used in [Bon+08; NC12] for example. Another alternative is to use modulo scheduling [RST92]. In order to guide the transformation, it is also common to use annotations as in [KL12] or `pragma` as for the OpenMP or OpenACC libraries. Sometimes transformations are guided by interaction with the programmer as in Parascope [Bal+89] in the past and in a few modern tools [Lar+12].

The OpenMP library is a major library for data and task parallelism. It is especially useful to parallelize `for` loops, iterating over an array for example. If operations on the elements of the array can be performed in parallel, the programmer indicates to OpenMP and the compiler that the operations of the loops may be split over threads. OpenMP will automatically create the threads and wait for them.

Parallelism through task and array manager

However OpenMP is not perfect for multiple reasons. The main reason is that OpenMP offers a small amount of scheduling policies managing the start time of the threads. Policies of OpenMP are efficient for homogeneous architectures having identical PEs, but not always for heterogeneous architectures. Many libraries are designed for this purpose, as StarPU [Aug+11], PaRSEC[1], or XKaapi [Gau+13]. Such libraries can be interfaced with OpenMP so that the programmer does not change the pragma annotations in the code, but the scheduling policy of OpenMP is replaced by the one of the library. Such libraries are usually called *runtime managers*. Many other parallelism libraries exist, especially for arrays: DASK[2] and Kokkos [ETS14].

In any case, with such parallelism libraries, programmers need to indicate by themselves where parallelism is available in the code and when synchronizations are needed. Dedicated tools may help to locate where parallelism is possible and when it is not, for example, STABILIZER [CB13] and COZ [CB18]. To locate where parallelism is possible is a difficult task, depending on the model used to represent applications. In

[1] http://icl.utk.edu/parsec/
[2] https://dask.org/

this subsection, we have listed the main programming languages and libraries supporting the thread MoC. In the next subsection, we present concepts and MoCs related to parallelism, especially data flow programming which is used in this thesis.

1.1.2 Software side

Modern programming languages such as Python or Java are widely used for regular applications. Threads or libraries upon threads are available in these languages and ease the expression of parallelism, but they require caution from the programmer and extensive analysis of the compilers or the libraries to be used efficiently. Indeed, managing data accesses between threads is error-prone, and introduce synchronizations to avoid concurrent data accesses, thus creating bottlenecks. A common source of errors with threads is the introduction of *deadlocks*, for example when two threads wait for the completion of the other one before continuing the execution of their own tasks. Another source of error is the memory management when exchanging or storing data. Some concepts and MoCs are dedicated to model specific behaviors of the applications, so that the design process is easier to use for programmers and easier to analysis for compilers. Such main concepts are: *control flow* and *data flow*.

Control flow and data flow

Control flow fixes the order of the computational tasks to execute. Especially, the `if/else` control flow conditional statements of *imperative* languages, such as Python or Java, enable the programmer to select a *branch* of instructions depending on the result of previous computations. In a way, it is the opposite of parallelism since only one branch is executed among all possible, the other branches of `if/else` statements will never be executed. Threads available in imperative languages are the only way to escape such exclusive choices and fixed orderings. Threads actually move part of the ordering of tasks from the programmer responsibility to the OS or runtime library responsibility. Imperative languages are control flow oriented.

On the other side, data flow focuses on the data exchange between the tasks to execute. The order of tasks is defined only between tasks exchanging data. Thus, two tasks which are not connected by any *data flow* can be executed in parallel. In data flow programming, parallelism is explicit. Programmers still have to express by themselves where parallelism is available, but such explicit parallelism is less error-prone and eases, in some cases, the automatic analysis made by compilers.

Nevertheless, both control flow and data flow are useful to express different kinds

of behaviors: exclusive choice for control flow and parallelism for data flow. Both concepts usually integrate a part of the other: threads in imperative languages and mutual exclusion in data flow languages. When possible, considering both at the same time is preferable [GBL99] for the designer but harder to analyse and to optimize. In the rest of this document, data flow is spelled *dataflow*, referring to both a flow of data and the concept of representing flows of data.

Control flow examples: state machines. State machines are very close to the concept of control flow when they are deterministic, that is when only one state is enabled at a time. Many kinds of state machines exist: Finite State Machine (FSM), automata, Kripke structures, Mealy machines, etc. While the aforementioned state machines do not have timing properties such as task Execution Times (ETs), some other kinds of states machines, Timed Input Output Automata (TIOA) [Dav+10; Jia+13], are refined with timed transitions from one state to another. State machines are heavily used to model systems in order to verify properties on it, as in model-checking, for example for real-time systems [Ost95]. UPPAAL[3] is a common model-checker for state machines with timing properties. However state machines generally are only one part of the system, and the computations performed at each state or each transition may rely on any programming language.

Dataflow examples: communicating process networks. Dataflow is especially needed for communicating systems where data exchanges have to be explicit because there is no common shared memory. In the seminal Kahn Process Networks (KPNs) [Kah74], tasks (called "processes") exchange data and are executed only when they have received enough data. Computations of a KPN are *deterministic*: for a given input, it always returns the same output. Other dataflow languages, slightly less generic and so easier to analyse, have been developed at the same time [Den74] or a few years later [DK82]. Dataflow Process Network (DPN) [LP95] extend KPNs with possibly non deterministic behavior. Petri nets [Pet66] are another important example although not only dedicated to communicating systems. They were originally thought as a more powerful alternative to state machines and can also have non deterministic behavior. Petri nets are powerful because they can formally represent lock systems and mutual exclusions; for example, see [BSZ16] for an implementation on a Kalray architecture. Petri nets can be extended with ETs [Zub91; Bér+05], or extended to dynamic systems [DA94], or be refined with hierarchy [Hid+08]. Expressiveness of Petri nets is large but implies that they are diffi-

[3] http://www.uppaal.org/

cult to analyze in the general case, especially because of their possible non deterministic behavior. However it is stil possible to perform some analysis as throughput anaylisis [Zub93]. Processing Graph Method (PGM) [Ste97; Kap97] proposes a MoC close to Petri nets, see [God97] for an application example. Besides, we can consider that all message passing middlewares, such as MPI[4] for distributed architectures with multiple memories or RabbitMQ[5] for inter-connected embedded systems, and even some OSs such as ROS[6], are dataflow oriented. Middlewares are libraries at the frontier between the software and hardware sides, right above the OS level if any. Finally, dataflow explicitly expresses task parallelism but sometimes implicitly data parallelism, for example if threads are created with the same processing task. Note that both kinds of parallelism are not always suitable depending on the application [CDY95]

Specialized languages

Many languages and MoCs are specialized to specific kind of computations or environments. We list here the most related to our work, at the frontier between real-time systems and stream processing. Real-time systems are usually constrained by periodic behaviors or maximum latency. Stream processing consists of applying the same process to a never ending *stream* of successive inputs.

Synchronous languages. Synchronous languages are heavily used in the domain of real-time systems where some timing properties are important to guarantee during the application execution [BB91]. Synchronous languages support to model and to check timing properties by the mean of periodic executions, that is for example, to execute a task every 10 millisecond. Among such languages, there is ESTEREL [BG92], control flow oriented. On the dataflow side, there are LUCID [WA85], Lustre [HLR92], Signal [LTL03] or even SynDEx [GLS99]. In its original publication [HLR92], Lustre is qualified as a "synchronous dataflow language", but this is not related to the Synchronous Data Flow (SDF) MoC of Edward A. Lee, which is not cited in the references of LUCID, Lustre and Signal. Nevertheless, dataflow synchronous language may execute SDF graphs thanks to a few transformations, as for Signal [SGL99; Bes+10].

Languages for real-time systems. A few languages focus on the specification of real-time constraints of complex application, as Giotto [HHK03]. Another language [PG16]

[4] https://www.open-mpi.org/
[5] https://www.rabbitmq.com/
[6] http://www.ros.org/

mixes Petri nets, SDF MoC, and synchronous languages for the modeling of Cyber-Physical Systems (CPSs). Other languages are dedicated to industrial systems, such as AutoSar[7] for cars and Architecture Analysis and Design Language (AADL)[8] for aircraft. Both languages support extensive modeling and analysis. For example, AADL supports synchronous and dataflow modeling and verification [Ma+13; Bes+14], and real-time properties modeling and verification [Dis+10].

Languages for stream processing. Stream processing applications often use dataflow languages to model and optimize their computations, such as the Canals [Dah+09] language. Canals originality is that it supports the description of the architecture and the task scheduler as well as the dataflow modeling of applications. There exist also the Brook [Lia+06] and SPUR [Zha+05] languages and corresponding compilers dedicated to dataflow stream applications. A more recent language also supports the description of task implementations and stream specifications [Wei+14]. An important category of stream processing applications is image processing, targeted by Diderot [Chi+12] for example. Diderot is more dedicated to the expression of the mathematical functions of the processing. Array-OL [Dem+95; Bou07] focuses on array traversals, and is dataflow oriented. Besides, Polka [Dem+99] is an old attempt to model distributed multimedia applications with CORBA.

Transformations towards dataflow. As many programmers first start by prototyping applications with imperative languages, some works study the automatic transformation of imperative languages to dataflow languages. For example, the DWhile [DFR11] language is directly embedded in the host language, such as C++, to provide dataflow specifications. Moreover, there exist an automatic transformation of Single Static Assignment (SSA) code towards "dataflow threads" [LPC12], and an automatic extraction of KPNs from `while loops` [Agu+15].

1.1.3 Hardware side

Multiple kinds of processor exist, generic or dedicated to specific computations. A processor usually refers to a Central Processing Unit (CPU), able to compute any common instruction. However, only in this chapter, we call "processor" any kind of PE, having a possibly reduced instruction set, or even none at all if implementing a specific circuit

[7] https://www.autosar.org/
[8] http://www.aadl.info

design. Personal computers usually embed one CPU multi-processor and one Graphics Processing Unit (GPU) card. GPUs are dedicated to data parallelism while CPUs support any kind of computations, but then are less efficient. Efficiency is measured in the number of floating point operations performed per second, sometimes also per Watt. CPUs operate at a higher frequency than GPUs but a GPU card may contain more than 100 processors while a CPU multi-processor only contains around 8 processors.

When prototyping an application, the target hardware architecture has to be chosen carefully, depending on the most suitable MoC for the application, and depending on the throughput constraints. Moreover, more specialized processors require more complex compilers to be used efficiently. Indeed, on the most specialized processors, there is no OS to manage thread parallelism, and the compiler has to compute itself the scheduling, that is the ordering and mapping of tasks. Hardware side and software side are not meant to be opposed; OSs, middleware libraries and compilers are here to adapt both sides to each other. To reach the optimal throughput, hardware has to perfectly match the software, and vice versa.

Among the common properties of processors, the most important is their clock frequency. All processors execute binary instructions or activate circuit gates at a maximum throughput being their frequency, generally between 2 and 3 GHz for CPUs. Each instruction may require multiple tics of the clock frequency to be completed, but a new instruction can be started at each tick, even if the previous ones are not completed. This is *hardware pipelining*. The only exception to pipelining is when an instruction must wait for the result of the previous one. However, *Out of Order* processors can automatically execute a next instruction before the awaiting one if it does not break any data dependence. In the next paragraphs and subsubsections, we briefly present the main kinds of processors.

Flynn's taxonomy [Fly72]. Flynn described four main kinds of processors, categorized by their ability to process single or multiple instructions on single or multiple data. Today, two kinds are commonly used: Single Instruction Multiple Data (SIMD) for GPUs and Multiple Instruction Multiple Data (MIMD) for cluster and grid of CPUs or CPUs multi-processors directly.

Models of Architecture (MoA). When a compiler transforms code to binary instructions, it may rely on a model of the processor. Many Model of Architectures (MoAs) exist, often integrated in and dedicated to the tools using them. MoAs also help to simu-

late and prototype architectures, as with Gem5[9]. In this thesis, we use the System-Level Architecture Model (S-LAM) [Pel+09a] MoA. For example, MoAs are used to analyze systems and to predict communication times [Cru91a; Cru91b; Boy+18], or Worst Case Execution Time (WCET) of tasks [HRP17].

Regular CPUs

Regular processors are the CPUs used in laptops or desktops. Such processors are generic, but already integrate multiple techniques to accelerate the computation. SSE/AVX binary instructions are one of those techniques. With such instructions, a unique CPU can perform the same instruction, such as an addition, on multiple data at once, that is data parallelism. Only a few data elements, 8 in general, may be *vectorized* and processed at the same time by this technique.

All modern CPUs integrate cache memory [Smi82] to buffer the data accesses to the main RAM. Such cache buffers are especially useful for data parallelism on arrays because when an array element is accessed on RAM, not only the element but also its neighbors are loaded in the buffer, more precisely in a cache line. Caches are faster than RAM and avoid time consuming direct data exchanges between the RAM and the CPU. CPUs multi-processors, generally integrate one main shared Last Level Cache (LLC), or multiple ones in the case of Non Uniform Memory Access (NUMA). NUMA implies coherence protocols to keep track of the freshest value of a data.

Finally, as CPUs are generic processors, each one integrates multiple kinds of Arithmetic Logic Units (ALUs), such as addition or multiplications. To not waste resources, many processors support Simultaneous Multi-Threading (SMT) which means that they actually execute multiple instructions in parallel and coming from different threads, only if they do not involve the same ALUs at a time. For instance, Intel[10] processors propose *hyperthreading* to execute two threads at a time on the same CPU.

Many-cores

Many-cores processors are the extreme case of MIMD processors, containing in general more than 100 identical processors on the same chip. Intel Xeon Phi is a standard professional many-cores. There exist also: Celerity [Dav+18], Epiphany[11], Bostan and Coolidge from Kalray[12], AEthereal [GH10], TILE64 [Bel+08], etc. Such many-cores

[9] https://www.gem5.org/
[10] https://www.intel.com
[11] https://www.parallella.org/2016/10/05/epiphany-v-a-1024-core-64-bit-risc-processor/
[12] https://www.kalrayinc.com/

processors require complex communication hardware between the *cores*, called Network on Chip (NoC). This kind of architectures is difficult to use efficiently with regular compilers and it requires a proper model [Li+17]. Similarly, the RAW [Wai+97; Lee+98; Tay+02] processor exposes some of its NoC circuit logic to the compiler to use it more efficiently, including for dataflow applications [Gor+02].

Ancestors of many-cores. Some old architectures also tried to integrate many processors together, such as the Cube-Connected Cycles [PV81] or the Wavefront [Kun+82] architecture and parallelism language, close to dataflow.

Exposed datapath

Exposed datapath architectures [Cor99; Bur+04] are even closer to the definition of dataflow, on the hardware side. In such architectures, communication connections between the many processors may be adapted to the dataflow of an application. FlexCore [Thu+07] is a multi-processor with exposed datapath, as well as the simulated architecture WaveScalar [Swa+07], or another targeting low energy and SIMD [Wae+15] As for many-cores, multi-processors with exposed datapath require specific compilation techniques [BJS16; BS17].

Embedded and specialized

Digital Signal Processors (DSPs) [Liu10] architectures, for instance, are multi-processors dedicated to a few kind of computations; they correspond to Application-Specific Instruction set Processors (ASIPs). ASIPs support only a reduced set of binary instructions and any application using only those instructions can be run on them. Field-Programmable Gate Array (FPGA) processors may also be seen as ASIPs but having no instruction set; indeed their logic circuits are reconfigurable but once configured they are dedicated to rapidly execute a few kind of pipelined computations only triggered by data inputs, they are far less generic than CPUs. Application-Specific Integrated Circuits (ASICs) are the most specialized kind of processors, whose circuits are fixed and optimized for only one application. When processors are specialized, they usually require less space and less energy for the electronic circuits and thus are more easily embedded with other components, such as sensors.

Qualcomm[13], Texas Instruments[14] and STMicroelectronics[15] are examples of large

[13] https://www.qualcomm.com/
[14] https://www.ti.com/
[15] https://www.st.com

producers of ASIPs. Smaller companies, such as Renesas[16] or Maxelers Technologies[17] produce even more specific processors. With ASIPs, compilers perform not only the translation of the programming code to the binary instructions or to the circuit design, but also complex memory mapping and scheduling, or floor-planning of the FPGA circuits.

Processors for real-time systems. Verification and hardware synthesis of real-time systems is challenging [HS06]. One of those challenges is to correctly model the behavior of hardware, whereas circuit complexity of CPUs multi-processors is still increasing. Thus, a few processors are designed specifically to ease the verification of real-time applications executed on them, such as T-CREST [Sch+15], used in the Patmos[18] project.

1.2 The SDF dataflow model and its flavors

The Synchronous Data Flow (SDF) MoC has been introduced by Lee and Messerschmitt [LM87b]. In this thesis, SDF always refers to the SDF MoC. The main advantage of SDF is to represent both data parallelism and task parallelism of dataflow applications with fixed and deterministic communications. Control flow is not available in SDF. Such static behavior allows for extensive automatic code generation and optimizations of the applications modeled with SDF. For example, designers are generally not required to allocate by themselves the memory of SDF applications. SDF especially targets stream applications which are executed indefinitely on new input data at each execution.

An application modeled with SDF is usually represented with SDF graphs. Graphs actually are a common and intuitive way to represent any dataflow MoC, as for the old BLODI language [KLV61]. Moreover, graphs are close to the specifications of dataflow architectures [AC86]. However, some languages and libraries support a programmatic expression of SDF graphs, such as DIF [Hsu+04]. Code written in other programming languages can even be analyzed in order to know if they fit into the SDF model [WR12; Zeb+08]. The analysis related in [WR12] relies on abstract interpretation [CC10] while [Zeb+08] relies on state machines describing communication rates of dataflow processing actors.

[16] https://www.renesas.com/eu/en/
[17] https://www.maxeler.com/products/
[18] http://patmos.compute.dtu.dk/

1.2.1 Original SDF

SDF graphs are directed multi-graphs $G = (V, E)$ composed of vertices, called *actors*, and edges, called *buffers*. Actors in V represent processing operations, while buffers in E represent the data communication between the different actors. Actors exchange data through their incoming and outgoing buffers. The abstract unit of data is called *token*. An example of SDF graph is depicted in Figure 1.2a. This graph does not expose task parallelism since there is only one path, however, as we shall see, this graph exposes data parallelism.

A buffer $e \in E$ is a First In First Out (FIFO) queue connecting its source actor $\text{src}(e) \in V$ to its destination $\text{dst}(e) \in V$. Each buffer e is annotated with rates: a production rate $\text{prod}(e) \in \mathbb{N}^*$ at the source of e, and a consumption rate $\text{cons}(e) \in \mathbb{N}^*$ at the destination of e. Production and consumption rates may not be equal: this is how data parallelism is expressed in SDF graphs. For example if an actor α sends 6 tokens to an actor β through buffer e, whereas β expects 3 tokens at the other end of the buffer e, it means that β will be executed twice: once on the first 3 tokens and another time on the last 3 tokens on e. The tokens produced by one execution of $\text{src}(e)$ are available to be consumed only after the end of the execution of $\text{src}(e)$, i.e. the *completion* of $\text{src}(e)$. The number of tokens initially present on a buffer e is denoted $d_0(e)$; these tokens are called a *delay*. In order to not deadlock, cycles of the graph require delays on at least one buffer of the cycle. This is the *liveness* property.

As the rates may not be equal on both sides of a buffer, there can be multiple executions of an actor $\alpha \in V$ in order to avoid underflow or overflow on the buffer. The graph is *consistent* if it ensures that buffers have bounded sizes. Then, it is possible to compute a unique *repetition vector* $\vec{r}$ giving the minimal number of executions of each actor needed to put the graph back to its initial state with the same number of tokens in each buffer. The consistency property can be formalized with the repetition vector as in the following Equation (1.1). This equation ensures that all data produced on a buffer e can be consumed, and vice versa.

$$\forall e \in E,\ \text{prod}(e) \times \vec{r}[\text{src}(e)] = \text{cons}(e) \times \vec{r}[\text{dst}(e)] \tag{1.1}$$

The repetition vector defines a graph *iteration* in which the actor executions are called *firings*, or equivalently *jobs* in the literature. Additionally, in this thesis, we consider that graph iterations always respect the data dependencies of buffers, even those broken by delays (except for cycles). Each actor in the SDF graph has a corresponding implementation code, written in any programming language. An actor firing corresponds

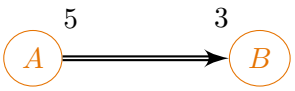
to the execution of its code.

The repetition vector can be computed by two algorithms. First algorithm [LM87c] uses a topology matrix which corresponds to a linear function giving the number of tokens present on each buffer for any repetition vector input. The repetition vector is the non null vector holding smallest Integer number of firings and belonging to the kernel of the topology matrix. Second algorithm uses a Breadth-First Search (BFS) graph traversal to compute a lowest common multiple of all production and consumption rates. The algorithm using graph traversal can be found in Section 3.1 of Bhattacharyya's book [BML12] dedicated to dataflow graphs. While both algorithms return the same repetition vector, the second one is faster since it requires less memory space and less computations. A similar version of this second algorithm is presented later in Section 1.4.3.

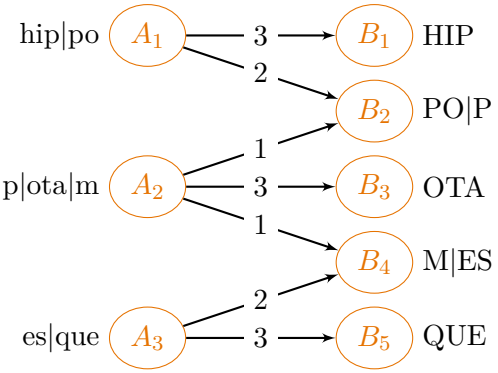
Thanks to the repetition vector, it is possible to expose data dependencies between each firing, in the Single-Rate Synchronous Data Flow (SRSDF) graph obtained from the original SDF. Such SRSDF graph is depicted in Figure 1.2b; it is sometimes referred to as Homogeneous Synchronous Data Flow (HSDF) graph in the literature. SRSDF graphs actually are a subset of SDF graphs where $\text{cons}(e) = \text{prod}(e)$ for each buffer e; consequently; each actor in an SRSDF graph is executed only once: $\vec{r} = \vec{1}$. The SRSDF version of an SDF graph always exists, but requires more space to be stored since all firings and data dependencies of the original SDF graph are exposed. To better distinguish SDF graphs from their SRSDF equivalent, we use a simpler representation as shown in Figure 1.3. In this thesis, buffers of SDF graphs are represented with double arrows $\Rightarrow$ while data dependencies of SRSDF graphs are represented with single arrows $\rightarrow$. The number of tokens of each data dependency is specified at the middle of the single arrows. In Figure 1.3, both graphs are SRSDF graphs, but we use the representation on the right to simplify the figures when the production and consumption rates of each buffer are equal, or do not need to be explicit.

Knowing all data dependencies, it is possible to schedule the firings, as shown in Figure 1.2c for one PE and in Figure 1.2e for four PEs. Additionally, Figures 1.2d and 1.2f depicts the number of tokens present at any time on the unique FIFO buffer between actors A and B. The data transmitted on the buffers, i.e. tokens, are represented on the SRSDF graph in Figure 1.2b. Actor A simply produces the French word "hippopotamesque"[19], while actor B rewrites it in uppercase. Here, one token corresponds to a letter of the word and vertical bars | respectively represent: where the tokens are

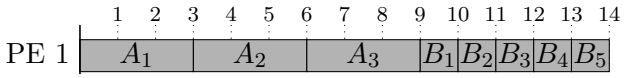
[19] "hippopotamesque" is a French adjective to describe the heaviness of anything. The funny part is that it is composed of five words of three letters, all valid at the French Scrabble. When cutting the word in three sections of five letters, it also matches with its etymological roots: *hippo*/horse of the *potam*/river and *esque* is a suffix for some French adjectives.



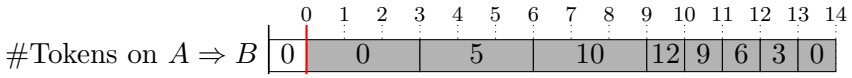
(a) SDF graph example, of repetition vector $[3, 5]^T$. Actor A produces the word "hippopotamesque" while actor B rewrites it in uppercase.



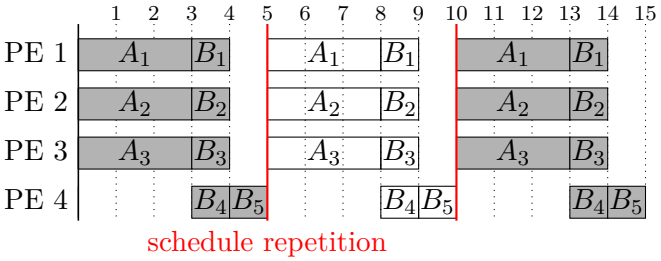
(b) SRSDF graph of 1.2a.



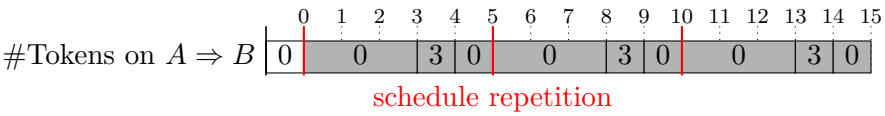
(c) Schedule example of 1.2a on one PE.



(d) FIFO buffer usage during schedule of 1.2c. Numbers in boxes are the number of tokens present at a time in the unique FIFO buffer between actors A and B.



(e) Schedule example of 1.2a on four PEs. Three repetitions of the static schedule are represented, separated by a red vertical line at time unit 5. Three complete graph iterations are represented, two with a gray background for all its firings, and the other without background color.



(f) FIFO buffer usage during schedule of 1.2e. Numbers in boxes are the number of tokens present at a time in the unique FIFO buffer between actors A and B.

Figure 1.2 – SDF graph example with its equivalent SRSDF graph and two possible static schedules with the corresponding buffer usage.

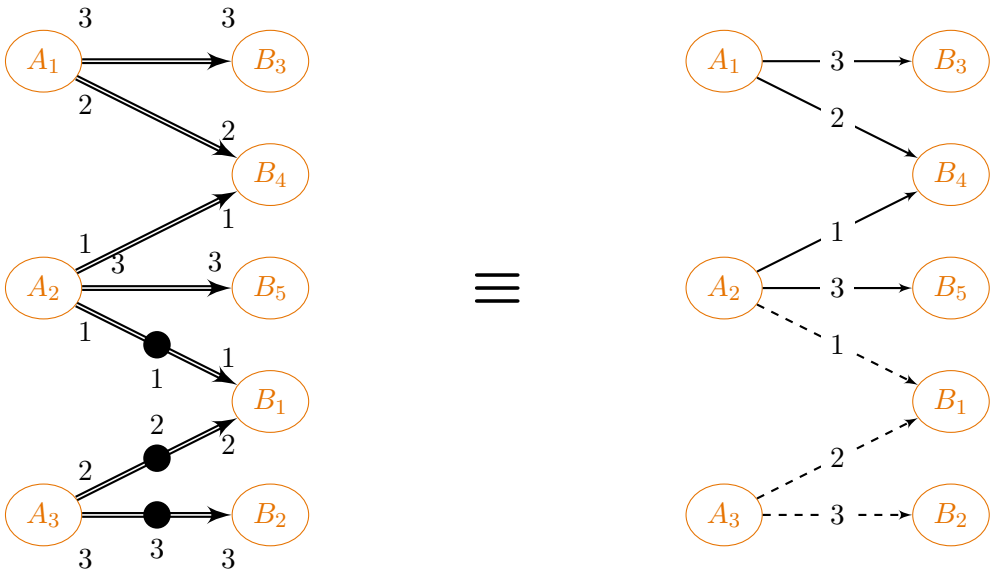
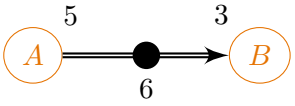


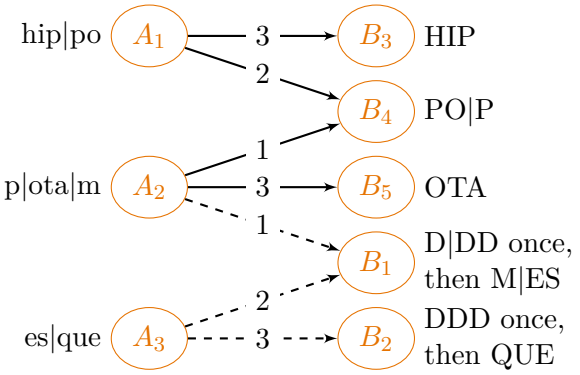
Figure 1.3 – SRSDF graph equivalent representations.

spread on the output buffers on the production side (firings of A), and where the tokens of the input buffers are joined on the consumption side (firings of B). Scheduling requires to know the ETs of each actor per kind of PE. In this thesis, ETs are not reported on the SDF graphs, but directly on the Gantt diagrams depicting their schedules.

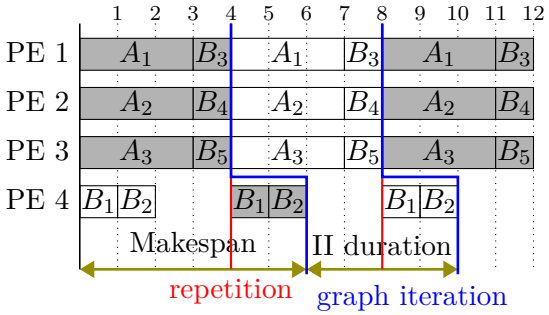
A delay on a buffer e breaks data dependencies because it enables firing $\mathrm{dst}(e)$ without having fired $\mathrm{src}(e)$. Such case is depicted in Figure 1.4. In the original publication describing SDF, delay semantics is not completely explicit and delays are manually set. However, the tokens of a delay may actually be set by the output of other actors [Arr+18]. Thanks to the delays, it is possible to increase the *throughput* of the schedule, as shown in Figure 1.4c which is repeating faster than on Figures 1.2c and 1.2e. The throughput is the average number of schedule repetitions per time unit, or equivalently the average number of graph iterations per time unit. In the case of indefinitely repeated static schedules, the throughput is defined as the inverse of the Initiation Interval (II) duration, i.e. the time between two schedule repetitions, as shown in Figure 1.4c. When there are delays, graph iterations do not match anymore with the schedule repetitions; the time until the end of the first graph iteration defines the *makespan* or *latency* of the scheduled application. A graph iteration is one end-to-end execution of the application represented by the graph. Delays on buffers outside cycles always increase the makespan, and imply that the makespan is greater than the II duration. In other words, there are more executions of the whole application per time unit, but each application execution



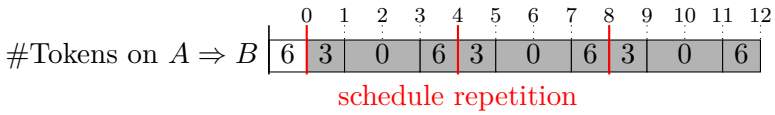
(a) SDF graph example, of repetition vector $[3, 5]^T$. Actor A produces the word "hippopotamesque" while actor B rewrites it in uppercase. A delay is present on the only buffer, breaking data dependencies of firings B_1 and B_2 on A_2 and A_3. The delay carries the six letters "dddddd" as initial data.



(b) SRSDF graph of 1.4a. Dependencies broken by the delay are represented with a dashed line.



(c) Schedule example of 1.4a on two PEs. Three repetitions of the static schedule are represented, separated by a red vertical line at time unit 4. A complete graph iteration is represented and delimited by the blue lines.



(d) FIFO buffer usage during schedule of 1.4c. Numbers in boxes are the number of tokens present at a time in the unique FIFO buffer between actors A and B.

Figure 1.4 – SDF graph example with its equivalent SRSDF graph and a possible static schedule with the corresponding buffer usage.

takes a longer time. Each different schedule repetition crossed by a single graph iteration is called a *pipeline stage*. In Figure 1.4c, there are two pipeline stages.

Finally, note that SDF graphs are a subclass of Petri nets, sometimes called Timed Weighted Event Graph (TWEG), where *places* correspond to buffers and *transitions* to actors. Petri nets are a dataflow oriented MoC, briefly presented in the previous Section 1.1.2. Thanks to their expressiveness, Petri nets have been rapidly used to model applications including stream dataflow applications, see Jennings' thesis [Jen81]

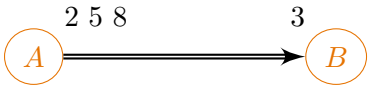
for example (page 56). However, Petri nets are difficult to analyse and may imply non deterministic selection of the dataflow path to execute when limiting parallelism with locks. Indeed, parallelism may be limited and non deterministic with Petri nets when multiple *transitions* pull tokens from the same *places*. At the opposite, SDF is completely deterministic and there is no explicit lock system. To introduce an implicit synchronization point, SDF requires to add a new actor which is fired only once and depends on all firings of actors preceding the new actor in the graph. Besides, SDF graphs are also a subclass of KPNs and Dataflow Process Networks (DPNs). SDF graphs are sometimes referred to as Multi-Rate Dataflow graphs in the literature.

1.2.2 Static SDF flavors

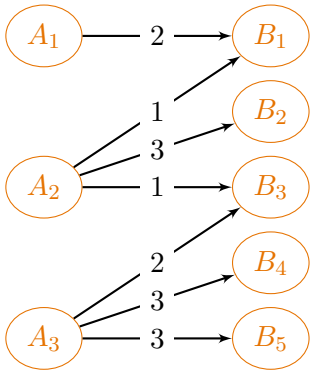
Thanks to its static communications leading to the computation of a repetition vector, and thanks to its expressiveness of data and task parallelism, SDF has been widely used and extended for dataflow stream applications. Hereafter, we briefly introduce extensions of SDF which still have static communications, i.e. fixed production and consumption rates on buffers.

Cyclo-Static Data Flow (CSDF). Cyclo-Static Data Flow (CSDF) [Bil+96] differs from SDF by the possibility to specify sequences of consecutive production or consumption rates which are indefinitely iterated over, hence *cyclic*. SDF is a subclass of CSDF, but CSDF has been developed after SDF. Figure 1.5 depicts an example of CSDF graph, its data dependencies, and a possible schedule. The production rate of actor A is "2 5 8", which means that at the end of its first firing, A produces only 2 tokens, 5 after the second firing and 8 after the last third firing. At its fourth firing, A loops over its production sequence and produces again 2 tokens. The repetition vector of CSDF graphs can be computed easily with a BFS algorithm as for SDF graphs. Instead of the production or consumption rate, it uses a rational number in $\mathbb{Q}$ being the sum of all rates in the sequence divided by the sequence length. Note that it is always possible to transform CSDF graphs into SDF graphs [PPL95], but in the worst case it requires to duplicate actors for each rate combination of its production and consumption sequences. The StreamIt language [TKA02] supports both SDF and CSDF models.

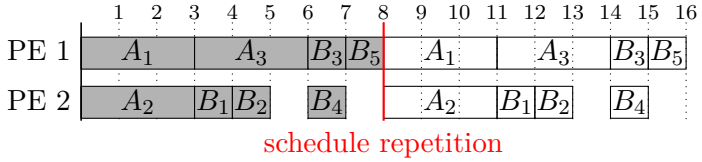
Extensions for multi-dimensional arrays Multidimensional dataflow [ML02] allows for considering exchanged data across multiple dimensions, and to have complex sampling patterns on these data. This model is especially useful for image processing. A



(a) CSDF graph example, of repetition vector $[3, 5]^T$.



(b) SRSDF graph of 1.5a.



(c) Schedule example of 1.5a on two PEs. Two repetitions of the static schedule are represented, separated by a red vertical line.

Figure 1.5 – CSDF graph example with its equivalent SRSDF graph and a possible schedule.

full review [KD13] has been made by the creator of the "Windowed" version [KHT06], which handles the case of elements in the halo of an image for example.

Extensions using polyhedral model SDF is especially useful to express data parallelism on arrays of data. However, a powerful analysis also exists to automatically find such data parallelism in nested `for` loops of regular imperative languages; this is the polyhedral analysis [Fea92]. Polyhedral Process Networks (PPN) [Ver10] combine polyhedral analysis with KPNs but only a few works focus on SDF graphs [FMG17].

Other extensions One important extension of SDF is the support of hierarchy, as with Interfaced-Based Synchronous Data Flow (IBSDF) [Pia+09]. Hierarchy eases the refinement of actors by other SDF graphs and IBSDF provides composition rules to keep consistent the whole SDF graph. SigmaC [Gou+11] is another extension, more dedicated to many-cores architecture [DLC14] compared to similar languages as StreamIt. Besides, there exist an extension of SDF with asynchronous communications between independent graphs [PPL94], and another extension modeling global states [PCH99].

1.2.3 Dynamic SDF flavors

Dynamic extension of SDF help to relax the constraint of fixed production and consumption rates on buffer. For example, multiple tools, such as Odyn [Dau+19], support the modification of the rates between each iteration of the graph. StreamDrive [SB19] also supports such inter-iteration rate modifications for KPNs, thus including SDF graphs.

With scenarii. Scenarii help to automatically trigger the modifications of the rates, for example in Scenario Aware Dataflow (SADF) [The+06], which uses Markov chains. In SADF, *detectors* detect actor scenarii and trigger accordingly the rate modifications, possibly intra-iteration. Not only rates but also actor ETs may be modified. Later, SADF has been refined with FSM [Stu+11], see [KSG15] for an implementation. Besides, Mode Aware Data Flow (MADF) [ZNS18] is inspired by SADF and more dedicated to real-time systems.

Parameterized ones. Bhattacharya introduced the Parameterized Synchronous Dataflow (PSDF) model supporting parameterized expression of the rates [BB01]. The parameters can be valuated by the output of other actors. PSDF relies on a hierarchic description of the SDF actors. Rate modifications may occur intra-iteration of the whole graph, but inter-iteration of the internal graphs. Parameterized Interfaced Meta-Model (PIMM) [Des+13] is quite similar to PSDF, with extra composition rules for hierarchy and simpler description of the parameters. Other works studied the safe intra-iteration modification of rates in non hierarchic graphs [FGP12]. A recent survey [BFG17] compares multiple other parameterized SDF models. Regarding the polyhedral model, there is a Parameterized extension of PPN (denoted P^3N) [ZNS11].

Reconfigurable topology. In a few SDF extensions, it is possible to modify the topology of the graph of the application, for example with switch actors [Buc94]. Boolean Parametric Dataflow (BPDF) [Bem15] is a parameterized model supporting the modification of the graph topology, by a specific Boolean flow enabling or disabling actors. PIMM also handles the disabling of actors by setting to 0 the rates of buffers connected to them. However, to our knowledge, only Reconfigurable Dataflow (RDF) [Fra+19] is able to perform complete inter-iteration reconfiguration of the graph, by removing or adding actors in it.

1.3 Scheduling of SDF graphs

Scheduling is a large research field, and this section presents only a few works related to this field. Scheduling may refer to the ordering or mapping of tasks, the ordering or routing of their communications, and to the allocation of the memory. Scheduling is actually applicable for every constrained resource, such as machines of a job shop for the *job shop* scheduling problem.

Scheduling problems are generally formalized either for independent tasks or for Directed Acyclic Graphs (DAGs) of dependent tasks, whereas we focus on SDF graphs. Nevertheless, in most of the cases, static SDF graphs are first unfolded to acyclic SRSDF graphs, that are DAGs if live, in order to schedule them. This unfolding is a standard procedure, but it might produce large SRSDF graphs when the SDF repetition vector specifies large number of firings; in a SRSDF graph, there are as many tasks as the sum of the SDF repetition vector. It is possible to stream the construction of data dependencies to avoid building the SRSDFs [Arr+19], and a few works mix the application model with schedule information, such as Dataflow Schedule Graph (DSG) [Wu+11] or a work using FSMs [Zeb+13]. An older work compares the impact of intermediate representations of SDF graphs on their schedule [Bam+02]. In our contribution on scheduling, in Chapter 3, we consider the standard unfolding of SDF graphs to their SRSDF equivalent.

In Section 1.3.1, we briefly present the main scheduling techniques. Tools dedicated to SDF graphs are listed in Section 1.3.2. Finally, analyses of SDF graphs, usually performed before the scheduling to compute its theoretical efficiency, are listed in Section 1.3.3.

1.3.1 Scheduling techniques

Multiple techniques can be used for scheduling. In the next subsubsections, we first list the main properties of schedulers, and then the most employed techniques to actually compute schedules.

Scheduling types

Scheduling corresponds to the ordering of tasks, and by extension, to the ordering of communications between tasks. Mapping, or partitioning, is the process of selecting the PEs on which a task will be executed. Mapping may be resolved during the scheduling process, and scheduling often refers to both mapping and scheduling. Similarly, the routing of communications, that is the selection of communication routes across a network,

may also be resolved during the scheduling process. Although less common, memory allocation may be considered during scheduling, as well as energy management or the optimization of any constrained resource.

Schedulers can be categorized thanks to some of the following properties. A scheduler:

- computes any of a task or communication *ordering*, or task mapping, a communication routing, or any allocation;
- is *online/dynamic*, ordering tasks as they are released, or *offline/static*, statically ordering tasks before their arrival;
- is *starvation-free* or non starvation-free, that is leading to not execute all given tasks (e.g. in case of deadlock);
- is *preemptive* or non-preemptive, that is enforcing tasks to execute until their completion without suspension;
- is *time-triggered* (e.g. by task periods) or *event-triggered* (e.g. self-timed data-driven);
- is global to multiple PEs, or *partitioned*, that is performed independently for each PE (only if mapping is already computed);
- supports *dependencies*, such as data communications between tasks, or not;
- supports dynamic *job migration* from one PE to another (only if online), or not;
- supports *batch processing*, that is to schedule independent tasks by groups according to their deadlines, or not;
- targets *unicore*, *homogeneous* or *heterogeneous* multi-processors;
- takes into account energy, processor temperature or frequency, etc.

Scheduler of real-time systems consider a few more constraints on tasks. Real-time systems generally consists of periodic tasks [LL73] whose executions have to be scheduled in a given periodic time window, every 10 millisecond for example. The deadlines of tasks are *implicit* when equal to their period. If a task is not completed before its *hard* deadline, the system is considered *non schedulable*. Periodic tasks are especially used for *safety critical systems* where determinism and verification is important [BS93]. Davis' survey [DB11] lists main modern scheduling types for real-time systems. Fixed Priority (FP) and Earliest Deadline First (EDF) are two common schedulers of periodic tasks;

they are online schedulers respectively based on a static priority assignment of tasks and on dynamic priorities depending on the deadlines. Some real-time models relax the deadlines hard constraints, such as weakly-hard constraints [QE12]. Some others, such as PolyGraph [Dub+19], release the periodic constraint on some tasks. Priority assignment and periods of tasks are set by the system designer, or computed offline by a scheduling synthesizer.

Timing properties of the tasks, such as latency from one task to another, may be verified after the scheduling, but some schedulers can take them into account during the scheduling process [Mai+18]. We present a few other metrics in the next subsubsection.

Scheduling metrics

One of the most important metric is the *processor utilization* factor U, derived from the first schedulability test [Hor74] of tasks executed with preemption and a global deadline on an homogeneous CPU multi-processor. In this case, U corresponds to the sum of the task ETs, divided by the global deadline. To ensure a schedulable system, U must be lower than m the number of PEs . U may also be computed separately on each PE, and then serves as a metric measuring *load-balancing* [Car08].

The *latency/makespan* of a DAG of tasks is defined as the elapsed time from the minimum start of the roots of the DAG to the maximum completion time of the leaves of the DAG, considering tasks executed in the graph topological order, that is a *graph iteration*. For example, in the static schedule of Figure 1.4c, the latency is 10 time units, from the start time of A_1 to the completion time of B_2 in the second repetition of the schedule. Indeed, delays create pipelining and the SDF graph is executed, in topological order, across two schedule repetitions. Note that SDF graphs may contain cycles, but we assume that enough delays are present to break the data dependencies, and thus their corresponding SRSDF graph is a DAG.

In this thesis, we focus on static schedules which are repeated indefinitely. Roots of the DAG of tasks generally correspond to input reads from a stream. Then, one new element of the stream is read at each repetition of the schedule. In the thesis, each repetition of the schedule is also called a *scheduler iteration*, which is different from a *graph iteration*. When there is pipelining, a graph iteration can cross multiple scheduler iterations, as in Figure 1.4c. The duration of a scheduler iteration is called II, equal to 8 time units in Figure 1.4c. In our context, the throughput of a graph is the inverse of the II duration, and it depends on the schedule.

Solving techniques

Scheduling has been proved NP-complete when minimizing the II duration , but some specific cases may be polynomial [KA99]. Heuristics are generally used, such as *list scheduling* which is based on a priority list of all tasks to execute[20]. List scheduling is not always optimal, and may even have counter-intuitive behavior in specific cases, such as increasing the II duration when increasing the number of PEs [Gra69]. List scheduling can be used offline, and also online, especially by runtime schedulers such as StarPU [Aug+11]. PREESM tool uses offline the FAST list scheduler [KAG96].

To compute optimal schedules, Integer Linear Programming (ILP) is used for a few decades [LM69], but at the cost of solving only small systems due to the large amount of computations that ILP requires. The benefit of using ILP is to specify an objective function, for example minimizing the II duration. Constraint Programming (CP) can also be used, for example with the OscaR[21] tool [DDP18]. CP provides more expressiveness for constraints but it requires a huge computational time when exploring the whole space of solutions. Local search is an alternative to explore only a subset of the whole solution space, see [Pra17] for an example using OscaR. Evolutionary Algorithms (EA) is another alternative to only explore a subset of solutions [ECP06]. Scheduling can also be solved with Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) techniques, as originally in [KS92].

In any case, it is always possible to specify multiple objectives for the schedule, but as they might be contradictory, there is not always *one* optimal solution; instead there exists a *Pareto front* of solutions. Each solution of a Pareto front is optimal according to at least one of the objectives, but not all. For multi-objective scheduling problems, multiple heuristics exist, such as the maximum diversity approach [MB08]. Some results of similar problems show that Linear Programming (LP)[22] may scale efficiently [MR14]. EA has also been used to solve multi-objective scheduling [KC02]. Exploring multiple solutions of the same scheduling problem is also referred to as Design Space Exploration (DSE) [Pim17].

Unfolding. To optimize the schedule of a stream application running forever, *unfolding* may be applied on the DAG of tasks. In this case, unfolding corresponds to consider multiple graph iterations at once, which is a kind of task replication, in order to use all the PEs more efficiently and so to increase the throughput. This technique has

[20] Fixed Priority (FP) and Earliest Deadline First (EDF) scheduling types rely on list scheduling.
[21] https://www.info.ucl.ac.be/~pschaus/oscar.html
[22] Linear Programming (LP) uses float numbers. It is less difficult to solve than ILP but is not optimal.

been employed for the execution of task graphs on DSPs [PM91]. It is especially useful when the task graph exposes less parallelism than the number of available PEs. Spasić [Spa17] also proposes an unfolding algorithm with task replication in order to execute SDF graphs with periodic actors while maximizing throughput or minimizing energy consumption.

Clustering. The opposite of unfolding is *clustering*, which is useful when the task graph exposes more parallelism than the number of available PEs. Clustering merges some tasks together. Bhattacharyya proposed multiple algorithms to perform clustering, as APGAN [BML97]. Clustering is not only useful to reduce the amount of parallelism, but also to reduce the size of the binary code generated by the compiler according to the schedule [Bha+95].

Pipelining. Software pipelining [Lam04; All+95] is another technique to improve the throughput of stream dataflow applications. Pipelining corresponds to the start of a new graph iteration while the previous one is not yet completed. This objective is similar to unfolding. For a DAG which is indefinitely executed with different input data, scheduling with pipelining is sometimes referred to as *pipelined workflow scheduling* [Ben+13b]. Pipelining has been widely studied for SDF [LM87a]; it corresponds to add delays on some buffers of the graph. When there are data dependencies, it is also possible to use *retiming* techniques [LS91] to create pipelines. Retiming corresponds to move delays from some buffers to others. Retiming and pipelining can be computed before calling the scheduler or at the same time if offline.

1.3.2 Tools and benchmarks

Multiple algorithms and tools are able to schedule SDF graphs or their SRSDF equivalent, for example onto regular CPUs [GR05], Other works target many-cores architectures, such as Kalray [Has+17], with the help of OpenMP. The latter work adds clustering and hierarchy to the work of Ha [KLH07]. The Diplomat framework [Bod+16] statically analyses code to transform it in the SDF model and to execute it on CPUs and GPUs with the help of OpenCL. OpenCL is a language and library dedicated to GPUs, which can also be executed on CPUs. Design Space Exploration (DSE) is supported by some works, such as [Sch+19] for an extension of SDF with dynamic actors. Multi-objective scheduling of SDF graphs has been considered for latency and throughput, thanks to ILP and heuristics [LGE12]. There exists an offline scheduler using SMT which tries to reduce memory and communication *contention* [Ska+18], and tightens the

schedule with the help of online measurements. Contention is the ET overhead due to the sharing of common resources such as cache memory or communication network.

All the aforementioned works use a large variety of techniques, and target a large variety of architectures. However, they focus on the scheduling problem and the code generation implementing the computed schedule. In the next subsubsection, we list some important works which combine scheduling for the hardware side, and modeling of the software side. For example, the StreamIt programming language [TKA02] supports the SDF model and is linked to a dedicated compiler performing scheduling. StreamIt also comes with an SDF benchmark. Tools to perform benchmarks of SDF applications are presented in the second subsubsection.

For development: modeling and code generation

Numerous work tackle the design of dataflow or real-time applications (some works, such as UML MARTE[23] are listed in [RAK15]) or tackle the compilation of such applications (some works, such as Silexica[24], are listed in [Leu+19]). However, we focus in this thesis on tools able to support both modeling and compilation or scheduling on given target architectures. Such tools are sometimes referred to as *co-design* tools since both application and architecture are modeled and refined during the design process. A survey on co-design tools for real-time systems [Trö+06] lists some of the most important tool, such as SynDEx [GLS99; GS03] and Ptolemy [Eke+03]. To our knowledge, none of this tool supports automatic configuration of the QoS of the designed application.

An industrial co-design tool example is Scade[25], which relies on the Lustre synchronous language[26]. Matlab/SimuLink tool is also widely used in the industry, see [Pag+14] for an example of a data-flow application modeled with SimuLink and executed on a many-cores processor from Tilera. LabView too can be used to model SDF graphs and execute them on FPGA processors [And+12a].

On the academic side, there is the Gaspard [Gam+11] co-design tool inspired by ArrayOL, relying on UML MARTE model and inter-operable with synchronous languages such as Signal or Lustre. PeaCE [Ha+08] is another co-design tool supporting SDF and FSMs. Other tools, such as MAPS [CLA13] support more generic models such as KPNs but are not specifically optimized for SDF graphs. TTool[27] and more specifically its

[23] https://www.omg.org/omgmarte/
[24] https://www.silexica.com/
[25] https://www.ansys.com/en/products/embedded-software/ansys-scade-suite
[26] See the paragraph on synchronous languages in Section 1.1.2.
[27] https://ttool.telecom-paristech.fr/

Diplodocus part [Apv+06] supports the UML SysML[28] model, which is an alternative to UML MARTE.

Other academic tools such as PREESM [Pel+14] and its runtime version Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPIDER) [Heu+14], specifically target SDF graphs, or more precisely the Parameterized Interfaced Synchronous Data Flow (PISDF) extension. It is possible to convert some UML MARTE specifications to the PISDF model [Amm+14]. Orcc [Yvi+13] is an ancestor of PREESM. CAPH [SB14] and OpenDF [Bha+08] support SDF graphs and target FPGA processors.

Synthesis of real-time properties of tasks. One objective of the design process is to avoid non schedulable systems. Algorithms exist to test the schedulability of SDF graphs with periodic actors and without preemption [Ben+12]. However, rather than ensuring schedulability or giving latency or buffer bounds [God97], it is even better to compute directly during the design step some properties such as priority assignment for FP scheduling [KM16]. Darts [BS11] and ADFG [Bou13; Hon+17] are more complete scheduling synthesizers since they even compute the periods of the SDF actors. One particularity of ADFG is to schedule directly the SDF graphs without expanding them to SRSDF. Darts and ADFG cannot be considered as co-design tools since they are tools targeting only homogeneous architectures and SDF applications without any modeling facility as a Graphical User Interface (GUI). Nevertheless, they take both application SDF graph and number of available homogeneous PEs as an input, and could be used as underlying scheduling component of any larger co-design tool. This has been demonstrated for ADFG to verify AADL dataflow specifications [Gau+19].

For benchmarks: use-cases, validation and generation of sample graphs

Many dataflow applications have been modeled with SDF graphs, such as a video decoder and encoder [OH02] or multiple telecommunication applications [Dar+16; Moo+08; PAN08]. A reinforcement learning application [HW07] has also been implemented in PISDF, as well as various image processing applications[29]. The StreamIt [TA10] benchmark contains other SDF and CSDF dataflow applications related to signal processing. There exists a refactored version of StreamIT for real-time system analysis [RP17][30]. Another benchmark [JY17] generates random C code corresponding to actors; it is based on the SDF^3 [SGB06b] SDF graphs generator and analyzer.

[28] https://www.uml-sysml.org/sysml/
[29] See PISDF implementations on: https://github.com/preesm/preesm-apps
[30] https://gitlab.inria.fr/brouxel/STR2RTS

Turbine [Bod+14][31] is a generator of SDF and CSDF graphs which can also ensure their liveness. Turbine exports SDF graphs in the file format of SDF^3. Such files can be converted to the file format of PISDF by a simple script. Task Graph For Free (TGFF) [DRW98] and GGen [Cor+10] focus on the generation of DAGs of tasks, a subclass of SDF graphs not exposing data parallelism.

Finally, a few tools focus on the evaluation of the schedules. For example, Cheddar [Sin+04] supports the SDF model and can simulate their periodic execution with the FP scheduling type to track events such as preemption.

1.3.3 Analysis for scheduling efficiency

As the scheduling process may be long and result in a non schedulable system, multiple algorithms have been developed to compute feasible bounds of metrics of the schedules. These metrics mainly are: latency, throughput, and the memory size required by the buffers. Bounds on these metrics are useful to asses the quality of the scheduling algorithms. Such bounds are usually computed before the scheduling process, and independently to it. If not specified in the next paragraphs, bounds are computed without resource constraints on the system.

Liveness. Liveness ensures that all tasks may be executed even if the SDF graph contains cycles. Delays are added to some buffers of cycles in order to break data dependencies, and the goal is generally to minimize the amount of delays needed for liveness [CR93]. There exist polynomial sufficient conditions for the delay sizes in live SDF graphs [MM08; MM09], and in live CSDF graphs [Ben+13a]. Liveness has also been studied in the case of self-timed execution [Gha+06a].

Throughput. Throughput may be limited by delays on cycles: the minimal delay sizes ensuring liveness may reduce the expressed data parallelism. At the opposite, delays placed on buffers outside cycles create pipelining and increase the throughput. The Maximum Cycle Mean metric [SB09] helps to compute the maximum throughput in such cases. Other optimal throughput analyses [Gro+12; Gha+06b] are based on max-plus algebra [Bac+92; Kom+18]. It is possible to estimate the throughput of large IBSDF hierarchical graphs [Der+17b; Der+17a]. Besides, the throughput of SDF graphs can be estimated while taking into account processor constraints [GAM19a].

[31] https://github.com/bbodin/turbine

Latency. Latency is also dependent on the delays of the SDF graph: delays generally increase the latency because one graph iteration is executed across multiple scheduler iterations. Latency has been evaluated for SDF graphs with periodic actors without preemption [Kha+16; GAM19b], and also with resource constraints on a many-cores processor [MB07]. A few works rely on symbolic execution to compute the minimum achievable latency, possibly under throughput and processor constraints [Gha+07].

Memory usage. It is possible to schedule SDF graphs such that buffer sizes are minimized [ČP93], for which throughput may not be optimal. At the opposite, it is possible to schedule SDF graphs such that the buffer size is minimized while ensuring the maximum throughput. In between, some works compute close to optimal buffer sizes while respecting a throughput constraint [WBS07]. The ADFG tool is also able to minimize the buffer sizes [BFG16b; BFG16a] for FP and EDF scheduling types. While all the aforementioned works considered that each buffer has its own memory, some other works consider buffers shared among multiple actors to have memory reuse [Den+07] and reduce the total memory size. Moreover, memory reuse can be optimized for SDF graphs once their static schedule is known [Des+16a; Des+16b]. The latter technique uses a graph of memory exclusion of the buffer accesses made by each actor in order to reuse the memory of some buffers by other buffers when their accesses never overlap in time during a scheduler iteration.

Trade-offs. As minimization of latency and maximization of throughput may be two contradictory objectives, trade-offs have been studied for the general case [SGB08] or for improved analysis with partial scheduling information [Dam+12]. The SDF^3 tool [SGB06b] proposes multiple analyses, including trade-offs between buffer sizes and throughput [SGB06a] which are also two contradictory objectives when considering one dedicated memory per buffer. Same trade-offs between throughput and buffer size have been studied for SDF graphs with periodic actors [Ben+10] and CSDF graphs with periodic actors [BMD13]. Other works handle constrained processors [Les17], for an extension of CSDF.

1.4 The PREESM tool

The PREESM [Pel+14] tool helps to design SDF applications. PREESM is developped at INSA Rennes as an open-source project[32]. It provides a GUI to ease the modeling,

[32] https://preesm.github.io

and multiple kinds of analyses are also implemented. Most importantly, PREESM has a synthesis part to generate automatically parallelized C code of the modeled applications. The models and files used by PREESM are described in Section 1.4.1. Then we focus on the definition of application parameters in Section 1.4.2, and on the computation of the repetition vector in Section 1.4.3. Finally, special kinds of actors are introduced in Section 1.4.4; they are especially useful to generate the SRSDF graph of an application.

1.4.1 Architecture of a PREESM project

The dataflow model supported by PREESM is called PISDF; it is the application of the Parameterized Interfaced Meta-Model [Des+13] to the SDF MoC. Regarding the MoA, PREESM uses S-LAM [Pel+09a]. As PREESM is an Eclipse[33] based platform, the files related to an application and its targeted architectures are grouped in a *project* containing multiple folders. The `Algo` folder contains the files representing the PISDF graphs of the application, while `Archi` contains the files representing the S-LAM architectures. Other configuration files are located in the folders `Code`, `Scenarios` and `Workflows`. A description of each project folder is given hereafter.

`Algo` project folder

This folder contains all the PISDF graphs of a project. The graphs are stored in an XML format with a `.pi` file extension. Moreover, all `.pi` files can be converted to, and automatically updated from, a GUI with corresponding `.diagram` files. The GUI supports the live editing of all properties of a PISDF graph: actors, buffers, production and consumption rates, parameters, etc... The GUI comes with an automatic layout algorithm to clarify the representation.

Since the PISDF MoC supports the hierarchical description of a graph, there might be multiple `.pi` files to represent a single PISDF graph. Indeed, an actor of the top level graph might be refined in another hierarchical graph and related `.pi` files. There is no limit to the number of subgraphs in the hierarchy. In this thesis only flat graphs, i.e. without hierarchy, are considered ; this is not a strong assumption because all hierarchical graphs can be automatically converted to their flat equivalent. When an actor is not refined by another subgraph, it has to be refined with the prototype of the function (coded in C) to execute. Eventually, all leaf actors of the whole graph are refined with a C function.

[33] https://www.eclipse.org/

`Archi` project folder

This folder contains all the S-LAM architectures of a project. The architectures are stored in an XML format with a `.slam` file extension. Moreover, all `.slam` files can be converted to, and automatically updated from, a GUI with corresponding `.layout` files. The GUI supports the live editing of all properties of a S-LAM architecture, especially PEs and data connections between them, with bandwidth specifications. Multiple `.slam` files can be present in the same `Archi` project folder.

In this thesis, only homogeneous CPU architectures are considered and we use only a subset of the S-LAM model. For example, the Intel i7-7820HQ @ 2.90GHz processor having 4 physical cores is modeled with 4 PEs each connected, thanks to an *undirected-DataLink*, to a single *parallelComNode* imitating the accesses to a shared RAM. The bandwidth of *parallelComNode* is arbitrarily set to 10^9 data unit per time unit, while we generally consider byte as data unit and nanosecond as time unit.

`Code` project folder

This folder contains all the C files of a project. Especially it contains the header `.h` files defining the function prototypes used as actor refinements. Multiple actors may be refined with the same prototype.

It is a common practice to store the implementations of the C functions in the `Code` project folder too. Another common practice is to create a `generated` subfolder as the target of the PREESM code generation. Thus all C files are closely located and it is easier to manage and compile them. The PREESM code generation creates one main `.c` file, plus one per PE. The purpose of the main file is to launch as many threads as PEs, each executing the instructions of its corresponding file. The generated files defines all the buffers and their static addresses on the memory stack.

A few other C files may be required, for example to wrap the communication means on a specific DSP (provided by the designer), or to instrument the code with check sums on the buffers content (provided by PREESM). It is also possible to instrument the code with timing measurements of every actor, which are stored in a custom `.csv` file during the execution.

`Scenarios` project folder

This folder contains all the *scenario* files of a project. The scenarios are stored in an XML format with a `.scenario` file extension. Multiple `.scenario` files can be present in the same `Scenarios` project folder. Each scenario file contains application information

which are specific to a given architecture, as the actor ETs. Thus they refer to exactly one top level PISDF graph and one S-LAM architecture. Moreover, a GUI supports the live editing of all the stored properties of a scenario, as well as the import of the ETs from `.csv` files resulting from the timing measurements done when instrumenting the code.

Among other properties of the scenario, they also store mapping constraints, i.e. whether or not an actor can be executed on a specific PE of the architecture. Values of PISDF parameters can be override in the scenario, and the size of buffer data types must be specified in the scenario. Finally, it is possible to specify the energy required by each actor firing and the power consumption of the PEs.

`Workflows` project folder

This folder contains all the *workflows* of a project. The workflows are stored in an XML format with a `.workflow` file extension. Multiple `.workflow` files can be present in the same `Workflows` project folder. Each workflow file defines the sequence of transformations to apply on one scenario and especially the PISDF graph it refers to. Eventually, the transformations lead to the scheduling of the application and to the code generation. Moreover, all `.workflow` files can be converted to, and automatically updated from, a GUI with corresponding `.layout` files. The GUI supports the live editing of all properties of a workflow, such as the add of a new transformation, for example the one to flatten a hierarchical PISDF graph.

Each transformation is called a *workflow task*. All the contributions of this thesis have been implemented as workflow tasks, whose names are given at the end of each chapter. Note that the workflow tasks form a kind of dataflow graph: a given root task outputs the scenario, the architecture and the top level PISDF graph, which can be the input of any other workflow task. Multiple intermediate representations are used between the workflows tasks, especially for the output of the scheduling workflow task. The obtained schedule can be displayed in a GUI thanks to another workflow task. Last but not least, workflows can be called in a script to avoid the Eclipse GUI, thus easing the automation of transformations on multiple projects.

1.4.2 Definition of PISDF graphs and parameters

When working in PREESM, the main interest of a designer is to develop PISDF graphs of the designed application. The easiest way is to use the dedicated GUI, which supports the live editing of a graph and its *parameters*, displayed in the same diagram and stored

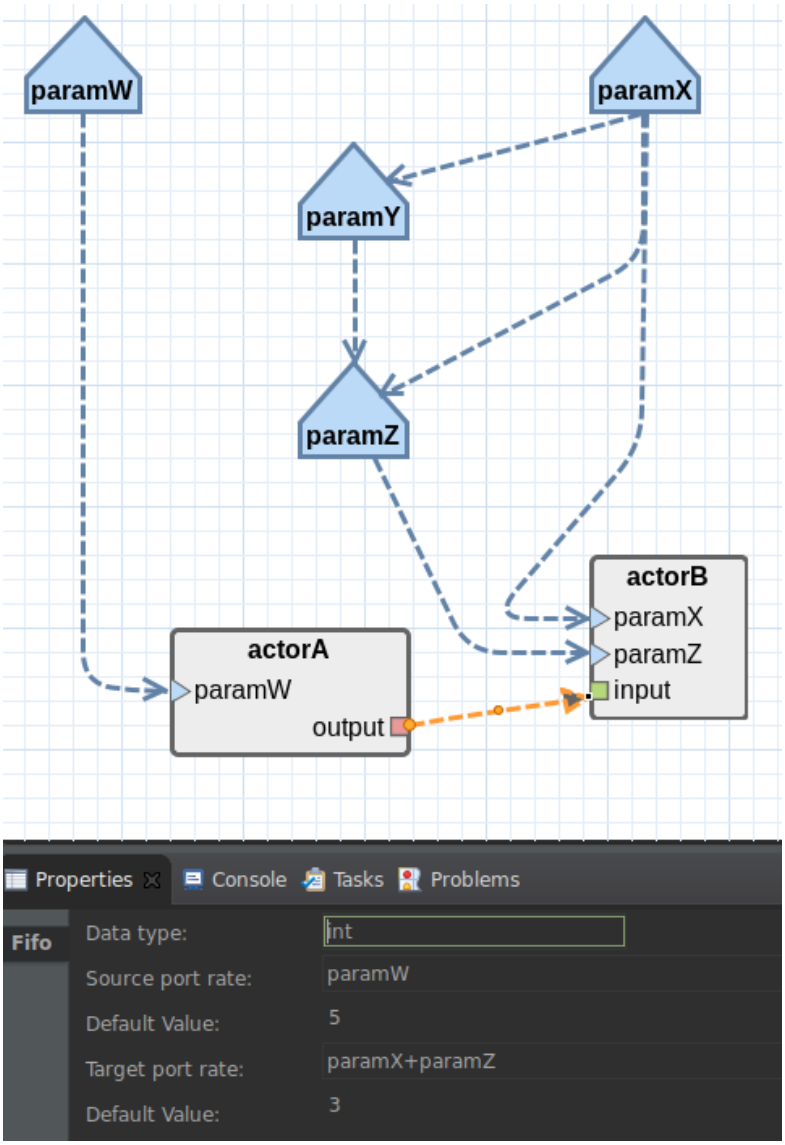


Figure 1.6 – Example of PISDF graph in PREESM. This example represents the same graph as in Figure 1.2a, but with parameterized production and consumption rates.

in the same `.pi` file. Parameters are arithmetic expressions which can be used in the definition of a few PISDF properties such as the production and consumption rates, and also used in the definition of a few scenario properties such as the actor ETs.

In the GUI, parameters are represented with a different shape and a different color than actors, as shown in Figure 1.6. Each parameter has a unique name, here suffixed by `param` to be recognized more easily in the figure. As parameters are named, their arithmetic expressions can depend on each other, and parameters eventually form a DAG whose roots (here `paramW` and `paramX`) can only hold constant expressions (there should be no cyclic dependency). A parameter can depend on multiple parameters, and

can be used by multiple other parameters or actors. Dependencies must be explicitly set by the designer. If an actor is dependent on a parameter, the parameter can be used in the production or production rates of buffers connected to this actor, and also as an argument of the C function refining this actor. The other arguments of the C function of an actor are the memory pointers to its input and output buffers, if any. In Figure 1.6, there is a single buffer named `output` for the producer `actorA` and `input` for the consumer `actorB`. The PISDF model also supports the use of an actor output in a parameter definition; then the parameter becomes *dynamic*, and it is not possible anymore to statically schedule the application. In this thesis we only consider *static* parameters, which are not depending on any actor output.

Concretely, the parameters especially ease the definition and the modification of production and consumption rates. In Figure 1.6, they are defined by the following set of equations in Listing 1.1, all related to the unique buffer denoted e for convenience. In the GUI, only the right hand sides of those equations are set into the text fields of the corresponding parameters or of the rates. For example, the arithmetic expressions of production and consumption rates are given here with the notations used in this thesis, respectively prod(e) and src(e), but in the GUI they are set directly in the `Source port rate` and `Target port rate` text fields of the currently selected buffer.

```
paramW = 5              paramX = 1
                        paramY = paramX*ceil(pi)
                        paramZ = paramX*ln(paramY)/ln(2)
prod(e) = paramW        src(e) = paramX+paramZ
```

Listing 1.1 – PISDF parameter definitions and use in Figure 1.6.

As you may notice, the definition of parameters is equivalent to a program in the SSA form, but without control flow structures. However, conditional if statements are supported inside the arithmetic expressions (more details are given in Section 5.1). Finally, since parameters are defined in an SSA form without cyclic control flow statements (they form a DAG), when all parameters are static they can be rapidly valuated, even while editing the graph in the GUI. Starting with the root parameters, the valuation process recursively propagates their values to the parameters depending on them. The valuation of a parameter is restricted to Integer numbers, it automatically truncates the floating point results of arithmetic expressions. The valuation is visible in the `Properties` tab of parameters and buffers, in the text field called `Default Value`.

1.4.3 Computation of the repetition vector

Once the parameters of a PISDF graph have been valuated, all the actor production and consumption rates are fixed and it is possible to compute the repetition vector, which respects Equation (1.1). To do so, a mere graph traversal can be used [BML12]. In the PREESM tool, this graph traversal is a Depth-First Search (DFS) taking into account cases which are specific to the PIMM model, especially hierarchy and rates equal to 0. As this thesis tackles only flat graphs, i.e. without hierarchy, the case of hierarchy is not detailed in the presented algorithm. To simplify, the case of hierarchy requires a recursive top-down approach, computing the repetition vector $\vec{r}_{\text{Parent}}$ of the parent graph first, and then multiplying the repetition vector $\vec{r}_{\text{Child}}$ of each child graph Child by its upper level repetition factor, i.e. by the scalar $\vec{r}_{\text{Parent}}$ [Child][34].

In PREESM, the computation of the repetition vector can be represented as two successive steps. The first step discovers all the largest *weakly connected components* of the flat PISDF graph, as detailed in Algorithm 1.1. A weakly connected component of a directed graph is a set of nodes such that an undirected path always exists between two nodes of the set[35]. As the definition involves undirected paths, the algorithm explores both the incoming $IE(\beta)$ and outgoing $OE(\beta)$ buffers of each actor β. During this first step, buffers where both the production and consumption rates are equal to 0 are ignored, see lines 20-22 and 27-29. Thus, if an actor has rates of all its buffers equal to 0, it is not executed at all, see lines 33-34. The visiting order of the actors during the first step is important and is reused for the second step. Indeed the second step actually sets a *firing ratio* between the currently visited actor and all its direct neighbors according to the DFS. A firing ratio is the local multiplicity of firings of an actor compared to one of its direct neighbors. All the firing ratios are finally used to compute the repetition vector, see lines 17-19 of Algorithm 1.2.

Note that Algorithms 1.1 and 1.2 set the repetition vector even for graphs having multiple connected components, which corresponds to the case of multiple independent applications described in the same graph. In such case, the algorithm automatically sets the repetition vector such that all connected components have the same throughput. For example, in the simplest case of a graph containing two unconnected actors without buffer, the repetition vector is $\vec{1}$: both actors are fired once.

However, the designer might need an unbalanced behavior, for example, with one

[34] In the parent graph, any child graph appears as an actor. The simplification concerns the rates of PISDF data interfaces in the child graphs: extra logic is automatically added to ensure that the repetition vector of any child graph is not modifying the repetition vector of its parent graph.

[35] At the opposite, cycles are *strongly* connected components of a directed graph.

Algorithm 1.1: Modified DFS before computing the repetition vector $\vec{r}$

```
1  function initializeCCs(V, r⃗) ▷ The DFS initializes all connected components (CC).
2      forall α ∈ V do
3          r⃗[α] ← 1;                                  ▷ By default each actor is executed once.
4      visitedActors ← ∅;
5      setOfCCs ← ∅;
6      forall α ∈ V do                    ▷ Main loop to discover all connected components (CC).
7          if α ∈ visitedActors then
8              continue ;
9          visitedActorsInCC ← ∅;
10         visitedBuffersInCC ← ∅;
11         toVisit ← ∅;
12         addFirst(toVisit, α);
13         while toVisit ≠ ∅ do              ▷ Traverse a single connected component (CC).
14             β ← head(toVisit);
15             remove(toVisit, β);
16             addLast(visitedActors, β);
17             addLast(visitedActorsInCC, β);
18             nbZeroBuffer ← 0;
19             forall e ∈ IE(β) do                          ▷ Iterate over the successors.
20                 if cons(e) = 0 and prod(e) = 0 then
21                     nbZeroBuffer ← nbZeroBuffer + 1;
22                     continue ;
23                 addLast(visitedBuffersInCC, e);
24                 if src(e) ∉ toVisit and src(e) ∉ visitedActorsInCC then
25                     addFirst(toVisit, src(e));
26             forall e ∈ OE(β) do                        ▷ Iterate over the predecessors.
27                 if cons(e) = 0 and prod(e) = 0 then
28                     nbZeroBuffer ← nbZeroBuffer + 1;
29                     continue ;
30                 addLast(visitedBuffersInCC, e);
31                 if dst(e) ∉ toVisit and dst(e) ∉ visitedActorsInCC then
32                     addFirst(toVisit, dst(e));
33             if nbZeroBuffer = #IE(β) + #OE(β) and nbZeroBuffer > 0 then
34                 r⃗[β] = 0; ▷ Not executed since all input and output rates are null.
35         cc ← new CC(visitedActorsInCC, visitedBuffersInCC);
36         addLast(setOfCCs, cc);
37     return setOfCCs;                    ▷ The discovered connected components (CC).
```

Algorithm 1.2: Computation of the repetition vector $\vec{r}$ based on the result of Algorithm 1.1

```
1  procedure computeRV(setOfCCs, r⃗)
2      forall cc ∈ setOfCCs do                ▷ Iterate over all connected components (CC).
3          forall α ∈ CC.visitedActorsInCC do      ▷ Initialize the firing ratios.
4              firingRatio(α) ← 0/1 = 0; ▷ The firing ratio is a fractional number. It
                 is relative to the direct predecessor in the DFS traversal.
5          forall e ∈ CC.visitedBuffersInCC do          ▷ Set the firing ratios.
6              ratioSrc ← firingRatio(src(e));
7              ratioDst ← firingRatio(dst(e));
8              if numerator(ratioSrc) = 0 and prod(e) > 0 and
                 numerator(ratioDst) > 0 then
9                  firingRatio(src(e)) ← ratioDst × cons(e)/prod(e);      ▷ Fractional ×.
10                 ratioSrc ← firingRatio(src(e));
11             if numerator(ratioDst) = 0 and cons(e) > 0 and
                 numerator(ratioSrc) > 0 then
12                 firingRatio(dst(e)) ← ratioSrc × prod(e)/cons(e);      ▷ Fractional ×.
13         multiple ← 1;
14         forall α ∈ CC.visitedActorsInCC do ▷ Compute the lowest common multiple
             (lcm) of all ratios in the connected component (CC).
15             ratio ← firingRatio(α);
16             multiple ← lcm{multiple, denominator(ratio)};
17         forall α ∈ CC.visitedActorsInCC do      ▷ Set the repetition vector r⃗.
18             ratio ← firingRatio(α);
19             r⃗[α] ← numerator(ratio) × multiple/denominator(ratio);      ▷ Is an Integer.
```

connected component executed twice more than the others. Such unbalanced repetition vector can also be useful to guarantee fairness in processor utilization of each connected component of the original graph. This unbalanced behavior can be enforced by making the whole graph *weakly connected*, i.e. having a single weakly connected component, thanks to a dummy actor connected with dummy buffers to every original actor having no incoming (or equivalently, outgoing) edge[36]. The dummy buffers have the smallest possible rate 1 on both sides, except for the one going to the twice more executed connected component, which takes the value 2 as a production rate.

1.4.4 Single-Rate graph and special actors

Once the repetition vector has been computed, it is possible to generate the SRSDF version of the original PISDF graph, as in Figure 1.4b. However, the graph in Figure 1.4b cannot be used in this form for the scheduling (because of delay management) and the code generation (because of varying number of inputs and outputs). Thus, a few special kinds of actors are used in PISDF in order to solve these problems. Figure 1.7 depicts the actual SRSDF graph[37]generated by PREESM instead of the one depicted in Figure 1.4b. The graph in Figure 1.7 contains 7 special actors, whose behavior is detailed in the next paragraphs. As the behavior of those special actors is defined by PREESM, their refinements in a C function do not have to be written by the designer. Moreover, some memory optimizations [Des+16b] are automatically applied to the special actors.

Init/End actors. These special actors ensure that any live SRSDF is indeed a DAG in PREESM. To simplify, we have stated earlier that all SRSDF graphs are DAGs however this is true only for live SDF graphs where data dependencies coming from and going to delays are replace by init and end actors, respectively. These actors are added automatically by PREESM during the SRSDF graph generation; the user does not have to manage them. An init actor only has an output buffer; it provides the data which

[36]The first method [LM87c] computing the repetition vector uses a topology matrix Γ instead of the graph traversal. Note that this original method is only for connected graphs and its main result is that a connected SDF graph is consistent if and only if $\dim(\ker(\Gamma)) = 1$, with the repetition vector $\vec{r}$ being the basis of this kernel. We conjecture a generalization of this equivalence: an unconnected SDF graph is consistent if and only if $\dim(\ker(\Gamma)) = \#CC$, with $\#CC$ being the number of its weakly connected components. If so, there exists a basis of $\ker(\Gamma)$, each basis vector being the repetition vector of one connected component, filled with zeros on actors not belonging to this connected component. The same relation $\dim(\ker(L)) = \#CC$ is already proved for the Laplacian matrix L of any undirected graph.

[37]Note that this automatically generated SRSDF graph has been modified to meet the indexing of firing used in the thesis: PREESM normally starts at 0 while we start at 1 in the thesis. Moreover, the generated names of the fork and join actor firings originally contains `implode` and `explode` instead of `fork` and `join` here.

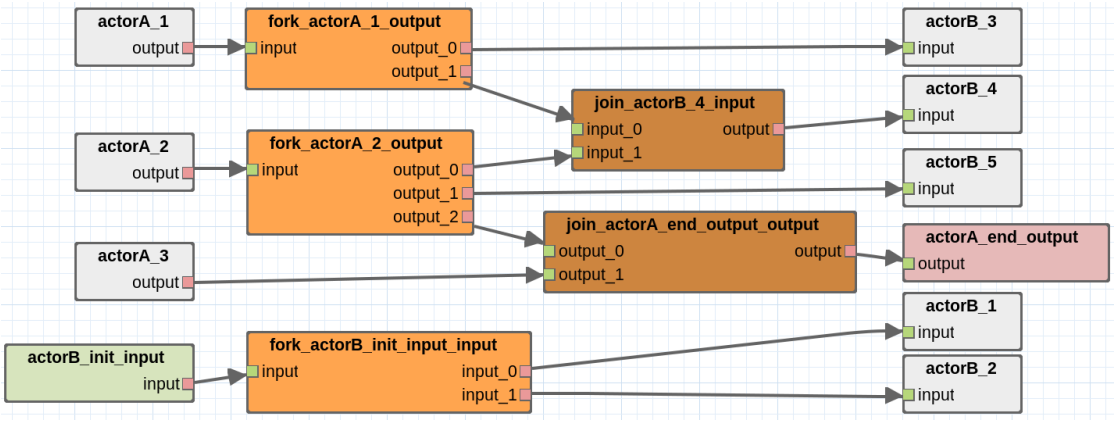


Figure 1.7 – Example of generated SRSDF graph in PREESM. This example represents the same graph as in Figure 1.4b, but with automatically added fork and join actors. Init and end actors respectively retrieves data from and stores data to the delay.

have been stored into its corresponding delay during the previous scheduler iterations. An end actor only has an input buffer; it stores the data into its corresponding delay during the current scheduler iteration.

Fork/Join actors. These special actors behave as scatter and gather operations, respectively. A fork actor scatters the data of its unique incoming buffer into all its outgoing buffers (in order of appearance, from top to bottom). A join actor gathers data of its incoming buffers (in order of appearance, from top to bottom) into its unique outgoing buffer. The sum of their incoming rates is equal to the sum of their outgoing rates. The fork and join actors are especially useful to ensure that all C functions of actor refinements always take the same number of arguments. For example, in Figure 1.4b, firings of A have two (for A_2) or three (for A_1 and A_3) output buffers whereas the original actor A has only one output in the SDF graph in Figure 1.4a. The use of fork actors solves this problem and all firings of A have only one output buffer in Figure 1.7. Thus, the same C function is used for all firings of A, with the original prototype `actorA(int * output)` if no PISDF parameters are used inside the C function, or `actorA(int paramW, int * output)` if considering the parameter defined in the PISDF graph in Figure 1.6. Fork and join actors are automatically generated by PREESM during the conversion into an SRSDF graph, but the designer can also add them manually before the conversion.

Broadcast/Roundbuffer actors. These special actors behave as duplicate and decimate operations. A broadcast actor copy the data of its unique incoming buffer into all its outgoing buffers, multiple times if the production rates are a multiple of the only input consumption rate. A roundbuffer actor retains only the last data of its incoming buffers (in order of appearance, from bottom to top) until reaching the production rate of its unique outgoing buffer. In the case of hierarchical graphs, broadcast and roundbuffer actors are automatically generated by PREESM during the conversion into a flat PISDF graph, in order to ensure that subgraphs do not modify the repetition vector of their parent graphs. The designer can also add broadcast and roundbuffer actors manually, and they are used in two of our contributions, see Figures 2.5 and 5.2.

1.5 Conclusion

Parallelism and architectures can be modeled with multiple means. In this section, we precise which models are used in this thesis and we detail the common notations.

1.5.1 Model used in this thesis

The dataflow model used in this thesis is PISDF, that is the application of PIMM to SDF. However, we do not use all properties of PISDF, in particular we do not consider dynamic configuration of the parameters nor hierarchical graphs. We focus on the static part of the PISDF model because it explicitly exposes both data and task parallelism, and is simple enough to perform extensive analyses on it. As PISDF supports any expression of parameters for production and consumption rates, delay sizes, or even ETs, it makes PISDF suitable for the automatic configuration of applications modeled with it.

To perform analyses on the PISDF model, we use the PREESM co-design tool, which also integrates the S-LAM architecture model. Our target architectures are modeled with S-LAM. PREESM offers a code generation feature, which implements synchronization barriers between each scheduler iteration. This feature corresponds to the scheduling Assumption 1 described in Section 3.2.2. To generate the code, PREESM creates one thread per PE. Each thread executes its tasks in a data-driven fashion.

1.5.2 Main notations

Main notations used in this thesis are detailed in Table 1.1. They are adapted from standard Burns' notations [Dav13] for real-time systems. Variables written in lower case

are used in formulae or for indexation. Variables written in upper case are kept for instance names in examples, such as actor names.

Symbol	Meaning
$G = (V, E)$	The considered SDF graph, with its set of actors V and buffers E.
α	Any actor in V.
π	Any periodic actor in V.
e	Any buffer in E.
$IE(\alpha)$	Set of incoming buffers of actor α.
$OE(\alpha)$	Set of outgoing buffers of actor α.
$d_0(e) \in \mathbb{N}$	Size of the delay on e.
$\text{src}(e) \in V$	Source actor of the buffer e.
$\text{prod}(e) \in \mathbb{N}$	Production rate of $\text{src}(e)$.
$\text{dst}(e) \in V$	Destination actor of the buffer e.
$\text{cons}(e) \in \mathbb{N}$	Consumption rate of $\text{dst}(e)$.
$\vec{r}$	Repetition vector of G, sorted in the lexicographic order of its actors.
$\vec{r}[\alpha]$	Number of firings of actor α as specified in the repetition vector.
α_j	j-th firing of actor α within one scheduler iteration.
α^i	firing of actor α within the i-th graph iteration.
$G^* = (V^*, E^*)$	The DAG being the SRSDF/HSDF version of G.
τ	Any task in G^*.
$C_\alpha \in \mathbb{N}$	Execution Time (ET) of actor α (more specifically, Worst Case Execution Time (WCET) in Chapter 3).
$T_\pi \in \mathbb{N}$	Period of periodic actor π.
$m \in \mathbb{N}$	Number of Processing Elements (PEs).

Table 1.1 – Main notations used in this thesis.

Chapter 2

Modeling nested `for` loops, with SDF graphs

Contents

2.1 SIFT keypoints detection application 45
2.2 Modeling of single loops having explicit parallelism 47
2.3 Modeling of nested loops having explicit parallelism 49
2.3.1 Iteration space splitting 50
2.3.2 SDF iterators 51
2.4 When and how to use SDF iterators? 52
2.4.1 When are needed SDF iterators? 53
2.4.2 How the processing code must be modified? 54
2.5 Evaluation 55
2.6 Related work 57
2.6.1 On specialized dataflow languages 57
2.6.2 On the clustering of dataflow graphs 58
2.7 Conclusion 59

Introduction

Signal processing applications are generally compute intensive and constrained in terms of throughput and latency. For example, the throughput of video displays is constrained in Frame Per Second (fps). Parallelization of such applications is the key to meet their throughput and latency requirements: when possible, data are processed simultaneously by different Processing Elements (PEs).

Parallelization of `for` loops can be achieved automatically in the code through OpenMP, or manually using threads. However, it is not possible to handle all the cases with OpenMP, as distributed memory. Moreover threads require to manually add synchronizations and communications in the code. Thus, applications are usually *modeled* in order to first expose their parallelism, and secondly analyze this available parallelism and synthesize efficient schedules. Here we use the SDF [LM87b] MoC which expresses parallelism in two ways: by the different paths in the graphs (task parallelism), and by the possible executions of the same process on different chunks of data (data parallelism).

One can model single `for` loops with SDF graphs as long as loops can be divided in sub-parts accessing chunks of data of equal size, to respect the SDF restriction of fixed amount of data communication. Yet, there is no general technique to model multiple nested `for` loops with SDF graphs, especially when bounds of the inner loops are varying.

The contribution of this chapter is the modeling with SDF graphs, of multiple perfectly nested loops having explicit parallelism and variable bounds in their inner loops. In loops having explicit parallelism, all iterations are independent. Perfectly nested loops perform computation only in the innermost loop. This contribution is motivated by two facts. First, variable amounts of data can be modeled with the CSDF [Bil+96] MoC, an extension of SDF; but previous experiments on modeling using the CSDF MoC have shown that this model is not easy to understand for designers and does not always offer a competitive benefit. CSDF modeling difficulty has been stated by the creators of the SDF-based language StreamIt [TKA02], in a review of their own work [TA10] (see section 5.2). Another option is to use dynamic dataflow MoCs such as Kahn Process Network [Kah74], but KPNs are hard to analyse and are not statically schedulable. Hence, we focus on SDF graphs instead. Second, we need to model nested loops with SDF graphs in order to finely control the granularity of the application representation. Moreover, the representation should be easily adaptable to the target architecture, especially to its number of PEs, while staying independent from the architecture.

A direct application of this contribution is the modeling of a computer vision feature detection application. Indeed, keypoints detection is performed on images at different

resolutions and different blur levels. Thus, nested loops iterate over images of different sizes so the loops have variable bounds.

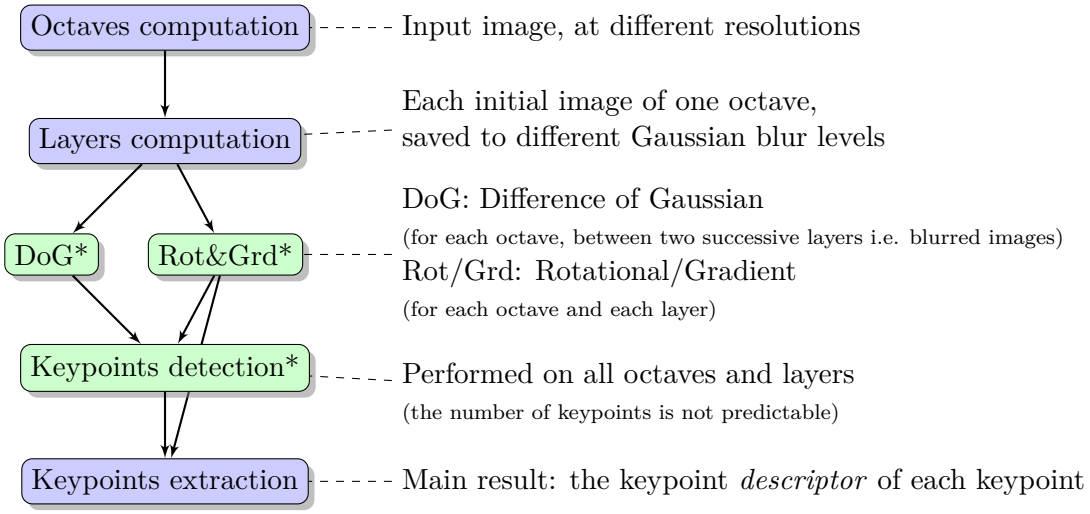
In this chapter, we introduce the notion of SDF *iterators* to model and optimize multiple nested loops having variable bounds. Iterators are demonstrated on a Scale Invariant Feature Transform (SIFT) keypoints detection [Low04] application, modeled with an SDF graph. Iterators help modeling and parallelizing SIFT detection, although some nested loops process images of variable sizes. At the same time, iterators help reducing the scheduling complexity since it is possible to adapt the number of parallel executions with regard to the number of PEs.

The chapter is organized as follows. The motivational SIFT application that serves as an example throughout the chapter is presented in Section 2.1. Then the parallelization of single loops with SDF graphs is recalled in Section 2.2. The main contribution, SDF iterators for multiple perfectly nested loops having explicit parallelism and variable bounds, is detailed in Section 2.3. The practical usage of such SDF iterators is detailed in Section 2.4. An evaluation of iterators in SDF graphs is presented in Section 2.5. Related work, in Section 2.6, is followed by a conclusion.

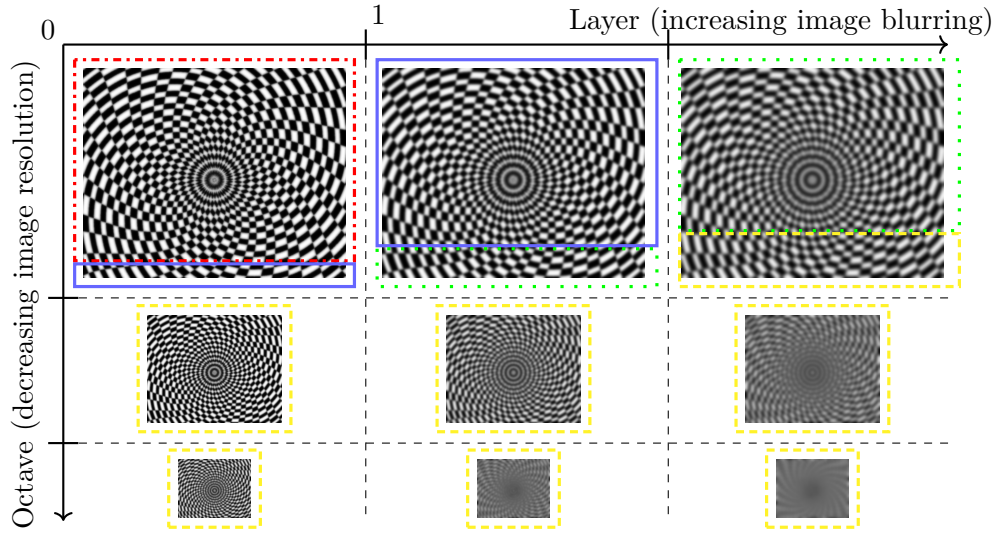
2.1 SIFT keypoints detection application

This section presents an overview of the SIFT application that is later modeled with SDF graphs. Scale Invariant Feature Transform (SIFT) computes keypoints by comparing points in the original image with the same points in blurred images obtained from the original one, and at different resolutions. Figure 2.1a details the main steps of SIFT: first the original image is upscaled once, and downscaled multiple times to build the images at various resolutions. Each resolution is called an *octave*. Then, the image at each octave is blurred multiple times. A blur level corresponds to a *layer*. All images are stored in a 4-dimensional (4-D) array; the dimensions are, in order: octave, layer, height, width. Difference of Gaussians (DoG), gradient, and rotational metrics are computed from this 4-D array. Each metric computation produces an array of the same size, except the DoG which produces one less layer. At last, keypoints detection is performed on these three 4-D arrays. Then, the extraction step refines the computed keypoints.

Two main problems arise when modeling SIFT detection with an SDF graph. First, the number of keypoints to detect is unknown since it depends on the image content. Second, the images to process stored in the 4-D array have different sizes depending on their octave, whereas the SDF MoC imposes data transfers of fixed size. The first problem is fixed by setting a limit on the number of detected keypoints. For the second



(a) SIFT workflow: green steps* are modeled with iterators in an SDF graph.



(b) Layers and octaves in SIFT with four different regions of equal processing amount. Original image is on top left. Blurring on abscissa and resolution downscale on ordinate.

Figure 2.1 – SIFT image processing application: main steps (Figure 2.1a) and data storage (Figure 2.1b).

problem, the naive way to model different octaves is to create a specific actor for each image resolution, which is not convenient because the model cannot be adapted to different numbers of octaves. Another difficulty is that the computation on the smallest image resolution, the last octave, is faster than the computation of the first octave by multiple orders of magnitude. Indeed, for an image of 640×800 pixel, the image is

upscaled once and downscaled five times by a factor 2 on each dimension; thus the ratio of the number of pixels in the first octave over the last is $4^6 = 4096$. If multiple PEs are available, an important question is how to parallelize the computation equally among them. Figure 2.1b illustrates this problem with three layers, three octaves, and four available PEs. One option is to assign each layer to a PE, but then a PE is not used. The opposite option is to assign each octave to a PE, but then a PE is not used, and computations are unbalanced. An example of equal distribution of the computation on the four PEs is shown in the boxes of the four colors red, blue, green and yellow (each with a specific dotted, dashed or straight line pattern). Each color encloses one quarter of the computation. On that example, boxes clearly do not match the image bounds.

The iterators introduced in Section 2.3 can handle this computation partitioning while staying in the SDF model. Iterators are used to model and parallelize the green steps in the workflow in 2.1a, in our case according to the number of available PEs. Iterators do not require duplicating any actor for each octave. Before describing the iterators in details, modeling and parallelization of single loops with SDF graphs is recalled in Section 2.2.

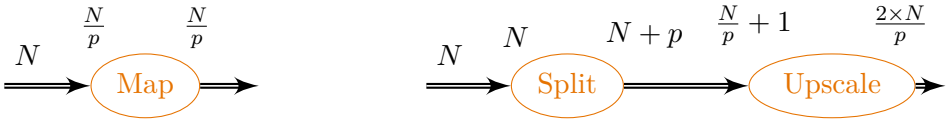
2.2 Modeling of single loops having explicit parallelism

This section discusses the modeling of single `for` loops with SDF graphs. `for` loops are a basic control structure of any imperative language. The code in Listing 2.1 illustrates a simple `for` loop. It iterates over an `input` array, processes each element and stores the result in an `output` array; both arrays having the same size `N`, it represents a map operation. Here the parallelism is *explicit*: there is no dependency between the statements of any iteration of the loop, and `process` is a pure function without side effects.

```
1  for (int i = 0; i < N; ++i) {
2    output[i] = process(input[i]);
3  }
```

Listing 2.1 – Simple one dimensional (1-D) `for` loop having explicit parallelism.

Figure 2.2a depicts the modeling with an SDF graph of a map operation with a controllable degree of parallelism. p is the degree of expressed data parallelism: the Map actor is executed p times, on chunks of data of size $\frac{N}{p}$, where p must be a divisor of N. If $p = N$, all data parallelism is expressed, however, it is not always useful to express



(a) Map in SDF, of repetition vector $[p]^T$.

(b) 1-D Upscale in SDF, of repetition vector $[1, p]^T$.

Figure 2.2 – Map and Upscale at a coarse-grain level.

all the parallelism, especially if the number of PEs is largely smaller than N. The code of the `Map` actor is almost the same as in Listing 2.1; the only difference lies in the loop index bound that is now `N/p` instead of `N`.

In image processing, a common operation is to perform an upscale, increasing the resolution of the image with interpolation. This operation is more generic than map since the output array does not have the same size as the input, and since multiple elements of the input array are accessed simultaneously to perform the interpolation. A code example is shown in Listing 2.2 for the 1-D case.

```
1  for (int i = 0; i < N-1; ++i) {
2    output[2*i] = input[i];
3    output[2*i+1] = interpolation(input[i], input[i+1]);
4  }
5  output[2*N-2] = input[N-1];
6  output[2*N-1] = input[N-1];
```

Listing 2.2 – Simple 1-D upscale, by interpolation on the element and its successor.

An upscale operation is similar to a map, but requires extra data to apply the interpolation on the borders of the chunks of the original array. The last element of a chunk is a copy of the first element of the next chunk. These extra data can be added by a copy actor preceding the upscale actor. The SDF modeling of an upscale operation is depicted in Figure 2.2b, where the interpolation actor is called Upscale, and the copy actor is called Split. Split is executed once while Upscale is executed p times. The code of the Split actor, in Listing 2.3, pre-processes the data to add an extra element to each chunk then processed by Upscale. The code of the Upscale actor, in Listing 2.4, is simpler and faster than the original one, in Listing 2.2, since the border case needs no more to be handled thanks to the copies performed by Split.

The upscale modeling pattern presented in Figure 2.2b is used to model the computation of the upscale of the input image in the first step of SIFT, as shown in Figure 2.1a.

```
1    for (int i = 0; i < p; ++i) {
2      // copy the whole chunk, can be optimized with memcpy
3      for (int j = 0; j < N/p; ++j) {
4            output[i*(N/p + 1) + j] = input[i*(N/p) + j];
5      }
6      // adds extra copy of last element of each chunk
7      output[i*(N/p + 1) + N/p] = input[i*(N/p) + N/P - 1];
8  }
```

Listing 2.3 – Spit SDF actor code.

```
1  for (int i = 0; i < N/p; ++i) {
2    output[2*i] = input[i];
3    output[2*i+1] = interpolation(input[i], input[i+1]);
4  }
```

Listing 2.4 – Upscale SDF actor code.

An image has two dimensions but the data parallelism is expressed only on the height of the image, divided by the number of PEs. The same pattern is also used for the second step of SIFT: the layers computation. However, the algorithm to compute the different layers consists of two successive 1-D Gaussian blurs on lines of the image, each blur performing a transposition. The Gaussian blur applies a 1-D stencil with two neighbors. As data parallelism is expressed through the height of the image in any case, data must be reordered between the two transpositions; this is creating an application bottleneck since this reordering is fully sequential. We now generalize the SDF modeling patterns seen in this section for single `for` loops to perfectly nested loops having explicit parallelism.

2.3 Modeling of nested loops having explicit parallelism

In this section, perfectly nested loops having explicit parallelism are considered. An example is given in Listing 2.5, with three perfectly nested loops. *Perfectly* nested loops do not contain any statement between the declaration of the loops: only the innermost loop contains statements, line 4 of Section 2.3.2. *Explicit parallelism* means that in the assignment statement on line 4, the indexes are the same on the right hand side and the left hand side: the iterations of the loops can be executed in any order without modifying the result. However, the index bounds of the inner loops may depend on

the outer loop indexes, as abstracted by the functions `f1` and `f2`, which can be any mathematical function. The functions giving the index bounds depend solely on their outer loop indexes, `i` for `f1` and `i,j` for `f2`; in our work, those functions cannot depend on the processed data. The parallelization of nested loops of the same form than in Listing 2.5 is described in Section 2.3.1, and their modeling with SDF graphs thanks to iterators is discussed in Section 2.3.2.

```
1  for (int i = 0; i < N1; ++i) {
2    for (int j = 0; j < f1(i); ++j) {
3      for (int k = 0; k < f2(i, j); ++k) {
4        output[i][j][k] = process(input[i][j][k]);
5      }
6    }
7  }
```

Listing 2.5 – Three perfectly nested `for` loops having explicit parallelism.

2.3.1 Iteration space splitting

An important property of the SDF MoC is that the rates of data exchanges are fixed. Thus, the only solution to model loops as in Listing 2.5 with SDF graphs is to split the whole iteration space into chunks of equal sizes. These chunks do not always match the loop bounds as depicted in Figure 2.1b. In Listing 2.5, the whole iteration space size $\mathcal{S}_{it}$ is $\sum_{i=0}^{N1}\left(\sum_{j=0}^{f1(i)} f2(i,j)\right)$. In this example, the iteration space size equals the total size of the array to process, it is a map operation. Following the structure of Listing 2.5, it is possible to define $\mathcal{S}_{it}$ for any number of d nested loops, as formalized in Equation (2.1). In Listing 2.5, there are $d = 3$ nested loops.

$$\mathcal{S}_{it} = \sum_{i_1=0}^{N1}\left(\sum_{i_2=0}^{f_1(i_1)}\left(\sum_{i_3=0}^{f_2(i_1,i_2)}\left(\dots\sum_{i_{d-1}=0}^{f_{d-2}(i_1,i_2,\dots,i_{d-2})} f_{d-1}(i_1,i_2,\dots,i_{d-1})\right)\right)\right) \tag{2.1}$$

The most straightforward way to cut the whole iteration space into chunks of equal size is to simulate the execution of the loops. A variable `iter` storing the number of performed iterations is incremented instead of calling the process function. Each time `iter` reaches a multiple of `chunk_size`, the loop indexes are recorded to be used as start/stop indexes when splitting the loops. This algorithm is written in Listing 2.6, with `chunk_size` being equal to any divisor of $\mathcal{S}_{it}$. The role of an SDF iterator is to

send the recorded indexes to split the real execution of the loops in the processing actor.

```
1  int iter = 0;
2  for (int i = 0; i < N1; ++i) {
3    for (int j = 0; j < f1(i); ++j) {
4      for (int k = 0; k < f2(i, j); ++k) {
5        if (iter++ % chunk_size == 0) {
6          record(i,j,k);
7        }
8      }
9    }
10 }
```

Listing 2.6 – Iteration space simulator for three perfectly nested `for` loops having explicit parallelism. The recorded indexes will be stored in the SDF iterator.

Note that this simulation can be done offline: the start/stop indexes only need to be saved in order to be used during the real execution of the loop (where the process is performed). Besides, this simulation can be easily adapted to any number d of nested loops: the structure is the same as the original nested loops.

2.3.2 SDF iterators

An SDF iterator actor outputs the start and stop indexes for each execution of the process actor modeling the nested loops. Thus, if the nested loops iteration space $\mathcal{S}_{it}$ is divided into p chunks, the iterator is executed once and the processing actor is executed p times. The code for the processing actor is similar to its original version in Listing 2.5, the only difference concerns the indexes that are set by the iterator output. The modeling of the perfectly nested loops with SDF graphs is depicted in Figure 2.3. $\mathcal{S}_{it}$ elements are sent to the Process actor, which is processing them by chunks of size $\frac{\mathcal{S}_{it}}{p}$. For each execution of Process, the iterator produces $2 \times d$ indexes: one start index and one stop for each loop of the d nested loops. These indexes `i,j,k,...` are the one recorded during the loop simulation presented in Listing 2.6, considering that the stop indexes of one execution of the process are the start indexes of its next execution. Hence, the p executions of Process can be performed in parallel. As the p chunks may not match with the dimension bounds, as for images in Figure 2.1b, the value of each recorded index may be lesser than the index bound of its corresponding loop; Section 2.4.2 details how to modify the processing actor code accordingly. Note that the map and upscale patterns described in Figure 2.2b can also be applied to this general case of nested loops.

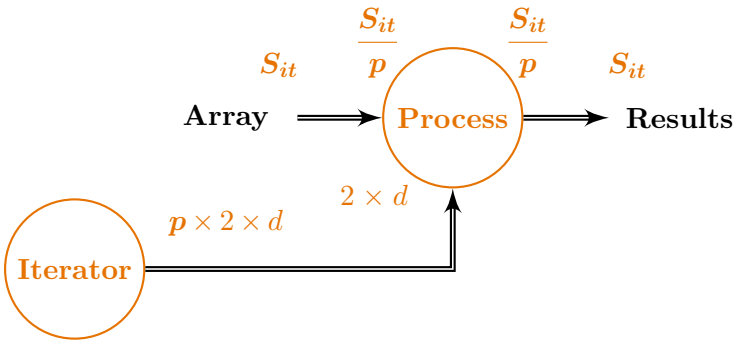


Figure 2.3 – Modeling of iterators with SDF. Repetition vector of Iterator and Process actors is: $[1, p]^T$.

Finally, iterators empower the designer to control data parallelism in SDF graphs for any perfectly nested loops having explicit parallelism. Iterators imply two drawbacks which can be overcome. The first drawback is the restriction to nested loops having explicit parallelism. However, this restriction can be removed in some cases, but not all, thanks to prior source-to-source code transformations, such as loop-skewing. The second drawback occurs when tagging an actor α, parallelized with an iterator, with a measured ET C_α. Indeed, the ET is then not only dependent on the actor code, but also on the chunk size. This drawback is easily overcome using symbolic expressions of the ET, as formulated in Equation (2.2).

$$C_\alpha(p) = \frac{C_\alpha(1)}{p} \tag{2.2}$$

So when actor α is parallelized over p chunks of equal size, the ET of each execution of a is equal to its sequential ET divided by the number of chunks. Equation (2.2) formalizes an ideal case of data parallelism and does not respect the Amdahl's law [Amd67], but more realistic symbolic expressions can be used instead. In the heterogeneous case, note that $C_\alpha(1)$ depends on the type of PE: while the amount of computations is equally distributed on each firing, their ET on different types of PEs may differ.

2.4 When and how to use SDF iterators?

This section details when and how to use the SDF iterators presented in Section 2.3.

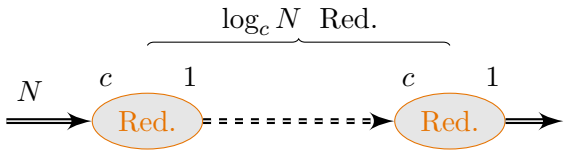


Figure 2.4 – Reduce of N elements in SDF, with chunk size $c > 1$. Repetition vector of Reduce actors is: $[\frac{N}{c}, \frac{N}{c^2}, \frac{N}{c^3}, ..., 1]^T$.

2.4.1 When are needed SDF iterators?

SDF iterators are required only when the loop indexes are used in the processing part. For example, loop indexes enable knowing the image bounds in the 4-D arrays of the SIFT application, and thus the processing is able to call specific code on borders of the image. The indexes are needed in SIFT also to access the whole array; Listing 2.8 provides a code example. SDF iterators are restricted to perfectly nested loops having explicit parallelism. Moreover, the bounds of the loops cannot depend on the processed data. If loops are not perfectly nested, it is still possible to use SDF iterators; however, in this case the computation times of the actor firings may be unbalanced.

On the contrary, SDF iterators are not required for standard map/reduce operations, even on multidimensional data, since map/reduce operations do not need to be aware of the current position in the input array. Indeed, in this case, only the size of the iteration space is needed. However, iterators may still be used, for example in order to handle padding data if the size of the iteration space is not a multiple of the number of processors. The map operation in the SDF MoC is depicted in Figure 2.2a, while the reduce operation is depicted in Figure 2.4. The reduce operation requires to create $\log_c N$ reduce actors in the corresponding SDF graph in order to control the degree of parallelism; each reduce actor consumes $\frac{N}{c}$ elements and produces 1. The repetition vector of these reduce actors is (in the same order as in the graph): $[\frac{N}{c}, \frac{N}{c^2}, \frac{N}{c^3}, ..., 1]^T$, where the total number of elements N is a power of the chunk size c. One drawback to this reduce operation modeling is to manually fill the SDF graph with the correct number of successive reduce actors. This drawback advocates for higher order languages to represent SDF graphs, as HoCL[1] developed by Jocelyn Sérot and inter-operable with PREESM.

[1] https://github.com/jserot/hocl

2.4.2 How the processing code must be modified?

Let's consider the code in Listing 2.7, that has to be modeled with an SDF graph and parallelized. This piece of code contains two nested non affine loops, iterating over a 2-D array stored in a row-major fashion (outer loop dimension first). The array contains 14 elements in total and will be split in two chunks, to be processed in parallel on two PEs.

```
1  float* array = float[1+2*2+3*3]; // = 14
2  // fill array with file data
3  ...
4  // 2-D non affine processing loop
5  for (int i = 0; i < 3; i ++) {
6    int bound = (i+1)*(i+1);
7    for (int j = 0; j < bound; j ++) {
8      processing1cell(array, i, j);
9    }
10 }
```

Listing 2.7 – Non affine 2-D `for` loop having explicit parallelism.

When adding the iterator to the SDF graph, the code of the actor to parallelize has to be adapted to its new inputs: the start and stop indexes of the loops. The modified code is presented in Listing 2.8. In the inner loops, the bounds cannot be used directly: they are used only if the upper loop is also using its start (respectively stop) index.

```
1  float* array = ; //provided as actor input
2  for (int i = start_i; i < stop_i; i ++) {
3    int begin_j = (i == start_i) ? start_j : 0;
4    int end_j = (i+1 == stop_i) ? stop_j : (i+1)*(i+1);
5    for (int j = begin_j; j < end_j; j ++) {
6      processing1cell(array, i, j);
7    }
8  }
```

Listing 2.8 – Non affine 2-D `for` loop SDF actor code.

In Listing 2.8, the size of the input array is not specified: only an address pointer is given. If the full array is split in two chunks, the SDF processing actor is fired twice by the SDF application execution framework, PREESM in our case. PREESM will feed each chunk with the correct data thanks to pointer arithmetic. The order of firings

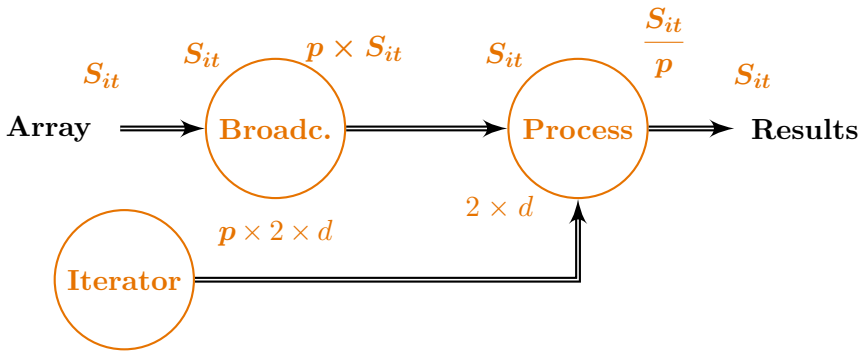


Figure 2.5 – Modeling of iterators with SDF, with broadcast actor. Repetition vector of Broadcast, Iterator and Process actors is: $[p, 1, p]^T$.

follows the order of the data: the second firing gets the second chunk (elements 7 to 13).

It is also possible to have the full array as an input of each firing, without extra memory consumption. This situation occurs in SIFT where the detection and extraction of the keypoints depends on pixels at different octaves, and thus depends on data spread everywhere in the 4-D array. To do so, a so-called `Broadcast` actor is added to the SDF graph, and provides multiple *virtual* copies of the 4-D input array: there are as many copies as there are firings of the processing actor. These copies are virtual since they refer to the same physical memory, by using the memory scripts [Des+16b] introduced by Karol Desnos et al. A graph extract corresponding to this solution is depicted in Figure 2.5. In this solution, the `Process` actor is still fired p times, and still processes only $\frac{S_{it}}{p}$ elements of the array, however it has access to the whole array.

2.5 Evaluation

SDF iterators are now used to model the SIFT keypoints detection application. In SIFT detection, images from different resolutions and blur levels are processed, which implies to iterate over a 4-D array with four corresponding perfectly nested loops. There is one loop per dimension of the array as written in Listing 2.9. The top loop iterates over the octaves and the second loop iterates over the layers. The two innermost loops iterating over the height and width of images have exponential bounds depending on the top loop index. With the octaves indexed by the variable i, the image height (respectively, width) to be processed is the biggest resolution height (resp. width) divided by 2^i. In Listing 2.9, the biggest image resolution is defined by the constants `image_height` and

image_width ; these constants are multiples of all possible values taken by 2^i and so a bit shift >> is used for the division.

```
1  float* array = ; // 4-D array to iterate
2  for (int i = 0; i < nb_octaves; i ++) {
3    for (int j = 0; j < nb_layers; j ++) {
4      for (int k = 0; k < (image_height >> i); k ++) {
5        for (int l = 0; l < (image_width >> i); l ++) {
6          processing1cell(array, i, j, k, l);
7        }
8      }
9    }
10 }
```

Listing 2.9 – Original non affine 4-D for loop in SIFT.

SDF iterators are used to model such loops, expressing a degree of parallelism $p \in \{1, 2, 4, 5, 10, 20\}$; this set contains the common divisors of the sizes of all the 4-D arrays. The SIFT detection code is a slightly modified version of the ezSIFT[2] implementation. The SDF model of SIFT is built with the PREESM [Pel+14] framework, which also performs static scheduling and generates the static parallelized code.

Another parallel version of SIFT has been implemented using OpenMP, with parallel for pragma above the loops iterating over the height of the images, which are the third inner loops as in Listing 2.9. With OpenMP, the keypoint detection step requires a critical section to add the detected keypoints into a shared list.

The execution times of the PREESM version and the OpenMP version of SIFT have been reported in Table 2.1. All experiments used an Intel Xeon E5-2650 v4 @ 2.20GHz processor (12 physical cores) and the GCC compiler version 5.4.0 (option -O2) on a single node of a cluster operated by Ubuntu 16.04 and managed by slurm. Both PREESM and OpenMP execution times are similar, the best speedup (in bold) is achieved alternatively by PREESM and OpenMP.

The number of scheduled tasks when unfolding the SDF graph of SIFT is also reported in Table 2.1. This number is the sum of the repetition vector, i.e. the sum of the minimal number of executions of each actor. As PREESM supports parameterized SDF graphs [Des+13], first introduced in [BB01], the expressed degree of parallelism p is set according to the number of targeted cores. The number of tasks is not multiplied by a

[2]See code on: https://github.com/robertwgh/ezSIFT

#Cores	#Tasks	PREESM Time (Speedup)	OpenMP Time (Speedup)
1	190	669 (0.96x)	645 (**ref.**)
2	293	412 (1.56x)	406 (**1.59x**)
4	355	277 (**2.33x**)	281 (2.29x)
5	386	263 (2.45x)	255 (**2.53x**)
10	541	171 (**3.77x**)	182 (3.54x)

Table 2.1 – Number of scheduled tasks, execution times in ms, and speedup for different number of cores. Execution time is an average on 200 runs.

factor equal to the number of cores since not all steps of SIFT are parallelized through iterators, as shown in Figure 2.1a.

This evaluation shows that it is possible to model and parallelize an application having perfectly nested `for` loops with variable index bounds thanks to SDF iterators. We achieve competitive performances against OpenMP, that we could even improve by adding delays in the SDF graph (delays create pipelining). A heuristic to automatically add delays is presented in Chapter 4. Moreover, with the SDF MoC, it is easier to express task parallelism mixed with data parallelism: for example, if a task is executed in parallel with a chunked loop. Finally, we are able to control the expressed degree of parallelism, although restricted to be a divisor of the size of the iteration space.

2.6 Related work

The modeling of nested `for` loops with SDF iterators is related to two main aspects: specialized dataflow languages, especially for image processing applications where at least two dimensions are considered, and dataflow graph clustering, since iterators impact on the degree of data parallelism.

2.6.1 On specialized dataflow languages

The most relevant dataflow MoC for image processing is the Multidimensional Synchronous DataFlow (MDSDF) MoC [ML02], expressing data parallelism across multiple dimensions. However this MoC does not solve the problem of variable image resolutions such as in the SIFT octaves; it requires to split the 4-D array in SIFT in as many 3-D arrays as the number of octaves, and thus the processing actors must be duplicated with a specific array input size for each octave. The 3-D arrays would have different dimension sizes according to the image resolution in each octave: $[\text{layer}, \text{height}_{(\text{octave})}, \text{width}_{(\text{octave})}]$.

Moreover, only a few tools, listed in [KD13], support the MDSDF MoC such as Array-OL [Dem+95; Bou07], or extend it, such as Windowed Synchronous DataFlow [KHT06].

The Brook stream language [Lia+06] supports a subset of MDSDF graphs, expressed directly as `C++` code. Brook only handles kernels with affine bounds, and thus cannot be used for variable image resolutions inducing exponential bounds. More generally, the same problem arises for all models relying on polyhedral analysis [Fea92], as the Polyhedral Process Network (PPN) [Ver10], a parameterized extension of it [ZNS11], or the OpenStream extension of OpenMP [CDF16]: they are dedicated to loops with affine bounds only. Extensive analyses and graph and code transformations allow to model any kind of loops with PPNs [NNS13] but then do not offer control of the degree of parallelism. Dynamic dataflow languages, such as the one supported by Orcc [Yvi+13], offer more flexibility on the application representation, however it is not possible to derive static schedules from such languages when their semantics are fully exploited.

2.6.2 On the clustering of dataflow graphs

Clustering is usually performed on graphs expressing more parallelism than available on the target architecture; clustering simplifies scheduling without increasing the application execution time [PBL95]. The standard way to reach a coarse representation is to limit the unfolding of the SDF graph into a precedence task graph. The unfolding is limited by merging different actors, or multiple executions of the same actor. This operation artificially reduces the repetition vector size, or decreases the values held by the repetition vector. For example, APGAN and RPMC algorithms [BML97] are two heuristics dedicated to merge actors of SDF graphs. Similar methods have been employed for SDF graphs under real-time constraints [ZBS13]. The StreamIT benchmark has also been successfully transformed into coarse SDF graphs for the RAW architecture [GTA06], by an unfolding technique using actor "fusion" (actor merging) and "fission" (actor duplication). Graph pattern detection and substitution is another way to merge actors [CS12]. Regarding reduction of the repetition vector size, there exists a *vectorization* algorithm [Rit+93] for an extension of the SDF MoC called Scalable Synchronous Dataflow, which inspired the recent Partial Expansion Graphs (PEG) [Zak+17]. PEG are scheduled under dynamic scheduling.

Another method is to completely unfold the SDF graph into a precedence task graph, and only then, to apply clustering algorithms; however, this significantly increases the clustering complexity. Clustering algorithms exist for precedence graphs, including compiler intermediate representations [LPC12]. Hierarchical SDF graphs have also been used

to model nested loops [PBR10], one per hierarchy, but they require an analysis of the iteration space. Both methods do not offer control on the degree of parallelism.

Finally, all the aforementioned clustering methods rely on algorithms that analyze the graph and create a coarse or hierarchical version of it, while the proposed iterators only require to replace the `for` loop index bounds by the iterator output. Only one iterator per iteration space is needed.

2.7 Conclusion

This chapter has demonstrated that it is possible to model a subclass of nested loops having variable index bounds with an SDF graph, and to control the degree of expressed parallelism. Thus, we can add to the observations of the StreamIt creators that the CSDF model is not only complicated to use, but is also not always compulsory: the SIFT application has been parallelized efficiently thanks to the SDF model and iterators.

The counter-intuitiveness of a MoC may be a argument to not use it, if another intuitive MoC exists and ensure equivalent performances. However, the ease to understand the abstraction provided by a MoC, is subjective. For example, other related work applied polyhedral techniques to multi-dimensional streams [LT19], motivating their contribution by the abstraction cost of the SDF model. We disagree with the following statement in the introduction of their paper: "*Multi-dimensional streams can be represented in such models* [Synchronous Data Flow (SDF)]*, albeit at the cost of abstraction, which makes this less natural for the programmer and restricts potential transformations in the compiler.*". Regarding the first assumption, it depends on the qualification of the programmer: if the programmer is used to the LabVIEW software for example, the SDF model will be perfectly natural for him/her. Regarding the second assumption, it is actually the opposite since we demonstrate in this chapter that we can model loops having non affine bounds. Nevertheless, as many programmers start by writing a code and rely on tools to automatically transform it and improve it (typically, as a compiler does), a future extension of this work is to automatically generate iterator code from a code analysis in order to help the programmer.

In this chapter, SDF iterators have been used in conjunction with parameterized rates of data consumption and production, for example, to adapt the number of firings of an actor to the number of available PEs. However, depending on the parallel paths in the SDF graph of the application, the best number of firings of an actor might be different from the number of PEs. A simple DSE algorithm is presented in Chapter 5 and solves this first problem. Another problem occurs when the number of firings explode: then,

scheduling becomes a long process limiting the DSE capacity. Yet, having numerous small firings may be useful to ensure that chunks of data fit in the cache memory. To solve this second problem, a fast and scalable scheduling algorithm is presented in Chapter 3. This scheduling algorithm is not only fast, but also takes into account partially periodic constraints, such as a video frame rate.

Dissemination and Implementation

The contribution presented in this chapter has been published in the SAMOS'19 conference [Hon+19]. The implementation of the main use-case, the SIFT application, is available on the preesm-apps public repository.

Chapter 3

Scheduling partially periodic SDF graphs

Contents

3.1 Background 64
3.1.1 Real-time systems and assumptions 64
3.1.2 Synchronous Dataflow graphs 64
3.2 Partially periodic constraints 66
3.2.1 Plain schedulability condition 67
3.2.2 With no scheduler iteration overlapping 68
3.3 Discussion on the schedulability conditions 75
3.3.1 Heuristic to run Algorithm 3.1 efficiently 75
3.3.2 A false positive to Algorithm 3.1 76
3.4 Scheduling SDF graphs with partially periodic constraints . 77
3.4.1 A fast scheduling algorithm for partially periodic SDF graphs . 78
3.4.2 Standard ILP formulation with Choco 82
3.4.3 Scheduler adaptation to extended periodic constraints 83
3.5 Evaluation 84
3.5.1 Partially periodic real-time applications 85
3.5.2 Schedulability check and scheduling 85
3.5.3 Gap between necessary conditions and scheduler 86
3.5.4 Gap between the proposed scheduler and preemptive EDF . . . 89
3.6 Related work 91
3.7 Conclusion 93

Introduction

Real-time systems correspond to systems whose tasks are constrained by deadlines. The tasks are scheduled so that the deadlines are met, inside threads of an operating system or directly on bare-metal. In order to perform their analysis and execution, such real-time systems are generally modeled with tasks having extra periodicity constraints for the deadlines, and precedence constraints for the data. For systems with only periodic tasks, synchronous languages and related tools as Esterel [BG92] and SynDEx [GLS99] are a good choice to check the schedulability and to compute a schedule. On the contrary, a few online schedulers [Foh95; LB00] focus on the execution of aperiodic, sporadic and periodic tasks together, but these schedulers do not consider precedences. Yet real-time systems have periodic components interacting with aperiodic components, and with precedence constraints here expressed in the SDF model. Our contribution aims to analyze the schedulability of such real-time systems, called *partially* periodic, and to schedule them systematically and efficiently. A few necessary conditions and an offline non-preemptive scheduling algorithm are introduced for this purpose. Both have been implemented in the PREESM tool [Pel+14].

Image signal processing systems and visual servoing are typical examples of partially periodic real-time systems where certain components are periodic. For example, a camera films at a periodic framerate and the images arrive to the aperiodic processing components as a stream. Other components may also be periodic, as the input of servo-motors which must be regularly updated. Thus the processing part often depends on periodic inputs and must provide periodically one or more outputs, but does not have to be periodic itself. The flexibility to deviate significantly from periodic operation arises, for example, if data is buffered between components. One possible use-case is the Simultaneous Localization And Mapping (SLAM) application: it constantly retrieves information from a camera or a LIDAR and then processes data to reconstruct a map of the environment and to move according to this map [Wen+18]. Sensor fusion [ZRW12] or other techniques [Gee+16] take advantage of camera *and* LIDAR at the same time.

This contribution focuses on real-time systems with periodic and aperiodic components, modeled with SDF graphs [LM87b]. SDF is commonly used to model image processing applications, as for SLAM with one camera [Pia+18]. SDF graphs of real-time systems often have imposed periodic inputs and outputs. However our approach is more flexible as any component of the system can be periodic. This flexibility is helpful in the case where multiple processing parts rely on different sensors.

Modeling systems is the first step of the design process. The systems then have

to be verified and scheduled. Unfortunately the offline non-preemptive scheduling time complexity is exponential in the number of tasks to get the optimal solution because it is in general NP-complete [KA99]. This complexity limits the design of real-time systems since optimal schedulers do not scale. In contrast, our approach gives results that are not optimal but that can be used to quickly build and assess prototypes of large applications. In other words, our approach is useful for the design space exploration of scheduling solutions. Optimal schedulers and timing property checkers may still have to be used. However, if they are used, it would only be after the prototyping step, on a small set of prototypes.

We focus on offline non-preemptive scheduling because of two main reasons. First, modern systems embed multi-processors where preemption, useful to perform multi-tasking on a unicore processor, is not always required when executing a single application as in our case. Preemptions may still be required to manage external inputs/outputs. Unpredictable external inputs/outputs are not modeled in our case; but the predictable periodic ones coming from sensors or addressed to actuators could be modeled thanks to our partially periodic constraints. Moreover, the absence of preemption prevents the overhead caused by context switching [LDS07] and simplifies the timing analysis. Second, as SDF graphs model only systems where all tasks and their precedences are known in advance, there is no necessity to have a reactive online scheduler. In our case a static schedule on each PE is used for a global self-timed execution of the system.

In this chapter, we consider applications modeled with an SDF graph, where some actors have periodic release times with implicit deadline. We say that such graph has *partially periodic constraints*. Given a number of identical PEs to execute the application and the WCET of each actor, the addressed problems are:

1. to quickly check the schedulability, without computing a schedule;

2. to compute an offline non-preemptive schedule satisfying the periodicity and precedence constraints.

In the context of this chapter, a schedule consists of a list containing the start times of all tasks and the PEs on which they are allocated. Communication times are not taken into account.

The notations used in this chapter and details about SDF graphs are introduced in Section 3.1. Then necessary conditions for the non-preemptive scheduling of SDF graphs with some periodic actors are expressed in Section 3.2. Section 3.3 discusses the algorithm checking if SDF graphs respect the necessary conditions. A greedy algorithm

to schedule graphs with some periodic actors is presented in Section 3.4. Finally, a discussion on this work, including an evaluation of the scheduling algorithm, is given in Section 3.5. The related work is presented in Section 3.6 and is followed by a conclusion.

3.1 Background

This work is related to real-time systems and dataflow graphs, whose important notions are discussed in the next two subsections. Note that the proposed algorithms only work for homogeneous multi-processors where communications are not taken into account.

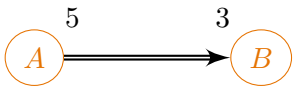
3.1.1 Real-time systems and assumptions

Real-time systems are composed of multiple computational tasks to execute before their deadlines. In this contribution, each task τ has either no real-time constraint or a periodic hard deadline. T_τ denotes the period of a periodic task τ. Tasks without periodic deadlines are called *aperiodic*. For periodically released tasks, their deadline d_τ (relative to their release time) is implicit, which means equal to their period. The WCET of each task τ is denoted C_τ. If a task τ is periodic, its period is greater than its WCET: $C_\tau \leq T_\tau$. In the Gantt diagrams of this chapter, the duration of a task execution corresponds to its WCET. Moreover, periodic releases and implicit deadlines are represented with orange down and up arrows, respectively.

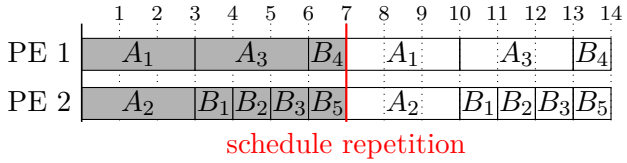
During the execution of a real-time system, the tasks must be ordered and mapped to the PEs in such a way that all tasks meet their deadlines (if any), which is not always possible. In this chapter, a *schedule* refers to the start times and to the static mapping of the tasks. When there is no schedule respecting the deadlines, the system is said to be not schedulable. In this work only offline data-driven non-preemptive schedulers are considered; the system repeats indefinitely a precomputed schedule. We consider that the system has m identical PEs.

3.1.2 Synchronous Dataflow graphs

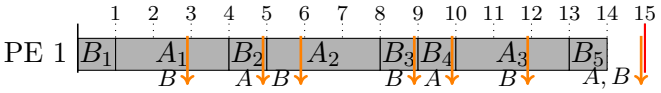
SDF graphs have already been introduced in Section 1.2.1. However, for this contribution, we would like to emphasize a few of their properties and introduce extra notations. In particular, we consider an indefinitely repeated static schedule; each repetition is called a *scheduler iteration*. Two examples of a schedule for an SDF graph are given in Figure 3.1, where the repetition vector is $[3, 5]^T$ (indexed by actor names in the lexicographic order). The repetition vector $\vec{r}$ defines a *graph iteration*, during which there



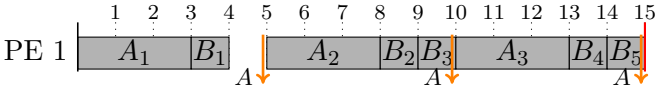
(a) SDF graph example, of repetition vector $[3, 5]^T$.



(b) Schedule example of 3.1a on two PEs. Two graph iterations are represented, separated by a red vertical line.



(c) Unicore schedule example of 3.1a, respecting actor periods $T_A = 5$ and $T_B = 3$, but not respecting the following precedences: $A_1 \rightarrow B_1$, $A_2 \rightarrow B_2$ and $A_3 \rightarrow B_4$.



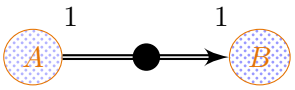
(d) Unicore schedule example of 3.1a, respecting the actor period $T_A = 5$ and all data dependencies. Actor B is not periodic here.

Figure 3.1 – SDF graph scheduling examples.

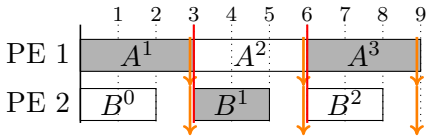
are as many firings as specified in $\vec{r}$, released according to the graph topological order. When the SDF graph contains delays, all firings of one *graph* iteration do not occur in the same *scheduler* iteration: this is *pipelining*. Figure 3.2 gives an example of such a pipelined schedule with delays: the firing of B consuming the data produced by the last firing of A happens one scheduler iteration after. Graph iterations are indexed as firing exponent in the Gantt diagram of Figure 3.2b.

In this chapter α_j is the j-th firing of α in one scheduler iteration. The WCET of an actor α, denoted C_α, is the same for each firing of α. Actors have uppercase names for examples, and lowercase names for formula variables. $\mathcal{P}$ denotes the set of periodic actors in G, and $\mathcal{N}$ denotes the set of aperiodic actors. The periods of the actors in $\mathcal{P}$ are defined by the user; but the graph consistency restricts their possible values. Indeed all periods are linearly related, as it will be demonstrated in Section 3.2.1.

Delays on buffers are allowed, with some restrictions for cycles in G. Indeed SDF graphs deadlock if there is no delays in cycles, and G is *live* if no deadlock occurs. We assume in the analysis that G is live, thanks to delays set by the user on one specific edge of each cycle. Then, such specific edges will not be considered during the analysis



(a) SDF graph having two periodic actors with a dependency and one delay. The repetition vector is $[1, 1]^T$.



(b) Schedule example of 3.2a on two PEs. The vertical red lines delimits the scheduler iterations. Three scheduler iterations are represented while only two graph iterations are completed (iterations 1 and 2, denoted as firing exponents).

Figure 3.2 – Example of scheduler and graph iterations, with pipelining.

and thus, the considered SDF graphs are seen as DAGs in this work. Such specific edges can also be computed automatically by a heuristic, which means that this cycle assumption might simply result from prior transformation. The computation to break cycles is detailed in the next chapter, in Section 4.3. It is also possible to ensure liveness by adding delays on multiple edges, but our restriction eases the necessary conditions presented in the next section. This restriction is not problematic since this way of breaking cycles is common in the image processing applications, as it will be seen later in Sections 4.3 and 4.4.

The case of cycles that are self-loops on an actor is also considered. Self-loops disable auto-concurrency and, as cycles, require delays in order to be consistent. *Auto-concurrency* means that multiple firings of an actor can be executed at the same time on different PEs. For a self-loop $l \in \mathcal{L} \subseteq E$, we assume $\text{cons}(l) = \text{prod}(l) = d_0(l)$.

3.2 Partially periodic constraints

Non-preemptive scheduling is often not the best strategy when considering only periodic actors since it may lead to use the PEs below their full capacity. For example, consider that two periodic actors A and B are scheduled as in Figure 3.1c. Actors A and B are periodic, but the only functional requirement on B is that there are 3 executions of A for 5 of B, according to the repetition vector of the SDF graph in Figure 3.1a. In Figure 3.1c, the periods are $T_A = 5$ and $T_B = 3$. When ignoring the precedences, the system is schedulable on one PE with the WCET respectively $C_A = 3$ and $C_B = 1$, and the PE even idles during 1 time unit. If the execution time of A is now $C_A = 3.1$ (instead of 3), the PE is still not used to its full capacity but the system is not schedulable anymore since B_3 would miss its deadline in any case. However, if B is not required to

be periodic, the system is schedulable on 1 PE with $C_A = 3.1$. When not ignoring the precedences, and with $C_A = 3$, releasing the periodic constraint on actor B also results in a schedulable system, as shown in Figure 3.1d.

Thus, in these cases, partially periodic constraints help to fully use the capacities of the PEs in the context of non-preemptive scheduling. In this section, we focus on necessary conditions for schedulability of SDF graphs with partially periodic constraints. The generic processor utilization necessary condition is recalled in Section 3.2.1 while a more precise one is established in Section 3.2.2.

3.2.1 Plain schedulability condition

A widely used necessary condition for schedulability of periodic tasks derives from the processor utilization factor [LL73] metric $U = \sum \frac{C_\tau}{T_\tau}$, without unit. U is the ratio of computations to perform per time unit. $U \leq m$ is a necessary but not sufficient condition, for all preemptive and non-preemptive schedulers of tasks with and without precedence constraints: if $U > m$ the system is not schedulable [Hor74].

In the case of weakly connected SDF graphs[1], all actors are connected and specifying the period of one actor π is equivalent to specifying a period for the whole graph. Indeed the graph period T_G will be $\vec{r}[\pi] \times T_\pi$ time unit. In Figure 3.1c, the graph period is 15, according to the periods $T_A = 5$ and $T_B = 3$, and to the repetition vector $[3, 5]^T$. Formally, in one graph iteration, the start time of the k-th firing of a periodic actor $\pi \in \mathcal{P}$ with an implicit deadline must occur in the following time interval:

$$[\![kT_\pi; (k+1)T_\pi - C_\pi]\!], \text{with } k \in [\![0; \vec{r}[\pi][\![\tag{3.1}$$

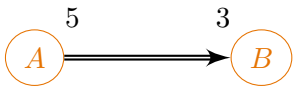
Consequently, on average $\vec{r}[\alpha]$ firings of an aperiodic actor α are executed during each graph period T_G, since the repetition vector imposes $\vec{r}[\alpha]$ firings of α for $\vec{r}[\pi]$ firings of π. Moreover, note that Equation (3.1) disables auto-concurrency of periodic actors.

As each periodic actor π defines a graph period $T_G = \vec{r}[\pi] \times T_\pi$ deriving from the unique repetition vector, this implies that all the obtained graph periods must be equal:

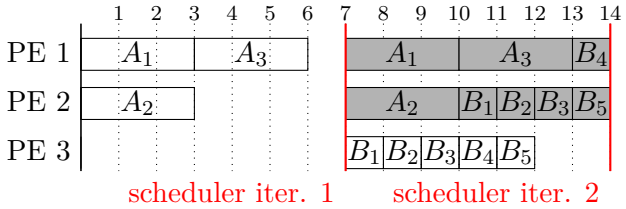
$$\exists! T_G, \forall \pi \in \mathcal{P}, T_G = \vec{r}[\pi] \times T_\pi \tag{3.2}$$

Note that the repetition vector is computed before the graph period; hence, the actor periods set by the user must be compatible with the repetition vector and they cannot alter it. Then, the processor utilization factor metric may be reformulated in the context

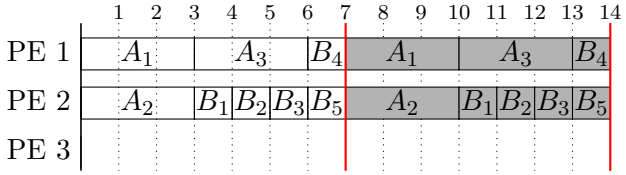
[1] See Section 1.4.3 for a definition of weakly connected graphs and a related discussion.



(a) SDF graph example, of repetition vector $[3, 5]^T$.



(b) ✘ Alternating schedule example of 3.3a on three PEs, not respecting Assumption 1.



(c) ✔ Normal schedule example of 3.3a on three PEs, respecting Assumption 1.

Figure 3.3 – Example and counter-example of Assumption 1.

of partially periodic constraints, considering the average number of aperiodic firings per graph period T_G. Equation (3.3) is a necessary but not sufficient schedulability condition for partially periodic SDF graphs.

$$m \geq U = \frac{\sum_{\alpha \in \mathcal{N}} \vec{r}[\alpha] \times C_\alpha}{T_G} + \sum_{\pi \in \mathcal{P}} \frac{C_\pi}{T_\pi} \tag{3.3}$$

The considered number of aperiodic firings per graph period is indeed an *average* value since the schedule might be alternating, as shown in Figure 3.3b.

3.2.2 With no scheduler iteration overlapping

Results of this subsection make the following assumption on the scheduler.

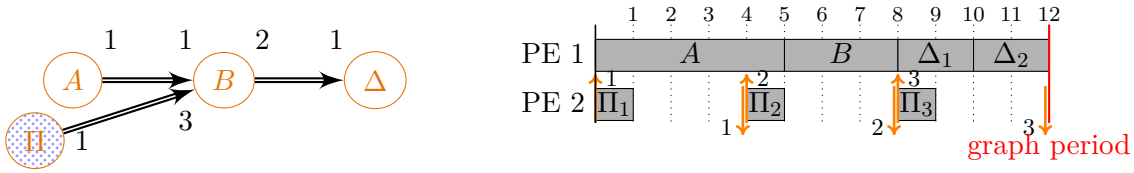
Assumption 1 (A1)**.** *For every actor α, as many firings as specified in the repetition vector $\vec{r}[\alpha]$ must have been completely executed before the next scheduler iteration begins.*

Figure 3.3 shows both an example and a counter-example of Assumption 1. Under Assumption 1, another necessary condition can be derived from the path lengths in the SDF graph with partially periodic constraints. This new necessary condition is complementary to the processor utilization factor condition.

A *scheduler* iteration is the static scheduling of the application that is indefinitely repeated until the application is stopped. This assumption is made to ease the scheduling, the code generation, and the memory allocation. Under Assumption 1, if there is a graph period, all firings of one scheduler iteration must be done during a time interval equal to this graph period. Assumption 1 is present in the PREESM SDF graph scheduler [Pel+14] where scheduler iterations cannot overlap in time and are separated by a synchronization barrier between all PEs. One graph period T_G separates two successive barriers, so it ensures that for each actor α, there are $\vec{r}[\alpha]$ executions of α between two barriers. As the schedule is indefinitely repeated, each actor firing, independently from each other of the same actor, can be seen as a periodic task of period equal to T_G. Note that Assumption 1 is similar to *K-periodic scheduling* [BMd12] with K being the repetition vector $\vec{r}$ in our case. While K-periodic scheduling imposes a periodic schedule of $K[\alpha]$ firings independently for each actor α, Assumption 1 enforces these periodic schedules to be synchronized with barriers.

Assumption 1 supports schedules pipelined with delays, as in the previous example depicted in Figure 3.2; such delays may actually help to break problematic data dependencies. Indeed, scheduling a partially periodic SDF graph without taking care of data dependencies may lead to buffer underflows and overflows as illustrated in Figure 3.4 where the SDF graph is however consistent. In this example the period of the actor Π is 4 time units (and $C_\Pi = 1$), and the graph period is 12. The Gantt diagram in Figure 3.4b respects the periodic constraint but not the data dependencies. An underflow occurs since B and Δ are executed before having received the data produced by the last firing of Π. To avoid such underflow, one possibility is to add delays on the graph. Note that the delays are predefined by the user, and are not computed nor checked by Algorithm 3.1, presented at the end of this section. However, such data dependency errors are checked easily on static schedules. Another possibility to avoid the underflow is to increase the actor period T_Π to 6 time units, as shown in Figure 3.5 which depicts a valid schedule of Figure 3.4a.

Figure 3.4 illustrates an intuitive necessary condition to check the schedulability: all actors depending on the tokens produced by the last execution of Π must be executed in the slack time of Π. This necessary condition derives from Assumption 1. The slack time of an actor corresponds to the interval of time between its period and its execution time. The slack time of Π is formally defined by $T_\Pi - C_\Pi$; it must be non negative in our case. A symmetrical necessary condition can be computed for the first execution of any periodic actor, this time with all its incoming data dependencies, which are all actors on a directed path leading to the periodic actor. In order to formalize these necessary



(a) SDF graph with dependencies on a periodic actor. The repetition vector is $[1, 1, 2, 3]^T$. $nblf^{\uparrow}_{\Pi}(B) = 1$, $nblf^{\uparrow}_{\Pi}(\Delta) = 2$

(b) Schedule example of graph 3.4a, generating underflow for the firings of B and Δ if $T_{\Pi} = 4$.

Figure 3.4 – Periodic actor Π generating an underflow.

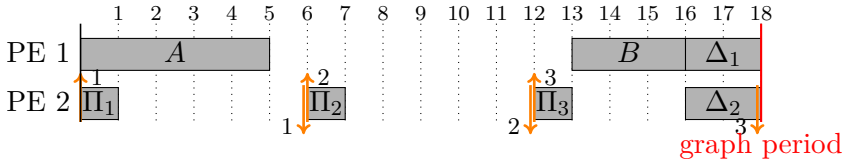


Figure 3.5 – Valid schedule example of graph 3.4a on two PEs and if $T_{\Pi} = 6$.

conditions, some notations and functions are introduced in the next paragraphs.

$\mathcal{D}^{\uparrow}_{\pi}$ denotes the set of actors in G that are transitively data dependent on an actor π: if $\alpha \in \mathcal{D}^{\uparrow}_{\pi}$, the last firing of π enables at least one firing of α. $\mathcal{D}^{\uparrow}_{\pi}$ can be computed by a unidirectional graph traversal from π. For the SDF graph in Figure 3.4a, the actors which are data dependent on the periodic actor Π are: $\mathcal{D}^{\uparrow}_{\pi} = \{B, \Delta\}$. The subgraph restriction of G containing only the actors in $\mathcal{D}^{\uparrow}_{\pi}$, is denoted $G^{\uparrow}_{\pi}$. In the notations of this chapter, the up arrow $\uparrow$ is always paired with a reference actor, π in $\mathcal{D}^{\uparrow}_{\pi}$. The up arrow emphasizes the fact that we are only considering the actors or firings which are data dependent on the reference actor π. Symmetrically, all presented equations can be reused for the first firing of π instead of the last, considering all actors $\mathcal{D}^{\downarrow}_{\pi}$ on which π is data dependent. For brevity, such equations are not shown.

The main metric to compute is the numbers of actor firings, enabled by a single firing of a periodic actor π. These numbers of firings allow us to compute lower bounds of the processor utilization factor. The analysis is simplified by restricting it to the last firing of π and all induced firings of its successors in $\mathcal{D}^{\uparrow}_{\pi}$. Note that the number of remaining dependent firings can be computed for two adjacent actors connected by a single buffer e: k firings of $\mathrm{src}(e)$ enable $\max\{0, \left\lceil \frac{k \times \mathrm{prod}(e) - d_0(e)}{\mathrm{cons}(e)} \right\rceil\}$ firings of $\mathrm{dst}(e)$. The term $k \times \mathrm{prod}(e)$ corresponds to the new tokens incoming on the buffer e. The ceiling operator is needed since the previous execution of the producer $\mathrm{src}(e)$ may have left unused tokens on e. At the end of the graph iteration, e contains exactly $d_0(e)$ delays.

The function computing the number of firings enabled by the last firing of a periodic actor π is denoted $nblf^{\uparrow}_{\pi}$, defined in Equation (3.4). $nblf^{\uparrow}_{\pi}$ is a recursive function, depending on the predecessor actors in the graphs $G^{\uparrow}_{\pi}$. The set of incoming edges to an actor α in $G^{\uparrow}_{\pi}$ is denoted $IE^{\uparrow}_{\pi}(\alpha)$; this set excludes self-loops $l \in \mathcal{L}$. In Figure 3.4a, $IE^{\uparrow}_{\pi}(B)$ only contains the edge coming from Π and not the one coming from A. Indeed, $A \notin \mathcal{D}^{\uparrow}_{\pi}$ and thus, $A \notin G^{\uparrow}_{\pi}$.

$$nblf^{\uparrow}_{\pi}(\alpha) = \max_{e \in IE^{\uparrow}_{\pi}(\alpha)} \left\{ 0, \left\lceil \frac{nblf^{\uparrow}_{\pi}(\mathrm{src}(e)) \times \mathrm{prod}(e) - d_0(e)}{\mathrm{cons}(e)} \right\rceil \right\} \tag{3.4}$$

The recursion stops at the root actor π, having no incoming edges (there is only one root, by construction of $G^{\uparrow}_{\pi}$), where $nblf^{\uparrow}_{\pi}$ holds the value 1 if π is periodic and 0 otherwise. Hence if the root actor is not periodic, $nblf^{\uparrow}_{\pi}$ takes the value 0 on all vertices and it does not help to find any necessary condition. For brevity, the proof of Equation (3.4) is given for one direct predecessor only of $\mathrm{dst}(e)$.

Proof. To prove Equation (3.4), let us consider the firings of the actor $\mathrm{dst}(e)$ before those that are induced by the last firing of π. By definition this number of firings is $\vec{r}[\mathrm{dst}(e)] - nblf^{\uparrow}_{\pi}(\mathrm{dst}(e))$. Under Assumption 1, there are exactly $\vec{r}[\alpha]$ firings of each actor α during one scheduler iteration. So this number can also be computed considering all tokens produced on e by $\mathrm{src}(e)$ during one scheduler iteration, before the last firing of π: this is why $\vec{r}[\mathrm{src}(e)] - nblf^{\uparrow}_{\pi}(\mathrm{src}(e))$ multiplies the production rate in the following equality.

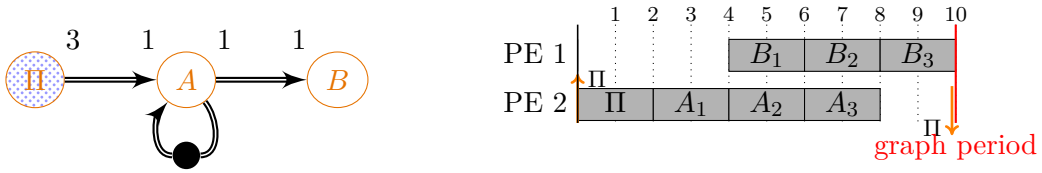
$$\vec{r}[\mathrm{dst}(e)] - nblf^{\uparrow}_{\pi}(\mathrm{dst}(e)) = \left\lfloor \frac{(\vec{r}[\mathrm{src}(e)] - nblf^{\uparrow}_{\pi}(\mathrm{src}(e))) \times \mathrm{prod}(e) + d_0(e)}{\mathrm{cons}(e)} \right\rfloor$$

Thus $nblf^{\uparrow}_{\pi}(\mathrm{dst}(e))$ is equal to:

$$\vec{r}[\mathrm{dst}(e)] - \left\lfloor \frac{\vec{r}[\mathrm{src}(e)] \times \mathrm{prod}(e) - nblf^{\uparrow}_{\pi}(\mathrm{src}(e)) \times \mathrm{prod}(e) + d_0(e)}{\mathrm{cons}(e)} \right\rfloor$$

Knowing that $\forall x \in \mathbb{R}, -\lfloor x \rfloor = \lceil -x \rceil$, it becomes:

$$nblf^{\uparrow}_{\pi}(\mathrm{dst}(e)) = \left\lceil \frac{\vec{r}[\mathrm{dst}(e)] \times \mathrm{cons}(e)}{\mathrm{cons}(e)} + \frac{-\vec{r}[\mathrm{src}(e)] \times \mathrm{prod}(e) + nblf^{\uparrow}_{\pi}(\mathrm{src}(e)) \times \mathrm{prod}(e) - d_0(e)}{\mathrm{cons}(e)} \right\rceil$$



(a) Sample SDF graph. The repetition vector is $[3, 3, 1]^T$

(b) Schedule example of graph 3.6a, on two PEs, with firings of a self-loop.

Figure 3.6 – Counter-example to generalization of Equation (3.6): actor A is not always executed alone.

From the consistency of G, see Equation (1.1), $\vec{r}[\mathrm{dst}(e)] \times \mathrm{cons}(e) - \vec{r}[\mathrm{src}(e)] \times \mathrm{prod}(e) = 0$, so the last formula can be simplified to Equation (3.4) (without the maximum, needed when they are multiple direct predecessors). □

The following formula is then a necessary condition for schedulability under Assumption 1. Equation (3.5) corresponds to the processor utilization factor of all firings depending on the last firing of a periodic actor. This processor utilization factor is computed over the slack time of the periodic actor, hence the division by $T_\pi - C_\pi$.

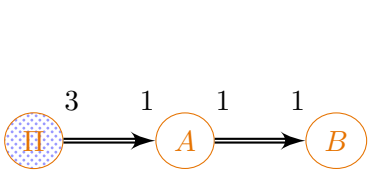
$$\forall \pi \in \mathcal{P}, \frac{\sum_{\alpha \in \mathcal{D}_\pi^\uparrow} nblf_\pi^\uparrow(\alpha) \times C_\alpha}{T_\pi - C_\pi} \leq m \tag{3.5}$$

Note that the maximum length of any graph path starting at a periodic actor π also provides a simple necessary condition of the schedulability: this length must be less than the slack time of π. A necessary condition for actors with self-loops is formalized in Equation (3.6).

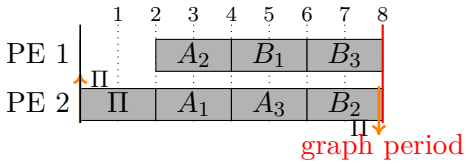
$$\forall \alpha \in \mathcal{D}_\pi^\uparrow \cap \mathcal{L},\ nblf_\pi^\uparrow(\alpha) \times C_\alpha \leq T_\pi - C_\pi \tag{3.6}$$

Unfortunately, our published contribution [Hon+20a] contains an error in the algorithm referencing Equation (3.6), and states: "Equation (3.6) can be extended to all paths between a periodic root π and leaves of the DAG $G_\pi^\uparrow$." Experimental results were not impacted by the error; we detail now why the previous quoted sentence is misleading. The necessary condition in Equation (3.6) is correct but its *direct* extension to all paths is wrong: firings dependent on the self-loop may be executed at the same time and the WCET cannot simply be summed. Figure 3.6 gives an example where actor A having a self-loop is executed at the same time as its outgoing dependency B (firings A_2 and B_1, A_3 and B_2).

Indeed the necessary condition expressed in Equation (3.6) can be extended to all



(a) Sample SDF graph. The repetition vector is $[3, 3, 1]^T$



(b) Schedule example of graph 3.7a, on two PEs, with the co-existence of firings of different actors at the same time (A_3 and B_1).

Figure 3.7 – Floor function underestimation example, as used in Equation (3.7).

paths between a periodic root π and leaves of the DAG $G^{\uparrow}_{\pi}$, but at the cost of an underestimation of the execution time. On each of these paths, each actor will be executed at least once except if there are enough delays before the actor; if no delays, the path length is greater or equal to the WCET sum of actors located on the path. Again, all these path lengths must be lower than the slack time $T_\pi - C_\pi$. It is possible to underestimate the number of times that the WCET of an actor must be added to the path length, given the number of PEs m. The generic underestimated necessary condition is given in Equation (3.7). For example, if $nblf^{\uparrow}_{\pi}(\alpha) = 6$ and $m = 3$, then C_α is added 2 times.

$$\forall \alpha \in \mathcal{D}^{\uparrow}_{\pi},\ C_\alpha \times \max\left\{1, \left\lfloor \frac{nblf^{\uparrow}_{\pi}(\alpha)}{m} \right\rfloor \right\} \leq T_\pi - C_\pi \tag{3.7}$$

Figure 3.7 is an intuitive example to demonstrate why the floor function is used in Equation (3.7): firings of different actors may be scheduled at the same time, and this overlap is not easily predictable. The only certainty for the path length after Π is about the concurrent execution of multiple firings of only one actor at a time. $nblf^{\uparrow}_{\Pi}(A) = nblf^{\uparrow}_{\Pi}(B) = 3$ but there is only two PEs in the system so only $\lfloor \frac{3}{2} \rfloor = 1$ execution time of each actor is added to the path length in this case.

Algorithm 3.1 checks the schedulability of a periodic actor π in $G^{\uparrow}_{\pi}$ thanks to the aforementioned necessary conditions: the one derived from the processor utilization factor, and the one derived from the path lengths. Note that the computation of the path lengths (using Equation (3.7) instead of Equation (3.6)) is similar to a Max-Plus algebra, as used for throughput analysis [Gha+06b]: "Max" operation is on line 17, and "Plus" operation is on line 21 of Algorithm 3.1. The efficiency of Algorithm 3.1 is discussed in the next section.

Finally, a very similar algorithm $nbff^{\downarrow}_{\pi}$ can be written for the first periodic firing of actor π in $G^{\uparrow}_{\pi}$, by exchanging the incoming edges $IE^{\uparrow}_{\pi}$ with the outgoing edges $OE^{\downarrow}_{\pi}$,

Algorithm 3.1: Modified BFS to compute $nblf^{\uparrow}_{\pi}$ and related necessary conditions

```
1  function nblfExt(π)
2      forall α ∈ G do                          ▷ Sets initial properties of each actor.
3          timeTo(α) ← 0;
4          nblf↑π(α) ← 0;
5          nbVisits(α) ← 0;
6      C_tot ← 0;
7      nblf↑π(π) ← 1;                           ▷ π is the periodic root.
8      D↑π ← ∅;                                 ▷ Outgoing depencies are not known yet.
9      queue ← ∅;
10     addLast(queue, π);
11     while queue ≠ ∅ do
12         α ← pop(queue);
13         D↑π ← D↑π ∪ {α};
14         forall e ∈ OE↑π(α) do
15             dest ← dst(e);
16             nbVisits(dest) ← nbVisits(dest) + 1;
17             timeTo(dest) ← max{timeTo(dest), timeTo(α)};
18             nblf↑π(dest) ← max{ nblf↑π(dest), ⌈ (nblf↑π(α) × prod(e) − d0(e)) / cons(e) ⌉ };   ▷ See Equation (3.4).
19             if nbVisits(dest) = #IE↑π(dest) and nblf↑π(dest) > 0 then
20                 addLast(queue, dest);
21                 timeTo(dest) ← timeTo(dest) + C_dest × max{1, ⌊ nblf↑π(dest) / m ⌋};
                   ▷ Update the path length to dest with an underestimation of its execution time. Generalization of Equation (3.7).
22                 if timeTo(dest) > T_π − C_π then
23                     return System not schedulable.
24     forall α ∈ D↑π − {π} do
25         C_tot ← C_tot + nblf↑π(α) × C_α;     ▷ See Equation (3.5).
26     if C_tot / (T_π − C_π) > m then
27         return System not schedulable.
```

the operator dst with src, and the variable *dest* with *srce*, and by switching production and consumption rates. Note that to study the first firing instead of the last, we can also reuse the exact same equations applied to the transpose graph of G. The transpose G^T of a graph G is its mirror, where all edges are directed in the opposite direction, inverting all data dependencies.

3.3 Discussion on the schedulability conditions

Algorithm 3.1, implementing the necessary conditions for the schedulability of SDF graph with partially periodic constraints, has a linear complexity in the number of edges in $G^{\uparrow}_{\pi}$. Thus if all actors in G are periodic, it may not be efficient to execute Algorithm 3.1 on each one: in specific cases the overall complexity can be more than quadratic in the number of vertices in G, as for the star graphs with directed paths going to/from a central vertex. In order to perform the algorithm on a subset of the periodic actors, a heuristic is presented in Section 3.3.1. Algorithm 3.1 faces another problem: as it relies only on necessary conditions, there are cases where the algorithm fails to detect a non schedulable system. This point is discussed in Section 3.3.2.

3.3.1 Heuristic to run Algorithm 3.1 efficiently

In this subsection a heuristic is given to execute Algorithm 3.1 on a small set of periodic actors: the one having small slack time and a low topological rank in G. Indeed a small period T_π will reduce the denominator in Equation (3.5), while a low topological rank may increase the numerator because it means that more actors are located after π. Thus this heuristic selects the actors being more discriminative regarding to the schedulability tests of Algorithm 3.1.

Delay placement assumption on cycles enables us to see the SDF graph as a DAG, so there is always a topological ordering existing. One topological ordering is used to select actors: it corresponds to an As Soon As Possible (ASAP) schedule of G^T, not constrained by the number of PEs. The ASAP topological ordering on G^T is denoted o^T and is used to select the periodic actors on which Algorithm 3.1 is called. Note that the actor WCETs are not taken into account in this topological ordering, only the structure of the graph is used. Such topological ordering can be computed with a BFS, having a linear complexity in the number of edges in G.

Considering the SDF graph in Figure 3.8, there are three topological ranks: one per actor in the longest graph path which is $A \to \Gamma \to E$. Thus $o^T(A) = 2$, $o^T(\Gamma) =$

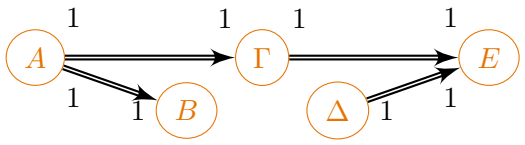


Figure 3.8 – Sample SDF graph of repetition vector $\vec{1}$, for topology ranks example. ASAP ordering on the graph: $o \equiv \{A, \Delta\} \prec \{B, \Gamma\} \prec \{E\}$. ASAP ordering on the transpose graph: $o^T \equiv \{E, B\} \prec \{\Gamma, \Delta\} \prec \{A\}$.

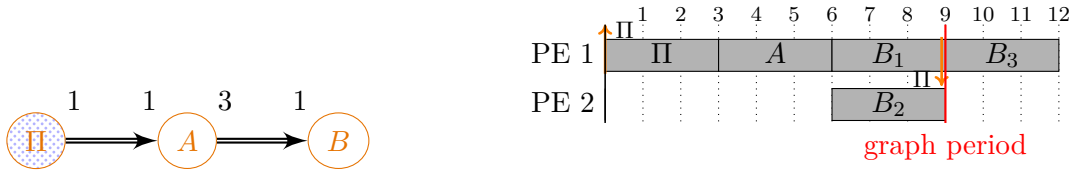
1, $o^T(E) = 0$. The other actors have the lowest topological ranks at which they can be executed, that is $o^T(B) = 0$, $o^T(\Delta) = 1$. Indeed B has no incoming edges in G^T so its rank is 0. The ASAP ordering on the transpose graph of Figure 3.8 can be equivalently represented as follows: $o^T \equiv \{E, B\} \prec \{\Gamma, \Delta\} \prec \{A\}$. On the original graph, the ASAP ordering is: $o \equiv \{A, \Delta\} \prec \{B, \Gamma\} \prec \{E\}$

Formally, the heuristic selects the periodic actors having the lowest $\frac{T_\pi - C_\pi}{o(\pi)}$. The number of selected actors with this heuristic is arbitrarily chosen by the user. Note that vertices with ASAP topological rank equal to 0 are not of interest since it means that they have no successors; Algorithm 3.1 is not run on such vertices.

3.3.2 A false positive to Algorithm 3.1

Algorithm 3.1 performs two schedulability tests. One is using the processor utilization factor U, Equation (3.5) lines 24−27, and thus does not consider the precedences between actors. Considering multiple PEs, it may lead to keep invalid schedules where $U < m$, but where a path from a periodic actor is longer than the slack time of this actor. This situation is precisely checked by the other schedulability test using path lengths, lines $21 - 23$ in Algorithm 3.1. Between these two situations, the algorithm may miss that U is too large on a small portion of the slack time: for example, if $nblf^{\uparrow}$of the last actor is greater than m. Thus, Algorithm 3.1 fails to find that the graph represented in Figure 3.9 is not schedulable with two PEs and $T_\Pi = T_G = 9$. Yet the critical path starting from Π in Figure 3.9a is equal to the slack time of Π and $U = \frac{15}{9}$ is less than the number of PEs $m = 2$.

Algorithm 3.1 may compute other false positive answers: only $G_\pi^{\uparrow}$ is considered and thus, periodic actors having a period smaller than T_π in G but not being in $G_\pi^{\uparrow}$ are not taken into account. To avoid that, Algorithm 3.1 can be refactored with a subfunction performing computations of lines $2 - 25$. The subfunction is called on π and on all the periodic actors having a period smaller than T_π, and not being connected by any path in G. For brevity, the full algorithm is not presented here.



(a) SDF graph with one periodic actor. The repetition vector is $[1, 3, 1]^T$.

(b) Shortest schedule of graph in Figure 3.9a on two PEs. $T_\Pi = T_G = 9$.

Figure 3.9 – A false positive to Algorithm 3.1: B_3 cannot be scheduled before the graph period.

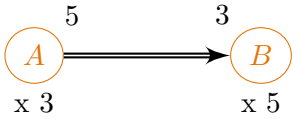
However, even if false positives appear, the system designer will logically continue its work by calling a scheduler. The scheduler will give a final answer: schedulable or not. In the next section, an offline scheduler is presented under Assumption 1. This scheduler is a heuristic and thus does not attempt to find an optimal scheduling, it rather focuses on quickly giving an answer to the designer.

3.4 Scheduling SDF graphs with partially periodic constraints

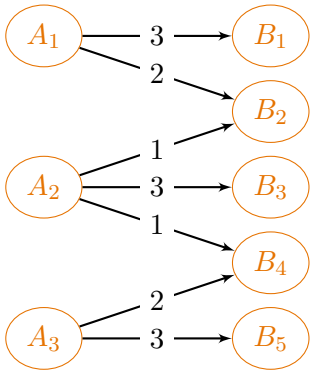
The problem studied in this section is the scheduling of G^* with partially periodic constraints. G^* is the SRSDF graph corresponding to the SDF graph G. In other words, G^* is the unrolled version of G, where data dependencies are expressed between firings instead of actors. Since all dependencies are explicitly expressed in G^*, it enables computing a static schedule. Each SDF actor α in G has $\vec{r}[\alpha]$ corresponding vertices in the SRSDF graph G^*, and on each edge e of G^* the rates of both sides are equal: $\text{prod}(e) = \text{cons}(e)$. In this section, *tasks* refer to vertices in G^*. As in G, delays break cycles so G^* is a DAG. An example of SDF graph G and its corresponding SRSDF version G^* is given in Figure 3.10, extracted from Figure 1.2. An example with delays has been given in Figure 1.4.

Scheduling of DAG of tasks has been widely studied [KA99] however the periodic case is specific since the start times of periodic actors are bounded in an interval of the form of Equation (3.1).

For example, the FAST algorithm [KAG96], based on a list scheduling heuristic and a neighborhood search, is not appropriate for periodic schedules. Especially its list scheduling heuristic relies on a classification of tasks belonging to, or connected to,



(a) SDF graph example, of repetition vector $[3, 5]^T$.



(b) DAG deduced from SDF example 3.10a.

Figure 3.10 – SDF graph G and its corresponding SRSDF version G^*.

the critical path of the DAG[2] with unlimited resources. The main difference between the problem studied here and the one solved by FAST and other standard list scheduling algorithms is that, because of partially periodic constraints, we have a bound on the schedule length. This bound is the graph period; it is both the shortest and the longest accepted schedule length. The bound implies that there might be no critical path in graphs with partially periodic constraints: idle time can be present between any successive tasks, as long as all tasks are completed within the graph period duration.

The scheduler that we have developed is presented in Section 3.4.1, while an optimal ILP formulation is summarized in Section 3.4.2. In Section 3.4.3, we briefly discuss how to adapt our scheduler to an extension of the periodic constraints.

3.4.1 A fast scheduling algorithm for partially periodic SDF graphs

The main difficulty to design a greedy list scheduling algorithm is to efficiently order the vertices before trying to schedule them. The FAST algorithm cannot be used as is but some of its techniques are reused in the presented Algorithm 3.3. As in FAST, Algorithm 3.3 relies on ASAP and As Late As Possible (ALAP) orderings.

The minimum start time ns of each task in G^* as well as the maximum start time xs are computed first. This is done in two successive rounds: 1) for all periodic actors, with Equation (3.1), 2) for all actors, with ASAP and ALAP. During round 2), ALAP schedule has a global deadline equal to the graph period. If any task τ has $\text{ns}(\tau) > \text{xs}(\tau)$, the algorithm stops: the system is not schedulable.

[2]The critical path of a DAG is the longest path of connected tasks according to their ETs and the communication times if specified. It gives the smallest possible schedule length.

Algorithm 3.3 is executed once all ns and xs have been computed, starting with the `schedule` procedure. The tasks are sorted according to their average start time $\frac{\text{xs}+\text{ns}}{2}$. This sorting criterion is a heuristic to balance the task executions over time. The list of tasks to allocate, *lt* line 8, contains only vertices having all their dependencies satisfied, initially the one having no incoming edges. If multiple tasks in *lt* have the same average start time, ns is used to break the tie, by increasing order. The algorithm performs a first fit approach: it selects the next task in *lt* and schedules it on the least loaded PE[3]. Subroutines called by the `schedule` procedure are briefly presented in Algorithm 3.2. The `allocateAndRemoveIfBefore` procedure, lines $13-24$ maps an allocated task on the PE most idling at the current time (shortest *finishTime*), and updates *lt* with tasks having a direct dependency on the currently allocated task (line 24). This procedure stops the scheduling process in two cases: 1) if a task is scheduled after its maximum start time xs, at lines $16-17$, 2) if the total idle time is more than the maximum possible, at line 10, according to the formula at line 5 of the `schedule` procedure. Thus Algorithm 3.3 stops as soon as it detects that it cannot schedule a task in G^*.

Algorithm 3.3 is *greedy* since if a task τ implies an idle time, the algorithm tries to schedule before τ the tasks τ_b having ns $<$ *predFinishTime*(τ), without delaying τ. This is the purpose of lines $13-17$ in Algorithm 3.3. The test line 18 in Algorithm 3.2 ensures that τ_b can be executed without delaying τ, and if not it prevents its allocation. Although Algorithm 3.3 is *greedy*, it is not subject to the Dhall's effect [DL78]. Dahll's effect occurs on multi-processors when two tasks are ready at the same time and have the same deadline, but the allocation order prevents from allocating both tasks because the smallest task may be allocated first on the least loaded PE, not leaving enough space for the other task. In Algorithm 3.3, as tasks are sorted by average start time from ASAP and ALAP scheduling, the largest task will have a shorter average start time and thus will appear sooner in the list of tasks ready to be scheduled.

The complexity of Algorithm 3.3 is upper bounded by the number of edges in G^* and by the linearithmic cost of the sorting operation on the vertices: $\mathcal{O}(\#E^* + \#V^*(m + \log(\#V^*)))$. The number of PEs m appears as a factor of $\#V^*$ since the list of PEs *lPEs* must remain sorted to select the least loaded PE for every vertex in the ready queue *lt*. The cost of transforming G in G^* is not included in this complexity, it is upper bounded by $\#E^*$, which depends on the number of firings specified in the repetition vector. At worst, the complexity of the transformation from G to G^* is exponential in the number

[3]Additionally, our implementation handles mapping constraints to specify which PEs support the execution of each SDF actor. So our first fit approach first selects the subset of PEs supporting the execution of the next firing to schedule, and then selects the least loaded PE in this subset.

Algorithm 3.2: Subroutines for partially periodic scheduling of tasks

```
1  procedure addReadyTasks(lt, τ, nbAllocs) ▷ Add tasks in the schedule queue only
   if their predecessors are allocated.
2      remove(lt, τ);
3      nbAllocs ← nbAllocs + 1;
4      forall e ∈ OE(τ) do                    ▷ Visit all successors of newly allocated τ.
5          dest ← dst(e);
6          nbVisits(dest) ← nbVisits(dest) + 1;;
7          predFinishTime(dest) ← max{predFinishTime(dest), finishTime(τ)};
8          if nbVisits(dest) = #IE(dest) then
9              add(lt, dest); ▷ If all predecessors of dst(e) have visited it, dst(e) is
                added in the queue.
10 procedure casIdleTime(startTimeDif, curIT, maxIT) ▷ Check and set remaining
   idle time. Raise an exception if curIT > maxIT.
11 function isThereAPEIdlingBefore(lPEs, deadline)   ▷ Returns true if one or
   multiple PEs idle before the deadline.
12 function possibleAllocationsBefore(lt, deadline) ▷ Returns tasks in lt which can
   start before the given deadline, ensuring that the selected tasks total
   execution time is not higher than the current idle time to the given deadline.
13 procedure
   allocateAndRemoveIfBefore(lt, τ, lPEs, deadline, maxIT, curIT, nbAllocs)
14     pesHead ← head(lPEs);
15     startTime ← max{predFinishTime(τ), finishTime(pesHead)};
16     if startTime > xs(τ) then
17         raise Scheduling failed.;
18     if startTime + C_τ > deadline then
19         return ;
       ▷ Various updates:
20     casIdleTime(startTime − finishTime(pesHead), curIT, maxIT);
21     finishTime(τ) ← startTime + C_τ;
22     nbAllocs ← nbAllocs + 1;
23     add(pesHead, τ, startTime);      ▷ Side effect: may modify the order of lPEs.
24     addReadyTasks(lt, τ);
```

Algorithm 3.3: Partially periodic scheduling of tasks

```
1  procedure schedule(tasks, graphPeriod, m)
2      forall τ ∈ tasks do                          ▷ Sets initial properties of each task.
3          nbVisits(τ) ← 0;
4          predFinishTime(τ) ← ns(τ);
5      maxIT ← m × graphPeriod − Σ_τ C_τ;
6      curIT ← 0;
7      nbAllocs ← 0;
8      lt ← list of currently ready tasks (initially without incoming edges), always
         maintained by increasing average start time i.e. (xs + ns)/2, and, if tie, by
         increasing ns;
9      lPEs ← list of schedules, one per PE, always maintained by increasing finish
         time;
10     while lt ≠ ∅ do
11         τ ← head(lt);
12         prevNbAllocations ← nbAllocs;
13         if isThereAPEIdlingBefore(c, predFinishTime(τ)) then   ▷ predFinishTime
             is the finish time of direct predecessors.
14             forall τ_b ∈ possibleAllocationsBefore(lt, lPEs, predFinishTime(τ)) do
15                 allocateAndRemoveIfBefore(
                     lt, τ_b, lPEs, predFinishTime(τ), maxIT, curIT, nbAllocs);
16             if prevNbAllocations < nbAllocs then
                   ▷ We restart the loop since new tasks may be ready now:
17                 continue ;
18         allocateAndRemoveIfBefore(lt, τ, lPEs, ∞, maxIT, curIT, nbAllocs);
```

of actors in G. However, it is not the case in the applications we have considered. Besides, note that the use of the parameter p in the Chapter 2 may limit this worst case complexity since p controls the degree of parallelism, and so the repetition vector. The transformation from G to G^* is a standard transformation, already implemented in the PREESM tool for example.

3.4.2 Standard ILP formulation with Choco

A scheduling model using ILP formulation has been developed in order to compare its performance with Algorithm 3.3. Although the formulation is purely ILP, the generic Choco[4] CP solver has been used for technical simplicity (especially for reification possibility, and for simple integration in PREESM). Also, note that there is no objective function in the formulation: the goal is only to test if there exists a valid schedule for a given number of PEs m. Choco stops on the first valid schedule encountered, and otherwise enumerates all possible schedules in order to prove that there is no solution.

The model uses 9 multi-dimensional arrays of variables in total. Only the first two arrays contain free variables[5].

V-3.1 start times of each task (free Integer), dimension $\Theta(\#V^*)$;

V-3.2 PE mapping of each task (free Boolean), dimension $\Theta(m \times \#V^*)$;

V-3.3 transpose matrix of V-3.2 (non free Boolean);

V-3.4 finish times of each task (non free Integer), obtained from V-3.1 plus WCETs;

V-3.5 if two tasks are on the same specific PE (non free Boolean), obtained from V-3.2 and V-3.3, dimension $\mathcal{O}(m \times \#V^* \times \#V^*)$;

V-3.6 if task τ_1 starts before τ_2 finishes (non free Boolean), obtained from V-3.1 and V-3.4, dimension $\mathcal{O}(\#V^* \times \#V^*)$;

V-3.7 if two tasks overlap temporally (symmetrical non free Boolean), obtained from V-3.6, dimension $\mathcal{O}(\#V^* \times \#V^*)$;

V-3.8 if two tasks are on the same unknown PE (symmetrical non free Boolean), obtained from V-3.5, dimension $\mathcal{O}(\#V^* \times \#V^*)$;

[4] http://www.choco-solver.org/

[5] A *free* variable has no predefined value and is set directly by the solver. A *non free* variable is defined by equations depending on free or non free variables. The solver sets the non free variables according to the equations defining it.

V-3.9 if two tasks overlap temporally and are on the same unknown PE (symmetrical non free Boolean, always false), obtained from V-3.7 and V-3.8, dimension $\mathcal{O}(\#V^* \times \#V^*)$.

The model size is bounded by the size of the transient Boolean matrix V-3.5 storing mapping of each couple of tasks: $\mathcal{O}(m \times \#V^* \times \#V^*)$. A transitive closure of the DAG G^* is computed before the model construction in order to reduce the size of this Boolean matrix. The transitive closure prevents from adding useless variables and redundant constraints to all transient matrices (variables V-3.5 to V-3.9) used to check mapping overlap: overlap between two tasks is checked only if there is no transitive precedence between the two tasks. The number of constraints is reduced by up to 16% thanks to the transitive closure. Last but not least, symmetries of the identical PEs are broken by enforcing some properties on the mapping matrix V-3.2, see [TPM13].

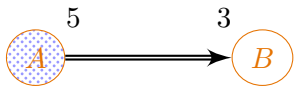
3.4.3 Scheduler adaptation to extended periodic constraints

Our scheduler can actually be adapted to handle a wider range of periodic constraints. For example, if considering specific deadlines instead of implicit ones, only the upper bound of the start time interval in Equation (3.1) needs to be refined with the given deadline. The deadline has to be greater than the WCET of its related periodic actor. The SRSDF graph traversal to compute the minimum and maximum start times ns and xs remains the same, using Equation (3.1) updated with a deadline. Algorithm 3.3 is not modified. The necessary conditions can still be used without any change, but as they do not consider the deadlines, they will be loose if an actor deadline is lower than its period. On the contrary, if an actor deadline is greater than its period, the necessary conditions are not *necessary* anymore.

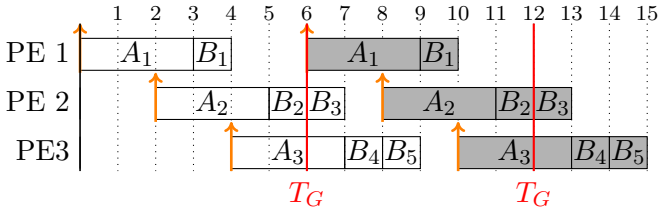
Another improvement is to consider the case of periodic actors having a WCET greater than their period: $T_\pi < C_\pi$. In such case, the interval of start times given by Equation (3.1) does not hold anymore since the upper bound is negative while the lower bound is positive. Simply removing the term C_π from the original equation removes the problem, but C_π has to be removed from the equation only for periodic actors π having $T_\pi < C_\pi$. Then, Equation (3.1) is modified into the following Equation (3.8).

$$[\![kT_\pi; (k+1)T_\pi[\![, \text{with } k \in [\![0; \vec{r}[\pi][\![\tag{3.8}$$

Note that Equation (3.8) enables auto-concurrency of such periodic actors, whereas the original Equation (3.1) disables it. Figure 3.11 gives an example for the periodic actor A.



(a) SDF graph example, of repetition vector $[3, 5]^T$.



(b) Possible schedule of 3.11a on three PEs. Here, periodic actor A has a period smaller than its WCET: $T_A = 2 < C_A = 3$. Assumption 1 is not respected.

Figure 3.11 – Schedule with an actor period smaller than its WCET.

In this case, multiple periodic executions of actor A overlap in time across two scheduler iterations and Assumption 1 cannot be respected. Regarding our scheduler, the graph traversal to compute the interval of start times has to be modified since xs is given by the ALAP ordering until the barrier of Assumption 1. Instead of the barrier, another limit can be used since the overlap of scheduler iterations occur during at most one graph period T_G. Then, xs is computed with the ALAP ordering given the limit of $2 \times T_G$, instead of $1 \times T_G$ in the original version. Moreover, the checking of the remaining idle time until the barrier, as done in lines 10 and 20 of Algorithm 3.2 cannot be used anymore and this function call must be removed. Last but not least, Algorithm 3.3 now has to ensure that overlap in time does not occur on the same PE. This is made possible by fixing an end time limit per each PE, set to the start time of the first firing allocated on it plus one graph period T_G. Multiple parts of Algorithms 3.2 and 3.3 have to modified to reflect this time limit per PE. Finally, it is possible to adapt the scheduler to the case of periodic actors π having $T_\pi < C_\pi$ at the cost of heavy modifications. However, the necessary conditions are not anymore valid for this case.

3.5 Evaluation

This section discusses how Algorithms 3.1 and 3.3 can be used in the design process of real-time systems, and presents an evaluation.

3.5.1 Partially periodic real-time applications

Only a few real-time use-cases are presented in the literature as *partially periodic* SDF graphs. Indeed it is often assumed that every component is periodic in order to ease the analysis and the code generation. Due to the lack of available partially periodic implementations, and to the simplicity of the existing ones, our algorithms have been evaluated on synthetic examples only. However, two small real examples are given hereafter. A pacemaker [Pel+09b] has been studied and modeled with the Architecture Analysis and Design Language (AADL)[6] and it is a partially periodic system. A critical subpart of this pacemaker is described using the CSDF model [Bil+96], an extension of SDF. In this subpart, two sensors (Motion and EKG) periodically send data to a processing component, each with its own period. A second example is in the telecommunication domain: the LTE standard has been studied and partially modeled with SDF graphs [Pel10]. In the LTE standard, the signals retrieved by the antenna are downsampled and periodically sent to the decoder. To the best of our knowledge, no open source benchmark exists that is explicitly partially periodic. Yet, the StreamIt [TKA02] benchmark contains dozens of signal processing applications in the SDF model. Periods are not specified in StreamIt but signal processing applications usually have one periodic input actor and another periodic output.

Finally it is possible to generate random SDF graphs with SDF^3 [SGB06b] and Turbine [Bod+14], but they do not generate partially periodic constraints. Thus, our experiments, detailed in Sections 3.5.3 and 3.5.4, have been performed on ten random DAGs generated by Turbine, and on one existing use-case of the SDF^3 data set. For the first experiment, partially periodic constraints are added to these generated graphs as a post-processing step that follows the graph generation.

3.5.2 Schedulability check and scheduling

The practical usage of the presented algorithms is summarized in Figure 3.12. In terms of the number of PEs m, the necessary conditions (Algorithm 3.1) give a schedulability lower bound (left part) while the scheduler (Algorithm 3.3) gives an upper bound (right part). For example, the lower bound is 4 PEs while the upper bound is 7 PEs in Figure 3.12. With less than 4 PEs, the application is proven to be *not* schedulable; with 7 PEs or more, the application is proven to be schedulable.

The algorithms have to be run iteratively with different values of m to shrink the bounds. The starting point of our necessary condition is given by the processor utilization

[6] http://www.aadl.info

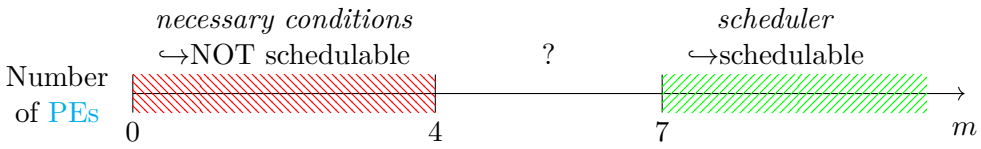


Figure 3.12 – Scheduling bounds on the number of PEs m.

factor necessary condition in Equation (3.3); the number of PEs is then incremented until our necessary condition returns true. This process could be automatized easily without execution time issue since both algorithms are fast.

Algorithms 3.1 and 3.3 and the optimal Choco model have all been implemented in the PREESM open-source[7] tool dedicated to the design of embedded systems from applications modeled with SDF graphs. Regarding the scheduler, note that it can actually be run for any unconnected DAG and thus is more generic than for SDF graphs with partially periodic constraints.

PREESM automatically generates the SRSDF graph G^* from a given SDF graph; the generation time of G^* is not included in the experiments. However, PREESM automatically adds *fork* and *join* tasks in G^* during the transformation, to respectively split and reassemble data to and from multiple firings of the same actor. These extra tasks are taken into account in the experiments, with a WCET equal to 1.

The next subsections report two experiments done on the algorithms presented in this chapter. Both experiments have been performed on a laptop with an Intel i7-7820HQ @ 2.90GHz processor.

3.5.3 Gap between necessary conditions and scheduler

This experiment measures the gap between the proposed necessary conditions, the proposed scheduler, and the optimal solution. The gap is measured in number of PEs required to synthesize a schedule while respecting all periodic constraints.

Dataset

The dataset contains ten random SDF DAGs generated with the Turbine tool [Bod+14]. Table 3.1 details the characteristics of the generated graphs. The first five graphs are small, with only ten actors, in order to make possible comparison with the optimal number of PEs computed by a CP solver. The generated graphs do not contain any

[7] https://preesm.github.io/

cycle nor delay. The degree, also called valency, of each actor α in the graphs is between 1 and 8: $1 < \#OE(\alpha) + \#IE(\alpha) < 8$.

One periodic actor π is set in the middle of the longest path of each SDF graph; largest number of firings breaks the tie between multiple possible actors. A second periodic actor is set on RandomDAG6-10. Their periods T_π are set manually, being the smallest integer such that a solution exists. Formally, T_π is the smallest integer which ensures: $\forall \tau \in G^*,\ \mathrm{ns}(\tau) < \mathrm{xs}(\tau)$.

Evaluation results

The evaluation has been performed as follows. For each scheduling algorithm, the result is the smallest number of PEs ensuring a valid schedule. For the necessary conditions, the result is the smallest number of PEs ensuring that no valid schedule exists for all lower number of PEs. All algorithms are run iteratively with an increasing number of PEs. Diagram in Figure 3.13 presents the results for the five small random graphs. The processor utilization factor *Utot* of the graph is given as reference on the left column. The execution time of the algorithms are given in the right part of Table 3.1 (in millisecond ms, and hour h). While the proposed algorithms always run in less than a second, Choco takes hours for the small graphs.

Choco is optimal but cannot solve problems with too many firings or PEs. It actually reaches timeout (T/O) of 12 hours for RandomDAG1 and RandomDAG4 with 9 PEs, and it also takes multiple hours to prove that the same graphs have no solutions for 8 PEs. The timeout of Choco is specified by an error interval, materialized by a small black line in Figure 3.13. The gap between the optimal solution and the proposed scheduler is at most two PEs, for RandomDAG4, and at best zero as for RandomDAG5.

Results for the five large graphs are depicted in Figure 3.14, without comparison with Choco because of the size of the problem (it would timeout in any case). On all ten graphs, the necessary conditions appear to be weakly discriminating: in most of the case, it states that the system is possibly schedulable as soon as $m > U_{tot}$. The necessary conditions are discriminating only for RandomDAG1 and RandomDAG6. Note that RandomDAG1 and RandomDAG6 are also the graphs requiring a longer execution time of the proposed scheduler. Yet two reasons may increase the complexity of the scheduler: more tasks ready at the same time (which increases the size of the sorted list l), or more idle time (which triggers the execution of lines $13 - 17$ in Algorithm 3.3). Further investigations are needed to characterize this phenomenon. Moreover, it could be interesting to study the evolution of the necessary conditions and the scheduler while

Name	#actors	WCET (average)	#firings	#deps. in G^*	$\vec{r}[\pi]$	T_G	Time Alg.3.1	Time Alg.3.3	Time Choco
RandomDAG1	10	100	141	280	5	2445	7 ms	27 ms	T/O (12 h)
RandomDAG2	10	100	184	373	7	4270	4 ms	13 ms	320327 ms
RandomDAG3	10	100	139	370	13	9412	4 ms	10 ms	993116 ms
RandomDAG4	10	100	143	329	3	1557	3 ms	11 ms	T/O (12 h)
RandomDAG5	10	100	136	251	3	1377	4 ms	7 ms	9046 ms
RandomDAG6	100	200	3226	6093	7/4	13496	10 ms	134 ms	–
RandomDAG7	100	200	2824	6239	5/10	15040	6 ms	68 ms	–
RandomDAG8	100	200	2978	6341	7/7	11711	5 ms	61 ms	–
RandomDAG9	100	200	2567	5600	6/6	8496	10 ms	89 ms	–
RandomDAG10	100	200	3358	6535	10/10	16680	8 ms	15 ms	–

Table 3.1 – Details of the random directed acyclic SDF graphs generated by Turbine, and execution time of the algorithms. RandomDAG1-5 contain one periodic actor π. RandomDAG6-10 contain two periodic actors.

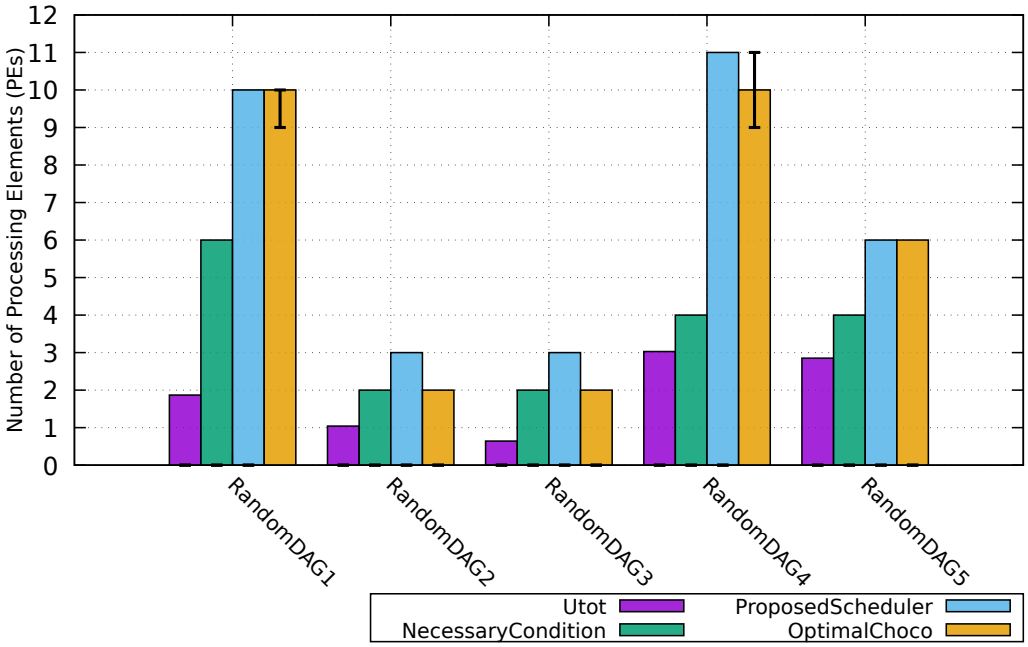


Figure 3.13 – Evaluation of the schedulability gap on the small random graphs RandomDAG1-5. Number of PEs is given for the four following algorithms (from left to right): total processor utilization factor, lower bound according to our necessary conditions, upper bound according to our scheduler, valid minimum according to the optimal ILP formulation.

increasing the number of periodic actors. In our experiments, only one and two periodic actors were considered.

3.5.4 Gap between the proposed scheduler and preemptive EDF

This experiment measures the graph period gap between the proposed non-preemptive offline scheduler Algorithm 3.3 and a standard preemptive real-time scheduler: EDF. The ADFG tool [Hon+17] is used as a reference, using Global EDF scheduling [BB11] with a synthesis algorithm adapted from the forced-forward demand bound function. ADFG considers that all actors are periodic and computes their optimal smallest period for a given number of PEs. Other synchronous languages and tools could have been used to perform the experiments, especially to compare with the non-preemptive fully periodic case. ADFG has been considered to ease the experiments since both ADFG and PREESM read the SDF^3 file format [SGB06b]. Moreover, ADFG also computes delays on the buffers; these delays are kept in the input of Algorithm 3.3. Then Algorithm 3.3

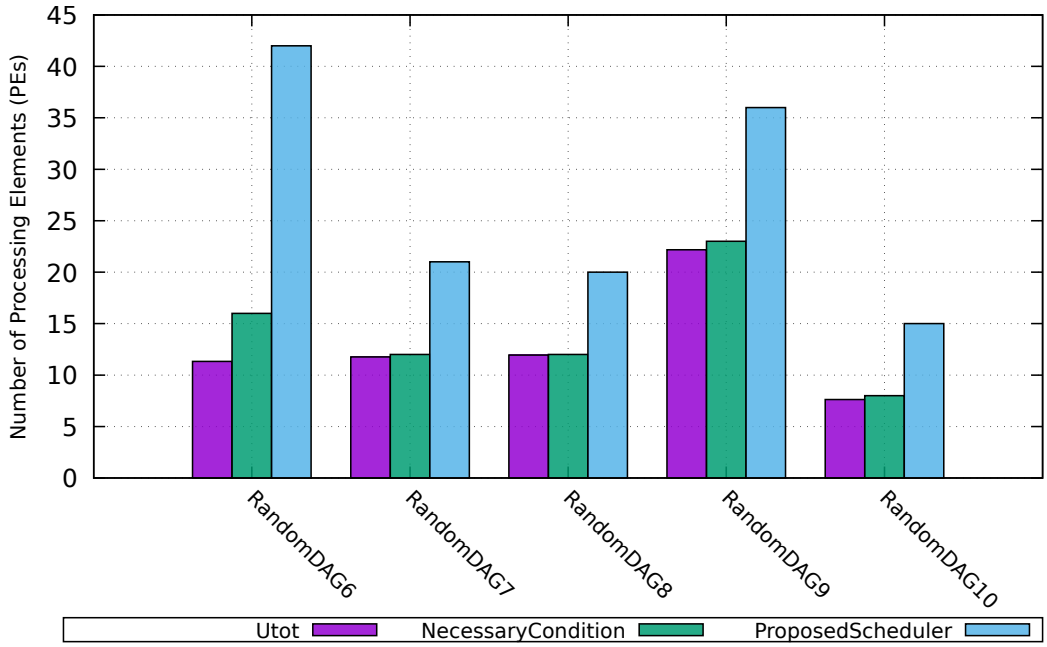
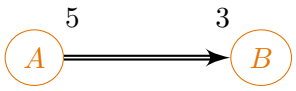


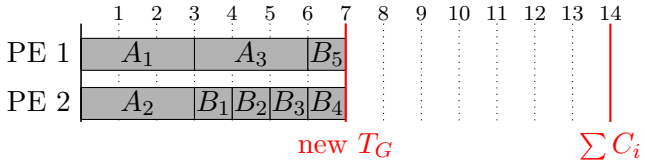
Figure 3.14 – Evaluation of the schedulability gap on the large random graphs RandomDAG6-10. Number of PEs is given for the three following algorithms (from left to right): total processor utilization factor, lower bound according to our necessary conditions, upper bound according to our scheduler.

is initially run with a graph period equal to the sum of all WCET, and the latest finish time of all firings found by Algorithm 3.3 is kept as a result. Indeed this last finish time is the smallest graph period achievable by the proposed scheduler and it may be smaller than the input graph period since Algorithm 3.3 is greedy and uses idle time as soon as possible. Figure 3.15 illustrates this refinement process. The refinement of the period does not modify the schedule: all maximum start times xs are reduced by the same amount (oldT_G − newT_G), while the minimum start times ns and the ordering by increasing average start time remain unchanged.

Results are depicted in Figure 3.16 for the Beamformer application from the StreamIt benchmark. Beamformer graph contains 57 actors and 70 edges. Each actor is fired only once, thus ensuring a fair comparison since ADFG does not handle auto-concurrency. WCETs of actors in Beamformer are already fixed in the graph file provided by ADFG. For each number of PEs, Algorithm 3.3 finds a graph period close to the optimal, and even optimal from 28 PEs. The optimal graph period is equal to the greatest WCET (5076) since numerous delays are added by ADFG and break the data dependencies.



(a) SDF graph example, of repetition vector $[3, 5]^T$.



(b) Schedule example of 3.15a, respecting the graph period $T_G = \sum C_i$. The graph period is refined to 7 since no task is executed after this limit.

Figure 3.15 – Graph period refinement example.

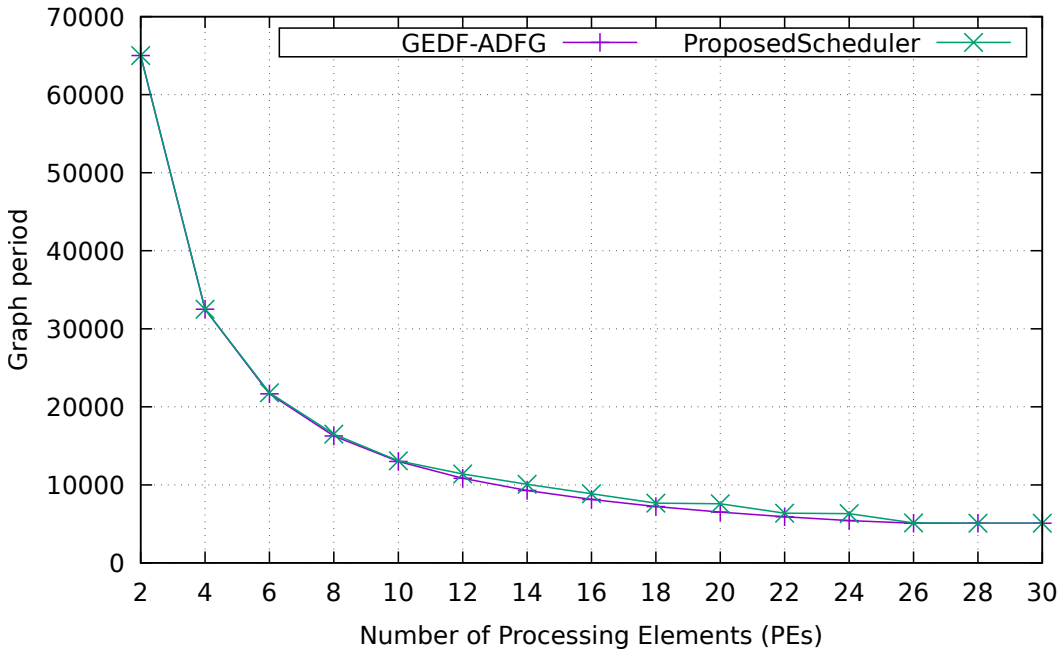


Figure 3.16 – Evaluation of the graph period computed by Algorithm 3.3 compared to ADFG with global EDF policy. Number of PEs in abscissa.

3.6 Related work

Our model of partially periodic constraints is a restriction of the PolyGraph model [Dub+19] to SDF graphs, but extended with deadlines. According to the PolyGraph model [Dub+19], $\mathcal{P}$ corresponds to actors having a *frequency*.

To our knowledge, only the MAPS [CLA13] tool accepts SDF graphs with partially

periodic constraints as specified in this chapter. However, MAPS does not exactly compute a schedule, but instead it checks if execution traces can be executed on the target architecture. As MAPS is not freely accessible, we were not able to compare to it. Deadlock [DRL16] and consistency [Dub+19] analysis of SDF graphs with partially periodic constraints have also been studied. Another work [Lou19], related to the PolyGraph model, shows an ILP formulation taking into account communication time to schedule partially periodic SDF graphs. The work [Lou19] solves a more complex scheduling problem than us since they take into account communication time, but is not as scalable and fast as our scheduling Algorithm 3.3 since they rely on ILP formulation.

Other tools and papers are closely related to our scheduling problem and are listed in the three next paragraphs, according to their category. We distinguish three categories of schedulers, dedicated to: 1) tasks with precedence and real-time constraints, 2) SDF graphs with periodic constraints, 3) SDF graphs with latency constraints.

Schedulers of real-time tasks with precedence constraints Real-time systems have been widely studied for online periodic scheduling, the most common schedulers being EDF and FP. Yet an offline part is still needed in most of the online schedulers: either to compute the deadlines as in the Chetto's algorithm [CSB90] to respect precedence constraints under EDF, or to compute the task priorities in the case of FP. Some online schedulers also take into account periodic and aperiodic tasks [LB00]; they still may need an offline pre-schedule [Foh95], and may rely on EDF [IF00]. Regarding offline non-preemptive scheduling, there exists an ILP formulation [XAP17] for sporadic and periodic tasks under EDF and FP. A CP solution for periodic tasks only has also been formulated [PNP15]. Both ILP and CP formulations have a high complexity and thus are not scalable.

Schedulers of SDF graphs with periodic constraints The Darts tool [BS11] is able to schedule SDF graphs under a throughput constraint, equivalent to a graph period constraint, for EDF and FP schedulers. ADFG [Hon+17] is similar to Darts, but it optimizes the throughput under a total buffer size constraint. SDF graphs can be modeled with synchronous languages such as Prelude [Pag+11], which generates code for EDF and FP schedulers. Still using synchronous language, activation clocks with precedences [Coh+06] can be composed and checked. Yet, in all the aforementioned tools of this paragraph, all SDF actors are periodic. Minimal actor periods can be computed independently from the scheduling policy [AAP15], but only for SRSDF graphs. Note that a polynomial algorithm [SEB18] exists for the unicore processor case under EDF,

with real-time tasks being specific SDF graphs. Other papers [BL06] specify throughput constraints on the only input or output actor of an SDF graph G, which is equivalent to specify a graph period T_G.

Schedulers of SDF graphs with latency constraints A latency constraint on an SDF graph G is equivalent to a throughput constraint or graph period if and only if the scheduler assumes Assumption 1 and there is no delay in G (except to break cycles). Indeed if delays are present, as in Figure 3.2, the latency may be higher than the graph period. Latency constraints for SDF graphs have been heavily studied, especially symbolically. For example, the latency has been analyzed either without scheduling assumption to derive upper and lower bounds [Kha+16], or with self-time scheduling of SDF [Gha+07] and SRSDF [MB07] graphs. Practically, the Ptolemy [Eke+03] tool supporting SDF graphs has been extended to perform timing verification, as latency, through system simulation [Guo+14]. Finally, there exists an offline scheduler accepting throughput and latency constraints on SDF graphs [LGE12]; it takes into account communications and computes the static schedule with ILP and heuristics. Thus all these tools tackle only a small subset of partially periodic constraints: the specific case of one graph period without any delay on the graph.

3.7 Conclusion

A few necessary conditions and an offline non-preemptive scheduling algorithm have been presented in order to analyze and synthesize the schedule of partially periodic SDF graphs. These results hold under a weak assumption on the execution of the systems: the presence of barriers at each graph period. The proposed algorithms have, at most, a linearithmic complexity and can thus be used on large cyber-physical systems modeled with SDF graphs. Experiments show that the proposed non-preemptive scheduler is fast, scalable, and efficient.

Next step is to extend the schedulability analysis to heterogeneous multi-processors, that are becoming the new standard for embedded systems. When targeting heterogeneous multi-processors, it is important to take into account communications, especially if the memory is distributed. A few efficient heuristics already exist and address the scheduling problem of DAGs on heterogeneous hardware taking into account communications, such as the HEFT [THM02] list scheduling algorithm. However HEFT uses average communication time while we need the worst case for the periodic constraints. A first naive idea is to map tasks on the slowest PE when they are allocated in the

greedy mode, that is when tasks are not located on the critical path (lines $13-17$ of Algorithm 3.3). When not in the greedy mode, we could use a best-fit mapping taking into account the communication time.

Another direction for future work is to take into account not only periodic constraints, but also latency constraints. For example, if an application contains both sensors and actuators as SDF actors, it might be necessary to guarantee a maximum time interval between a sensor measurement and the corresponding actuator response. In the SDF model, such latency constraint raises the following question: which firing must be considered to compute the latency between two actors? Moreover, can this latency constraint be set across multiple iterations of the whole application? Scheduling tasks according to such constraints is tedious since latency constraints may be localized inside a part only of the application graph and would require a kind of local priority when allocating all tasks of this part.

Thanks to its rapidity and scalability, the proposed scheduler is fast enough to be used in brute-force DSE. This point will be demonstrated in Chapter 5. Finally, the presented necessary conditions made a few assumptions on cycles, which compulsorily have to contain delays on one buffer. In Chapter 4, we detail a heuristic to automatically place delays on regular buffers, and also in cycles.

Dissemination and Implementation

The contribution presented in this chapter has been published in the RTNS'20 conference [Hon+20a]. The algorithms presented in this chapter have been implemented as workflow tasks of the PREESM tool. See the following task description for the implementation of Algorithm 3.1:

- `org.ietr.preesm.pimm.algorithm.checker.periods.` \
  `PeriodsPreschedulingChecker`

See the following task description for the implementation of Algorithms 3.2 and 3.3:

- `pisdf-mapper.periodic.DAG`
- `pisdf-synthesis.void-periodic-schedule`
- `pisdf-synthesis.simple`

Chapter 4

Pipelining SDF graphs automatically

Contents

4.1 Admissible graph cuts for pipelining 97
4.1.1 Properties of admissible cuts and definitions 97
4.1.2 An ILP formulation to compute all admissible cuts 98
4.1.3 Initialization of delays 101
4.1.4 Impact on scheduling 102
4.2 Automatic pipelining of SDF graphs 103
4.2.1 Computing topological graph cuts 103
4.2.2 Selecting best topological graph cuts 106
4.3 Automatic cycle breaking of SDF graphs 108
4.4 Evaluation 111
4.4.1 Theoretical throughput gain: regular applications 113
4.4.2 Theoretical throughput gain: widely parallel applications . . . 114
4.4.3 Practical experimentation 116
4.5 Related work 117
4.6 Conclusion 119

Introduction

A common way to represent pipelining in SDF graphs is to add *delays* on buffers. Delays represent initial data in buffers, which break data dependencies and create *pipeline stages*. This method has already been proved to be efficient on SDF applications [LM87a] but usually requires to add the delays manually in the SDF graph or to call heuristics [KM08]. Indeed, computing the optimal throughput of an application is a problem of high complexity that also requires computing the scheduling, the mapping and the pipelining.

In this chapter we propose a fast heuristic to automatically pipeline an application modeled with an SDF graph. Our pipelining heuristic computes the size and the placement of delays on buffers of any SDF graph. The placement corresponds to an *admissible cut* of the graph; it is a set of buffers. Delays are initial data present on the buffers before starting to execute the application. The token values that the delays carry are not computed, but they can remain undefined in the case of SDF DAGs. Following the semantics introduced in [Arr+18], we assume all delays being permanent, which means that their token values are transmitted from one scheduler iteration to the next. The heuristic is performed before mapping and scheduling the application, and is thus suboptimal but fast and scalable. As a consequence, any scheduler may benefit from the pipelining heuristic, including the one developed in Chapter 3 for partially periodic constraints. As an input, the heuristic requires the number of targeted homogeneous PEs and application profiling information, i.e. the Execution Times (ETs) of actors. The heuristic is parameterizable: the user chooses the number of pipeline stages that he wants to add. Various experiments demonstrate that the heuristic increases the throughput of the majority of the tested applications. When adding one pipeline, the heuristic finds the solution ensuring optimal throughput for 19 applications out of 24 tested.

The chapter is organized as follows. Section 4.1 defines the notion of pipeline for SDF graphs as well as related properties. Equations to assert the validity of the pipelines are also presented. The main contribution, automatic pipelining of SDF graphs, is developed in Section 4.2. A secondary contribution, about automatic cycle breaking is detailed in Section 4.3. Extensive experiments follow in Section 4.4 with both theoretical evaluation of the throughput gain and actual measurements on hardware. The main drawback of pipelining, memory footprint increase, is also quantified. Related work is presented in Section 4.5. Finally, Section 4.6 concludes this chapter.

4.1 Admissible graph cuts for pipelining

Admissible graph cuts for pipelining correspond to *feed-forward* graph cuts as defined for the design of integrated circuits [Par07]. Main properties of admissible graph cuts are presented in Section 4.1.1. An ILP formulation to compute all admissible cuts of an SDF graph is detailed in Section 4.1.2. The initialization of token values carried by the delays are discussed in Section 4.1.3. Finally, Section 4.1.4 precises the advantages and drawbacks of pipelining under the specific case of scheduling Assumption 1.

4.1.1 Properties of admissible cuts and definitions

A graph cut is a set of edges which, if removed, disconnects the graph in two or more components. In a feed-forward graph cut, all edges of the cut are going in the same direction. Hence, such cut cannot contain an edge from a cycle. The direction of an edge is deduced from the topological ranks of the actors. An example of actor ranks of ASAP and ALAP topological orderings is presented in Figure 4.1a. Lowest actor rank 1 correspond to actors without input buffers, and the highest actor rank correspond to actors without output buffers. Note that these topological ranks differ from Figure 3.8 in Chapter 3: in the present chapter ranks start at value 1, and the transpose graph is not used[1]. The topological orderings are computed on the raw SDF graph without taking into account the presence of delays.

A pipeline is created by adding delays on all buffers of a feed-forward graph cut, in order to break the data dependencies. For example, the feed-forward graph cut (dashed line) between topological ranks 1 and 2 of the graph in Figure 4.1a breaks the data dependencies between actors A and B, and A and Δ. The pipeline increases the throughput of the graph, as depicted in Figure 4.1c. In this schedule example and in the next ones, we assume that there is a synchronization barrier, represented by a red vertical line, at the end of each scheduler iteration: this is Assumption 1. The barrier is only used to simplify the examples; the heuristic presented in section 4.2 does not require it. Moreover, in Figure 4.1a as in all SDF graphs illustrating this chapter, production and consumption rates are all equal to 1 for simplification. This implies that all SDF graphs in this chapter have their repetition vector $\vec{r} = \vec{1}$; single arrows emphasize this property. Nevertheless, all results remain valid for any rate value.

We define the throughput of an SDF graph as the inverse of the *II* duration, that is the duration to periodically execute one *scheduler iteration*. A scheduler iteration contains

[1] ASAP ordering is equivalent to the reversed ALAP ordering of the transpose graph. However, we use here a slightly modified version of ALAP, as it will be detailed in Section 4.2.1.

as many firings as specified in the repetition vector $\vec{r}$. On the left part of Figure 4.1c, without pipeline, the II duration is 3, whereas on the right part, the II duration is only 2 thanks to the pipeline. Note that, depending on the topological ordering, the graph cuts may not be identical.

Scheduler iterations differ from *graph iterations*. Both of them contain as many firings as specified in the repetition vector $\vec{r}$, but a graph iteration contains the firings in the order respecting the data dependencies broken by the delays. A graph iteration corresponds to the end-to-end processing of a given data in the graph. This implies that graph iterations are distributed over multiple scheduler iterations when pipeline delays are present. See Figure 1.4c for an example were one graph iteration is spread over two scheduler iterations.

To create one pipeline on an SDF graph, the size of a delay on a buffer e must be equal to the number of tokens consumed on e during a whole graph iteration, as specified in Equation (4.1). It is the direct application of the consistency property defined in Equation (1.1).

$$d_0(e) = \text{prod}(e) \times \vec{r}[\text{src}(e)] = \text{cons}(e) \times \vec{r}[\text{dst}(e)] \tag{4.1}$$

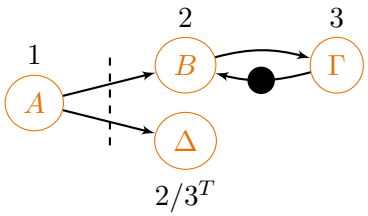
Thus, dependencies between all firings of producer actor $\text{src}(e)$ and receiver actor $\text{dst}(e)$ are broken. If multiple feed-forward graph cuts contain the same buffer, the delay sizes are summed. In this chapter, a *pipeline* is a synonym for feed-forward graph cut, and n pipelines divide the execution of the application in $n+1$ *pipeline stages*. The number of pipeline stages actually correspond to the number of scheduler iterations needed to complete one graph iteration.

4.1.2 An ILP formulation to compute all admissible cuts

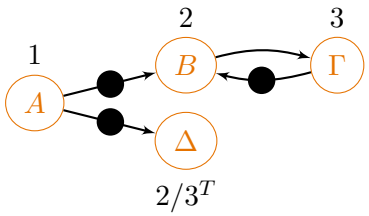
Thanks to Michael Masin and his team[2], we were able to formulate the admissible feed-forward cut constraint as the following recursive Equation (4.2). The equality must be respected for every actor $\alpha \in V$. It introduces the notion of *actor delay*, denoted $\daleth$[3]. An actor delay corresponds to a shift of data on all its input. The unit of the actor delay function $\daleth$ is the number of pipeline stages: if $\daleth(\alpha) = 2$, there are two pipeline stages until actor α.

[2] This PhD has been partly founded by the European Union H2020 project Cerbero. Michael Masin was the leader of the Cerbero project, while working at IBM Research Labs in Haifa (Israel). We also partly worked together on the entanglement problem discussed in Section 5.2.

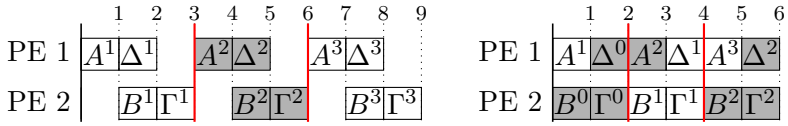
[3] $\daleth$ literal name is "daleth".



(a) Graph example annotated with ASAP and ALAP topological orderings. ALAP ordering is specified with ^{T}, only if different from ASAP.



(b) Same graph with pipeline delays added on the cut represented with a dashed line on 4.1a.



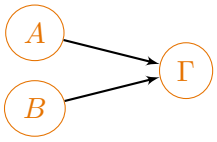
(c) Two schedule examples of graph 4.1a on two PEs: on the left without pipeline, on the right with one pipeline between ranks 1 and 2. Firing exponents denote their graph iteration.

Figure 4.1 – Topological ordering and schedule example without and with pipeline.

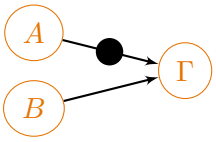
$$\forall e \in \{b \in E | \, \mathrm{dst}(b) = \alpha\}, \neg(\alpha) = \neg(\mathrm{src}(e)) + \frac{d_0(e)}{\mathrm{cons}(e) \times \vec{r}\,[\alpha]} \tag{4.2}$$

The recursion stops on actors having no incoming buffer, where the actor delay is set to 1 by default: such actors are executed during the first pipeline stage. As in Chapter 3, it is assumed that the user sets enough delays on at least one buffer of any cycle. These buffers are ignored for the generation of the ILP constraints, otherwise the formula Equation (4.2) would imply an indefinite recursion.

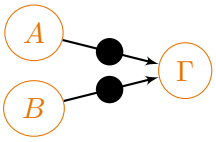
Valid and invalid delay placement examples are given in Figure 4.2 and illustrate the preceding Equation (4.2). Let's consider that actors A and B both produce one Integer equal to the number of times they have been executed so far. Actor Γ checks that both Integers have the same value. When executing the original graph on Figure 4.2a, Γ consumes successively $(1,1)(2,2)(3,3)(4,4)\ldots$ Tuples (i,j) represent the head of its two incoming buffers, from actor A for i and actor B for j. When executing graph with the misplaced delay carrying the initial value 0, on Figure 4.2b, Γ consumes successively $(0,1)(1,2)(2,3)(3,4)\ldots$ Finally, when executing graph with well placed delays both carrying the initial value 0, on Figure 4.2c, Γ consumes successively $(0,0)(1,1)(2,2)(3,3)(4,4)\ldots$ which only shifts the original output but does not modify it. Hence, admissible graph cuts ensure to only shift the application result without modifying it.



(a) Original graph without delay.



(b) ✘ Graph with misplaced delay (cannot result from an admissible graph cut).



(c) ✔ Graph with well placed delays (resulting from an admissible cut).

Figure 4.2 – Delay placement examples, resulting from invalid and admissible graph cuts.

The ILP formulation of Equation (4.2) contains two arrays of non free Integer variables plus one free Integer variable per buffer[4]. There is a fraction in Equation (4.2), however the denominator is a non zero Integer constant and both sides of the equation can be safely multiplied by it.

V-4.1 actor delay $\urcorner$ (non free Integer), dimension $\Theta(\#V)$;

V-4.2 buffer delay d_0 (free Integer), dimension $\Theta(\#E)$;

V-4.3 transient variable to express Equation (4.2) per buffer e, which must be equal to $\urcorner(\mathrm{dst}(e))$ of array V-4.1 (non free Integer), dimension $\Theta(\#E)$.

The main constraint formalized in Equation (4.3) is put on actors $\alpha \in V$ having no outgoing buffer ($OE(\alpha) = \emptyset$): the maximum value of their actor delay $\urcorner$ must be equal to the number of pipeline stages wanted by the user. There is no objective function since we want to list all admissible cuts.

$$\max_{OE(\alpha)=\emptyset} \{\urcorner(\alpha)\} = \#\mathrm{Stages} \tag{4.3}$$

In the implementation, we also force the delays to be a multiple of the pipeline size in Equation (4.1). Thus, the fraction in Equation (4.2) can be equivalently formulated with another free Integer variable q replacing d_0, as shown in Equation (4.4).

$$\forall e \in \{b \in E | \, \mathrm{dst}(b) = \alpha\}, \urcorner(\alpha) = \urcorner(\mathrm{src}(e)) + q, \quad q \in [\![1; \#\mathrm{Stages}]\!] \tag{4.4}$$

As the placement validity is defined recursively from the actors having no outgoing buffers, delays may be distributed over the whole paths going to an actor, and not only on its direct incoming buffers. Figure 4.3 illustrates this possibility.

[4]See Footnote 5 on Page 82 for a short definition of *free* and *non free* variables in our context.

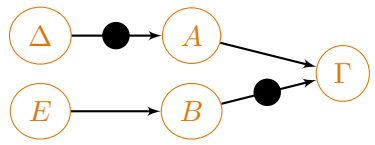


Figure 4.3 – Graph with valid delay placement distributed on the paths. There are two pipeline stages: $\urcorner(\Delta) = \urcorner(E) = \urcorner(B) = 1$ and $\urcorner(A) = \urcorner(\Gamma) = 2$.

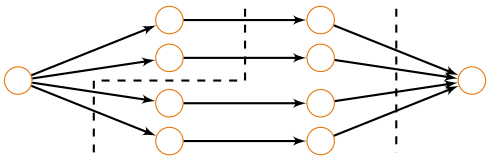


Figure 4.4 – Split-join graph with four parallel branches. 2 admissible cuts are represented with a dashed line, among 81 possible.

Unfortunately, the number of admissible graph cuts may be large. An example is given with a commonly used split-join graph topology [TPM13], which is a subcategory of SDF graph. Although the graph represented in Figure 4.4 only contains 4 parallel paths with 3 buffers each, $3^4 = 81$ cuts are admissible. Indeed, if k paths connect a split actor to a join actor, each path having b buffers, the total number of feed-forward graph cuts is equal to b^k. Because the number of admissible graph cuts may grow exponentially with the number of edges of the graph, exploring them all is not feasible. For this reason, our heuristic algorithm will only explore a subset of the admissible cuts. For example, our heuristic considers at most 3 admissible cuts for the graph in Figure 4.4. Those admissible cuts are detailed in the next section.

4.1.3 Initialization of delays

In the previous Section 4.1.2, it is stated in the explanations of Figure 4.2c that admissible cuts only shift the application result without modifying it. Yet, this is true only for SDF graphs being DAGs, such as Figures 4.2c, 4.3 and 4.4. Such graphs are said to be be *stateless* and the token values carried by the pipeline delays of admissible cuts can be undefined. Indeed, for the SDF DAGs, pipelining will only shift the application results and the first firings based on the undefined token values are executed but they will not modify the next results since there is no state.

However, cycles model such states: delays breaking the cyclic data dependencies carry the states of cycles, even across consecutive graph iterations. Then, any shift of input modifies the state of the cycle and so the application behavior. SDF graphs having cycles are said to be *stateful*. For example, it is the case of Figure 4.1b, between actors B and Γ. Thus, the token values of pipeline delays have to be chosen carefully when applying admissible cuts on SDF graphs having cycles; the token values have to be chosen by the designer so that the cycle states remain coherent with the original

application behavior. Fortunately, for some implementations such as SIFT, cycles are present but the initial token values of delays are discarded by the internal code of the actors thanks to a firing counter. More precisely, the actors may operate as a switch and select the value of an input buffer having a delay or not. In this specific case, even if the SDF graph of SIFT has cycles, the token values of delays breaking those cycles may remain undefined since those tokens are discarded by the internal code of the actors.

Finally, note that the semantics of SDF *Setter* actors [Arr+18] introduces the initialization of the token values of delays, set by the output of other actors in the graph. Symmetrically, this semantics also defines *Getter* actors which pop the token value of a delay to use it as an input of another actor in the graph. This semantics is especially useful to model the initialization of variables in `for` loops having dependencies between their iterations and to retrieve the value of the variables after the last loop iteration. Metaphorically, *Setter* actors are executed during the prologue of the application while the *Getter* actors are executed during its epilogue.

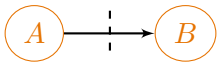
4.1.4 Impact on scheduling

Delays are a property of the SDF MoC by itself, independent to the scheduler used to execute any SDF graph. Thus, adding pipeline delays to an SDF graph can be performed without knowing the actual scheduler. In particular, pipelined SDF graphs are compatible with partially periodic constraints defined in Chapter 3, and they can be scheduled with the Algorithm 3.3.

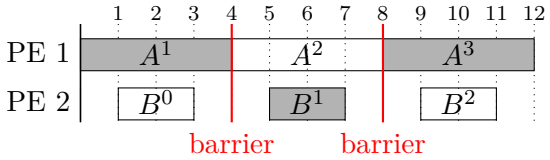
Actually, pipelining may be a way to diminish the drawbacks of Assumption 1 used by Algorithm 3.3. Assumption 1 implies the presence of a global barrier avoiding any overlap between the scheduler iterations. On the contrary, the main objective of pipelining is to cut the SDF graphs so that graph iterations are spread over multiple scheduler iterations, artificially supporting overlap even under Assumption 1. Ideally, the graph cuts are located on the parallelism bottlenecks of the graphs, in order to fill the potential idle time of the PEs which could be created by the parallelism bottlenecks. The idle time of the PEs can be filled with pipelined firings of other graph iterations. As a by-product, cuts should be selected so that the processor utilization factor U increases[5].

However, note that pipelining with delays is not always sufficient to avoid the drawbacks of Assumption 1. Figure 4.5 depicts an example where it is better, with regard to the throughput, to make the scheduler iterations overlap without Assumption 1 (Figure 4.5c, II duration is 3) than to pipeline the SDF graph executed under Assumption 1

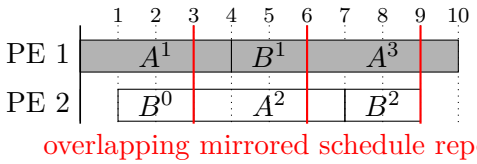
[5]See Equation (3.3) for the definition of processor utilization factor.



(a) Simple graph with two actors. The only possible cut for pipelining is represented with a dashed line.



(b) ✘ Schedule of 4.5a on two PEs *with* the barrier of Assumption 1: graph cut for pipelining does not help to reach the optimal throughput. Firing exponents denote their graph iteration.



(c) ✔ Schedule of 4.5a on two PEs *without* barrier and without graph cut: it increases the throughput compared with 4.5b. Firing exponents denote their graph iteration. Two static schedules (one and its mirror exchanging the PEs) actually alternate every 3 time units.

Figure 4.5 – Schedule example where pipelining does not compensate for the presence of a global barrier.

(Figure 4.5b, II duration is 4). Figure 4.5c, without pipeline delays, is the optimal solution; but in the case of indefinitely repeated static schedules, it requires to mirror the schedule every 3 time units (exchanging PE 1 with PE 2) and to discard B_0.

In the next section, we propose a heuristic to automatically pipeline SDF graphs. This heuristic tries to cut the graphs where the degree of data and task parallelism is low, to avoid idle time of the PEs.

4.2 Automatic pipelining of SDF graphs

The automatic pipelining heuristic has two main steps: (1) generation of all topological graph cuts, (2) selection of topological graph cuts. The first step, described in Section 4.2.1, computes a subset of admissible cuts. The second step, detailed in Section 4.2.2, selects a few cuts among the cuts computed in step (1).

4.2.1 Computing topological graph cuts

The heuristic selects a subset of admissible graph cuts: *topological* graph cuts according to the ASAP and ALAP topological orderings. A topological graph cut of rank cr contains all buffers coming from an actor of rank lower than cr and going to an actor of

rank higher than or equal to cr. Such topological cut is admissible if none of its buffers is part of a directed cycle of the graph.

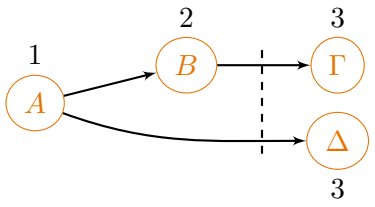
The number of admissible topological graph cuts is upper bounded by the diameter of the graph, that is the number of buffers on the longest path. For example, the graph depicted in Figure 4.4 admits only 3 topological cuts according to ASAP graph ordering, whereas this graph admits 81 admissible cuts in total. Moreover, in the case of Figure 4.4, ASAP and ALAP graph orderings are identical so the same 3 graph cuts are considered for both topological orderings.

In order to build the ASAP and ALAP topological orderings, a cycle analysis of the graph is run first: the Johnson's algorithm [Joh75] computes all simple cycles of a directed graph. Johnson's algorithm upper bounds the complexity of the whole heuristic. The buffers being part of cycles are recorded to later filter the admissible cuts. For example, the topological cut of rank 3 in the SDF graph depicted in Figure 4.1a is invalid since there is a cycle between actors B and Γ. If a cut contains at least one buffer belonging to a cycle, then the cut is not admissible.

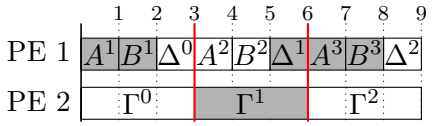
Note that it is assumed that the user sets enough delays on at least one buffer of any cycle, so that this buffer breaks the data dependency of the cycle. Alternatively, the heuristic presented in Section 4.3 can be used to automatically break the cycle dependencies. Thanks to this assumption, ASAP and ALAP orderings are computed by a mere breadth first search on the graph, not visiting the buffer breaking each cycle. Thus, any cyclic SDF graph is seen as a DAG during the graph traversal.

The number of admissible topological graph cuts is small and upper bounded by the graph diameter, enabling our heuristic to be fast. The admissible topological graph cuts naturally include all cuts located at sequential bottlenecks of the application, so they are the best candidates to increase the application performances by pipelining. Formally, sequential bottlenecks are located on single paths of the graph: when two successive actors of ranks $cr-1$ and cr are the only actors having these ranks. Selecting such cuts particularly benefits the applications having single paths and their repetition vector equal to $\vec{1}$.

Only ASAP and ALAP topological orderings are considered in order to limit the number of explored cuts. These two topological orderings are complementary and it is useful to consider both. Indeed, depending on the ordering, cuts of same rank may not contain all the same buffers and give more options to balance the computation between them. Two examples are given in Figures 4.6 and 4.7 to illustrate the different possibilities offered by ASAP and ALAP topological orderings. Cuts on each graph have the same rank and give the same II in the related schedules, but the cut obtained with

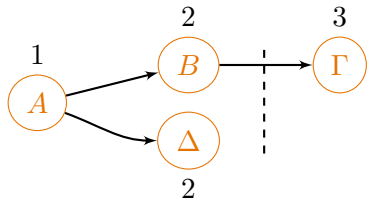


(a) Graph example with ALAP topological ordering. Delays are added in the represented cut of rank 3.

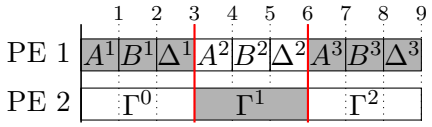


(b) Schedule of Figure 4.6a on two PEs. Firing exponents denote their graph iteration.

Figure 4.6 – Graph cut example and related schedule for ALAP topological ordering.



(a) Graph example with ASAP topological ordering. Delays are added in the represented cut of rank 3.

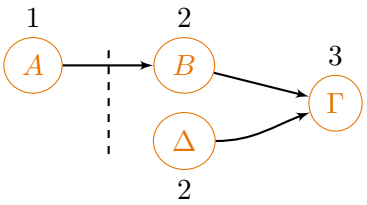


(b) Schedule of Figure 4.7a on two PEs. Firing exponents denote their graph iteration.

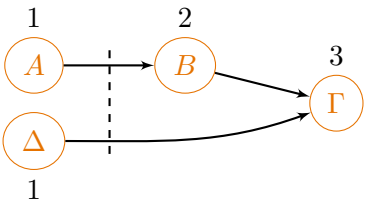
Figure 4.7 – Graph cut example and related schedule for ASAP topological ordering.

ASAP ordering avoids adding useless delays. Note that the represented graph and the actors PEs are different from Figure 4.1.

Finally it is important to note that we actually use a modified version of ALAP topological ordering, otherwise Equation (4.2) might not always be respected. The modified ALAP version enforces that all actors having no incoming buffers (or only incoming buffers breaking cycles) have the lowest topological rank. A counter-example and a valid example are given in Figure 4.8.



(a) ✘ Regular ALAP topological ordering creating invalid graph cut.



(b) ✔ Modified ALAP topological ordering to have valid (i.e. feed-forward) graph cuts.

Figure 4.8 – Graph cut examples for regular and modified ALAP topological ordering.

4.2.2 Selecting best topological graph cuts

To select the best topological graph cuts, the presented heuristic relies on a map linking topological ranks to an estimate of the Execution Time (ET) of all their actors. By definition, all actors having the same topological rank can be executed in parallel. We introduce a few notations to formalize the computation of this map. C_α denotes the ET of an actor α. The number of firings of α is $\vec{r}[\alpha]$. The rank of α is rank(α). The number of PEs is m. The ET estimate of rank cr, denoted rankLoad(cr), is computed as follows in Equation (4.5).

$$\text{rankLoad}(cr) = \frac{\sum_{\text{rank}(\alpha)=cr} \left\lceil \frac{\vec{r}[\alpha]}{m} \right\rceil \times C_\alpha}{\#\{\alpha | \, \text{rank}(\alpha) = cr\}} \tag{4.5}$$

The main purpose of Equation (4.5) is to provide a metric indicating if cutting before actors of rank cr improves the throughput, that is to balance the computation before and after the cut. To do so, we actually compare the estimated ET of all ranks before the cut of rank cr, $\sum_{1 \leq i < cr}$ rankLoad(i), with the estimated ET of all ranks after the cut of rank cr, $\sum_{cr \leq i}$ rankLoad(i). However, it is needed to weight the ranks according to the amount of parallelism that they contain, so that the graph is cut where it matters the most: on single paths for example. Thus, Equation (4.5) contains two divisions in order to reduce the weight of already parallel ranks: the repetition factor is divided by m, and the whole sum is divided by the number or actors in the considered rank. More precisely, numerator and denominator of rankLoad are averaged for ASAP and ALAP topological orderings; it is not specified in the equation for readability. Here Equation (4.5) is presented for identical PEs, but it can be rewritten for heterogeneous systems by considering the average ET time on each type of PE.

The selection of cuts is parameterized by a pair of two integers denoted H x, y: the number of cuts wanted by the user x, selected among the number of balanced cuts to consider y. We always have x lower than or equal to y, and y lower than the highest actor topological rank. y helps to define a first set of equally distributed topological graph cuts. To do so, the sum of all rankLoad(i) is divided by $y + 1$, giving an average stage load avgStageLoad. Then we enumerate ASAP cuts by increasing order of their rank, while summing their rankLoad and storing the rank of the closest cuts to a multiple of avgStageLoad. At most y ranks are preselected by this mean. The same operation is performed on ALAP cuts sorted by decreasing order of their rank. As cuts computed by ASAP and ALAP orderings may not be identical, there might be two possible cuts per preselected rank (only the ranks are stored in the aforementioned enumeration, not the cuts themselves). Considering the example in Figures 4.6 and 4.7, avgStageLoad = 2 and

our heuristic preselects the cut ranks 2 and 3. When both ASAP and ALAP topological cuts are valid on the same rank, only the one with the smallest delay sizes is kept, according to Equation (4.1). Thus, at most $2 \times y$ balanced cuts are preselected for the last step.

The last step of the heuristic is to select x cuts among the $2 \times y$ balanced cuts. This is done by two means: removing cuts that are too close from each other, and then selecting the one using less delays. Two topological cuts of rank cr_1 and cr_2 are considered too close from each other if the sum of their intermediate estimated ET is lower than avgStageLoad, as formalized in Equation (4.6).

$$\text{avgStageLoad} > \sum_{cr_1 \leq i < cr_2} \text{rankLoad(i)} \tag{4.6}$$

Considering the example in Figures 4.6 and 4.7, the ASAP topological graph cut of rank 3 depicted in Figure 4.7a is finally selected for the configuration H $1, 1$, since it is the one implying less delays (all rates and pipeline delays are equal to 1 in this example).

An example of preselection and final selection of cuts is depicted in Figure 4.9. 4 cuts are preselected by the heuristic with configuration H $2, 3$ on the given input SRSDF graph having 9 actors in line. Each actor is executed only once and its ET is equal to 10. The two cuts with a dashed line correspond to the cuts found during the first enumeration of ASAP cuts. The two cuts with a dotted line correspond to the cuts found during the second enumeration of ALAP cuts. Note that there are less than 3 cuts preselected by each traversal because an extra condition stops the traversal when the sum of remaining rankLoad is higher than avgStageLoad = 22. The current value of the sum of rankLoad and the closest multiple of avgStageLoad when a cut is preselected is recalled below the cut in Figure 4.9. The ranks of the preselected cuts are: $4, 6, 7, 5$, in order of appearance. Except between cuts 4 and 7, none of the other pair of ranks respects Equation (4.6). The removal procedure first sorts the cuts by the size of their pipeline delays, and then starts in the reverse order of appearance to remove the largest cuts in delay size. In this case, all cuts imply the same delay size, and the first two cuts to compare are the cuts 5 and 7. As cuts 5 and 7 are too close to each other and imply the same delay size, the highest rank is removed by default: 7. Then only three preselected cuts remain: cuts $4, 6, 7$ and the removal procedure stops since three is the number of preselected cuts asked by the configuration H $2, 3$. Finally, the heuristic selects the first two of the remaining cuts: 4 and 6. An evaluation on real SDF applications is provided in Section 4.4.

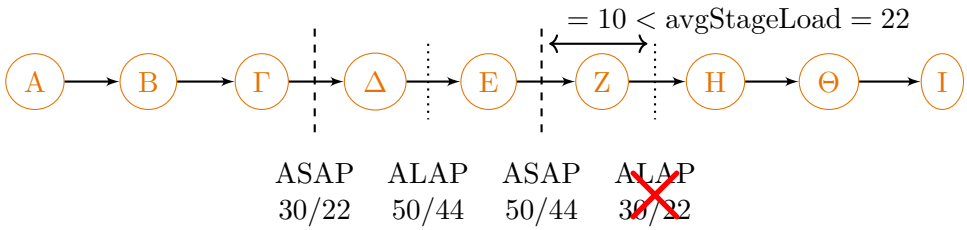


Figure 4.9 – Preselected and final cuts computed by the delay placement heuristic with configuration H $2, 3$ on a sample chain graph. Dotted cuts correspond to the preselected cuts while the dashed cuts correspond to the 2 final cuts. Each actor is fired once and has an ET equal to 10.

4.3 Automatic cycle breaking of SDF graphs

In the current chapter and in Chapter 3, an assumption is made on cycles: they have to contain at least one buffer with a delay set by the user. This delay ensures liveness of the cycle in the SDF graph. In this section, we propose a heuristic to automatically add such delay in the cycles. Of course, another solution to ensure liveness is to add multiple delays of smaller sizes on different buffers of the same cycle. Thus, the presented heuristic is a sufficient but not necessary condition to ensure liveness of SDF graphs having cycles.

First, we recall that delays in cycles may have a strong impact on the application behavior. Unlike pipeline delays on DAGs, they do not only shift the result but also modify it. An example is given in Figure 4.10. Let's consider that actor A computes the sum of its two inputs and copy the result on its two outputs. The self-loop buffer, which creates a cycle, has one delay which stores a kind of arithmetic carry: actor A sums the n-th array input i_n with the previous output $o_{n-\ldots}$. If the delay stores a unique token value $\overline{d_0}[0]$, then actor A behaves like a sum reduction of the input array and all intermediate sums are sent to the Results actor. Formally, it corresponds to the recursive sequence of Equation (4.7).

$$\forall n \in \mathbb{N}, \quad o_n = o_{n-1} + i_n, \quad o_0 = \overline{d_0}[0] \tag{4.7}$$

Now if the delay on the self-loop buffer contains two tokens, the behavior is modified and Equation (4.7) becomes Equation (4.8). Outputs with even (respectively, odd) indexes only store the sum reduction of previous array elements with even (respectively, odd)

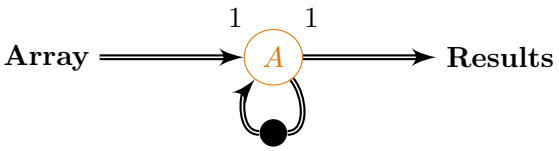


Figure 4.10 – SDF actor having a self-loop.

indexes[6].

$$\forall n \in \mathbb{N}, \quad o_n = o_{n-2} + i_n, \quad o_0 = \overline{d_0}[0],\ o_1 = \overline{d_0}[1] \tag{4.8}$$

Only the user know the behavior he wants, and thus the number of delays needed on a cycle. However, a placement pattern appears in the applications[7] modeled with the PREESM framework. Our heuristic only considers this pattern, where the plausible delay breaking a cycle $\mathcal{C}$ on buffer e corresponds to a local pipeline delay according to the repetition vector of the cyle only, as stated in Equation (4.9). According to the example in Figure 4.10, it corresponds to Equation (4.7) for only one token on the self-loop buffer.

$$d_0|_{\mathcal{C}}(e) = \text{cons}(e) \times \vec{r}|_{\mathcal{C}}[\text{dst}(e)] \tag{4.9}$$

The repetition vector of the cyle $\vec{r}|_{\mathcal{C}}$ is obtained by dividing the original repetition vector by the greatest common divisor (gcd) of all actors in the cycle $\mathcal{C}$. Equation (4.10) formally defines $\vec{r}|_{\mathcal{C}}$.

$$\vec{r}|_{\mathcal{C}} = \frac{\vec{r}}{\text{gcd}_{\alpha \in \mathcal{C}}\{\vec{r}[\alpha]\}} \tag{4.10}$$

Equation (4.9) provides a plausible delay size to break a cycle on a specific buffer. Our heuristic also selects the buffer where to place the delay. To do so, the heuristic relies on a classification of actors belonging to the cycle. Actors may be:

- an *entry* point of the cycle if they have incoming buffers outside the cycle;
- an *exit* point of the cycle if they have outgoing buffers outside the cycle;
- *both* entry and exit point;
- *normal* if all buffers connected to it belongs to the cycle.

Figure 4.11 illustrates entry, exit, and normal actors. In Figure 4.10, actor A is both an entry and an exit point.

[6] Besides, note that with two tokens on the self-loop buffer of Figure 4.10, the auto-concurrency of actor A is limited to two firings at a time. With only one token, as for Equation (4.7), auto-concurrency is not possible at all.

[7] https://github.com/preesm/preesm-apps

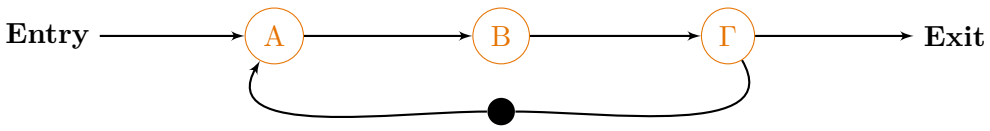


Figure 4.11 – Cycle example with one entry actor (A), one exit actor (Γ) and one normal actor (B). A delay (as chosen by our heuristic) breaks the cycle data dependencies.

The most common pattern seen in the application developped in PREESM corresponds to the example of Figure 4.11 where there are one entry actor, one exit actor, multiple normal actors from the entry to the exit, and no actor from the exit to the entry. The delay is placed on the buffer from the exit to the entry.

For the analysis, the heuristic builds a string representing the categories of actors in the cycle. Normal actors are ignored: the heuristic relies only on the relative placement of entries and exits. Entry actors are coded with the letter `i` while the exit actors are coded with the letter `o`. If an actor is both an entry and an exit, it is coded with the letter `b`. Graph in Figure 4.11 would be coded `io` or `oi` (all circular permutations are possible).

The heuristic selects the buffer breaking the cycle by matching the cycle string with regular expressions. An excerpt of the algorithm is presented as pseudo-code in Algorithm 4.1. The heuristic works only when actors of each kind (entry or exit or both) are not interleaved, otherwise it returns any buffer of the cycle (like in the cycle of the form `ioioi` for example). When not interleaved (patterns on lines 1, 4 and 6), the heuristic selects a buffer on the only path between the last exit and the first entry. Note that regular expression patterns on lines 4 and 6 are a cyclic permutation of each other; they are the most common cases. However, they do not return symmetrical answers. To be symmetrical and indepenent from the cycle permutations, it should be returned on line 5 the buffer preceding the first entry after the last exit instead.

The heuristic selects the buffer placement and the delay size to break cycles. Multiple cycles may share the same actors and buffers, but they are processed separately by the heuristic. If the same buffer e is selected for multiple cycles, the delay size is the maximum of each $d_0|_{\mathcal{C}}(e)$. Unfortunately, we did not formally prove that this solution ensures the liveness of complex applications having nested cycles. The proof is kept for future work.

Last but not least, the heuristic does not set the token value $\overline{d_0|_{\mathcal{C}}}$ of the delay. A neutral element would be the best default token value (as 0 for a sum, or 1 for a multiplication), but the SDF MoC is independent from the code of the actor, so the neutral

Algorithm 4.1: Selection of buffer breaking cycles

```
1 if matching(i*b?i) or matching(o*b?o*) then
2 |   if exactly one b present then
3 |   |   return buffer preceding the unique "both" actor;
4 else if matching(i*b?o+i*) then
5 |   return buffer following the last "exit" actor;
6 else if matching(o*i+b?o*) then
7 |   return buffer preceding the first "entry" actor;
8 else
9 |   return any buffer of the cycle;
```

element cannot be guessed automatically. However, depending on the application, the delay on cycles can be undefined data; it implies that some input or output of the first firings of actors located after the delay are not taken into account. This happens in the SDF graph of SIFT application, which contains cycles having delays with undefined token value. The loop actors having an incoming buffer with a delay also take another incoming buffer from a repetition counter actor. The repetition counters enumerate the current number of firings in the current scheduler iteration, and enable the loop actors to select the correct input in their internal code. The correct input may come from the buffer having a delay with initial undefined token value of from another incoming buffer.

4.4 Evaluation

The presented heuristic to pipeline SDF graphs is evaluated on various applications coming from the StreamIt [TKA02] benchmark, the examples provided with the SDF^3 [SGB06b] tool, and the applications[8] provided with the PREESM [Pel+14] tool. These applications represent a panel of state of the art signal and image processing algorithms, as well as more complex telecommunications, video coding and computer vision applications. The heuristic results are compared by throughput gain, relative to the sequential non-pipelined throughput on a single Processing Element (PE).

Three different evaluations are performed. In Section 4.4.1, the theoretical throughput gain is computed based on the schedule length, i.e. the II duration, obtained after adding the pipelines selected by the heuristic. This throughput gain is theoretical since no actual execution of the application is performed. A comparison is made with the optimal throughput gain among all admissible cuts, for applications amenable to

[8] https://github.com/preesm/preesm-apps

an exhaustive exploration. Large applications are detailed in Section 4.4.2. Finally in Section 4.4.3, the throughput and memory footprint increases are measured on actual executions of applications running on hardware.

The other heuristic to break cycles detailed in Section 4.3 has not been extensively evaluated. We only have checked that the size and the placement of delays according to the heuristic breaking cycles are exactly the same as the original size and placement of delays set by the designers. On the two tested applications of PREESM having cycles (SIFT and stereo), delays set by the heuristic breaking cycles are identical to those originally set by the designer. In the remaining part of this section, the *heuristic* word always refer to the one pipelining SDF graphs detailed in Section 4.2.

All experiments have been run with the PREESM open-source[9] tool, on a laptop with an Intel i7-7820HQ @ 2.90GHz processor (4 physical cores) and the GCC compiler version 7.5.0 (option `-O2`) on Ubuntu 18.04. For all selected applications, the execution time of the proposed heuristic is between 1 and 18 ms (maximum reached for SIFT). Note that the StreamIt/SDF[3] applications are all stateless in our experiments, except h263decoder (noAC) having self-loops. Self-loops disable auto-concurrency of an actor: multiple firings are serialized. For simplicity, the implementation of the delay placement heuristic in PREESM currently supports only homogeneous systems with identical PEs. Indeed, we have measured the throughput gain with the scheduler proposed in Chapter 3 which supports only homogeneous systems. However, the heuristic, in particular Equation (4.5), could easily be updated to take into account heterogeneous systems, by considering the average ET time on all types of PE.

Main characteristics of the applications are presented directly in the results tables. In the second column, MAP is the Maximum number of Actors in Parallel in the SDF graph; MAP equals the maximum number of parallel paths in the graph. When known, the total number of admissible graph cuts is specified in the column labeled #Cuts. Note that multiple versions of SIFT and sobel-morpho applications are considered: their graph is identical but they do not have the same number of firings. Some of their actors are fired a number of times equal to a multiple of a parameter p. Only SIFT and stereo contain directed cycles in their SDF graph. In all results tables, the columns labeled by H x, y contain the throughput gain obtained by the heuristic selecting x pipelines among y balanced pipelines. Columns labeled by O x contain the optimal throughput gain, over all admissible cuts, for x pipelines. Lines of results tables without any value printed in bold means that the throughput gain is similar for all setups; otherwise, the value in bold corresponds to the best gain along the line.

[9] https://preesm.github.io/

4.4.1 Theoretical throughput gain: regular applications

Theoretical throughput gain obtained with the heuristic is presented in Table 4.1, for three setups: no pipeline, one pipeline among one, three pipelines among three. Most applications have a repetition vector $\vec{r}$ equal to $\vec{1}$, except Chain4.2noAC (which contains self-loops), cd2dat, h263decoder, modem, mp3decoder, samplerate and satellite. Chain4.2noAC and Chain4.1 are toy examples made to fit the best cases of the heuristic; they correspond to the graph depicted in Figure 4.4, with only one path instead of four.

In Table 4.1, the best throughput gain is obtained by the heuristic with 3 cuts (H 3,3) for 11 of the 17 applications. More importantly, the heuristic finds a close to the optimal throughput with 1 cut for all applications except mp3decoder. The number of admissible cuts generating a throughput gain lower than or equal to H 1,1 is reported as a percentage of the total number of admissible cuts, in column %. On average, H 1,1 reaches a better throughput gain than 91% of the admissible cuts. The set of all admissible cuts has been generated thanks to Equations (4.3) and (4.4) implemented in the generic Choco[10] CP solver. Note that two applications are not compared with the optimal gain, FMRadio and Vocoder, because they admit too many cuts. These applications, and three others, are discussed in section 4.4.2.

On DCT and h263decoder, the throughput gain is less than 2.0, even with 3 pipelines: this comes from too few actors in the original graphs (respectively 8 and 4), having unbalanced ETs. This configuration leads the heuristic to find only 2 graph cuts for DCT and h263decoder, even if 3 pipelines were asked by the user. The number of effectively selected cuts is specified as an exponent. The same behavior happens for modem and mp3decoder applications: only 2 cuts are selected whereas 3 pipelines were asked. To avoid this problem, only 2 pipelines among 3 are requested for the PREESM applications, see Table 4.2 Indeed, in these applications the ETs are greatly unbalanced and the ET of the longest actor represents up to 47% of the sequential ET of sobel-morpho (p1).

For the PREESM applications evaluated in table 4.2, the heuristic reaches the best throughput in 7 cases out of 9. SIFT application is a difficult case: its SDF graph is widely parallel (up to 30 parallel paths) and contains multiple cycles. Moreover, its parallel paths have unbalanced ET. In this situation, selecting topological cuts is not the best option and 1 optimal cut (O 1) even reaches a better throughput than 2 cuts from the heuristic (H 2,3): for SIFT (p1) and SIFT (p2). However, when more balanced

[10] http://www.choco-solver.org/

Name	MAP	$\#V$	#Cuts	H 0	H 1,1	O 1	%	H 3,3
Chain4.1	1	4	3	1.0	2.0	2.0	100	**3.9**
Chain4.2noAC	1	4	3	1.4	2.3	2.3	100	**3.4**
BitonicSort	4	40	141	1.6	2.9	2.9	100	**3.6**
cd2dat	1	6	5	4.0	4.0	4.0	80	4.0
DCT	1	8	7	1.0	1.8	1.8	100	1.8[2]
DES	3	53	128	1.2	2.2	2.2	96	**2.4**
FFT	1	17	16	1.0	2.0	2.0	100	**3.7**
FMRadio	12	43	—	3.1	**3.3**	—	—	**3.3**
h263decoder (noAC)	1	4	3	1.8	1.8	1.9	100	**2.0**[2]
modem	1	6	5	2.0	**3.3**	3.3	100	3.3[2]
mp3decoder	2	14	33	3.7	3.7	**3.8**	66	3.7[2]
MPEG2noparser	3	23	140	1.1	2.2	2.2	100	**2.7**
samplerate	1	6	5	4.0	4.0	4.0	60	4.0
SAR	2	44	63	1.0	1.8	1.8	100	**2.3**
satellite	3	22	90	4.0	4.0	4.0	68	4.0
TDE	1	29	28	1.0	1.9	1.9	100	**3.4**
Vocoder	17	114	—	1.2	2.1	—	—	**2.6**

Table 4.1 – Characteristics and throughput gain with delays (H) of SDF benchmark applications, on four PEs. H 0 corresponds to no pipeline. H 1,1 corresponds to one pipeline selected among one. O 1 corresponds to the optimal single stage pipeline. % is the percentage of cuts worst than or equal to the heuristic. H 3,3 corresponds to three pipelines selected among three.

parallelism is expressed, for SIFT (p4), the heuristic configuration H 2,3 once again is better than the other setups.

4.4.2 Theoretical throughput gain: widely parallel applications

This subsection evaluates the applications revealing the main advantage of the presented heuristic: no explosion of the number of cuts to test when the SDF graph is already parallel. Indeed, all evaluated applications in Table 4.3 admit between 10^5 and 10^{10} cuts, which makes it impossible to evaluate the throughput of each cut by performing scheduling and mapping. Moreover, the number of possibilities also explodes with the number of pipelines asked: it is equal to the number of cut combinations without repetition (binomial coefficient): $\binom{\#\text{Cuts}}{\#\text{Stages}-1}$.

Table 4.3 presents results for the applications already having parallelism expressed in their graph: MAP is between 12 and 17 for all of them. In this experiment, the throughput is evaluated on 64 PEs for the heuristic setup H 3,3 selecting 3 pipelines.

Name	MAP	$\#V$	#Cuts	H 0	H 1,1	O 1	%	H 2,3
SIFT (p1)	30	54	868	1.2	1.6	**2.2**	92	1.6
SIFT (p2)	30	54	868	2.3	2.8	**3.7**	91	3.0
SIFT (p4)	30	54	868	3.5	3.5	3.6	80	**3.7**
sobel-morpho (p1)	1	6	5	1.0	**2.0**	2.0	100	2.0
sobel-morpho (p2) *	1	6	5	1.7	2.4	2.4	100	**2.6**
sobel-morpho (p3) *	1	6	5	2.3	**3.5**	3.5	100	3.4
sobel-morpho (p4)	1	6	5	2.3	2.8	**3.3**	40	3.3
stereo	3	28	3631	3.3	3.9	**3.9**	99	3.9
lane-detection *	3	11	24	1.0	1.7	1.7	100	**2.5**

Table 4.2 – Throughput gain with delays (H) of SDF benchmark applications, on four PEs. H 0 corresponds to no pipeline. H 1,1 corresponds to one pipeline selected among one. O 1 corresponds to the optimal single stage pipeline. % is the percentage of cuts worst than or equal to the heuristic. H 2,3 corresponds to two pipelines selected among three possibilities.

Having 64 PEs ensures to observe the effect of the pipelines instead of the inherent task parallelism. Indeed, the maximum number of actors in parallel MAP (17) is almost 4 times smaller than the number of PEs. The maximum theoretical throughput gain with unlimited PEs, denoted Max Θ, is given as a reference. All applications in Table 4.3 are acyclic, so Max Θ is computed by dividing the sequential ET of the application by the ET of its longest actor, as if each buffer had a pipeline delay. Adding 3 pipelines increases the throughput gain from a factor 2 (for FMRadio) to 3 (for ChannelVocoder).

Name	MAP	$\#V$	#Cuts	H 0	H 3,3	Max Θ
Beamformer	12	57	1.7×10^7	8.9	19.0	25.6
ChannelVocoder	17	55	1.3×10^{10}	11.1	33.2	33.4
Filterbank	16	85	4.3×10^8	10.5	30.5	32.2
FMRadio	12	43	2.6×10^5	6.0	12.7	13.1
Vocoder	17	114	3.0×10^{10}	1.2	2.7	2.8

Table 4.3 – Throughput gain with delays (H) of SDF benchmark applications, on sixty-four PEs. H 0 corresponds to no pipeline. H 3,3 corresponds to three pipelines selected among three possibilities. Max Θ corresponds to the maximum possible throughput gain, with unlimited PEs.

4.4.3 Practical experimentation

In this subsection, the throughput and memory measurements come from actual executions on hardware, on the same laptop used for all experiments, having 4 PEs. Memory is allocated after the scheduling process, with buffer merging [Des+16b] optimizations activated. The memory needed is computed by PREESM, and compared with the sequential version on 1 PE for reference.

Results are provided in Table 4.4, for an average of 100 executions for SIFT and stereo, and 10000 executions for sobel-morpho and lane-detection. Note that the scheduler used in this practical experimentation differs from the one used in the theoretical experimentation, it is a list scheduler taking into account communications [KAG96]. Indeed, due to an implementation bug in the memory allocation process at the time of this experiment, it was not possible to measure accurately the memory size when using the scheduler of Chapter 3.

In Table 4.4, the heuristic especially improves the throughput of SIFT and sobel-morpho with $p = 1$ and $p = 2$, that is, when the application is not parallel enough. Yet, for lane-detection which has $\vec{r} = \vec{1}$, the heuristic only slightly increases the throughput, while increasing the memory by a factor 1.9. The theoretical throughput gain of lane-detection is 2.5, that is two times higher than reality. We assume that this gap is due to the variability of the ET of the display actor, representing 28% of the application sequential execution time. Extended experiments should be performed to confirm this hypothesis. Also, synchronization points added by PREESM may be non-negligible. None of the applications reaches the throughput expected in the theoretical evaluation.

An interesting point is that selecting 1 cut among 2 (H 1,2) gives better results than 1 among 1 for half of the cases. Such heuristic setups may compensate the case of unbalanced ETs or cycles, especially for SIFT (p2). Moreover for SIFT (p2) the H 1,2 setup greatly reduces the memory footprint compared to H 1,1: from a factor 3.0 to 1.1. Finally, the heuristic offers a trade-off between memory footprint and throughput. This trade-off is especially needed for memory bounded application as SIFT requiring 197 MBytes (reference). In the worst case, for sobel-morpho (p4), adding one pipeline decreases the throughput while greatly increasing the memory (3.3 times). The memory increase is due to the graph cut location: between buffers transmitting numerous data, and thus it causes additional time for memory copies and synchronizations.

In Table 4.5, we also compare the practical throughput increase of the PREESM legacy (Leg.) scheduler (FAST initial step [KAG96]) and the one developed for partially periodic constraints (Per.) described in Algorithm 3.3. Results are roughly similar

Name	H 0		H 1,1		H 1,2		H 2,3	
	Sp.	Mem.	Sp.	Mem.	Sp.	Mem.	Sp.	Mem.
SIFT (p1)	1.2	1.1	**1.6**	2.1	1.4	1.3	1.3	1.8
SIFT (p2)	1.8	1.1	1.9	3.0	**2.4**	1.2	2.2	2.3
SIFT (p4)	**2.5**	1.2	2.2	1.1	2.2	1.1	2.5	1.8
sobel-morpho (p1)	0.9	1.0	1.3	2.2	1.3	2.6	**1.6**	3.8
sobel-morpho (p2)	1.7	1.6	2.3	2.1	**2.5**	2.4	2.1	3.4
sobel-morpho (p3) *	2.3	2.1	2.4	2.8	**2.5**	2.6	2.5	3.2
sobel-morpho (p4)	**2.5**	2.0	1.9	2.6	2.2	2.3	2.4	3.3
stereo	2.2	1.1	2.3	1.1	2.3	1.1	**2.4**	1.1
lane-detection *	1.0	1.0	1.1	1.8	1.1	1.7	**1.2**	1.9

Table 4.4 – Throughput and memory increases with delays (H), on four PEs, for different parallelism parameters (p). Specific mapping constraints are enforced for applications marked with *: read and display actors are alone on their core if there is a pipeline.

for both schedulers, except for the lane-detection application, where the periodic one is slower than the sequential time for H 0, H 1,1 and H 1,2. Further investigations are needed to characterize this phenomenon but the first fit mapping strategy of Algorithm 3.3 is a plausible cause. This first fit strategy is more likely to allocate two successive data dependent firings on two different PEs, introducing a costly synchronization lock between the firings.

4.5 Related work

Pipelining and more generally retiming has been extensively studied in the context of Very Large Scale Integration (VLSI) circuit design [LS91; Par07]. Pipelining legality was formally defined by Parhi [Par07] for a subset of SDF graphs: SRSDF graphs, which always have their repetition vector equal to $\vec{1}$. It was also studied for software pipelining [Lam04; All+95], with retiming methods used in this context [CDR98]. Those works only concern SRSDF graphs. Our work focuses on pipelining SDF graphs, avoiding the costly conversion to SRSDF and thus reducing the analysis complexity.

Pipelining of SDF graphs was originally proposed by Lee and Messerschmitt [LM87a] as an optimization. Gordon et al. [GTA06] proposed a heuristic to pipeline a partially unfolded SDF graph, as well as Kudlur et al. [KM08]. The heuristic presented by Gordon et al. relies on a first transformation of the original actors, in order to balance the ETs and to adapt the amount of parallelism. The *Stream Graph Modulo Scheduling* presented

Name	H 0		H 1,1		H 1,2		H 2,3	
	Leg.	Per.	Leg.	Per.	Leg.	Per.	Leg.	Per.
SIFT (p1)	1.2	1.1	**1.6**	1.0	1.4	1.1	1.3	1.0
SIFT (p2)	1.8	1.6	1.9	**2.6**	2.4	1.6	2.2	1.4
SIFT (p4)	**2.5**	2.4	2.2	2.1	2.2	2.1	2.5	2.2
sobel-morpho (p1)	0.9	0.8	1.3	1.4	1.3	1.3	1.6	**1.7**
sobel-morpho (p2)	1.7	1.4	2.3	**2.6**	2.5	2.6	2.1	2.0
sobel-morpho (p3) *	2.3	2.2	2.4	**2.7**	2.5	2.5	2.5	2.4
sobel-morpho (p4)	2.5	**2.6**	1.9	2.6	2.2	2.1	2.4	2.3
stereo	2.2	2.0	2.3	2.1	2.3	2.0	**2.4**	2.2
lane-detection *	1.0	0.3	1.1	0.6	1.1	0.5	**1.2**	1.1

Table 4.5 – Throughput increases with delays (H), on four PEs, for different parallelism parameters (p). Leg. is using the legacy PREESM scheduler while Per. is using Algorithm 3.3. Specific mapping constraints are enforced for applications marked with *: read and display actors are alone on their core if there is a pipeline.

by Kudlur et al. relies on an ILP formulation to set the unfolding limit, and it requires the Initiation Interval (II) length as an input of their algorithm. In the work of Udupa et al. [UGT09], the SDF graph is completely unfolded to its SRSDF equivalent allowing for a fine tune of the added delays. They propose an ILP formulation to compute the stage of each actor firing; this formulation requires a few minutes to be solved for large graphs of the StreamIt benchmark. On the contrary, our heuristic works on the original unfolded SDF graph and it requires a maximum number of pipelines as an input, in order to minimize the II accordingly. Our heuristic is faster and more scalable than the aforementioned works, however note that their algorithms also perform scheduling at the same time while we do not.

Scheduling has been largely explored in optimal and heuristic forms [KA99; MG13]. Other works look at combining pipelining with scheduling, restricted to SRSDF graphs [YH09] or acyclic SDF graphs [YH12]. Our work separates pipelining from scheduling. Scheduling is computed afterwards on the pipelined graph, taking advantage of original data and task parallelism, as well as temporal parallelism.

Finally, multiple works [Zhu+16; Liv+07] addressed the optimal search for a retiming to reduce the makespan of a graph. Additionally, [Zhu+16] accepts a constraint on the maximum number of PEs, at the cost of non-optimality. Both use symbolic execution of a partially unfolded SDF graph to find a retiming. In our contribution we focus on the pipelining of an SDF graph in its reduced original form to provide a fast heuristic. We

do not perform any execution, symbolic or not. It is also possible to retime SDF graphs by adding initial data in the buffers so that all firings of an actor can be performed in parallel [KLE17]. The aforementioned work [KLE17] is especially convenient to execute the same algorithm on multiple data at a time, similarly to a GPU. In our work, we focus on the balancing of the ETs in the different pipeline stages, for generic CPU, DSP or even FPGA.

4.6 Conclusion

A fast heuristic to automatically pipeline SDF applications at coarse grain has been presented and actually improves the throughput of the evaluated applications. The heuristic is able to quickly pipeline applications containing up to billions of admissible cuts. Our algorithm limits its exploration to a few cuts to reduce analysis time, and experiments show that this method is very often close to the optimal solution. The presented heuristic is especially useful when considering a large number of PEs.

However, the last experiment in Section 4.4.3 shows a gap between the theoretical throughput gain and the practical gain, always lower than expected. This gap is observed for both our pipelining heuristic and the theoretical optimal solution. Even if pipelining is a powerful optimization, its memory usage is a major drawback. The last experiment also shows that when it is possible to adapt the parallelism grain, it is more efficient to express enough parallelism inside the application than to pipeline it. Thus, the efficiency of pipelining is heavily dependent on the choice of various parameters of the application, and on the scheduling and mapping processes. In a joint work with the team of Michael Masin of IBM Research Labs, we have tried to develop an integrated LP formulation tackling multiple aspects: the pipelining, the scheduling, the mapping and the selection of parameters. This integrated formulation of multiple problems at once is challenging, and although we are close to a solution, we did not succeed yet. This formulation is briefly discussed in Chapter 5, and an alternative DSE solution is presented.

Our heuristic can be improved on various directions. For example, by adding smaller delays to break the dependencies between only a certain number of firings instead of all. Instead of setting delay sizes $d_0(e)$ as a multiple of Equation (4.1), the size could be a multiple of $\text{lcm}\{\text{cons}(e), \text{dst}(e)\}$, ensuring only a pipeline local to the buffer e. A greater feature would be to compute the size of the added delays symbolically: as an equation of the application parameters if any. This feature is challenging but would be especially useful for the DSE approach presented in Chapter 5. Finally, this heuristic is only one optimization method among various others, as the most related to this work:

retiming. Retiming is especially useful when considering graphs having cycles. We have also presented a heuristic to break data dependencies of cycles, and a combination of our two heuristics and classic retiming techniques is kept for future work.

Dissemination and Implementation

The contribution presented in this chapter, except Section 4.3, has been published in the SAMOS'20 conference [Hon+20b] (see also the video presentation here ↗). The algorithms presented in this chapter have been implemented as workflow tasks of the PREESM tool. See the following task description for the implementation of the algorithm described in Section 4.2:

- `pisdf-delays.setter` ↗

Chapter 5

Configuration of parameterized SDF graphs

Contents

5.1 PREESM parameters . . . 123
5.1.1 What can be parameterized? . . . 123
5.1.2 How to use PiMM parameters? . . . 125
5.1.3 Malleable parameters: design choice in PiMM . . . 126
5.2 DSE: entangled problems . . . 128
5.2.1 Entanglement of the objectives . . . 128
5.2.2 Entanglement of the solving methods . . . 129
5.3 An exhaustive DSE method . . . 132
5.4 Improving DSE with automatic delay placement . . . 134
5.4.1 Adding delays to improve the results . . . 135
5.4.2 Precomputing the number of cuts . . . 136
5.5 A naive heuristic for Integer malleable parameters . . . 137
5.6 Evaluation . . . 139
5.6.1 Test application: live video SIFT . . . 139
5.6.2 Experiments . . . 141
5.7 Related work . . . 144
5.7.1 Multi-objective DSE . . . 144
5.7.2 Parameterized SDF MoCs . . . 145
5.7.3 Parallelism adaptation . . . 146
5.8 Conclusion . . . 147

Introduction

Multiple extensions of the SDF MoC support parameters, some of which are listed in Section 1.2.3. Parameters are used to modify the application behavior during its execution, or to explore different ways to implement it, that is Design Space Exploration (DSE). In particular, parameters of PISDF applications can modify the consumption and production rates, the delay sizes, the actor ETs and energy consumption. Parameters may also define the Quality of Service (QoS) of the application, as the image resolution. In this chapter, we study the offline DSE of static PISDF applications, in order to select and fix the best values of all their parameters. As the parameter values are fixed once for all the indefinitely repeated executions of the static applications, the schedule, and especially the throughput and latency can be optimized offline[1]. The case of dynamic applications and online DSE is quite more challenging and is not considered here.

DSE is especially needed to adapt the application to the target architecture. For example, in Chapter 2, a parameter p is used to control the degree of data parallelism of an SDF graph. Experiments reported in Table 4.2 of Chapter 4 show that for the sobel-morpho application, best throughput is achieved with one pipeline and $p = 3$ while there are $m = 4$ PEs available. The goal of the DSE is to automatically select this configuration, but only if the designer accepts the extra latency due to the pipeline. Thus, DSE not only adapts the application to the architecture, but it also has to respect constraints and trade-offs specified by the designer.

As a last contribution of this thesis, we introduce in this chapter a method to perform DSE on PISDF graphs, while respecting constraints and trade-offs regarding throughput, latency, energy, or directly any parameter for QoS. Unfortunately, both memory and communication contentions are not taken into account in our DSE. This contribution uses all the previous ones detailed in Chapters 2 to 4. It is based on the new notion of *malleable parameters* which introduce choice among multiple expressions defining the value of a parameter.

This chapter is organized as follows. Section 5.1 presents the usage of parameters in the PREESM tool. Section 5.2 describes a few challenges of DSE, due to the entanglement of design objectives on one side, and the entanglement of the solving methods on the other side. An exhaustive DSE workflow is detailed in Section 5.3. To improve the DSE results, delays can be added automatically with the method presented in Chapter 4. This procedure is detailed in Section 5.4. Moreover, a dichotomous DSE heuristic

[1] Other offline optimized properties are the memory allocation and the communication synchronizations. However, the memory allocation and the communications are not taken into account in our DSE.

is presented in Section 5.5. Both approaches are evaluated and results are shown in Section 5.6. Finally, related work is presented in Section 5.7 and Section 5.8 concludes this contribution.

5.1 PREESM parameters

This section presents the usage of static parameters available in the PREESM tool. Static parameters are a subset of the PISDF MoC. The PISDF model is the application of the PIMM [Des+13] meta-model on SDF graphs. We restrict to static parameters in order to generate statically optimized code, in which all the needed memory is pre-allocated for every data communication on the buffers.

In Section 5.1.1, we detail which entities of the PISDF model can be parameterized and their impact on the application. Following Section 5.1.2 gives examples of the parameter syntax, and gives tricks to model multiple situations. Finally, we introduce the *malleable parameters* in Section 5.1.3; malleable parameters have been developed for the purpose of DSE.

5.1.1 What can be parameterized?

In the PREESM implementation, parameters are graphically represented by pentagons with a rectangular base, as depicted in Figure 5.1. Each *parameterizable entity* has incoming arrows for each parameter it depends on. In Figure 5.1, the `tot_image_size` parameter (bottom right of the picture) depends on `image_width` and `image_height`; the arithmetic expression of `tot_image_size` simply is: `image_height*image_width`.

All the parameterizable entities are:

E-1 parameters themselves in a PISDF graph;

E-2 data production and consumption rates of any buffer in a PISDF graph;

E-3 period of any non hierarchical actor in a PISDF graph, as well as the top graph period itself (see Chapter 3);

E-4 size of a delay (see Chapter 4) in a PISDF graph;

E-5 Execution Times (ETs) of actors, set in the *scenario* of PREESM;

E-6 energy of actors, set in the *scenario* of PREESM.

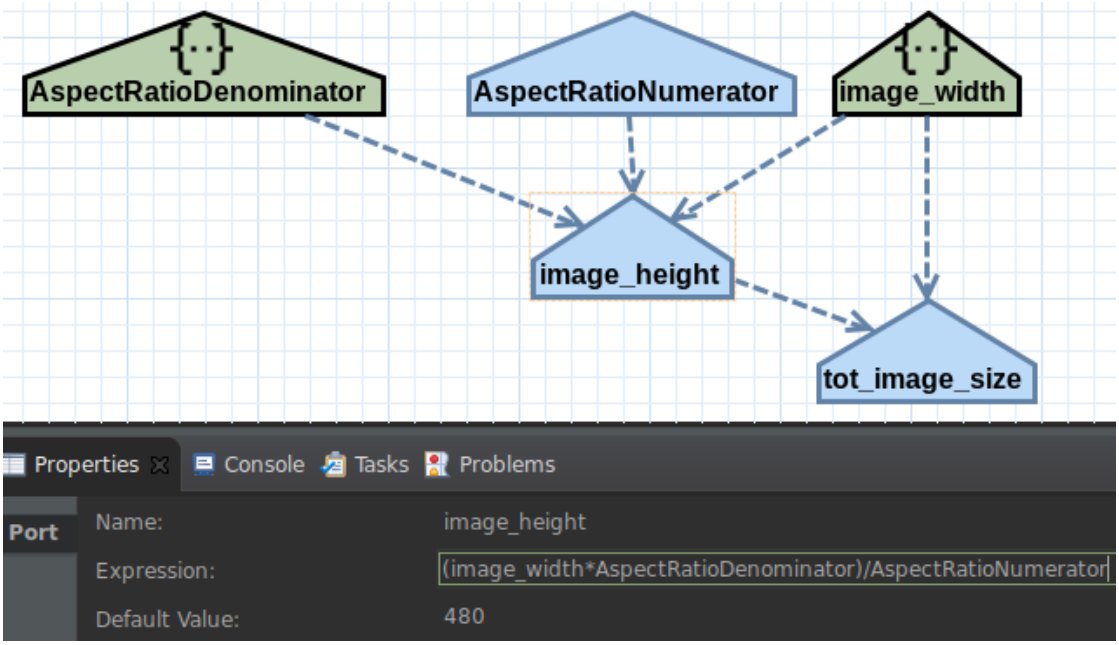


Figure 5.1 – Example of parameters and their dependencies, here to express image resolution choices. Screenshot of the SIFT application PISDF graph in PREESM.

These parameterizable entities accept arithmetic expressions composed of predefined mathematical functions and parameters. In the *scenario* model of PREESM used to store application information dedicated to a specific architecture, the ET and energy of actors are parameterizable entities, but not the data sizes. When the data consumption and production rates are parameterized, parameters impact on the repetition vector (see parameter p for the degree of data parallelism in Chapter 2). If used in any parameterized entity, a parameter also has a direct impact on the scheduling process. Last but not least, a parameter can be an argument of the `C` function associated with each actor.

Each parameter has a unique name and contains an arithmetic expression, possibly depending on other parameters or predefined mathematical functions such as logarithm[2]. Dependencies between parameters must be explicitly given by the designer. The parameters form a DAG whose roots[3] hold only numerical values. The designer has to specify the parameter expressions of the regular parameters and the numerical values of the root parameters. As parameters have unique names and as their dependencies must not

[2]A few common mathematical functions are provided by the PREESM implementation, and it is possible to code more complex functions directly in the Java code of PREESM (which needs to be recompiled then).

[3]The roots of the DAG of parameters are the parameters having no incoming dependencies, i.e. the parameters being a single numerical value.

```
if(key==<key1>, <value1>, if(key==<key2>, <value2>, <defaultValue>))
```

Listing 5.1 – Parameter expression implementing a dictionary, here depending on key parameter `key`.

create any cycle, they can be seen as an SSA program, ensuring that their expressions can all be valuated to a number. The valuation has to be an Integer number in the PISDF model, but internal computations may use floating point numbers. For example, both expressions `9.9/3.3` and `10.0/3.0` are valuated to 3.

5.1.2 How to use PiMM parameters?

The syntax of PREESM parameters is simple and does not authorize assignments. This limitation benefits the case of dynamic parameters, to avoid overhead while valuated them with the dynamic version of PREESM: SPIDER. If the expression contains semi-colon characters, it will be only evaluated until the first semi-colon occurrence (from left to right). The only authorized control flow statement is the conditional statement `if(condition, ifTrueStatement, otherwise)`. The conditional statement can be used to express a dictionary as in Listing 5.1. However, due to the SSA form, the dictionary parameter has to be duplicated for each possible input parameter `key`.

In some cases, it might be necessary to guarantee that a parameter is a multiple or divisor of another. For example, it happens for the degree of parallelism parameter p in Chapter 2, which has to be a divisor of the size of an actor input array. Similarly in the SIFT application[4], a maximum number of keypoints has to be detected by a data-parallel actor of the application. It is then necessary to ensure that each firing of the data-parallel actor has at least one keypoint to detect, and that all firings have the same value. Listing 5.2 presents the computation of this local number of keypoints to detect, according to the maximum total number of keypoints defined by the user, and the degree of parallelism.

Parameters are valuated to Integer numbers only, although they support floating point numbers inside expressions. Then it is possible to use rational numbers in $\mathbb{Q}$, defined as two parameters: one Integer value for the numerator and another one for the denominator. This is especially useful to express the aspect ratio selection of an image resolution: for example, `16:9` or `4:3`. This solution is depicted in Figure 5.1: the

[4]The SIFT application is briefly introduced in Chapter 2.

```
max(1, floor(nKeypointsMaxUser/parallelismLevel))
```

Listing 5.2 – Parameter expression of `nLocalKptmax` in the SIFT application. This parameter computes the number of keypoints detected by each firing of the corresponding actor `detect_keypoints`.

common numerator is equal to 9 while the denominator (top left of the figure) can be set to 16 or 12.

Finally, actor production and consumption rates can be equal to 0 in very specific occasions, and otherwise are only positive Integers. A *null* buffer where both production and consumption rates are equal to 0 is ignored during the analysis; consequently if an actor is only connected to such null buffers, it is not fired at all. This behavior may lead to inconsistencies during the PISDF graph execution: if an actor has a null buffer input, what argument should be given to the underlying `C` function? Thus, only *special actors* of PISDF, whose `C` function is predefined by PREESM, are authorized to have some of their buffers with null production rate or consumption rates. Special actors mainly correspond to standard duplicate/upsample or decimate/downsample actors which respectively copy multiple times their unique input on all their outputs or copy only once a subset of all their inputs on their unique output[5]. For example, this behavior enables the designer to avoid executing actors forming a path between a duplicate and a decimate actors, while not removing the path and actors from the application graph. If a parameter is used to put all rates of the path to the value 0 with an `if` statement, this is especially useful to easily test different configuration of the application. Figure 5.2 illustrates this possibility, where a path containing two actors is not executed depending on a parameter value. In any case, the special actors around the non executed path must still have a non null buffer connecting them[6], here from `out2` to `in2`.

5.1.3 Malleable parameters: design choice in PiMM

Parameters in PREESM contain a single expression set by the designer, so the designer has to change the expression manually to explore a new configuration of its application.

[5] In the PISDF model, these special actors are called: `broadcast` (for duplication), `roundbuffer` (for decimation), `fork` (for split) and `join` (for concatenation).

[6] More precisely, the special actors require in any case a non 0 consumption rate on at least one input buffer, and a non 0 production rate on at least one output buffer. Moreover, if one rate of the buffer is 0, then the rate on the other side must also be 0.

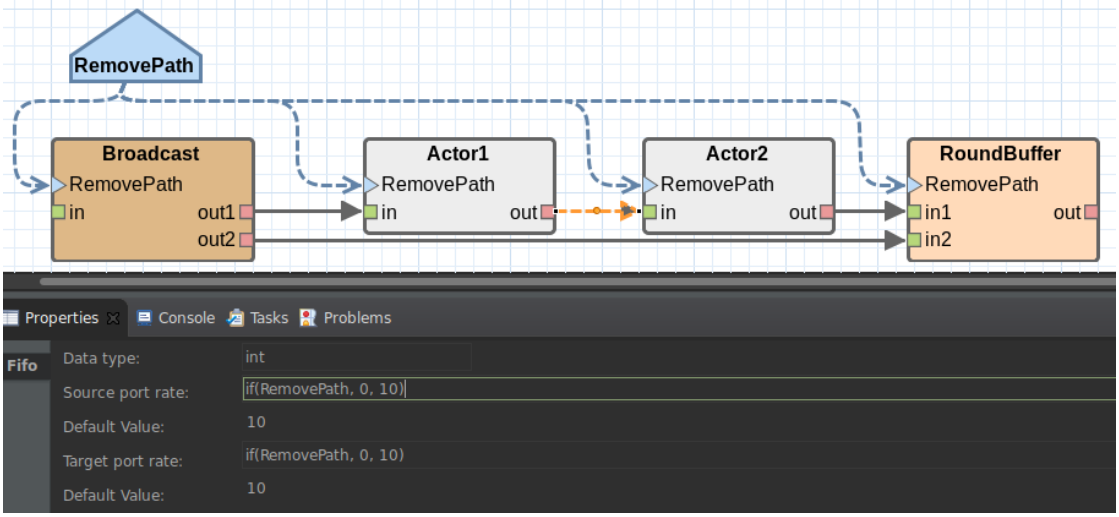


Figure 5.2 – Example of a parameterized path: Actor1 and Actor2 are executed only if parameter RemovePath is not 0. The path to remove is well delimited by two special actors (colored in orange, on left and right). Screenshot of PREESM, with rate expressions of the buffer from Actor1 to Actor2.

In order to perform an automatized DSE of the application configuration, we introduce here a new kind of parameters, called *malleable parameters*. Technically, malleable parameters inherit from regular parameters and behave like them during the analysis of the PISDF graph. Regular parameters may contain multiple sub-expressions delimited by semi-colons, but only the first one is valuated (from left to right). On the contrary, malleable parameters come with an optional specific analysis and synthesis algorithm performing the DSE: it tests all combinations of sub-expressions stored in the malleable parameters. The DSE then selects the best point according to designer objectives, by setting the best sub-expression of each malleable parameter as its default (i.e. first) sub-expression. The DSE will be detailed in the following Sections 5.3 to 5.5.

In the GUI of PREESM, malleable parameters are identified by a different color (green instead of blue) and an extra **{..}** symbol on top. Figure 5.1 contain two malleable parameters: one to set the image width and the other one to set the aspect ratio. The image height and the total image size derive from these two malleable parameters. The expression of the `AspectRatioDenominator` malleable parameter is: `12;9` since the aspect ratio is either `4:3=16:12` or `16:9`.

As they technically behave as regular parameters, malleable parameters can depend on other malleable parameters. Any parameterized entity can depend on a malleable parameter. The space of possible application configurations due to malleable param-

eters corresponds to the Cartesian product of their sub-expression sets. For example, 10 malleable parameters each having two sub-expressions lead to $2^{10} = 1024$ possible combinations. Because of this combinatorial explosion and because no symbolic analysis is performed during the DSE, more complex expressions as interval definitions are not supported yet.

5.2 DSE: entangled problems

Design Space Exploration (DSE) helps to compare various configurations of an application and to select the best according to objectives given by the designer. Unfortunately, these objectives may be contradictory. Moreover, depending on what variables are computed by the DSE, i.e. the solving method, the variables themselves may be entangled and have complex relationships between each other. In this section, we briefly list main entanglements of the objectives (in Section 5.2.1) and main entanglements of the solving methods (in Section 5.2.2), and we specify which objectives and solving methods are used in our DSE. Note that DSE also refers to the selection of best target architectures, however we only consider application configuration selection in this contribution: the architecture is already predefined by the application designer.

5.2.1 Entanglement of the objectives

DSE require objectives, expressed as metrics to minimize, to maximize, or to limit, in order to select the best application configurations, meeting most objectives. We list below the most important metrics, and give a few details about their entanglement:

M-1 throughput (redundant if graph period set);

M-2 latency (entangled with M-1 when latency is expressed in number of scheduler iterations);

M-3 power/energy (entangled with M-1 since both depend on processor frequency);

M-4 memory (entangled with M-1 and M-2 to compute mutual exclusions of buffer usage, and to store delays creating latency);

M-5 communication (entangled with M-1, M-3 and M-4 to compute mutual exclusions of network usage, and to store transmitted data);

M-6 application QoS (entangled with all above when it depends on any parameter);

M-7 DSE execution time (entangled with all above);

M-8 generated code size (depending on the scheduler, and so on all above).

In our contribution, we consider only metrics M-1 to M-3, M-6 and M-7. Regarding metric M-6 for the QoS, it can depend on any parameter such as the image resolution, which defines the amount of computations in a video application. Metric M-8 is partially considered since it may depend on the degree of parallelism parameter (metric M-6). Metric M-4 is partially considered since it may depend on the latency (metric M-2). Metric M-5 is not considered at all and we currently restrict the architecture to single homogeneous multi-processors.

The fact that metrics are mutually entangled is not always a problem: if a mathematical relation is known between two metrics, then it is possible to merge the objectives referring to these metrics during the DSE. For example, the ADFG [Hon+17] real-time periodic scheduling synthesizer offers an option to convert throughput loss in memory gain, depending on a graph partitioning imbalance ratio. This relation is made possible thanks to the ADFG specific scheduling model of SDF graphs (periodic firings without auto-concurrency). However, such relations are not always known, and not always predictable. For example, the repetition vector of an SDF application depends on the lowest common multiple of all production and consumption rates, whereas the exact distribution of prime numbers in $\mathbb{N}$ is not known yet.

When no relation is known between the metrics of the objectives, there is no single best configuration and instead there is a Pareto front of best configurations. To enforce a single best configuration, it is possible to use a weighted sum of the objectives or to prioritize them[7]. In our contribution, we consider a prioritization of the objectives: an objective is considered only if the ones of higher priority are already optimized.

5.2.2 Entanglement of the solving methods

Multiple variables are defined or computed during the design and synthesis processes respectively. The main variables are the mapping and scheduling of firings, the routing and scheduling of communications, the placement of delays, the estimation of ETs and energy of firings, and the memory allocation. These variables are all entangled with the scheduling of firings. Besides, note that delays are usually defined or computed at the

[7]More complex functions aggregating the objectives exist, as the Choquet integral. Besides, some aggregation methods (usually convex functions) may find a single best configuration while discarding other dominant configurations regarding to the actual Pareto front (see the discarded configuration x_4 in Figure 1 (b) of [AGZ19].

SDF graph G level, before the scheduling of the corresponding SRSDF graph G^*[8]. It is one of the most common way to simplify the synthesis process: the entangled variables are solved by successive and independent solving methods, each method focusing on the computation of a specific category of variables, such as memory allocation. Considering all entanglements and related variables at the same time is difficult already in the domain of timing verification [Mai+18]; to synthesize all these variables at the same time is even more difficult. There exists a global CP formulation [RS14] for SDF graphs, but it does not take into account delay synthesis nor QoS configuration. In the work [RS14], the CP formulation is especially useful to not explore all possible schedules: it decreases the scheduling execution time but then the result may not be optimal.

ETs are also required to generate a concrete static schedule[9] but they are modified by it if memory and communication contention are taken into account [NYP16]. One way to avoid contention is to perform spatial or temporal isolation [Per+16]. Also, there exist heuristics to minimize the memory contention with SDF graphs, as one heuristic [Tra+19] dedicated to the Kalray many-core processor, however they require a precise knowledge of the memory accesses during the execution of an actor firing. Such knowledge can be modeled with PREM (PRedictable Execution Model) [Pel+11] or memory access patterns [Gho+12; Wan+14], but PREESM does not support any model about memory access. Thus, both memory and communication contentions are not taken into account in our DSE. When considering real-time constraints, such as periodic tasks, a few works are able to perform timing verification during the schedule synthesis [Did+19], but it does not adapt the QoS. In our DSE, we consider timing verification of partially periodic SDF graphs via the scheduler presented in Chapter 3 which automatically stops when the system is not schedulable. Non schedulable configurations are discarded. The support of partially periodic constraints and its scalability are the main advantages of the scheduler presented in Chapter 3, compared with the numerous schedulers developed for DAGs of tasks which could be used instead.

In our work, we consider that variables are solved separately in the following order:

S-1 modeling of ETs and energy per actor firing, PREESM can generate instrumented code to infer an averaged ET estimation[10];

S-2 valuation of all parameterized expressions and computation of repetition vector;

[8] For example, *Stream Graph Modulo Scheduling* [KM08] places delays after a first partial unfolding of the graph.

[9] A concrete static schedule specifies the start time and end time of each task, whereas some abstract schedules only give partial information, as an ordered list of tasks to execute.

[10] Either directly, or using PAPIFY [Mad+19] for a better accuracy.

S-3 if asked, flattening of the PISDF graph and call to the delay placement heuristic of Chapter 4;

S-4 unfolding of PISDF graph G in the SRSDF equivalent DAG G^*;

S-5 scheduling and mapping of firings (at the same time);

S-6 scheduling and routing of communications (at the same time);

S-7 memory allocation (buffer addresses of each firing input and output);

S-8 code generation (one file per Processing Element (PE)).

The generated code corresponds to the firings and communication function calls, distributed over threads according to the static non-preemptive scheduling on each PE in the target architecture. One thread is created per PE.

Note that during the PhD, we have worked with the team of Michael Masin at IBM Research Labs[11] to formulate efficient solving methods computing all entangled variables. However we did not succeed yet since the problem formulation involves many variables at the same time, which makes it difficult to verify and optimize. Two formulations have been tried:

- a suboptimal LP relaxation relying on the IBM ILOG CPLEX constraint solver which successively solves the variables of each category;
- an asymptotically optimal formulation relying on a non public tool extending the CPLEX capacities with a test-and-retry feature automatically generating formulations which randomly fix a subset of variables at each try.

Both formulations must be iterated multiple times to refine the solution.

The most complex feature of those IBM formulations was the memory allocation, which is optimized for the duplication special actors of PREESM. The optimization keeps the memory address of the data to duplicate and provide the same address to all outputs of the duplication actor instead of actually copying it. Such optimization is needed for SIFT which would use up to 2 Gbytes otherwise, and 10 times less with the equivalent optimization already present in PREESM thanks to memory scripts [Des+16b]. This optimization requires to double the number of buffer variables: *logical* buffers and *physical* buffers. Logical buffers respect the SDF original semantics while physical buffer

[11] See Footnote 2 on Page 98.

implement the optimization. Related equations also require a formulation of the dependencies between firings[12], itself depending on the size of delays. Despite our efforts, both formulations still encounter problems which have not been solved yet. For example, the optimal formulation does not respect the delay placement rules detailed in Section 4.1.

5.3 An exhaustive DSE method

The exhaustive DSE method that we have implemented in PREESM tests all possible combinations of malleable parameter sub-expressions of an application for a single target architecture. Each combination of malleable parameter sub-expressions is one possible configuration of the application, and defines one DSE point. The DSE is run after having modeled the application with PISDF, especially its malleable parameters and the ETs of its actors. After describing the main steps of the DSE, we detail the objectives used to select the best DSE point.

Concretely, the exhaustive DSE algorithm tests all possible DSE points by performing for each DSE point the workflow steps S-2 to S-5 according to the steps listed in Section 5.2.2. Step S-1 is the definition of parameterized entities, such as ETs; it is user responsibility. Energy and ETs are specific to the given architecture; all are set in the PREESM scenario file. The repeated steps S-2 to S-5 respectively:

S-2 valuate the given combination of malleable parameter sub-expressions;

S-3 optionally add delays with heuristic of Chapter 4 (see also Section 5.4);

S-4 convert the PISDF graph into its SRSDF unfolded form

S-5 schedule it with algorithm proposed in Chapter 3.

At the end of step S-5, metrics of the current DSE point are recorded and compared to the best current point, according to the objectives which are detailed below.

The objectives are an input of the algorithm. They may refer to the following metrics (encoded each with one letter):

- Throughput `T` inverse (corresponds to II duration);
- Latency `L` (maximum value of actor delay $\urcorner_{\max}$ defined in Equation (5.1)) and Makespan `M` (Latency multiplied by II duration);

[12] Generic equations for the dependencies between firings of the PISDF model have been formulated by Florian Arrestier in [Arr+19].

- Energy `E` and Power `P` (Energy of all firings divided by II duration).

The objectives using these metrics are either full minimization (encoded with a 0) or threshold to not exceed (encoded with any positive Integer). An example is given in Listing 5.3. The order of appearance of an objective defines its priority.

The total energy consumption and the II duration are computed during the scheduling process, respectively at the beginning and at the end of the process. The unit of the threshold objectives are the same as the metrics they refer to. The latency is evaluated right before the scheduling process (between steps S-4 and S-5) thanks to a graph traversal applying the $\urcorner_{\max}$ Max-Plus algebra variant of actor delay $\urcorner$ (see Equation (4.2)). The definition of actor delay $\urcorner$ implies that all incoming buffers create the same actor delay (i.e. feed-forward graph cut) so that it respects the original application semantics if adding new delays. However, here the designer may have added delays which do not form a feed-forward graph cut, and thus the equality is replaced by a maximum over all incoming buffers. $\urcorner_{\max}$ is formally defined in Equation (5.1); its unit is the number of pipeline stages, or equivalently the number of scheduler iterations on which one SDF graph iteration is spread. In the remaining part of this chapter, latency refers to $\urcorner_{\max}$ while makespan refers to $\urcorner_{\max}$ multiplied by II duration[13].

$$\forall \alpha \in V, \urcorner_{\max}(\alpha) = \max_{\{b \in E \mid \mathrm{dst}(b)=\alpha\}} \left\{ \urcorner_{\max}(\mathrm{src}(e)) + \frac{d_0(e)}{\mathrm{cons}(e) \times \vec{r}\,[\alpha]} \right\} \tag{5.1}$$

```
1. Comparisons :  T>L>P
2. Thresholds  :  0>2>10.4
3. Params objectives : >+Hvideo/image_height          \
                       >+Hvideo/AspectRatioDenominator
```

Listing 5.3 – Sample objective input for the DSE algorithm. The first line gives the priorities of each objective (highest priority on the left). The second line specify the threshold to not exceed for each objective of the first line (in the same order). A 0 encodes minimization. The third line gives the parameter objectives.

A second set of objectives can optionally be given by the designer to ask the mini-

[13] This makespan is actually an upper bound since the first firing in the SDF graph topological order may happen at any time during a scheduler iteration if some delays are located after this firing. Same reasoning is valid for the last firing in topological order, which might be the first inside a scheduler iteration if delays are located right before it. The lower bound of the makespan is $(\urcorner_{\max} - 1) \times T$ with T equal to the II duration.

mization or the maximization of any parameter of the PISDF graph. These objectives may be used to control the QoS of the application. These objectives are also prioritized, and are appended to the objectives based on metrics. Last line of Listing 5.3 gives an example of two parameter objectives. Character + encodes maximization while - encodes minimization.

Each objective is internally transformed as a DSE point comparator. Each DSE point contains all metrics and parameters valuation of the corresponding combination of malleable parameter sub-expressions. Minimization and maximization comparators are obvious, but positive thresholds require a specific implementation: two points are considered equivalent only if both have their metric lower than the threshold. For example, the power threshold comparator code is detailed in Listing 5.4.

```
1 public int compare(DSEpointIR arg0, DSEpointIR arg1) {
2   double power0 = (double) arg0.energy / (double) arg0.durationII;
3   double power1 = (double) arg1.energy / (double) arg1.durationII;
4   if (power0 > threshold || power1 > threshold) {
5     return Double.compare(power0, power1);
6   }
7   return 0;
8 }
```

Listing 5.4 – Code of the power threshold comparator of DSE points.

Finally, when testing each DSE point, the DSE algorithm calls a *global* comparator which iterates over all single objective comparators in the priority order. An objective comparator is called only if the ones of higher priorities did not return 0. In other words, the global comparator stops the comparison as soon as one DSE point is better than the other according to the currently tested objective. The global comparator code is detailed in Listing 5.5. Note that any kind of objective is allowed to appear multiple times in the list, possibly with different thresholds.

5.4 Improving DSE with automatic delay placement

When the highest priority DSE objective is the II duration minimization (i.e. throughput maximization), it might be useful to pipeline the applications as demonstrated in Chapter 4. After describing the general conditions to call the automatic pipelining heuristic, we detail the formulae used to compute the number of pipeline stages to add.

```
public int compare(DSEpointIR o1, DSEpointIR o2) {
  // variable ``comparators''
  // is the list of all objectives ordered by priority
  for (final Comparator<DSEpointIR> comparator : comparators) {
    final int res = comparator.compare(o1, o2);
    if (res != 0) {
      return res;
    }
  }
  return 0;
}
```

Listing 5.5 – Code of the global comparator of DSE points.

5.4.1 Adding delays to improve the results

Pipeline delays may improve the throughput, and thus especially benefit the II duration minimization objective. It may also help to respect a specified graph or actor period. However, adding delays increase the latency and makespan metrics. Moreover, reducing the II duration may increase the power consumption (since same energy for a smaller II). Due to these drawbacks, the automatic delay placement is only an optional feature of the DSE. Each original DSE point, is tested first without the delay placement heuristic. If the point does not already respect the DSE objectives, it is tested a second time after one call of the delay placement heuristic. At worst, this DSE improvement doubles the original number of tested DSE points.

Moreover, the delay placement heuristic is never called if a makespan minimization or a latency minimization objective has a highest priority than a II duration objective (minimization or threshold). As soon as the objectives include a latency threshold, the latency threshold becomes an upper bound of the number of accepted pipeline stages. If the minimum latency threshold is equal to 1 pipeline stage, no delay is accepted in the application and thus the delay placement heuristic is never called. Additionally, the heuristic is not called if any of power, latency or makespan threshold objective is not met. Indeed, adding pipeline stages to reduce the II duration will only worsen those objectives.

The delay placement heuristic works even if some size of delays are parameterized (see parameterizable entity E-4) since their valuation is resolved first. Unfortunately, the delay placement heuristic does not take into account the delays already present in

the graph, so it may add delays at the same place as preexisting delays. However, the number of pipeline stages to add is limited by the difference of any latency threshold and the latency of the current DSE point as we will see in the next subsection.

5.4.2 Precomputing the number of cuts

The delay placement heuristic presented in Chapter 4 requires two inputs: the number of graph cuts to add x, and the number of balanced graph cuts to preselect y. When calling the heuristic, we always set $y \leftarrow x + 1$ so that we introduce slightly more choices than necessary. Now we detail how x is computed.

Let's note L_{cur} the latency of the graph resulting from the current DSE point. By definition, $L_{\text{cur}} = \max_{\alpha \in V_{\text{cur}}} \{\ulcorner_{\max}(\alpha)\}$. Also, L_{THR} denotes the minimum value of any latency threshold objective. Then as stated at the end of the previous subsection, we obtain Equation (5.2).

$$x \leq L_{\text{THR}} - L_{\text{cur}} \tag{5.2}$$

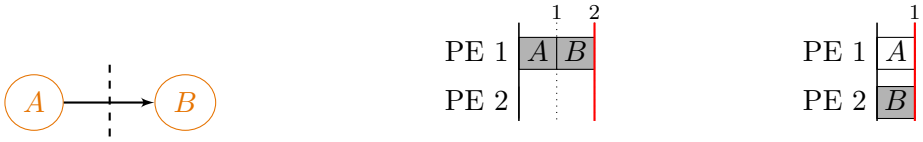
Based on the scheduling of the current DSE point without added delays, we know its II duration T_{cur}. If a too small graph period was set, it may happen that the scheduling process did not work, however in this case we know the relation $T_{\text{cur}} = T_{G_{\text{cur}}}$. Then it is possible to estimate how far was the current schedule from the ideal case where the processor utilization factor is full: $U_{\text{cur}} = m$. The estimate corresponds to an upper bound of the cuts necessary to reach this ideal case, as stated in Equation (5.3). In the ideal case $U_{\text{cur}} = m$, the fraction is equal to 1 and thus no cut is needed. Hence we add the term -1 to convert number of pipeline stages to number of cuts. x is without unit since we divide time unit by time unit.

$$x \leq \left\lceil \frac{T_{\text{cur}} \times m}{\sum_{\alpha \in V} \vec{r}_{\text{cur}}[\alpha] \times C_\alpha} \right\rceil - 1 \tag{5.3}$$

Then, if any II duration threshold objective is present, x is refined according to it. In this case, the number of cuts is upper bounded by the ratio of the current II duration T_{cur} on the threshold T_{THR}, as stated in Equation (5.4).

$$x \leq \left\lceil \frac{T_{\text{cur}}}{T_{\text{THR}}} \right\rceil - 1 \tag{5.4}$$

Figure 5.3 gives an example on the usage of T_{THR} to upper bound x. Considering $T_{\text{THR}} = 1$ and $T_{\text{cur}} = 2$, Equation (5.4) states to add 1 cut. If adding the only possible cut, between actors A and B, we indeed obtain a schedule where $T_{\text{THR}} = T_{\text{cur}}$ which



(a) Graph example. The only possible cut is represented with a dashed line.

(b) Two schedule examples of graph 5.3a on two PEs: on the left $T_{\text{cur}} = 2$ without any cut, on the right $T_{\text{cur}} = 1$ with one cut between actors A and B.

Figure 5.3 – Example to estimate number of cuts in DSE.

corresponds to the ideal case $U_{\text{cur}} = m$.

For makespan threshold, we use a ratio with the ideal case $U = m$. The number of cuts has to be less than the makespan threshold M_{THR} divided by the minimum ideal II duration. The upper bound is defined in Equation (5.5).

$$x \leq \left\lceil \frac{M_{\text{THR}} \times m}{\sum_{\alpha \in V} \vec{r}_{\text{cur}}[\alpha] \times C_\alpha} \right\rceil - 1 \tag{5.5}$$

Finally, x is selected as the maximum Integer respecting all Equations (5.2) to (5.5). If any of the considered threshold is not present in the objectives, then the corresponding equation is ignored. At worst, if there is no threshold at all, x is refined with Equation (5.3) only. The delay placement heuristic is not called if $x \leq 0$ since it would be useless. Note that Equations (5.2) and (5.5) are strong limits on the number of cuts, whereas Equations (5.3) and (5.4) are only informative upper bounds which are not forbidden to exceed. If exceeding bounds of Equations (5.3) and (5.4), memory usage will most probably increase while throughput gain will be negligible.

5.5 A naive heuristic for Integer malleable parameters

Another way to improve the DSE is to only explore a subset of all possible DSE points, so that the DSE execution time decreases. Randomly selecting the subset to explore may remove interesting points, so instead we focus on the malleable parameters holding only Integer numbers in their sub-expressions. Indeed, we expect a relationship between the evolution of any malleable parameter value and the evolution of some metrics of the corresponding DSE points. For example, if the value of a specific malleable parameter increases, the total number of firings may increase as well. Of course this is not always true, yet this is the only possible expectation without further symbolic analysis of the parameterized expressions. Hence, we have implemented a heuristic performing a kind of

dichotomy on the Integer numbers of malleable parameters. The heuristic works only for malleable parameters whose all sub-expressions are Integer numbers. For other malleable parameters, the heuristic is not called and all their combinations of sub-expressions are explored.

The heuristic is a kind of dichotomy on the sorted sets of values of all malleable parameters holding only Integer numbers. We refer to such malleable parameters as *Integer malleable parameters*. Two values are selected in the set of values of each Integer malleable parameters, and these two values replace temporarily the full set of possible values; an example is detailed in the next paragraph. Then, all combinations of other non Integer malleable parameter sub-expressions with those two values are explored, which gives a temporary best DSE point. The combination of sub-expressions of this best point is tracked to reduce the full set of values of each Integer malleable parameter. The non selected value of each Integer malleable parameter is removed, as well as all values below (respectively, above) if it was the smallest (respectively, greatest) value of the two values previously selected.

The set reduction process is iterated until all sets of values of the Integer malleable parameters contain only one value. The final best DSE point is selected among all combinations of sub-expressions over all iterations. The two sample values representing a set at each iteration[14] are symmetrically located at one third and two thirds of the set size. Thus, if $s_{\max}$ is the maximum size of all sets, approximately $\log_{\frac{3}{2}}(s_{\max})$ iterations are needed in total. $\log_{\frac{3}{2}}(s_{\max})$ is an over approximation of the number of iterations since the maximum set size minus 2 (for the two selected values to test at each iteration) might not be a multiple of 3. More precisely, $\left\lfloor \frac{s_{\text{cur}}-1}{3} \right\rfloor + 1$ values are removed from the set containing s_{cur} values at each iteration. For an Integer malleable parameter holding 5 different values, 3 iterations are required to fix it, whereas $\log_{\frac{3}{2}}(s) \approx 3.419$. Figure 5.4 depicts a set reduction example of an Integer malleable parameter holding 13 Integer values.

At each iteration, all combinations of non Integer malleable parameter sub-expressions are tested, along with the only two values of any Integer malleable parameter. Consequently, the heuristic is useful especially when most of the malleable parameters are Integer malleable parameters.

[14] The set may actually have been already reduced to a unique value. In such case, the parameter is considered as fixed.

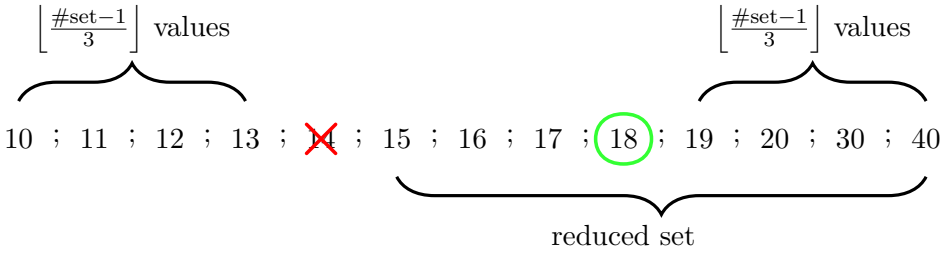


Figure 5.4 – Dichotomy bounds example of an Integer malleable parameter. Only values 14 and 18 have been tested in the previous DSE iteration because they split the set in three subsets of equivalent sizes. Here the temporarily best value is 18: it gives a better DSE point than the value 14. Thus, the reduced set of values for the next DSE iteration contains only values greater than 14.

5.6 Evaluation

In this section, we present an evaluation of the DSE method detailed in the previous sections. The evaluation is exclusively based on the SIFT video application, executed on an homogeneous architecture with 4 PEs. The DSE execution time have been measured for the exhaustive DSE and for its delay and Integer heuristics. The best DSE point found is discussed for these different DSE cases.

5.6.1 Test application: live video SIFT

The SIFT live video application is a modified version of SIFT presented in Section 2.1. Note that this version does not depend on any external image processing library. The live video version uses a webcam stream to retrieve the input image and as an output, it displays the keypoints above the input image in real-time. The application runs indefinitely and its frame rate is limited either by the SIFT processing time or by the webcam frame rate (30 fps maximum in our case, hardware dependent).

Malleable parameters

In order to control the processing time of the SIFT application, we have defined the following malleable parameters in its PISDF graph:

MP-1 `imgDouble` to upsample the original image once via a parameterized path (true by default);

MP-2 `AspectRatioDenominator` to select the image aspect ratio, `4:3` (= `16:12`, by default) or `16:9`;

MP-3 `image_width` to select the image width, `640` (by default) or `320` pixels;

MP-4 `parallelismLevel` to select the degree of data parallelism (corresponding to the parameter p introduced in Chapter 2) in $\{1; 2; 4; 5; 10\}$ (`4` by default);

MP-5 `delayRead` and `delayDisplay` to add delays on pre-processing and post-processing (false by default);

MP-6 `NumeratorFrequency` to imitate a frequency selection, `1:1` (by default) or `3:4` or `5:4`;

MP-7 `nKeypointsMaxUser` for the number of detected keypoints, in $\{80; 90; ...; 150; 160\}$ (`100` by default).

All those malleable parameters hold only Integer values, so the heuristic presented in Section 5.5 can be called to reduce the number of tested DSE points.

Main objectives of the DSE are a throughput threshold (to ensure 30 fps) and a makespan threshold (below 100 ms to not disturb the user[15]). The next parameter objectives gives a higher priority on the image resolution (`image_height`) and the quality of the processing (`imgDouble`). The objective input is detailed in Listing 5.6.

```
1. Comparisons :  T>M
2. Thresholds  :  33000000>100000000
3. Params objectives : >+Hvideo/image_height          \
                       >+SIFT/imgDouble               \
                       >+Hvideo/AspectRatioDenominator\
                       >-Hvideo/parallelismLevel      \
                       >-Hvideo/NumeratorFrequency    \
                       >+Hvideo/nKeypointsMaxUser
```

Listing 5.6 – Objective input for the DSE evaluation of the SIFT video application.

Delay placement

Two malleable parameters (`delayRead/Display`, MP-5 in the preceding list) are used to add pipeline delays delimiting pre- and post-processing (such as color scale to grey

[15] A very subjective and limited experiment has shown that seeing his/her own reflection with a delay of more than 100ms on a webcam is annoying.

scale conversion). The placement of these pipeline delays has not been deduced from the heuristic presented in Chapter 4. Indeed, the heuristic depends on the repetition vector of the application, which depends on the malleable parameter `parallelismLevel`. The heuristic also depends on the ETs of the actors, modified by all the aforementioned parameters, except MP-5 for the delays. As the delay placement should be recomputed for each configuration, we have chosen to set by ourselves two possible placements, and also to call the delay heuristic after the scheduling process, as seen in Section 5.4. If delays are added by the heuristic at the same place as specified by `delayRead/Display`, their sizes are summed (so it is a useless DSE point).

Execution times and energy modeling

Timings are linearly parameterized by the number of pixels, by the degree of parallelism (as done in Equation (2.2)), and by the frequency. For example, the `yuv_to_rgb` pre-processing actor has the following timing:

```
(1150000.0*tot_image_size*DenominatorFrequency)/
(RefTotSize*parallelismLevel*NumeratorFrequency)
```

Here `1150000` is the ET of actor `yuv_to_rgb` measured in nanosecond for the default number of pixels `RefTotSize`. This linear computation of the ET is an ideal case which does not respect the Amdahl's law [Amd67], but it is sufficient regarding to the high degree of data parallelism of the processing actors and to the small number of PEs in the target architecture.

Unfortunately, we did not performed actual measurements for the energy consumption. However, the parameterizable expressions of energy support common energy model (for example, see equations (2.4) and (2.5) in the thesis of E. Nogues [Nog16]). Currently, the DSE computes the energy as the sum of all energy expressions per actor firing, without considering the energy of the processor alone. The power is obtained by dividing the computed energy by the II duration. Moreover, the frequency minimization objective that we define corresponds to a better power efficient strategy than *idle-to-race* as stated in [Hol+14]. Another reason to not model energy in our experiments is that Dynamic Voltage and Frequency Scaling (DVFS) [IY98] is automated on our laptop processor and it is difficult to enforce its frequency.

5.6.2 Experiments

Experiments were performed on a laptop with an Intel i7-7820HQ @ 2.90GHz processor (4 physical cores) and the GCC compiler version 7.5.0 (option `-O2`) on Ubuntu 18.04.

Note that after the DSE, a different scheduler (legacy PREESM scheduler, FAST initial step [KAG96]) is called. This scheduler is longer to execute and cannot be used in the DSE, especially because it takes communications into account and because it performs a best fit mapping while the DSE scheduler only performs first fit mapping (see Chapter 3).

Two experiments have been performed. The first experiment assesses the choice of the degree of parallelism of SIFT introduced in Chapter 2. The second experiment details the DSE execution times and best DSE points depending on the DSE options (exhaustive, delay heuristic, Integer heuristic).

Throughput versus degree of parallelism

Figure 5.5 depicts an excerpt of the exhaustive DSE with the SIFT malleable parameters MP-1 to MP-4. The II duration is represented in function of the application configuration, with one line per degree of parallelism. The II duration is computed offline by the scheduler, it is not measured on actual executions of the application. The orange horizontal line corresponds to the II duration threshold objective ensuring 30 fps.

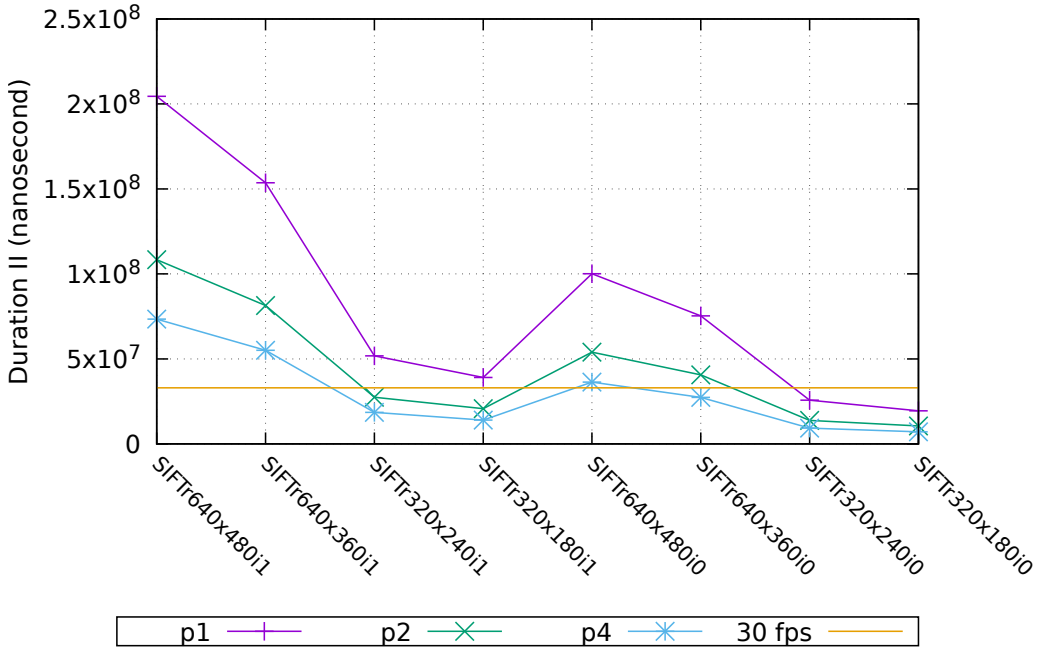


Figure 5.5 – Duration of II as function of the SIFT application configuration, for different degrees of parallelism p and four PEs. `r640x480i0` encodes the image resolution and the value of the `imgDouble` parameter. All other parameters are fixed to their nominal value.

As expected, the II duration decreases when the degree of parallelism increases; their lines never cross each other. The degree of parallelism 4 (denoted `p4` in Figure 5.5) has the highest number of DSE points below the 30 fps II duration threshold. There are also four PEs in the target architecture. Thanks to the parameter objectives detailed in Listing 5.6, the exhaustive DSE method selects the best point having an image resolution of 640x360, without the initial image upsample path (malleable parameter MP-1 set to false), and with the degree of parallelism of 4. Note that even if including the delay malleable parameter MP-5 in the DSE, the best point remains unchanged. This result demonstrates that for the SIFT application, it is better to adapt the degree of parallelism to the number of available PEs, than to pipeline the application. Moreover, the DSE is able to select an acceptable configuration, which meets all threshold requirements: II duration and makespan.

DSE execution time versus best point quality

DSE execution times are reported in Table 5.1, with and without delay and Integer heuristics. Each point takes around half a second (538 millisecond) to be explored. However this is an average since the scheduling process is linearithmic in the repetition vector, itself depending on the malleable parameter values. Half a second per point is not so fast, and advocates for the Integer heuristic to decrease the number of explored DSE points.

The number of explored points and the best one are also reported in Table 5.1. As for the previous experiments, delays are almost never added, except with the Integer heuristic (`delayRead`). Using both heuristics for malleable parameters MP-1 to MP-5 divides by a factor two the number of explored points (160 to 80). On one side, the delay heuristic triggers slightly more points to explore, and on the other side, the Integer heuristic drastically decreases the number of points to explore. For example, with both heuristics for malleable parameters MP-1 to MP-7, only 316 points are explored over 4320 in total. The heuristics do not modify the best point quality: the same image resolution is selected for all types of DSE. However, with malleable parameters MP-1 to MP-7, the highest frequency is selected, allowing for an increase of the aspect ratio, from `16:9` to `16:12`, and an increase of the number of keypoints, from `100` to `160`.

Finally, the degree of parallelism selected by our DSE is always equal to the number of available PEs, 4 in this experiment. In the meantime, delays are almost never added. In this experiment on the SIFT application, it is more efficient to adapt the degree of parallelism than to pipeline the application graph. This conclusion cannot be generalized

to any graph and any number of PEs. Indeed the functions that we used to model the actor ETs are inversely proportional to the degree of parallelism, which might favor parallelism over pipelining. In the general case, this linear model is too optimistic. For example, doubling the number of PEs executing a task would divide by less than two the total ET of that task.

DSE type	M. Param.	#Points	DSE time	Best point	Delays
Exhaustive	MP-1 to MP-5	160	1 m 26 s	r640x360i0p4	none
Delay	MP-1 to MP-5	186	1 m 43 s	r640x360i0p4	none
Integer	MP-1 to MP-5	68	32 s	r640x360i0p4	`delayRead`
Delay+Integer	MP-1 to MP-5	80	38 s	r640x360i0p4	none
Exhaustive	MP-1 to MP-7	4320	37 m 19 s	r640x480i0p4	none
Delay+Integer	MP-1 to MP-7	316	2 m 57 s	r640x480i0p4	none

Table 5.1 – Results of DSE on SIFT video for different configurations of the DSE algorithm. Column #Points correspond to the number of points actually explored by the DSE, possibly different than the total number of DSE points (higher if delay heuristic, lower if number heuristic).

5.7 Related work

In our contribution, we explore multiple application configurations on the same architecture, with a unique scheduler. Thus, our work is related to multi-objective DSE, to parameterized SDF applications, and also to parallelism grain adaptation.

5.7.1 Multi-objective DSE

Numerous works target DSE of applications on multi-processors [Pim17]. Joint energy consumption and real-time scheduling optimization have been widely studied using DVFS, for both offline [WRG16] and online [GGP17] cases, and even considering the temperature [AGZ19]. To these works, we can add a few tools also taking into account energy and real-time constraints, which compute the best mapping [Abd14] or the best mapping and frequency at the same time [Yan+19]. Finer compilation optimizations may also result in energy trade-offs, such as function inlining [MF20]. However, in our work, we stay at a coarse representation level of the applications, and we did not investigate the multiple existing compilation optimizations, especially the SSE/AVX vector extensions. Moreover, we focus on SDF graphs whereas none of the aforementioned work

supports it.

More specifically, there exists a previous work optimizing makespan and throughput objectives for scheduling of SDF graphs, using ILP formulation and heuristics [LGE12]. However, their SDF input graph is not parameterized. Considering parameters, the IGOR tool [Smi+19] helps to deduce the impact of each parameter, but it is especially useful if there are numerous possible values per parameter. In our experiments, half of the parameters are binary decision variables (malleable parameters MP-1 to MP-3 and MP-5).

Closely to the SDF model, the MASES tool [YKG18] optimizes throughput, makespan and processor utilization of SRSDF graphs. Another tool [Kan+12] performs multi-objective DSE of a variant of KPNs, especially focusing on the mapping and the network usage. Code generation and DSE [Sch+19] has also been studied for an extended version of SDF with dynamic actors, but it does not compute the placement of pipeline delays.

Additionally, many DSE tools consider multiple variables while optimizing a single objective. The only objective usually is the throughput (see DSE of SRSDF applications with communication contention [KBB06]), or the schedulability according to any requirement (for example, see DSE of Unified Modeling Language (UML) applications [Apv+06]). Yet they may integrate other objectives if expressed as strong requirements, as it is the case of FoRTReSS [Duh+15] for the DSE of an SRSDF variant on FPGAs, taking into account communications and floorplanning. Finally, DAMSHE methodology [Sur+19] also takes into account heterogeneous hardware with both FPGAs and regular CPUs, but the DAMHSE methodology is not an automatic DSE: it is driven by the designer.

5.7.2 Parameterized SDF MoCs

A complete survey of parameterized SDF MoC has been done by Bouakaz et al. [BFG17]. One of the oldest work [BB01] separates the parameter valuation from their usage; all parameters must be resolved and fixed before each graph (or *subgraph*) iteration. Other work [TB06] analyses SDF graphs having intervals as production and consumption rates.

Parameterized SDF graph raise interesting problems about when the parameters can change their value during the graph iterations. However, in our work, all parameters are fixed for all iterations of the application. Consequently, we do not support parameters depending on any actor output. Such limitation is an advantage for other aspects, especially the memory usage which is predicted and optimized after scheduling.

Other parameterized models take advantage of symbolic analysis of the rates, as

does BPDF (Boolean Parametric DataFlow) [Bem15]; but BPDF rate expressions are restricted to multiplications. Unfortunately, our DSE algorithm does not benefit from the symbolic expression of parameters: they are all evaluated and fixed before scheduling. Thus we cannot benefit from promising parametric scheduling [PGB19], which defines regions of equivalent schedules for tasks without dependencies. Similar parameter synthesis techniques already exist for Timed Automata, as the IMITATOR tool [And+12b].

5.7.3 Parallelism adaptation

An important aspect of parameters is their influence over the parallelism of the application. They may modify both data parallelism (through rate divisions or multiplications) and task parallelism (trough path selection, with rates equal to 0 in PREESM).

For example, a method identifies *hot actors*, i.e. bottlenecks, and provides a heuristic to split them [Far+11]. Similarly, multiple tools rely on SDF graph unfolding to obtain efficient schedules [KM08; Zhu+16]. There exists also a Mixed Integer Linear Programming (MILP) formulation to select the level of intra-parallelism [Zho+16] (here, note that each task may require multiple processors at the same time). Besides, an extension of the SDF MoC, Scalable Synchronous Dataflow, supports the automatic modification of the repetition factor of some actors via a *vectorization* algorithm [Rit+93]. Graph pattern detection and substitution is another way to reduce parallelism [CS12].

In our experiments, a single malleable parameter was controlling the degree of data parallelism of multiple actors. Finer optimizations could configure this degree of data parallelism independently for each actor in order to produce the best mix between task parallelism and data parallelism according to the graph topology. To our knowledge, this specific problem has not been widely studied for SDF graphs, but algorithms exist for a kind of DAGs with parameterized data parallelism [Mar+18] executed on heterogeneous target architectures [NSC07]. In the domain of DAG scheduling, such parameterized parallelism corresponds to *moldable tasks* if the degree of parallelism is static, and to *malleable tasks* if the degree of parallelism can be changed dynamically. Algorithms developed for moldable tasks may integrate multiple objectives as makespan and energy minimization [DS10]. This fine data parallelism adaptation is especially needed when considering clustered architectures with constraints on input and output data locality [BDS02]. Unfortunately, our scheduler does not consider neither data locality nor communication times to move data.

Finally, only a few models support the modification of task parallelism, since it modifies the application semantics. However, it is an important feature when dynamically

adapting the QoS. The RDF (Reconfigurable DataFlow) model [Fra+19] supports it via graph rewriting. The BPDF model [Bem15] supports the parameterized removal of existing actors, but not their addition, via a Boolean activation input. The PREESM tool supports the parameterized removal of existing actors, but not their addition, with a few restrictions on the paths containing them.

5.8 Conclusion

Our DSE approach helps to design dataflow processing applications having minimum QoS requirements, while minimizing other objectives, as latency for example. Instead of assessing multiple target architectures or multiple possible mappings, the presented DSE explores the available application configurations expressed with PISDF malleable parameters. As in another work dedicated to the design of a real-time computer vision application [AVA19], our DSE approach especially helps to compute the appropriate degree of parallelism p, yet only if this degree of parallelism is modeled with a malleable parameter in our case. If the processor frequency is also modeled with a malleable parameter, we are able to find a suitable one given throughput or energy objectives.

Our approach naively performs an exhaustive search by default, and does not benefit from a symbolic analysis of the application parameters. However, efficient heuristics for scheduling, pipelining and parameter value selection, help to perform our DSE algorithm in a reasonable time: a few minutes at most in our experiments. The presented DSE can be improved in many directions, but such improvements would most likely increase the DSE execution time.

Among possible improvements of our DSE, the most important one is to adapt its internal tools (for scheduling and pipelining) to heterogeneous target architectures. Such heterogeneous architectures also call for communication modeling (especially if memory is distributed), which triggers the problem of communication contention on shared buses for example. Memory is also a source of contention, and the memory footprint minimization is not currently available as an objective. Both communication and memory modelings require to heavily extend the capabilities of the DSE algorithms relying on the PISDF application model and the S-LAM architecture model used by PREESM.

Another important future work is to consider multiple applications to configure together. It is easy to add a dummy actor connecting two application graphs together so that both applications are seen as a single graph by our algorithms. With this method, the fairness of the processor usage between the two applications may also be configured with malleable parameters on the rates of the edges coming from the dummy actor.

However, connecting the graphs of multiple applications implies that DSE objectives as makespan and throughput are not anymore specific to each application, which is a major drawback. A simpler future work is to implement other methods to select the best DSE point according to the given objectives. The current method selects the best point by prioritizing the objectives, while the domain of Multi-Criteria Decision Analysis (MCDA) offers numerous other methods. Finally, the easiest improvement to perform is to test multiple DSE points in parallel: for now, all points are tested sequentially, one after the other.

Dissemination and Implementation

The contribution presented in this chapter has not been published nor submitted yet for peer-review. The algorithms presented in this chapter have been implemented as workflow tasks of the PREESM tool. See the following task description for the implementation of all DSE algorithms described in this chapter:
- `pisdf-mparams.setter`

Conclusion

In this thesis, we have studied the problem of configuration of parameterized Synchronous Data Flow (SDF) graphs with Quality of Service (QoS) constraints on heterogeneous multi-processor architectures with hardware constraints. In order to solve this problem, we have proposed four contributions regarding modeling, scheduling, pipelining and configuration itself. Unfortunately, these contributions are limited to homogeneous multi-processors. Nevertheless, experiments have shown that the configuration can be performed in a reasonable time, a few minutes, with multiple parameters as the image resolution and the multi-processor frequency. Energy, throughput and latency constraints are also taken into account. Contributions are summarized in the next section. A discussion on future work closes this thesis.

Summary of contributions

The four contributions presented in this thesis cover different aspects of the design of a dataflow processing application. The first design step is the modeling of the application; then the application has to be configured to meet some constraints coming from hardware or software side, as the frame rate of a camera. Design Space Exploration (DSE) is a common technique to perform this configuration, and in our case it uses scheduling and pipelining techniques. However, due to the exponential number of possible configurations, an exhaustive DSE may be prohibitive. Thus, most of the algorithms presented in this thesis are heuristics dedicated to accelerate the DSE and to scale to large SDF graphs with dozens of actors. Our main use-case, the Scale Invariant Feature Transform (SIFT) image processing application contains around 70 actors and is constrained with a minimum throughput and a maximum latency. Thanks to the contributions we are able to model SIFT with a parameterized SDF graph and to automatically configure its image resolution to respect both throughput and latency constraints. All contributions have been implemented in the PREESM open-source tool.

Modeling Two contributions, detailed in Chapters 2 and 3, are related to modeling. First we have shown in Chapter 2 how to model a category of imperative `for` loops with the SDF model, so that theses `for` loops can be parameterized and adapted to the number of Processing Elements (PEs) in the target architecture. This contribution eases the design process since it avoids using more complex dataflow models as the Cyclo-Static Data Flow (CSDF) Model of Computation (MoC). Then, in Chapter 3, we have discussed the analysis of SDF graphs having periodic constraints on some of their actors, to represent the behavior of some components of a processing application, especially its sensors and actuators.

Scheduling While modeling is the first design step usually manually performed by the designer, scheduling is the main next step, to automatically execute the modeled application. In Chapter 3, a fast non-preemptive offline scheduler has been introduced to schedule SDF applications, even when having periodicity constraints. This scheduler is highly scalable and experiments show that is has a fair quality compared to an optimal one: global Earliest Deadline First (EDF).

Pipelining A common optimization of SDF graphs is to pipeline them by adding *delays* to break some of their data dependencies. While this optimization may dramatically improve the throughput of an application, it also increases its memory footprint and its latency. In Chapter 4, we define *admissible* graph cuts to insert delays on SDF graphs so that the application semantics is not modified. We also present a pipelining heuristic to automatically compute valid delay placements on a subset of such admissible cuts. This pipelining heuristic has been tested on various SDF graphs, and it is especially efficient when considering a large number of PEs in the target architecture. The heuristic is performed before the scheduling step during the design process.

Configuration Our last contribution, presented in Chapter 5, uses all the aforementioned contributions to select the best configuration of a parameterized SDF graph according to some constraints on energy, latency, throughput and QoS. We propose an exhaustive DSE algorithm and two heuristics to configure the application according to the constraints. The heuristics either reduce the number of explored configurations to accelerate the DSE, or improve each explored configuration by automatically performing an additional pipelining step. Our three DSE algorithms find a suitable configuration of the SIFT image processing application, optimizing the image resolution under throughput and latency constraints.

Future work

Combined together, our contributions enable us to configure a parameterized SDF graph on a multi-processor homogeneous architecture. However, multi-processors, and especially Multi-Processor System-on-Chips (MPSoCs) are now embedding various kinds of processors which are specialized to reduce the energy consumption or to accelerate the computations of specific algorithms. All our contributions have to be adapted to take into account heterogeneous multi-processor architectures. We think that such adaptation is achievable in short-term for the modeling, pipelining and configuration contributions, but our scheduler requires more work, especially if integrating communication costs.

More broadly, we think that promising but difficult future work is to develop an end-to-end global approach to accurately configure applications: from modeling to actual execution. It implies at least two challenges: (1) powerful symbolic analysis to derive the importance of each variable of a system, and (2) simulation and actual execution of configurations to better match the reality. Regarding challenge (1), symbolic analysis could allow us to parameterize the repetition vector, even when parameters use complex arithmetic expressions. Regarding challenge (2), only a subset of the complete behavior of an application in its actual environment is effectively and accurately modeled. Thus, we always need to confront reality. Unfortunately, our configuration algorithm does not use experimental results to refine its best, theoretical, configuration. For instance, Execution Times (ETs) can be estimated by Parallel and Real-time Embedded Executives Scheduling Method (PREESM) but they are not automatically refined.

Finally, in this thesis we have focused on static applications and offline synthesis, for scheduling, and pipelining and configuration. However, another global future work is to adapt our contributions to dynamic environments, for example, in the SDF runtime manager Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPIDER). A possible use-case is the reconfiguration of the degree of parallelism p and the QoS when one or multiple PEs are not anymore available, due to a failure or a power shortage. SDF iterators may already be modeled with SPIDER. In this case, the iterator not only stores the values of the indexes, but it also computes them. But computing the best suitable degree of parallelism as done in the last contribution may still require multiple minutes, which is too long. We believe that both aforementioned challenges (1) and (2) would help this future work: symbolic analysis would accelerate the *re*configuration and online actual executions would detect the need for reconfiguration.

Hereafter we detail future work specific to each contribution.

Modeling On modeling, the main long-term future work is to automatically transform applications written with an imperative programming language to the SDF MoC. Indeed this complex design step is often performed manually by the designer. Such transformation is difficult first because the SDF MoC is less expressive than almost all imperative languages, and second because a suitable dataflow granularity has to be extracted. Regarding periodicity constraints, the main long-term future work is to complete them with latency constraints. Indeed, an actuator may be dependent on a sensor activation and be required to operate before a specific deadline, shorter than the period if any.

Scheduling The main future challenge for our scheduling algorithm is to consider heterogeneous multi-processors and communication times between the processors. However, taking into account communications also imply taking into account communication contention. Similarly, we did not take into account neither memory contention nor data locality. This challenge is especially difficult to solve if the scheduler has to remain fast: adding new variables and constraints will most probably reduce its rapidity and scalability. Moreover, it requires a precise model of available means of communication on the target architecture, and a precise model of memory accesses. A short-term future work, is to adapt our scheduling algorithm to heterogeneous multi-processors without taking into account communications.

Pipelining Pipelining of SDF graphs may increase the memory footprint of the application. Our heuristic focuses on a subset of valid delay placements to be fast, thus potentially excluding good solutions which imply a smaller memory footprint. Yet, adding more solutions to explore will result in a longer execution time of the heuristic. Another way to improve the pipelining heuristic is to benefit from a symbolic analysis of the SDF graph parameters in order to output the placement and parameterized sizes of delays to add instead of their concrete Integer sizes. This long-term future work would especially benefit the last contribution regarding automatic configuration.

Configuration The current configuration is naive and close to brute-force. A symbolic analysis of the SDF graph parameters would allow us to infer rapidly precious information without performing scheduling. One of this information is the repetition vector. However, such symbolic analysis is a long-term future work if considering complex arithmetic expressions in parameters. Another long-term future work is to consider multiple applications at once. This is already possible by connecting the SDF graphs of the applications, but then the configuration objectives are not specific anymore to each

application. A short-term future work is to integrate multiple methods to compare the explored configurations. Currently, objectives have to be prioritized, which allows for a best configuration definition. Different methods, including Pareto front generation, would benefit to the designer when some objectives cannot be prioritized.

Bibliography

[AAP15] H. I. Ali, B. Akesson, and L. M. Pinho. "Generalized Extraction of Real-Time Parameters for Homogeneous Synchronous Dataflow Graphs". In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Mar. 2015, pp. 701–710 (cit. on p. 92).

[Abd14] N. Abdallah. "Multiprocessor real-time partitioning with Quality of Service requirements and energy constraints". Theses. Université de Nantes, Feb. 2014. URL: https://hal.archives-ouvertes.fr/tel-01332443 (cit. on p. 144).

[AC86] Arvind and D. E. Culler. "Dataflow Architectures". In: *Annual Review of Computer Science* 1.1 (1986), pp. 225–253. eprint: https://doi.org/10.1146/annurev.cs.01.060186.001301. URL: https://doi.org/10.1146/annurev.cs.01.060186.001301 (cit. on p. 13).

[Agu+15] M. A. Aguilar, J. F. Eusse, R. Leupers, G. Ascheid, and M. Odendahl. "Extraction of Kahn Process Networks from While Loops in Embedded Software". In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. Aug. 2015, pp. 1078–1085 (cit. on p. 9).

[AGZ19] A. Abdi, A. Girault, and H. Zarandi. "ERPOT: A Quad-Criteria Scheduling Heuristic to Optimize Execution Time, Reliability, Power Consumption and Temperature in Multicores". In: *IEEE Transactions on Parallel and Distributed Systems* 30.10 (Oct. 2019), pp. 2193–2210. URL: https://hal.inria.fr/hal-02400019 (cit. on pp. 129, 144).

[All+95] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. "Software Pipelining". In: *ACM Comput. Surv.* 27.3 (Sept. 1995), pp. 367–432. URL: http://doi.acm.org.insis.bib.cnrs.fr/10.1145/212094.212131 (cit. on pp. 26, 117).

[Amd67] G. M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. URL: https://doi.org/10.1145/1465482.1465560 (cit. on pp. 52, 141).

[Amm+14] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and M. Abid. "MARTE to PiSDF transformation for data-intensive applications analysis". In: *Design & Architectures for Signal & Image Processing (DASIP)*. Madrid, Spain, Oct. 2014. URL: https://hal.archives-ouvertes.fr/hal-01122725 (cit. on p. 28).

[And+12a] H. Andrade, J. Correll, A. Ekbal, A. Ghosal, D. Kim, J. Kornerup, R. Limaye, A. Prasad, K. Ravindran, T. N. Tran, M. Trimborn, G. Wang, I. Wong, and G. Yang. "From Streaming Models to FPGA Implementations: ERSA'12 Industrial Regular Paper". In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. WORLDCOMP'12. July 2012, p. 1. URL: http://worldcomp-proceedings.com/proc/proc2012/ersa.html (cit. on p. 27).

[And+12b] É. André, L. Fribourg, U. Kühne, and R. Soulat. "IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems". In: *FM 2012: Formal Methods*. Ed. by D. Giannakopoulou and D. Méry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 33–36 (cit. on p. 146).

[Apv+06] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. "A UML-based Environment for System Design Space Exploration". In: *2006 13th IEEE International Conference on Electronics, Circuits and Systems*. Dec. 2006, pp. 1272–1275 (cit. on pp. 28, 145).

[Arr+18] F. Arrestier, K. Desnos, M. Pelcat, J. Heulot, E. Juarez, and D. Menard. "Delays and States in Dataflow Models of Computation". In: *SAMOS XVIII*. Pythagorion, Greece, July 2018. URL: https://hal.archives-ouvertes.fr/hal-01850252 (cit. on pp. 17, 96, 102).

[Arr+19] F. Arrestier, K. Desnos, E. Juarez, and D. Menard. "Numerical Representation of Directed Acyclic Graphs for Efficient Dataflow Embedded Resource Allocation". In: *ACM Trans. Embed. Comput. Syst.* 18.5s (Oct. 2019). URL: https://doi.org/10.1145/3358225 (cit. on pp. 22, 132).

[Aug+11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–198. URL: http://hal.inria.fr/inria-00550877 (cit. on pp. 5, 25).

[AVA19] T. Amert, S. Voronov, and J. Anderson. "OpenVX and Real-Time Certification: The Troublesome History". In: *2019 IEEE Real-Time Systems Symposium (RTSS)*. 2019, pp. 312–325 (cit. on p. 147).

[Bac+92] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and linearity*. 1st. Wiley, 1992 (cit. on p. 29).

[Bal+89] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. "The parascope editor: an interactive parallel programming tool". In: *Supercomputing, 1989. Supercomputing '89. Proceedings of the 1989 ACM/IEEE Conference on*. Nov. 1989, pp. 540–550. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5348986 (cit. on p. 5).

[Bam+02] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya. "Intermediate Representations for Design Automation of Multiprocessor DSP Systems". In: *Design Automation for Embedded Systems* 7.4 (Nov. 2002), pp. 307–323. URL: https://doi.org/10.1023/A:1020307222052 (cit. on p. 22).

[BB01] B. Bhattacharya and S. S. Bhattacharyya. "Parameterized Dataflow Modeling for DSP Systems". In: *IEEE TRANSACTIONS ON SIGNAL PROCESSING* 49 (2001), pp. 2408–2421 (cit. on pp. 21, 56, 145).

[BB11] M. Bertogna and S. Baruah. "Tests for Global EDF Schedulability Analysis". In: *J. Syst. Archit.* 57.5 (May 2011), pp. 487–497. URL: http://dx.doi.org/10.1016/j.sysarc.2010.09.004 (cit. on p. 89).

[BB91] A. Benveniste and G. Berry. "The synchronous approach to reactive and real-time systems". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1270–1282 (cit. on p. 8).

[BDS02] V. Boudet, F. Desprez, and F. Suter. *One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication*. Research Report RR-4591, LIP RR-2002-34. INRIA, LIP, 2002. URL: https://hal.inria.fr/inria-00071994 (cit. on p. 146).

[Bel+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect". In: *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*. Feb. 2008, pp. 88–598 (cit. on p. 11).

[Bem15] E. Bempelis. "Boolean Parametric Data Flow Modeling - Analyses - Implementation". Theses. Université Grenoble Alpes, Feb. 2015. URL: https://tel.archives-ouvertes.fr/tel-01148698 (cit. on pp. 21, 146, 147).

[Ben+10] M. Benazouz, O. Marchetti, A. Munier-Kordon, and P. Urard. "A new approach for minimizing buffer capacities with throughput constraint for embedded system design". In: *ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010*. May 2010, pp. 1–8 (cit. on p. 30).

[Ben+12] A. Benabid-Najjar, C. Hanen, O. Marchetti, and A. Munier-Kordon. "Periodic Schedules for Bounded Timed Weighted Event Graphs". In: *IEEE Transactions on Automatic Control* 57.5 (May 2012), pp. 1222–1232 (cit. on p. 28).

[Ben+13a] M. Benazouz, A. Munier-Kordon, T. Hujsa, and B. Bodin. "Liveness Evaluation of a Cyclo-static DataFlow Graph". In: *Proceedings of the 50th Annual Design Automation Conference*. DAC '13. Austin, Texas: ACM, 2013, 3:1–3:7. URL: http://doi.acm.org.insis.bib.cnrs.fr/10.1145/2463209.2488736 (cit. on p. 29).

[Ben+13b] A. Benoit, Ü. V. Çatalyürek, Y. Robert, and E. Saule. "A Survey of Pipelined Workflow Scheduling: Models and Algorithms". In: *ACM Comput. Surv.* 45.4 (Aug. 2013). URL: https://doi.org/10.1145/2501654.2501664 (cit. on p. 26).

[Bér+05] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux. "Comparison of Different Semantics for Time Petri Nets". In: *Automated Technology for Verification and Analysis*. Ed. by D. A. Peled and Y.-K. Tsay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 293–307 (cit. on p. 7).

[Bes+10] L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin. "Compilation of Polychronous Data Flow Equations". In: *Synthesis of Embedded Software*. Ed. by S. K. Shukla and J.-P. Talpin. Springer, 2010, pp. 1–40. URL: https://hal.inria.fr/inria-00540493 (cit. on p. 8).

[Bes+14] L. Besnard, A. Bouakaz, T. Gautier, P. Le Guernic, Y. Ma, J.-P. Talpin, and H. Yu. "Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony". In: *Science of Computer Programming*. Science of Computer Programming (July 2014), p. 20. URL: https://hal.inria.fr/hal-01095010 (cit. on p. 9).

[BFG16a] A. Bouakaz, P. Fradet, and A. Girault. *Symbolic Analysis of Dataflow Graphs (Extended Version)*. Research Report 8742. Inria - Research Centre Grenoble – Rhône-Alpes, Jan. 2016. URL: https://hal.inria.fr/hal-01166360 (cit. on p. 30).

[BFG16b] A. Bouakaz, P. Fradet, and A. Girault. "Symbolic Buffer Sizing for Throughput-Optimal Scheduling of Dataflow Graphs". In: *RTAS 2016 - 22nd IEEE Real-Time Embedded Technology & Applications Symposium*. Vienne, Austria, Apr. 2016. URL: https://hal.inria.fr/hal-01253168 (cit. on p. 30).

[BFG17] A. Bouakaz, P. Fradet, and A. Girault. "A Survey of Parametric Dataflow Models of Computation". In: *ACM Trans. Des. Autom. Electron. Syst.* 22.2 (Jan. 2017), 38:1–38:25. URL: http://doi.acm.org/10.1145/2999539 (cit. on pp. 21, 145).

[BG92] G. Berry and G. Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. URL: http://www.sciencedirect.com/science/article/pii/0167642392900005V (cit. on pp. 8, 62).

[Bha+08] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet. "OpenDF - A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems". In: *Multi-Core Computing. MCC 2008. First Swedish Workshop on*. Ronneby, Sweden, Nov. 2008, p. CD. URL: https://hal.archives-ouvertes.fr/hal-00340437 (cit. on p. 28).

[Bha+95] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee. "Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms". In: *IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications* 42.3 (Mar. 1995), pp. 138–150 (cit. on p. 26).

[Bil+96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. "Cycle-static Dataflow". In: *Trans. Sig. Proc.* 44.2 (Feb. 1996), pp. 397–408. URL: http://dx.doi.org/10.1109/78.485935 (cit. on pp. 19, 44, 85).

[BJS16] A. Bhagyanath, T. Jain, and K. Schneider. "Towards Code Generation for the Synchronous Control Asynchronous Dataflow (SCAD) Architectures". In: *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016.* 2016, pp. 77–88. URL: https://doi.org/10.6094/UNIFR/10641 (cit. on p. 12).

[BL06] S. S. Bhattacharyya and W. S. Levine. "Optimization of signal processing software for control system implementation". In: *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*. Oct. 2006, pp. 1562–1567 (cit. on p. 93).

[BMd12] B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. "K-Periodic schedules for evaluating the maximum throughput of a Synchronous Dataflow graph". In: *2012 International Conference on Embedded Computer Systems (SAMOS)*. July 2012, pp. 152–159 (cit. on p. 69).

[BMD13] B. Bodin, A. Munier-Kordon, and B. D. de Dinechin. "Periodic schedules for Cyclo-Static Dataflow". In: *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*. Oct. 2013, pp. 105–114 (cit. on p. 30).

[BML12] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software synthesis from dataflow graphs*. Vol. 360. Springer Science & Business Media, 2012 (cit. on pp. 15, 36).

[BML97] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. "APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations". In: *Design Automation for Embedded Systems* 2.1 (Jan. 1997), pp. 33–60. URL: https://doi.org/10.1023/A:1008806425898 (cit. on pp. 26, 58).

[Bod+14] B. Bodin, Y. Lesparre, J.-M. Delosme, and A. Munier-Kordon. "Fast and Efficient Dataflow Graph Generation". In: *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '14. Sankt Goar, Germany: ACM, 2014, pp. 40–49. URL: http://doi.acm.org.insis.bib.cnrs.fr/10.1145/2609248.2609258 (cit. on pp. 29, 85, 86).

[Bod+16] B. Bodin, L. Nardi, P. H. J. Kelly, and M. F. P. O'Boyle. "Diplomat: Mapping of Multi-kernel Applications Using a Static Dataflow Abstraction". In: *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Sept. 2016, pp. 241–250 (cit. on p. 26).

[Bon+08] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 101–113. URL: http://doi.acm.org/10.1145/1375581.1375595 (cit. on p. 5).

[Bou07] P. Boulet. *Array-OL Revisited, Multidimensional Intensive Signal Processing Specification*. Research Report RR-6113. INRIA, 2007, p. 24. URL: https://hal.inria.fr/inria-00128840 (cit. on pp. 9, 58).

[Bou13] A. Bouakaz. "Real-time scheduling of dataflow graphs". Theses. Université Rennes 1, Nov. 2013. URL: https://tel.archives-ouvertes.fr/tel-00945453 (cit. on p. 28).

[Boy+18] M. Boyer, B. Dupont De Dinechin, A. Graillat, and L. Havet. "Computing Routes and Delay Bounds for the Network-on-Chip of the Kalray MPPA2 Processor". In: *ERTS 2018 - 9th European Congress on Embedded Real Time Software and Systems*. Toulouse, France, Jan. 2018. URL: https://hal.archives-ouvertes.fr/hal-01707911 (cit. on p. 11).

[BS11] M. Bamakhrama and T. Stefanov. "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications". In: *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*. Oct. 2011, pp. 195–204 (cit. on pp. 28, 92).

[BS17] A. Bhagyanath and K. Schneider. "Exploring the Potential of Instruction-Level Parallelism of Exposed Datapath Architectures with Buffered Processing Units". In: *Application of Concurrency to System Design (ACSD)*. Ed. by A. Legay and K. Schneider. Zaragoza, Spain: IEEE Computer Society, 2017, pp. 106–115 (cit. on p. 12).

[BS93] J. Bowen and V. Stavridou. "Safety-critical systems, formal methods and standards". In: *Software Engineering Journal* 8.4 (July 1993), pp. 189–209 (cit. on p. 23).

[BSZ16] S. Bliudze, A. Simalatsar, and A. Zolotukhina. *Modelling Resource Dependencies*. Tech. rep. EPFL-REPORT-218599. EPFL IC IINFCOM RiSD, June 2016. URL: http://infoscience.epfl.ch/record/218599 (cit. on p. 7).

[Buc94] J. T. Buck. "A Dynamic Dataflow Model Suitable for Efficient Mixed Hardware and Software Implementations of DSP Applications". In: *Proceedings of the 3rd International Workshop on Hardware/Software Co-design*. CODES '94. Grenoble, France: IEEE Computer Society Press, 1994, pp. 165–172. URL: http://dl.acm.org/citation.cfm?id=947185.947212 (cit. on p. 21).

[Bur+04] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. "Scaling to the end of silicon with EDGE architectures". In: *Computer* 37.7 (July 2004), pp. 44–55 (cit. on p. 12).

[Car+12] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov. "LARA: An Aspect-oriented Programming Language for Embedded Systems". In: *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*. AOSD '12. Potsdam, Germany: ACM, 2012, pp. 179–190. URL: http://doi.acm.org/10.1145/2162049.2162071 (cit. on p. 4).

[Car08] I. Caragiannis. "Better Bounds for Online Load Balancing on Unrelated Machines". In: *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '08. San Francisco, California: Society for Industrial and Applied Mathematics, 2008, pp. 972–981. URL: http://dl.acm.org/citation.cfm?id=1347082.1347188 (cit. on p. 24).

[CB13] C. Curtsinger and E. D. Berger. "STABILIZER: Statistically Sound Performance Evaluation". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 219–228. URL: http://doi.acm.org/10.1145/2451116.2451141 (cit. on p. 5).

[CB18] C. Curtsinger and E. D. Berger. "Coz: Finding Code That Counts with Causal Profiling". In: *Commun. ACM* 61.6 (May 2018), pp. 91–99. URL: http://doi.acm.org/10.1145/3205911 (cit. on p. 5).

[CC10] P. Cousot and R. Cousot. "A gentle introduction to formal verification of computer systems by abstract interpretation". In: *Logics and Languages for Reliability and Security*. Ed. by J. Esparza, O. Grumberg, and M. Broy. NATO Science Series III: Computer and Systems Sciences. IOS Press, 2010, pp. 1–29. URL: https://hal.inria.fr/inria-00543886 (cit. on p. 13).

[CDF16] A. Cohen, A. Darte, and P. Feautrier. *Static Analysis of OpenStream Programs*. Research Report RR-8764. CNRS ; Inria ; ENS Lyon, Jan. 2016, p. 26. URL: https://hal.inria.fr/hal-01184408 (cit. on p. 58).

[CDR98] P.-Y. Calland, A. Darte, and Y. Robert. "Circuit retiming applied to decomposed software pipelining". In: *IEEE Trans. Parallel and Distributed Systems* 9.1 (Jan. 1998), pp. 24–35 (cit. on p. 117).

[CDY95] S. Chakrabarti, J. Demmel, and K. Yelick. "Modeling the Benefits of Mixed Data and Task Parallelism". In: *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '95. Santa Barbara, California, USA: Association for Computing Machinery, 1995, pp. 74–83. URL: https://doi-org.insis.bib.cnrs.fr/10.1145/215399.215423 (cit. on p. 8).

[Chi+12] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. "Diderot: A Parallel DSL for Image Analysis and Visualization". In: *SIGPLAN Not.* 47.6 (June 2012), pp. 111–120. URL: http://doi.acm.org/10.1145/2345156.2254079 (cit. on p. 9).

[CLA13] J. Castrillon, R. Leupers, and G. Ascheid. "MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs". In: *IEEE Transactions on Industrial Informatics* 9.1 (Feb. 2013), pp. 527–545 (cit. on pp. 27, 91).

[Coh+06] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. "N-synchronous Kahn Networks: A Relaxed Model of Synchrony for Real-time Systems". In: *SIGPLAN Not.* 41.1 (Jan. 2006), pp. 180–193. URL: http://doi.acm.org/10.1145/1111320.1111054 (cit. on p. 92).

[Cor+10] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. "Random Graph Generation for Scheduling Simulations". In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. SIMUTools '10. Torremolinos, Malaga, Spain: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010, 60:1–60:10. URL: http://dl.acm.org/citation.cfm?id=1808143.1808219 (cit. on p. 29).

[Cor99] H. Corporaal. "TTAs: Missing the ILP complexity wall". In: *Journal of Systems Architecture* 45.12 (1999), pp. 949–973. URL: http://www.sciencedirect.com/science/article/pii/S1383762198000460 (cit. on p. 12).

[ČP93] M. Čubrić and P. Panangaden. "Minimal memory schedules for dataflow networks". In: *International Conference on Concurrency Theory*. Springer. 1993, pp. 368–383 (cit. on p. 30).

[CR93] P. Chrzastowski-Wachtel and M. Raczunas. "Liveness of weighted circuits and the diophantine problem of Frobenius". In: *Fundamentals of Computation Theory*. Ed. by Z. Ésik. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 171–180 (cit. on p. 29).

[Cru91a] R. L. Cruz. "A calculus for network delay. I. Network elements in isolation". In: *IEEE Transactions on Information Theory* 37.1 (Jan. 1991), pp. 114–131 (cit. on p. 11).

[Cru91b] R. L. Cruz. "A calculus for network delay. II. Network analysis". In: *IEEE Transactions on Information Theory* 37.1 (Jan. 1991), pp. 132–141 (cit. on p. 11).

[CS12] L. Cudennec and R. Sirdey. "Parallelism Reduction Based on Pattern Substitution in Dataflow Oriented Programming Languages". In: *International Conference on Computational Science, ICCS 2012*. Ed. by Y. S. Hesham Ali, D. Khazanchi, M. Lees, G. D. van Albada, J. Dongarra, P. M. Sloot, and J. Dongarra. Vol. 9. University of Nebraska, Omaha. Omaha, Nebraska, United States: Elsevier B.V., June 2012, pp. 146–155. URL: https://hal.inria.fr/hal-00706943 (cit. on pp. 58, 146).

[CSB90] H. Chetto, M. Silly, and T. Bouchentouf. "Dynamic Scheduling of Real-time Tasks Under Precedence Constraints". In: *Real-Time Syst.* 2.3 (Sept. 1990), pp. 181–194. URL: http://dx.doi.org/10.1007/BF00365326 (cit. on p. 92).

[DA94] R. David and H. Alla. "Petri nets for modeling of dynamic systems: A survey". In: *Automatica* 30.2 (1994), pp. 175–202. URL: http://www.sciencedirect.com/science/article/pii/0005109894900248 (cit. on p. 7).

[Dah+09] A. Dahlin, J. Ersfolk, G. Yang, H. Habli, and J. Lilius. "The Canals Language and Its Compiler". In: *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '09. Nice, France: ACM, 2009, pp. 43–52. URL: http://dl.acm.org/citation.cfm?id=1543820.1543829 (cit. on p. 9).

[Dam+12] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. “Modeling static-order schedules in synchronous dataflow graphs”. In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2012, pp. 775–780 (cit. on p. 30).

[Dar+16] M. Dardaillon, K. Marquet, T. Risset, J. Martin, and H.-P. Charles. “A New Compilation Flow for Software-Defined Radio Applications on Heterogeneous MPSoCs”. In: *ACM Transactions on Architecture and Code Optimization* 13 (2016). URL: https://hal.inria.fr/hal-01396143 (cit. on p. 28).

[Dau+19] B. Dauphin, R. Pacalet, A. Enrici, and L. Apvrille. “Odyn: Deadlock Prevention and Hybrid Scheduling Algorithm for Real-Time Dataflow Applications”. In: *22nd Euromicro Conference on Digital System Design, DSD 2019, Kallithea, Greece, August 28-30, 2019*. IEEE, 2019, pp. 88–95. URL: https://doi.org/10.1109/DSD.2019.00023 (cit. on p. 21).

[Dav+10] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. “Timed I/O Automata: A Complete Specification Theory for Real-time Systems”. In: *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*. HSCC ’10. Stockholm, Sweden: ACM, 2010, pp. 91–100. URL: http://doi.acm.org/10.1145/1755952.1755967 (cit. on p. 7).

[Dav+18] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor. “The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips”. In: *IEEE Micro* 38.2 (2018), pp. 30–41 (cit. on p. 11).

[Dav13] R. I. Davis. “Burns Standard Notation for real time scheduling”. English. In: *Real-Time Systems*. Ed. by N. Audsley and S. Baruah. CreateSpace Independent Publishing Platform, Mar. 2013, pp. 38–41 (cit. on p. 41).

[DB11] R. I. Davis and A. Burns. “A Survey of Hard Real-time Scheduling for Multiprocessor Systems”. In: *ACM Comput. Surv.* 43.4 (Oct. 2011), 35:1–35:44. URL: http://doi.acm.org/10.1145/1978802.1978814 (cit. on p. 23).

[DDP18] R. De Landtsheer, J.-C. Deprez, and C. Ponsard. “Optimal Mapping of Task-Based Computation Models over Heterogeneous Hardware Using Placer”. In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS ’18. Copenhagen, Denmark: Association for Computing Machinery, 2018, pp. 17–21. URL: https://doi-org.insis.bib.cnrs.fr/10.1145/3270112.3270136 (cit. on p. 25).

[Dem+95] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J.-C. Dufourd, and J.-L. Marro. “Array-OL : proposition d’un formalisme tableau pour le traitement de signal multidimensionnel”. In: 1995 (cit. on pp. 9, 58).

[Dem+99] I. Demeure, L. Leboucher, N. Rivierre, and F. Singhoff. *Modélisation et support d’applications multimédias réparties*. Tech. rep. ENSTA, 1999 (cit. on p. 9).

[Den+07] K. Denolf, M. Bekooij, J. Cockx, D. Verkest, and H. Corporaal. “Exploiting the Expressiveness of Cyclo-Static Dataflow to Model Multimedia Implementations”. In: *EURASIP Journal on Advances in Signal Processing* 2007.1 (June 2007), p. 084078. URL: https://doi.org/10.1155/2007/84078 (cit. on p. 30).

[Den74] J. B. Dennis. "First version of a data flow procedure language". In: *Programming Symposium*. Ed. by B. Robinet. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 362–376 (cit. on p. 7).

[Der+17a] H. Deroui, K. Desnos, J.-F. Nezan, and A. Munier-Kordon. "Relaxed Subgraph Execution Model for the Throughput Evaluation of IBSDF Graphs". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. SAMOS, Greece, July 2017. URL: https://hal.archives-ouvertes.fr/hal-01569593 (cit. on p. 29).

[Der+17b] H. Deroui, K. Desnos, J.-F. Nezan, and A. Munier-Kordon. "Throughput Evaluation of DSP Applications based on Hierarchical Dataflow Models". In: *International Symposium on Circuits and Systems (ISCAS)*. Baltimore, United States, May 2017. URL: https://hal.archives-ouvertes.fr/hal-01514641 (cit. on p. 29).

[Des+13] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi. "PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration". In: *13th International Conference on Embedded Computer Systems: Architecture, Modeling and Simulation (SAMOS XIII)*. Samos, Greece, July 2013, pp. 41–48. URL: https://hal-ensta-bretagne.archives-ouvertes.fr/hal-00877492 (cit. on pp. xviii, xxviii, 21, 31, 56, 123).

[Des+16a] K. Desnos, M. Pelcat, J. F. Nezan, and S. Aridhi. "Distributed Memory Allocation Technique for Synchronous Dataflow Graphs". In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. Oct. 2016, pp. 45–50 (cit. on p. 30).

[Des+16b] K. Desnos, M. Pelcat, J. F. Nezan, and S. Aridhi. "On Memory Reuse Between Inputs and Outputs of Dataflow Actors". In: *ACM Transactions on Embedded Computing Systems (TECS)* 15.2 (Feb. 2016), p. 30. URL: https://hal.archives-ouvertes.fr/hal-01284333 (cit. on pp. 30, 39, 55, 116, 131).

[DFR11] C. Demetrescu, I. Finocchi, and A. Ribichini. "Reactive Imperative Programming with Dataflow Constraints". In: *CoRR* abs/1104.2293 (2011). arXiv: 1104.2293. URL: http://arxiv.org/abs/1104.2293 (cit. on p. 9).

[Did+19] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton, and A. Graillat. "Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware". In: *ACM Transactions on Architecture and Code Optimization* 16.3 (Aug. 2019), pp. 1–27. URL: https://hal.inria.fr/hal-02422789 (cit. on p. 130).

[Dis+10] P. Disseaux, A. Plantec, M. Kerboeuf, and F. Singhoff. "AADL design patterns and tools for modelling and performance analysis of real-time systems." In: *5th european congress ERTSS Embedded Real-Time Software and System*. France, May 2010. URL: http://hal.univ-brest.fr/hal-00661001 (cit. on p. 9).

[DK82] A. L. Davis and R. M. Keller. "Data Flow Program Graphs". In: *Computer* 15.2 (Feb. 1982), pp. 26–41. URL: http://dx.doi.org/10.1109/MC.1982.1653939 (cit. on p. 7).

[DL78] S. K. Dhall and C. L. Liu. "On a Real-Time Scheduling Problem". In: *Operations Research* 26.1 (1978), pp. 127–140. URL: http://www.jstor.org/stable/169896 (cit. on p. 79).

[DLC14] X. K. Do, S. Louise, and A. Cohen. “Comparing the StreamIt and ΣC Languages for Manycore Processors”. In: *Fourth International workshop on Data-Flow Models for extreme scale computing (DFM 2014, associated with PACT)*. Edmonton, Canada, 2014. URL: https://hal.archives-ouvertes.fr/hal-01257246 (cit. on p. 20).

[DRL16] P. Derler, K. Ravindran, and R. Limaye. “Specification of precise timing in synchronous dataflow models”. In: *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. Nov. 2016, pp. 85–94 (cit. on p. 92).

[DRW98] R. P. Dick, D. L. Rhodes, and W. Wolf. “TGFF: task graphs for free”. In: *Hardware/Software Codesign, 1998. (CODES/CASHE ’98) Proceedings of the Sixth International Workshop on*. Mar. 1998, pp. 97–101 (cit. on p. 29).

[DS10] F. Desprez and F. Suter. “A Bi-Criteria Algorithm for Scheduling Parallel Task Graphs on Clusters”. In: *10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Melbourne, Australia, May 2010, pp. 243–252. URL: https://hal.archives-ouvertes.fr/hal-00533904 (cit. on p. 146).

[Dub+19] P. Dubrulle, C. Gaston, N. Kosmatov, A. Lapitre, and S. Louise. “A Data Flow Model with Frequency Arithmetic”. In: *Fundamental Approaches to Software Engineering*. Ed. by R. Hähnle and W. van der Aalst. Cham: Springer International Publishing, 2019, pp. 369–385 (cit. on pp. 24, 91, 92).

[Duh+15] F. Duhem, F. Muller, R. Bonamy, and S. Bilavarn. “FoRTReSS: a flow for design space exploration of partially reconfigurable systems”. In: *Design Automation for Embedded Systems* 19.3 (Sept. 2015), pp. 301–326. URL: https://hal.archives-ouvertes.fr/hal-01134008 (cit. on p. 145).

[ECP06] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel. “Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design”. In: *IEEE Transactions on Evolutionary Computation* 10.3 (June 2006), pp. 358–374 (cit. on p. 25).

[Eke+03] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. “Taming heterogeneity - the Ptolemy approach”. In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144 (cit. on pp. 27, 93).

[ETS14] H. C. Edwards, C. R. Trott, and D. Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3202–3216. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257 (cit. on p. 5).

[Far+11] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. “Orchestration by Approximation: Mapping Stream Programs Onto Multicore Architectures”. In: *SIGPLAN Not.* 47.4 (Mar. 2011), pp. 357–368. URL: http://doi.acm.org/10.1145/2248487.1950406 (cit. on p. 146).

[Fea92] P. Feautrier. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”. In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pp. 389–420. URL: https://doi.org/10.1007/BF01379404 (cit. on pp. 20, 58).

[FGP12] P. Fradet, A. Girault, and P. Poplavko. “SPDF: A Schedulable Parametric Data-Flow MoC”. In: *Design Automation and Test in Europe, DATE’12*. Dresden, Germany, Mar. 2012. URL: https://hal.inria.fr/hal-00744376 (cit. on p. 21).

[FL11] P. Feautrier and C. Lengauer. "Polyhedron Model". In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 1581–1592. URL: https://doi.org/10.1007/978-0-387-09766-4_502 (cit. on p. 5).

[Fly72] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960 (cit. on p. 10).

[FMG17] R. Fontaine, L. Morel, and L. Gonnord. *Combining dataflow programming and polyhedral optimization, a case study*. Technical Report RT-0490. Inria Rhône-Alpes ; CITI - CITI Centre of Innovation in Telecommunications and Integration of services ; LIP - ENS Lyon, July 2017, p. 40. URL: https://hal.archives-ouvertes.fr/hal-01572439 (cit. on p. 20).

[Foh95] G. Fohler. "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems". In: *Proceedings 16th IEEE Real-Time Systems Symposium*. Dec. 1995, pp. 152–161 (cit. on pp. 62, 92).

[Fra+19] P. Fradet, A. Girault, R. Krishnaswamy, X. Nicollin, and A. Shafiei. "RDF: Reconfigurable Dataflow". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2019, pp. 1709–1714 (cit. on pp. 21, 147).

[Gam+11] A. Gamatié, S. Le Beux, É. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser. "A Model-Driven Design Framework for Massively Parallel Embedded Systems". In: *ACM Trans. Embed. Comput. Syst.* 10.4 (Nov. 2011), 39:1–39:36. URL: http://doi.acm.org/10.1145/2043662.2043663 (cit. on p. 27).

[GAM19a] P. Glanon, S. Azaiez, and C. Mraidha. "Analyzing Throughput for Cyber-Physical Systems modeled with Synchronous Dataflow". In: *Proceedings of the Cyber-Physical Systems PhD Workshop 2019, an event held within the CPS Summer School "Designing Cyber-Physical Systems - From concepts to implementation", Alghero, Italy, September 23, 2019*. Ed. by L. Pulina. Vol. 2457. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 1–9. URL: http://ceur-ws.org/Vol-2457/1.pdf (cit. on p. 29).

[GAM19b] P. Glanon, S. Azaiez, and C. Mraidha. "Estimating Latency for Synchronous Dataflow Graphs Using Periodic Schedules". In: *Verification and Evaluation of Computer and Communication Systems - 13th International Conference, VECoS 2019, Porto, Portugal, October 9, 2019, Proceedings*. Ed. by P. Ganty and M. Kaâniche. Vol. 11847. Lecture Notes in Computer Science. Springer, 2019, pp. 79–94. URL: https://doi.org/10.1007/978-3-030-35092-5%5C_6 (cit. on p. 30).

[Gau+13] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin. "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures". In: *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Boston, Massachusetts, United States, May 2013. URL: https://hal.inria.fr/hal-00799904 (cit. on p. 5).

[Gau+19] T. Gautier, C. Guy, A. Honorat, P. Le Guernic, J.-P. Talpin, and L. Besnard. "Polychronous automata and their use for formal validation of AADL models". In: *Frontiers of Computer Science* 13.4 (Aug. 2019), pp. 677–697. URL: https://hal.inria.fr/hal-01411257 (cit. on p. 28).

[GBL99] A. Girault, Bilung Lee, and E. A. Lee. "Hierarchical finite state machines with multiple concurrency models". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.6 (1999), pp. 742–760 (cit. on p. 7).

[Gee+16] T. Gee, J. James, W. V. D. Mark, P. Delmas, and G. Gimel'farb. "Lidar guided stereo simultaneous localization and mapping (SLAM) for UAV outdoor 3-D scene reconstruction". In: *2016 International Conference on Image and Vision Computing New Zealand (IVCNZ)*. Nov. 2016, pp. 1–6 (cit. on p. 62).

[GGP17] B. Gaujal, A. Girault, and S. Plassart. *Dynamic Speed Scaling Minimizing Expected Energy Consumption for Real-Time Tasks*. Research Report RR-9101. UGA - Université Grenoble Alpes ; Inria Grenoble Rhône-Alpes ; Université de Grenoble, Oct. 2017, pp. 1–35. URL: https://hal.inria.fr/hal-01615835 (cit. on p. 144).

[GH10] K. Goossens and A. Hansson. "The aethereal network on chip after ten years: Goals, evolution, lessons, and future". In: *Design Automation Conference*. June 2010, pp. 306–311 (cit. on p. 11).

[Gha+06a] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk. "Liveness and Boundedness of Synchronous Data Flow Graphs". In: *2006 Formal Methods in Computer Aided Design*. Nov. 2006, pp. 68–75 (cit. on p. 29).

[Gha+06b] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. "Throughput Analysis of Synchronous Data Flow Graphs". In: *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*. ACSD '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 25–36. URL: http://dx.doi.org/10.1109/ACSD.2006.33 (cit. on pp. 29, 73).

[Gha+07] A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. "Latency Minimization for Synchronous Data Flow Graphs". In: *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. Aug. 2007, pp. 189–196 (cit. on pp. 30, 93).

[Gho+12] A. Ghosal, R. Limaye, K. Ravindran, S. Tripakis, A. Prasad, G. Wang, T. N. Tran, and H. Andrade. "Static dataflow with access patterns: Semantics and analysis". In: *DAC Design Automation Conference 2012*. June 2012, pp. 656–663 (cit. on p. 130).

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors". In: *Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on*. Mar. 1999, pp. 74–78 (cit. on pp. xviii, xxviii, 8, 27, 62).

[God97] S. Goddard. "Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application". In: *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. June 1997, pp. 60–71 (cit. on pp. 8, 28).

[Gor+02] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. "A Stream Compiler for Communication-exposed Architectures". In: *SIGARCH Comput. Archit. News* 30.5 (Oct. 2002), pp. 291–303. URL: http://doi.acm.org.insis.bib.cnrs.fr/10.1145/635506.605428 (cit. on p. 12).

[Gou+11] T. Goubier, R. Sirdey, S. Louise, and V. David. "ΣC: A Programming Model and Language for Embedded Manycores". In: *Algorithms and Architectures for Parallel Processing: 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I*. Ed. by Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 385–394. URL: http://dx.doi.org/10.1007/978-3-642-24650-0_33 (cit. on p. 20).

[GR05] J. Gummaraju and M. Rosenblum. “Stream programming on general-purpose processors”. In: *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’05)*. Nov. 2005, 12 pp.–354 (cit. on p. 26).

[Gra69] R. L. Graham. “Bounds on Multiprocessing Timing Anomalies”. In: *SIAM Journal of Applied Mathematics* 17 (1969), pp. 416–429 (cit. on p. 25).

[Gro+12] R. de Groote, J. Kuper, H. Broersma, and G. J. M. Smit. “Max-Plus Algebraic Throughput Analysis of Synchronous Dataflow Graphs”. In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. Sept. 2012, pp. 29–38 (cit. on p. 29).

[GS03] T. Grandpierre and Y. Sorel. “From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations”. In: *Formal Methods and Models for Co-Design, 2003. MEMOCODE ’03. Proceedings. First ACM and IEEE International Conference on*. June 2003, pp. 123–132 (cit. on p. 27).

[GTA06] M. I. Gordon, W. Thies, and S. Amarasinghe. “Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs”. In: *SIGPLAN Not.* 41.11 (Oct. 2006), pp. 151–162. URL: http://doi.acm.org.insis.bib.cnrs.fr/10.1145/1168918.1168877 (cit. on pp. 58, 117).

[Guo+14] L. Guo, Q. Zhu, P. Nuzzo, R. Passerone, A. Sangiovanni-Vincentelli, and E. A. Lee. “Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems”. In: *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Oct. 2014, pp. 1–10 (cit. on pp. xix, xxviii, 93).

[Ha+08] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo. “PeaCE: A Hardware-software Codesign Environment for Multimedia Embedded Systems”. In: *ACM Trans. Des. Autom. Electron. Syst.* 12.3 (May 2008), 24:1–24:25. URL: http://doi.acm.org/10.1145/1255456.1255461 (cit. on p. 27).

[Has+17] J. Hascoët, K. Desnos, J.-F. Nezan, and B. Dupont De Dinechin. “Hierarchical Dataflow Model for Efficient Programming of Clustered Manycore Processors”. In: *28th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2017)*. Seattle, WA, United States, July 2017. URL: https://hal.archives-ouvertes.fr/hal-01564019 (cit. on p. 26).

[Heu+14] J. Heulot, M. Pelcat, K. Desnos, J. Nezan, and S. Aridhi. “Spider: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for multicore DSPS”. In: *2014 6th European Embedded Design in Education and Research Conference (EDERC)*. 2014, pp. 167–171 (cit. on p. 28).

[HHK03] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. “Giotto: a time-triggered language for embedded programming”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 84–99 (cit. on p. 8).

[Hid+08] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. “DFL: A dataflow language based on Petri nets and nested relational calculus”. In: *Information Systems* 33.3 (2008), pp. 261–284. URL: http://www.sciencedirect.com/science/article/pii/S0306437907000634 (cit. on p. 7).

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. “Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE”. In: *IEEE Transactions on Software Engineering* 18.9 (Sept. 1992), pp. 785–793 (cit. on p. 8).

[Hol+14] S. Holmbacka, E. Nogues, M. Pelcat, S. Lafond, and J. Lilius. "Energy Efficiency and Performance Management of Parallel Dataflow Applications". In: *The 2014 Conference on Design & Architectures for Signal & Image Processing*. Madrid, Spain, Oct. 2014. URL: https://hal.archives-ouvertes.fr/hal-01078573 (cit. on p. 141).

[Hor74] W. A. Horn. "Some simple scheduling algorithms". In: *Naval Research Logistics Quarterly* 21.1 (1974), pp. 177–185. URL: http://dx.doi.org/10.1002/nav.3800210113 (cit. on pp. 24, 67).

[HRP17] D. Hardy, B. Rouxel, and I. Puaut. "The Heptane Static Worst-Case Execution Time Estimation Tool". In: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Vol. 8. International Workshop on Worst-Case Execution Time Analysis. Dubrovnik, Croatia, June 2017, p. 12. URL: https://hal.sorbonne-universite.fr/hal-01590444 (cit. on p. 11).

[HS06] T. A. Henzinger and J. Sifakis. "The Embedded Systems Design Challenge". In: *Proceedings of the 14th International Conference on Formal Methods*. FM'06. Hamilton, Canada: Springer-Verlag, 2006, pp. 1–15. URL: http://dx.doi.org/10.1007/11813040_1 (cit. on p. 13).

[Hsu+04] C.-J. Hsu, F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya. "DIF: An Interchange Format for Dataflow-Based Design Tools". In: *Computer Systems: Architectures, Modeling, and Simulation*. Ed. by A. D. Pimentel and S. Vassiliadis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 423–432 (cit. on p. 13).

[Hug+08] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. "From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite". In: *ACM Trans. Embed. Comput. Syst.* 7.4 (Aug. 2008). URL: https://doi.org/10.1145/1376804.1376810 (cit. on pp. xix, xxviii).

[HW07] H. van Hasselt and M. A. Wiering. "Reinforcement Learning in Continuous Action Spaces". In: *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. Apr. 2007, pp. 272–279 (cit. on p. 28).

[IF00] D. Isovic and G. Fohler. "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints". In: *Proceedings 21st IEEE Real-Time Systems Symposium*. Nov. 2000, pp. 207–216 (cit. on p. 92).

[IJT91] F. Irigoin, P. Jouvelot, and R. Triolet. "Semantical Interprocedural Parallelization: An Overview of the PIPS Project". In: *Proceedings of the 5th International Conference on Supercomputing*. ICS '91. Cologne, West Germany: ACM, 1991, pp. 244–251. URL: http://doi.acm.org/10.1145/109025.109086 (cit. on p. 5).

[IY98] T. Ishihara and H. Yasuura. "Voltage scheduling problem for dynamically variable voltage processors". In: *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*. Aug. 1998, pp. 197–202 (cit. on p. 141).

[Jen81] S. F. Jennings. "Petri net models of program execution in data flow environments". PhD thesis. Iowa State University, 1981 (cit. on p. 18).

[Jia+13] Y. Jiang, Z. Li, H. Zhang, Y. Deng, X. Song, M. Gu, and J. Sun. "Design and Optimization of Multi-clocked Embedded Systems Using Formal Technique". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 703–706. URL: http://doi.acm.org/10.1145/2491411.2494575 (cit. on p. 7).

[Joh+12] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. "ABS: A Core Language for Abstract Behavioral Specification". In: *Formal Methods for Components and Objects*. Ed. by B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 142–164 (cit. on p. 4).

[Joh75] D. B. Johnson. "Finding all the elementary circuits of a directed graph". In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84 (cit. on p. 104).

[JY17] J. Jang and H. Yang. "Executable Dataflow Benchmark Generation Technique for Multi-core Embedded Systems". In: *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*. RSP '17. Seoul, South Korea: ACM, 2017, pp. 50–56. URL: http://doi.acm.org/10.1145/3130265.3130323 (cit. on p. 28).

[KA99] Y.-K. Kwok and I. Ahmad. "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors". In: *ACM Comput. Surv.* 31.4 (Dec. 1999), pp. 406–471. URL: http://doi.acm.org/10.1145/344588.344618 (cit. on pp. 25, 63, 77, 118).

[KAG96] Y.-K. Kwok, I. Ahmad, and J. Gu. "FAST: a low-complexity algorithm for efficient scheduling of DAGs on parallel processors". In: *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*. Vol. 2. Aug. 1996, 150–157 vol.2 (cit. on pp. 25, 77, 116, 142).

[Kah74] G. Kahn. "The Semantics of Simple Language for Parallel Programming". In: *IFIP Congress*. 1974, pp. 471–475 (cit. on pp. 7, 44).

[Kan+12] S. Kang, H. Yang, L. Schor, I. Bacivarov, S. Ha, and L. Thiele. "Multi-objective mapping optimization via problem decomposition for many-core systems". In: *2012 IEEE 10th Symposium on Embedded Systems for Real-time Multimedia*. Oct. 2012, pp. 28–37 (cit. on p. 145).

[Kap97] D. J. Kaplan. "An introduction to the processing graph method". In: *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*. Mar. 1997, pp. 46–52 (cit. on p. 8).

[KBB06] M. Khandelia, N. K. Bambha, and S. S. Bhattacharyya. "Contention-conscious transaction ordering in multiprocessor DSP systems". In: *IEEE Transactions on Signal Processing* 54.2 (Feb. 2006), pp. 556–569 (cit. on p. 145).

[KC02] J. Knowles and D. Corne. "On metrics for comparing nondominated sets". In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*. Vol. 1. May 2002, 711–716 vol.1 (cit. on p. 25).

[KD13] J. Keinert and E. F. Deprettere. "Multidimensional Dataflow Graphs". In: *Handbook of Signal Processing Systems*. Ed. by S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala. New York, NY: Springer New York, 2013, pp. 1145–1175. URL: https://doi.org/10.1007/978-1-4614-6859-2_35 (cit. on pp. 20, 58).

[Kha+16] J. Khatib, A. Munier-Kordon, E. C. Klikpo, and T.-C. Kods. "Computing latency of a real-time system modeled by Synchronous Dataflow Graph". In: *Real-Time Networks and Systems RTNS*. Brest, France, Oct. 2016, pp. 87–96. URL: https://hal.archives-ouvertes.fr/hal-01449892 (cit. on pp. 30, 93).

[KHT06] J. Keinert, C. Haubelt, and J. Teich. "Modeling and Analysis of Windowed Synchronous Algorithms". In: *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*. Vol. 3. May 2006, pp. III–III (cit. on pp. 20, 58).

[KL12] C. Kessler and W. Löwe. "Optimized Composition of Performance-aware Parallel Components". In: *Concurr. Comput. : Pract. Exper.* 24.5 (Apr. 2012), pp. 481–498. URL: http://dx.doi.org/10.1002/cpe.1844 (cit. on p. 5).

[KLE17] S. Kanur, J. Lilius, and J. Ersfolk. "Detecting data-parallel synchronous dataflow graphs". In: *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. Sept. 2017, pp. 1–6 (cit. on p. 119).

[KLH07] S. Kwon, C. Lee, and S. Ha. "Data-Parallel Code Generation from Synchronous Dataflow Specification of Multimedia Applications". In: *2007 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*. Oct. 2007, pp. 91–96 (cit. on p. 26).

[KLV61] J. L. Kelly, C. Lochbaum, and V. A. Vyssotsky. "A block diagram compiler". In: *The Bell System Technical Journal* 40.3 (1961), pp. 669–678 (cit. on p. 13).

[KM08] M. Kudlur and S. Mahlke. "Orchestrating the Execution of Stream Programs on Multicore Platforms". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 114–124. URL: http://doi.acm.org/10.1145/1379022.1375596 (cit. on pp. 96, 117, 130, 146).

[KM16] E. C. Klikpo and A. Munier-Kordon. "Preemptive scheduling of dependent periodic tasks modeled by synchronous dataflow graphs". In: *Real-Time Networks and Systems RTNS*. Brest, France, Oct. 2016, pp. 77–86. URL: https://hal.archives-ouvertes.fr/hal-01449876 (cit. on p. 28).

[Kom+18] J. Komenda, S. Lahaye, J.-L. Boimond, and T. van den Boom. "Max-plus algebra in the history of discrete event systems". In: *Annual Reviews in Control* 45 (2018), pp. 240–249. URL: http://www.sciencedirect.com/science/article/pii/S1367578818300129 (cit. on p. 29).

[KS92] H. Kautz and B. Selman. "Planning As Satisfiability". In: *Proceedings of the 10th European Conference on Artificial Intelligence*. ECAI '92. Vienna, Austria: John Wiley & Sons, Inc., 1992, pp. 359–363. URL: http://dl.acm.org/citation.cfm?id=145448.146725 (cit. on p. 25).

[KSG15] R. v. Kampenhout, S. Stuijk, and K. Goossens. "A Scenario-Aware Dataflow Programming Model". In: *2015 Euromicro Conference on Digital System Design*. Aug. 2015, pp. 25–32 (cit. on p. 21).

[Kun+82] S.-Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao. "Wavefront Array Processor: Language, Architecture, and Applications". In: *IEEE Trans. Comput.* 31.11 (Nov. 1982), pp. 1054–1066. URL: https://doi-org.insis.bib.cnrs.fr/10.1109/TC.1982.1675922 (cit. on p. 12).

[Lam04] M. S. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *SIGPLAN Not.* 39.4 (Apr. 2004), pp. 244–256. URL: https://doi.org/10.1145/989393.989420 (cit. on pp. 26, 117).

[Lar+12] P. Larsen, R. Ladelsky, J. Lidman, S. McKee, S. Karlsson, and A. Zaks. "Parallelizing more Loops with Compiler Guided Refactoring". In: *Parallel Processing (ICPP), 2012 41st International Conference on*. Sept. 2012, pp. 410–419. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6337602 (cit. on p. 5).

[LB00] G. Lipari and G. Buttazzo. "Schedulability analysis of periodic and aperiodic tasks with resource constraints". In: *Journal of Systems Architecture* 46.4 (2000), pp. 327–338. URL: http://www.sciencedirect.com/science/article/pii/S1383762199000090 (cit. on pp. 62, 92).

[LDS07] C. Li, C. Ding, and K. Shen. "Quantifying the cost of context switch". In: *Proceedings of the 2007 workshop on Experimental computer science*. ACM. 2007 (cit. on p. 63).

[Lee+98] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. "Space-time Scheduling of Instruction-level Parallelism on a Raw Machine". In: *SIGOPS Oper. Syst. Rev.* 32.5 (Oct. 1998), pp. 46–57. URL: http://doi.acm.org/10.1145/384265.291018 (cit. on p. 12).

[Les17] Y. Lesparre. "Efficient evaluation of mappings of dataflow applications onto distributed memory architectures". Theses. Université Pierre et Marie Curie - Paris VI, Mar. 2017. URL: https://tel.archives-ouvertes.fr/tel-01624553 (cit. on p. 30).

[Leu+19] R. Leupers, M. A. Aguilar, J. Castrillon, and W. Sheng. "Software Compilation Techniques for Heterogeneous Embedded Multi-Core Systems". In: *Handbook of Signal Processing Systems*. Ed. by S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala. Cham: Springer International Publishing, 2019, pp. 1021–1062. URL: https://doi.org/10.1007/978-3-319-91734-4_28 (cit. on p. 27).

[LGE12] J. Lin, A. Gerstlauer, and B. L. Evans. "Communication-aware Heterogeneous Multiprocessor Mapping for Real-time Streaming Systems". In: *Journal of Signal Processing Systems* 69.3 (Dec. 2012), pp. 279–291. URL: https://doi.org/10.1007/s11265-012-0674-6 (cit. on pp. 26, 93, 145).

[Li+17] Y. Li, Q. Wang, Y. Li, L. Han, Y. Gao, and Q. Mu. "A Cost Model for Heterogeneous Many-Core Processor". In: *Parallel Architecture, Algorithm and Programming*. Ed. by G. Chen, H. Shen, and M. Chen. Singapore: Springer Singapore, 2017, pp. 566–578 (cit. on p. 12).

[Lia+06] S. Liao, Z. Du, G. Wu, and G.-Y. Lueh. "Data and computation transformations for Brook streaming applications on multiprocessors". In: *International Symposium on Code Generation and Optimization (CGO'06)*. Mar. 2006, 12 pp.–207 (cit. on pp. 9, 58).

[Liu10] D. Liu. "Application Specific Instruction Set DSP Processors". In: *Handbook of Signal Processing Systems*. Ed. by S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala. Boston, MA: Springer US, 2010, pp. 415–447. URL: https://doi.org/10.1007/978-1-4419-6345-1_16 (cit. on p. 12).

[Liv+07] N. Liveris, C. Lin, J. Wang, H. Zhou, and P. Banerjee. "Retiming for Synchronous Data Flow Graphs". In: *2007 Asia and South Pacific Design Automation Conference*. Jan. 2007, pp. 480–485 (cit. on p. 118).

[LL73] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. URL: http://doi.acm.org/10.1145/321738.321743 (cit. on pp. 23, 67).

[LM69] E. L. Lawler and J. M. Moore. "A Functional Equation and Its Application to Resource Allocation and Sequencing Problems". In: *Management Science* 16.1 (1969), pp. 77–84. URL: http://www.jstor.org/stable/2628367 (cit. on p. 25).

[LM87a] E. Lee and D. Messerschmitt. “Pipeline interleaved programmable DSP’s: Synchronous data flow programming”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 35.9 (Sept. 1987), pp. 1334–1345 (cit. on pp. 26, 96, 117).

[LM87b] E. A. Lee and D. G. Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (Sept. 1987), pp. 1235–1245 (cit. on pp. xviii, xxi, xxviii, 13, 44, 62).

[LM87c] E. A. Lee and D. G. Messerschmitt. “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing”. In: *IEEE Trans. Comput.* 36.1 (Jan. 1987), pp. 24–35. URL: http://dx.doi.org/10.1109/TC.1987.5009446 (cit. on pp. 15, 39).

[Lou19] S. Louise. “Graph Transformations and Derivation of Scheduling Constraints Applied to the Mapping of Real-Time Distributed Applications”. In: *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2019, pp. 295–303 (cit. on p. 92).

[Low04] D. G. Lowe. “Distinctive Image Features from Scale-Invariant Keypoints”. In: *International Journal of Computer Vision* 60.2 (Nov. 2004), pp. 91–110. URL: https://doi.org/10.1023/B:VISI.0000029664.99615.94 (cit. on pp. xxiii, xxx, 45).

[LP95] E. A. Lee and T. M. Parks. “Dataflow process networks”. In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801 (cit. on p. 7).

[LPC12] F. Li, A. Pop, and A. Cohen. “Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs”. In: *IEEE Micro* 32.4 (2012), pp. 19–31. URL: https://hal.archives-ouvertes.fr/hal-00906099 (cit. on pp. 9, 58).

[LS91] C. E. Leiserson and J. B. Saxe. “Retiming synchronous circuitry”. In: *Algorithmica* 6.1 (June 1991), pp. 5–35. URL: https://doi.org/10.1007/BF01759032 (cit. on pp. 26, 117).

[LT19] J. Leben and G. Tzanetakis. “Polyhedral Compilation for Multi-dimensional Stream Processing”. In: *ACM Trans. Archit. Code Optim.* 16.3 (July 2019), 27:1–27:26. URL: http://doi.acm.org/10.1145/3330999 (cit. on p. 59).

[LTL03] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. *Polychrony for system design*. Research Report RR-4715. INRIA, 2003. URL: https://hal.inria.fr/inria-00071871 (cit. on p. 8).

[Ma+13] Y. Ma, H. Yu, T. Gautier, P. Le Guernic, J.-P. Talpin, L. Besnard, and M. Heitz. “Toward Polychronous Analysis and Validation for Timed Software Architectures in AADL”. In: *The Design, Automation, and Test in Europe (DATE) conference*. Grenoble, France, Mar. 2013, p. 6. URL: https://hal.archives-ouvertes.fr/hal-00763379 (cit. on p. 9).

[Mad+19] D. Madroñal, F. Arrestier, J. Sancho, A. Morvan, R. Lazcano, K. Desnos, R. Salvador, D. Menard, E. Juarez, and C. Sanz. “PAPIFY: Automatic Instrumentation and Monitoring of Dynamic Dataflow Applications Based on PAPI”. In: *IEEE Access* 7 (2019), pp. 111801–111812 (cit. on p. 130).

[Mai+18] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis. *A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems*. Tech. rep. TR-2018-9. Verimag Research Report, 2018 (cit. on pp. 24, 130).

[Mar+18] L. Marchal, B. Simon, O. Sinnen, and F. Vivien. "Malleable task-graph scheduling with a practical speed-up model". In: *IEEE Transactions on Parallel and Distributed Systems* 29.6 (June 2018), pp. 1357–1370. URL: https://hal.inria.fr/hal-01687189 (cit. on p. 146).

[MB07] O. M. Moreira and M. J. G. Bekooij. "Self-Timed Scheduling Analysis for Real-Time Applications". In: *EURASIP Journal on Advances in Signal Processing* 2007.1 (2007), p. 083710. URL: http://dx.doi.org/10.1155/2007/83710 (cit. on pp. 30, 93).

[MB08] M. Masin and Y. Bukchin. "Diversity Maximization Approach for Multiobjective Optimization". In: *Operations Research* 56 (2008), pp. 411–424 (cit. on p. 25).

[MF20] K. Muts and H. Falk. "Multi-Criteria Function Inlining for Hard Real-Time Systems". In: *Proceedings of the 28th International Conference on Real-Time Networks and Systems*. RTNS 2020. Paris, France: Association for Computing Machinery, 2020, pp. 56–66. URL: https://doi.org/10.1145/3394810.3394819 (cit. on p. 144).

[MG13] A. Malik and D. Gregg. "Orchestrating Stream Graphs Using Model Checking". In: *ACM Trans. Archit. Code Optim.* 10.3 (Sept. 2013), 19:1–19:25 (cit. on p. 118).

[ML02] P. K. Murthy and E. A. Lee. "Multidimensional synchronous dataflow". In: *IEEE Transactions on Signal Processing* 50.8 (Aug. 2002), pp. 2064–2079 (cit. on pp. 19, 57).

[MM08] O. Marchetti and A. Munier-Kordon. "Minimizing Place Capacities of Weighted Event Graphs for Enforcing Liveness". In: *Discrete Event Dynamic Systems* 18.1 (Mar. 2008), pp. 91–109. URL: https://doi.org/10.1007/s10626-007-0035-y (cit. on p. 29).

[MM09] O. Marchetti and A. Munier-Kordon. "A sufficient condition for the liveness of weighted event graphs". In: *European Journal of Operational Research* 197.2 (Sept. 2009), pp. 532–540. URL: https://hal.archives-ouvertes.fr/hal-01197183 (cit. on p. 29).

[Moo+08] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. "Cache Aware Mapping of Streaming Applications on a Multiprocessor System-on-Chip". In: *2008 Design, Automation and Test in Europe*. Mar. 2008, pp. 300–305 (cit. on p. 28).

[MR14] M. Masin and T. Raviv. "Linear programming-based algorithms for the minimum makespan high multiplicity jobshop problem". In: *Journal of Scheduling* 17.4 (Aug. 2014), pp. 321–338. URL: https://doi.org/10.1007/s10951-014-0376-y (cit. on p. 25).

[NC12] C. Nugteren and H. Corporaal. "Introducing 'Bones': A Parallelizing Source-to-source Compiler Based on Algorithmic Skeletons". In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. GPGPU-5. London, United Kingdom: ACM, 2012, pp. 1–10. URL: http://doi.acm.org/10.1145/2159430.2159431 (cit. on p. 5).

[NNS13] D. Nadezhkin, H. Nikolov, and T. Stefanov. "Automated Generation of Polyhedral Process Networks from Affine Nested-loop Programs with Dynamic Loop Bounds". In: *ACM Trans. Embed. Comput. Syst.* 13.1s (Dec. 2013), 28:1–28:24. URL: http://doi.acm.org.insis.bib.cnrs.fr/10.1145/2536747.2536750 (cit. on p. 58).

[Nog16] E. Nogues. “Energy optimization of Signal Processing on MPSoCs and its Application to Video Decoding”. Theses. INSA de Rennes, June 2016. URL: https://tel.archives-ouvertes.fr/tel-01359031 (cit. on p. 141).

[NSC07] T. N’Takpé, F. Suter, and H. Casanova. “A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms”. In: *6th International Symposium on Parallel and Distributed Computing - ISPDC 2007*. Hagenberg, Austria, July 2007, pp. 35–42. URL: https://hal.inria.fr/inria-00151812 (cit. on p. 146).

[NYP16] V. Nélis, P. M. Yomsi, and L. M. Pinho. “The Variability of Application Execution Times on a Multi-Core Platform”. In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Ed. by M. Schoeberl. Vol. 55. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 6:1–6:11. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6899 (cit. on p. 130).

[OH02] H. Oh and S. Ha. “Fractional Rate Dataflow Model and Efficient Code Synthesis for Multimedia Applications”. In: *SIGPLAN Not.* 37.7 (June 2002), pp. 12–17. URL: http://doi.acm.org/10.1145/566225.513834 (cit. on p. 28).

[Ost95] J. S. Ostroff. “Abstraction and composition of discrete real-time systems”. In: *Proc. of CASE* 95.370-380 (1995), p. 17 (cit. on p. 7).

[Pag+11] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. “Multi-task implementation of multi-periodic synchronous programs”. In: *Discrete Event Dynamic Systems* 21.3 (2011), pp. 307–338. URL: https://hal.inria.fr/inria-00638936 (cit. on p. 92).

[Pag+14] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. “The ROSACE case study: From Simulink specification to multi/many-core execution”. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2014), pp. 309–318 (cit. on p. 27).

[PAN08] M. Pelcat, S. Aridhi, and J. F. Nezan. “Optimization of automatically generated multi-core code for the LTE RACH-PD algorithm”. In: *DASIP 2008*. Bruxelles, Belgium, Nov. 2008, pp. URL: https://hal.archives-ouvertes.fr/hal-00336477 (cit. on p. 28).

[Par07] K. K. Parhi. *VLSI digital signal processing systems : design and implementation*. John Wiley \& Sons, 2007 (cit. on pp. 97, 117).

[PBL95] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. “A hierarchical multiprocessor scheduling system for DSP applications”. In: *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*. Vol. 1. Oct. 1995, 122–126 vol.1 (cit. on p. 58).

[PBR10] J. Piat, S. S. Bhattacharyya, and M. Raulet. “Loop transformations for interface-based hierarchies IN SDF graphs”. In: *21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*. Rennes, France, July 2010, pp. 341–344. URL: https://hal.archives-ouvertes.fr/hal-00560028 (cit. on p. 59).

[PCH99] C. Park, J. Chung, and S. Ha. “Extended synchronous dataflow for efficient DSP system prototyping”. In: *Proceedings Tenth IEEE International Workshop on Rapid System Prototyping. Shortening the Path from Specification to Prototype (Cat. No.PR00246)*. July 1999, pp. 196–201 (cit. on p. 20).

[Pel+09a] M. Pelcat, J. F. Nezan, J. Piat, J. Croizer, and S. Aridhi. "A System-Level Architecture Model for Rapid Prototyping of Heterogeneous Multicore Embedded Systems". In: *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*. nice, France, Sept. 2009, 8 pages. URL: https://hal.archives-ouvertes.fr/hal-00429397 (cit. on pp. 11, 31).

[Pel+09b] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha. "Handling Mixed-criticality in SoC-based Real-time Embedded Systems". In: *Proceedings of the Seventh ACM International Conference on Embedded Software*. EMSOFT '09. Grenoble, France: ACM, 2009, pp. 235–244. URL: http://doi.acm.org/10.1145/1629335.1629367 (cit. on p. 85).

[Pel+11] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. "A Predictable Execution Model for COTS-Based Embedded Systems". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2011, pp. 269–279 (cit. on p. 130).

[Pel+14] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. F. Nezan, and S. Aridhi. "PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming". In: *6th Embedded Design Education and Research Conference EDERC*. Italy, Sept. 2014, p. 36. URL: https://hal.archives-ouvertes.fr/hal-01059313 (cit. on pp. xxiii, xxx, 28, 30, 56, 62, 69, 111).

[Pel10] M. Pelcat. "Rapid Prototyping and Dataflow-Based Code Generation for the 3GPP LTE eNodeB Physical Layer mapped onto Multi-Core DSPs". Theses. INSA de Rennes, Sept. 2010. URL: https://tel.archives-ouvertes.fr/tel-00578043 (cit. on p. 85).

[Per+16] Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet. "Temporal isolation of hard real-time applications on many-core processors". In: *RTAS : Real-Time Embedded Technology & Applications Symposium*. VIENNE, Austria, Apr. 2016. URL: https://hal.archives-ouvertes.fr/hal-01393382 (cit. on p. 130).

[Pet66] C. A. Petri. "Communication with automata". In: 1966. URL: https://edoc.sub.uni-hamburg.de/informatik/volltexte/2010/155/pdf/diss_petri_engl.pdf (cit. on p. 7).

[PG16] F. Pereira and L. Gomes. "Combining Data-Flows and Petri Nets for Cyber-Physical Systems Specification". In: *Technological Innovation for Cyber-Physical Systems*. Ed. by L. M. Camarinha-Matos, A. J. Falcão, N. Vafaei, and S. Najdi. Cham: Springer International Publishing, 2016, pp. 65–76 (cit. on p. 8).

[PGB19] J. V. Pinxten, M. Geilen, and T. Basten. "Parametric Scheduler Characterization". In: *ACM Trans. Embed. Comput. Syst.* 18.5s (Oct. 2019). URL: https://doi.org/10.1145/3358226 (cit. on p. 146).

[Pia+09] J. Piat, S. S. Bhattacharyya, M. Pelcat, and M. Raulet. "Multi-Core Code Generation From Interface Based Hierarchy". In: *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*. Sophia Antipolis, France, Sept. 2009, online. URL: https://hal.archives-ouvertes.fr/hal-00440479 (cit. on p. 20).

[Pia+18] J. Piat, P. Fillatreau, D. Tortei, F. Brenot, and M. Devy. "HW/SW co-design of a visual SLAM application". In: *Journal of Real-Time Image Processing* (Nov. 2018). URL: https://doi.org/10.1007/s11554-018-0836-2 (cit. on p. 62).

[Pim17] A. D. Pimentel. “Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration”. In: *IEEE Design Test* 34.1 (Feb. 2017), pp. 77–90 (cit. on pp. 25, 144).

[PM91] K. K. Parhi and D. G. Messerschmitt. “Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding”. In: *IEEE Transactions on Computers* 40.2 (Feb. 1991), pp. 178–195 (cit. on p. 26).

[PNP15] W. Puffitsch, E. Noulard, and C. Pagetti. “Off-line mapping of multi-rate dependent task sets to many-core platforms”. In: *Real-Time Systems* 51.5 (Sept. 2015), pp. 526–565. URL: https://doi.org/10.1007/s11241-015-9232-1 (cit. on p. 92).

[PPL94] J. L. Pino, T. M. Parks, and E. A. Lee. “Mapping multiple independent synchronous dataflow graphs onto heterogeneous multiprocessors”. In: *Proceedings of 1994 28th Asilomar Conference on Signals, Systems and Computers*. Vol. 2. Oct. 1994, 1063–1068 vol.2 (cit. on p. 20).

[PPL95] T. M. Parks, J. L. Pino, and E. A. Lee. “A comparison of synchronous and cyclo-static dataflow”. In: *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*. Vol. 1. Oct. 1995, 204–210 vol.1 (cit. on p. 19).

[Pra17] C. Pralet. “An Incomplete Constraint-Based System for Scheduling with Renewable Resources”. In: *Principles and Practice of Constraint Programming*. Ed. by J. C. Beck. Cham: Springer International Publishing, 2017, pp. 243–261 (cit. on p. 25).

[PV81] F. P. Preparata and J. Vuillemin. “The Cube-connected Cycles: A Versatile Network for Parallel Computation”. In: *Commun. ACM* 24.5 (May 1981), pp. 300–309. URL: http://doi.acm.org.insis.bib.cnrs.fr/10.1145/358645.358660 (cit. on p. 12).

[QE12] S. Quinton and R. Ernst. “Generalized Weakly-Hard Constraints”. In: *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Applications and Case Studies - Volume Part II*. ISoLA’12. Heraklion, Crete, Greece: Springer-Verlag, 2012, pp. 96–110. URL: https://doi.org/10.1007/978-3-642-34032-1_13 (cit. on p. 24).

[RAK15] M. Rashid, M. W. Anwar, and A. M. Khan. “Toward the tools selection in model based system engineering for embedded systems—A systematic literature review”. In: *Journal of Systems and Software* 106 (2015), pp. 150–163. URL: http://www.sciencedirect.com/science/article/pii/S016412121500103X (cit. on p. 27).

[Rit+93] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr. “Optimum vectorization of scalable synchronous dataflow graphs”. In: *Proceedings of International Conference on Application Specific Array Processors (ASAP ’93)*. 1993, pp. 285–296 (cit. on pp. 58, 146).

[RP17] B. Rouxel and I. Puaut. “STR2RTS: Refactored StreamIT benchmarks into statically analysable parallel benchmarks for WCET estimation & real-time scheduling”. In: *OASIcs-OpenAccess Series in Informatics*. Vol. 57. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017 (cit. on p. 28).

[RS14] K. Rosvall and I. Sander. “A constraint-based design space exploration framework for real-time applications on MPSoCs”. In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–6 (cit. on p. 130).

[RST92] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. "Code Generation Schema for Modulo Scheduled Loops". In: *Proceedings of the 25th Annual International Symposium on Microarchitecture*. MICRO 25. Portland, Oregon, USA: IEEE Computer Society Press, 1992, pp. 158–169. URL: http://dl.acm.org.insis.bib.cnrs.fr/citation.cfm?id=144953.145795 (cit. on p. 5).

[SB09] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. 2nd. Boca Raton, FL, USA: CRC Press, Inc., 2009 (cit. on p. 29).

[SB14] J. Sérot and F. Berry. "High-Level Dataflow Programming for Reconfigurable Computing". In: *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*. Oct. 2014, pp. 72–77 (cit. on p. 28).

[SB19] A. Stoutchinin and L. Benini. "StreamDrive: a Dynamic Dataflow Framework for Clustered Embedded Architectures". In: *J. Signal Process. Syst.* 91.3-4 (2019), pp. 275–301. URL: https://doi.org/10.1007/s11265-018-1351-1 (cit. on p. 21).

[Sch+15] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. "T-CREST: Time-predictable multi-core architecture for embedded systems". In: *Journal of Systems Architecture* 61.9 (2015), pp. 449–471. URL: http://www.sciencedirect.com/science/article/pii/S1383762115000193 (cit. on p. 13).

[Sch+19] T. Schwarzer, J. Falk, S. Müller, M. Letras, C. Heidorn, S. Wildermann, and J. Teich. "Compilation of Dataflow Applications for Multi-Cores Using Adaptive Multi-Objective Optimization". In: *ACM Trans. Des. Autom. Electron. Syst.* 24.3 (Mar. 2019), 29:1–29:23. URL: http://doi.acm.org/10.1145/3310249 (cit. on pp. 26, 145).

[SEB18] A. Singh, P. Ekberg, and S. Baruah. "Uniprocessor scheduling of real-time synchronous dataflow tasks". In: *Real-Time Systems* (May 2018). URL: https://doi.org/10.1007/s11241-018-9310-2 (cit. on p. 92).

[SGB06a] S. Stuijk, M. Geilen, and T. Basten. "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs". In: *2006 43rd ACM/IEEE Design Automation Conference*. July 2006, pp. 899–904 (cit. on p. 30).

[SGB06b] S. Stuijk, M. Geilen, and T. Basten. "SDF^3: SDF For Free". In: *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. Turku, Finland: IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006, pp. 276–278. URL: http://www.es.ele.tue.nl/sdf3 (cit. on pp. 28, 30, 85, 89, 111).

[SGB08] S. Stuijk, M. Geilen, and T. Basten. "Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs". In: *IEEE Transactions on Computers* 57.10 (Oct. 2008), pp. 1331–1345 (cit. on p. 30).

[SGL99] I. Smarandache, T. Gautier, and P. Le Guernic. "Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints". In: *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*. LNCS vol. 1709. Toulouse, France: Springer, Sept. 1999, pp. 1364–1383. URL: https://hal.archives-ouvertes.fr/hal-00548887 (cit. on p. 8).

[Sin+04] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. “Cheddar: A Flexible Real Time Scheduling Framework”. In: *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-time &Amp; Distributed Systems Using Ada and Related Technologies*. SIGAda ’04. Atlanta, Georgia, USA: ACM, 2004, pp. 1–8. URL: http://doi.acm.org/10.1145/1032297.1032298 (cit. on p. 29).

[Ska+18] S. Skalistis, F. Angiolini, G. D. Micheli, and A. Simalatsar. “Safe and Efficient Deployment of Data-Parallelizable Applications on Many-Core Platforms: Theory and Practice”. In: *IEEE Design Test* 35.4 (Aug. 2018), pp. 7–15 (cit. on p. 26).

[Smi+19] F. Smirnov, B. Pourmohseni, M. Glaß, and J. Teich. “IGOR, Get Me the Optimum! Prioritizing Important Design Decisions During the DSE of Embedded Systems”. In: *ACM Trans. Embed. Comput. Syst.* 18.5s (Oct. 2019). URL: https://doi.org/10.1145/3358204 (cit. on p. 145).

[Smi82] A. J. Smith. “Cache Memories”. In: *ACM Comput. Surv.* 14.3 (Sept. 1982), pp. 473–530. URL: http://doi.acm.org.insis.bib.cnrs.fr/10.1145/356887.356892 (cit. on p. 11).

[Spa17] J. Spasić. “Improved Hard Real-Time Scheduling and Transformations for Embedded Streaming Applications”. PhD thesis. Universiteit Leiden, Nov. 2017. URL: https://openaccess.leidenuniv.nl/handle/1887/59459 (cit. on p. 26).

[Ste97] R. S. Stevens. “The processing graph method tool (PGMT)”. In: *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*. July 1997, pp. 263–271 (cit. on p. 8).

[Stu+11] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. “Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications”. In: *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. July 2011, pp. 404–411 (cit. on p. 21).

[Sur+19] L. Suriano, F. Arrestier, A. Rodríguez, J. Heulot, K. Desnos, M. Pelcat, and E. de la Torre. “DAMHSE: Programming heterogeneous MPSoCs with hardware acceleration using dataflow-based design space exploration and automated rapid prototyping”. In: *Microprocessors and Microsystems* 71 (2019), p. 102882. URL: http://www.sciencedirect.com/science/article/pii/S0141933118303107 (cit. on p. 145).

[Swa+07] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. “The WaveScalar Architecture”. In: *ACM Trans. Comput. Syst.* 25.2 (May 2007), 4:1–4:54. URL: http://doi.acm.org/10.1145/1233307.1233308 (cit. on p. 12).

[TA10] W. Thies and S. Amarasinghe. “An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design”. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’10. Vienna, Austria: ACM, 2010, pp. 365–376. URL: http://doi.acm.org/10.1145/1854273.1854319 (cit. on pp. 28, 44).

[Tay+02] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. “The Raw microprocessor: a computational fabric for software circuits and general-purpose programs”. In: *IEEE Micro* 22.2 (Mar. 2002), pp. 25–35 (cit. on p. 12).

[TB06] J. Teich and S. S. Bhattacharyya. "Analysis of Dataflow Programs with Interval-limited Data-rates". In: *Journal of VLSI signal processing systems for signal, image and video technology* 43.2 (June 2006), pp. 247–258. URL: https://doi.org/10.1007/s11265-006-7274-2 (cit. on p. 145).

[The+06] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. "A scenario-aware data flow model for combined long-run average and worst-case performance analysis". In: *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.* July 2006, pp. 185–194 (cit. on p. 21).

[THM02] H. Topcuoglu, S. Hariri, and Min-You Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), pp. 260–274 (cit. on p. 93).

[Thu+07] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing". In: *2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation.* July 2007, pp. 18–25 (cit. on p. 12).

[TKA02] W. Thies, M. Karczmarek, and S. P. Amarasinghe. "StreamIt: A Language for Streaming Applications". In: *Proceedings of the 11th International Conference on Compiler Construction.* CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 179–196. URL: http://dl.acm.org/citation.cfm?id=647478.727935 (cit. on pp. 19, 27, 44, 85, 111).

[TPM13] P. Tendulkar, P. Poplavko, and O. Maler. "Symmetry Breaking for Multi-criteria Mapping and Scheduling on Multicores". In: *Formal Modeling and Analysis of Timed Systems.* Ed. by V. Braberman and L. Fribourg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 228–242 (cit. on pp. 83, 101).

[Tra+19] H. N. Tran, A. Honorat, J.-P. Talpin, T. Gautier, and L. Besnard. "Efficient Contention-Aware Scheduling of SDF Graphs on Shared Multi-bank Memory". In: *ICECCS 2019 - 24th International Conference on Engineering of Complex Computer Systems.* Hong Kong, China: IEEE, Nov. 2019, pp. 114–123. URL: https://hal.inria.fr/hal-02193639 (cit. on p. 130).

[Trö+06] M. Tröngren, D. Henriksson, O. Redell, C. Kirsch, J. El-Khoury, D. Simon, Y. Sorel, H. Zdenek, and K.-E. Årézn. *Co-design of control systems and their real-time implementation : a tool survey.* eng. Trita-MMK, 2006:11. Stockholm: Department of Machine Design, Royal Institute of Technology, 2006 (cit. on p. 27).

[UGT09] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. "Software Pipelined Execution of Stream Programs on GPUs". In: *2009 International Symposium on Code Generation and Optimization.* 2009, pp. 200–209 (cit. on p. 118).

[Ver10] S. Verdoolaege. "Polyhedral Process Networks". In: *Handbook of Signal Processing Systems.* Ed. by S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala. Boston, MA: Springer US, 2010, pp. 931–965. URL: https://doi.org/10.1007/978-1-4419-6345-1_33 (cit. on pp. 20, 58).

[WA85] W. W. Wadge and E. A. Ashcroft. *LUCID, the Dataflow Programming Language.* San Diego, CA, USA: Academic Press Professional, Inc., 1985 (cit. on p. 8).

[Wae+15] L. Waeijen, D. She, H. Corporaal, and Y. He. "A Low-Energy Wide SIMD Architecture with Explicit Datapath". In: *Journal of Signal Processing Systems* 80.1 (July 2015), pp. 65–86. URL: https://doi.org/10.1007/s11265-014-0950-8 (cit. on p. 12).

[Wai+97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. "Baring it all to software: Raw machines". In: *Computer* 30.9 (Sept. 1997), pp. 86–93 (cit. on p. 12).

[Wan+14] G. Wang, R. Allen, H. Andrade, and A. Sangiovanni-Vincentelli. "Communication storage optimization for static dataflow with access patterns under periodic scheduling and throughput constraint". In: *Computers & Electrical Engineering* 40.6 (2014), pp. 1858–1873. URL: http://www.sciencedirect.com/science/article/pii/S0045790614001281 (cit. on p. 130).

[WBS07] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. "Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs". In: *2007 44th ACM/IEEE Design Automation Conference*. June 2007, pp. 658–663 (cit. on p. 30).

[Wei+14] H. Wei, S. Zuckerman, X. Li, and G. R. Gao. "A Dataflow Programming Language and its Compiler for Streaming Systems". In: *Procedia Computer Science* 29 (2014), pp. 1289–1298. URL: http://www.sciencedirect.com/science/article/pii/S1877050914002932 (cit. on p. 9).

[Wen+18] S. Wen, M. Sheng, C. Ma, Z. Li, H. K. Lam, Y. Zhao, and J. Ma. "Camera Recognition and Laser Detection based on EKF-SLAM in the Autonomous Navigation of Humanoid Robot". In: *Journal of Intelligent & Robotic Systems* 92.2 (Oct. 2018), pp. 265–277. URL: https://doi.org/10.1007/s10846-017-0712-5 (cit. on p. 62).

[WR12] M. Wipliez and M. Raulet. "Classification of Dataflow Actors with Satisfiability and Abstract Interpretation". In: *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)* 3.1 (2012), pp. 49–69. URL: https://hal.archives-ouvertes.fr/hal-00717361 (cit. on p. 13).

[WRG16] J. Wang, P. S. Roop, and A. Girault. "Energy and timing aware synchronous programming". In: *International Conference on Embedded Software, EMSOFT'16*. Pittsburgh, United States: ACM, Oct. 2016, p. 10. URL: https://hal.inria.fr/hal-01412100 (cit. on p. 144).

[Wu+11] H. H. Wu, C. C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya. "A Model-Based Schedule Representation for Heterogeneous Mapping of Dataflow Graphs". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. May 2011, pp. 70–81 (cit. on p. 22).

[XAP17] J. Xiao, S. Altmeyer, and A. Pimentel. "Schedulability Analysis of Non-preemptive Real-Time Scheduling for Multicore Processors with Shared Caches". In: *2017 IEEE Real-Time Systems Symposium (RTSS)*. Dec. 2017, pp. 199–208 (cit. on p. 92).

[Yan+19] S. Yang, S. l. Nours, M. m. Real, and S. Pillement. "Mapping and Frequency Joint Optimization for Energy Efficient Execution of Multiple Applications on Multicore Systems". In: *2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2019, pp. 29–34 (cit. on p. 144).

[YH09] H. Yang and S. Ha. "Pipelined data parallel task mapping/scheduling technique for MPSoC". In: *2009 Design, Automation, Test in Europe Conference Exhibition*. 2009, pp. 69–74 (cit. on p. 118).

[YH12] Yuankai Chen and Hai Zhou. "Buffer minimization in pipelined SDF scheduling on multi-core platforms". In: *17th Asia and South Pacific Design Automation Conference*. 2012, pp. 127–132 (cit. on p. 118).

[YKG18] W. Yu, J. Kornerup, and A. Gerstlauer. "MASES: Mobility And Slack Enhanced Scheduling For Latency-Optimized Pipelined Dataflow Graphs". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. SCOPES '18. Sankt Goar, Germany: ACM, 2018, pp. 104–109. URL: http://doi.acm.org/10.1145/3207719.3207733 (cit. on p. 145).

[Yvi+13] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet. "Orcc: Multimedia Development Made Easy". In: *Proceedings of the 21st ACM International Conference on Multimedia*. MM '13. ACM, 2013, pp. 863–866 (cit. on pp. 28, 58).

[Zak+17] G. F. Zaki, W. Plishker, S. S. Bhattacharyya, and F. Fruth. "Implementation, Scheduling, and Adaptation of Partial Expansion Graphs on Multicore Platforms". In: *Journal of Signal Processing Systems* 87.1 (Apr. 2017), pp. 107–125. URL: https://doi.org/10.1007/s11265-016-1107-8 (cit. on p. 58).

[ZBS13] J. T. Zhai, M. A. Bamakhrama, and T. Stefanov. "Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems". In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. May 2013, pp. 1–8 (cit. on p. 58).

[Zeb+08] C. Zebelein, J. Falk, C. Haubelt, and J. Teich. "Classification of General Data Flow Actors into Known Models of Computation". In: *2008 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*. June 2008, pp. 119–128 (cit. on p. 13).

[Zeb+13] C. Zebelein, C. Haubelt, J. Falk, T. Schwarzer, and J. Teich. "Representing mapping and scheduling decisions within dataflow graphs". In: *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*. Sept. 2013, pp. 1–8 (cit. on p. 22).

[Zha+05] D. Zhang, Z.-Z. Li, H. Song, and L. Liu. "A Programming Model for an Embedded Media Processing Architecture". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Ed. by T. D. Hämäläinen, A. D. Pimentel, J. Takala, and S. Vassiliadis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 251–261 (cit. on p. 9).

[Zho+16] Z. Zhou, W. Plishker, S. S. Bhattacharyya, K. Desnos, M. Pelcat, and J.-F. Nezan. "Scheduling of Parallelized Synchronous Dataflow Actors for Multicore Signal Processing". In: *J. Signal Process. Syst.* 83.3 (June 2016), pp. 309–328. URL: http://dx.doi.org.insis.bib.cnrs.fr/10.1007/s11265-014-0956-2 (cit. on p. 146).

[Zhu+16] X. Zhu, M. Geilen, T. Basten, and S. Stuijk. "Multiconstraint Static Scheduling of Synchronous Dataflow Graphs Via Retiming and Unfolding". In: *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* 35.6 (June 2016), pp. 905–918 (cit. on pp. 118, 146).

[ZNS11] J. T. Zhai, H. Nikolov, and T. Stefanov. "Modeling adaptive streaming applications with Parameterized Polyhedral Process Networks". In: *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2011, pp. 116–121 (cit. on pp. 21, 58).

[ZNS18] J. T. Zhai, S. Niknam, and T. Stefanov. "Modeling, Analysis, and Hard Real-Time Scheduling of Adaptive Streaming Applications". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2636–2648 (cit. on p. 21).

[ZRW12] X. Zhang, A. B. Rad, and Y.-K. Wong. "Sensor Fusion of Monocular Cameras and Laser Rangefinders for Line-Based Simultaneous Localization and Mapping (SLAM) Tasks in Autonomous Mobile Robots". In: *Sensors*. 2012 (cit. on p. 62).

[Zub91] W. Zuberek. "Timed Petri nets definitions, properties, and applications". In: *Microelectronics Reliability* 31.4 (1991), pp. 627–644. URL: http://www.sciencedirect.com/science/article/pii/002627149190007T (cit. on p. 7).

[Zub93] W. M. Zuberek. "Throughput analysis of simple closed timed Petri net models". In: *Proceedings of 36th Midwest Symposium on Circuits and Systems*. Aug. 1993, 930–933 vol.2 (cit. on p. 8).

List of published contributions

[Hon+17] A. Honorat, H. N. Tran, L. Besnard, T. Gautier, J.-P. Talpin, and A. Bouakaz. "ADFG: a scheduling synthesis tool for dataflow graphs in real-time systems". In: *Real-Time Networks and Systems*. Grenoble, France, Oct. 2017, pp. 1–10. URL: https://hal.inria.fr/hal-01615142 (cit. on pp. 28, 89, 92, 129).

[Hon+19] A. Honorat, K. Desnos, M. Pelcat, and J.-F. Nezan. "Modeling Nested for Loops with Explicit Parallelism in Synchronous DataFlow Graphs". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Embedded Computer Systems: Architectures, Modeling, and Simulation 19th International Conference, SAMOS 2019, Samos, Greece, July 7–11, 2019, Proceedings. Pythagorion, Samos Island, Greece, July 2019, pp. 269–280. URL: https://hal.archives-ouvertes.fr/hal-02267487 (cit. on p. 60).

[Hon+20a] A. Honorat, K. Desnos, S. S. Bhattacharyya, and J.-F. Nezan. "Scheduling of Synchronous Dataflow Graphs with Partially Periodic Real-Time Constraints". In: *Real-Time Networks and Systems*. Paris, France, June 2020. URL: https://hal.archives-ouvertes.fr/hal-02883663 (cit. on pp. 72, 94).

[Hon+20b] A. Honorat, K. Desnos, M. Dardaillon, and J.-F. Nezan. "A Fast Heuristic to Pipeline SDF Graphs". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Embedded Computer Systems: Architectures, Modeling, and Simulation 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings. Pythagorion, Samos Island, Greece, July 2020, pp. 139–151. URL: https://hal.archives-ouvertes.fr/hal-02993338 (cit. on p. 120).

Acronyms

AADL Architecture Analysis and Design Language. xxviii, 9, 85

ALAP As Late As Possible. vi, 78, 79, 84, 97, 99, 103–107

ALU Arithmetic Logic Unit. 11

ASAP As Soon As Possible. vi, 75, 76, 78, 79, 97, 99, 103–107

ASIC Application-Specific Integrated Circuit. 12

ASIP Application-Specific Instruction set Processor. 12, 13

BFS Breadth-First Search. ix, 15, 19, 74, 75

CP Constraint Programming. 25, 82, 86, 92, 113, 130

CPS Cyber-Physical System. 9

CPU Central Processing Unit. 3, 9–13, 24, 26, 32, 119, 145

CSDF Cyclo-Static Data Flow. v, 19, 20, 28–30, 44, 59, 85, 150

DAG Directed Acyclic Graph. 22, 24–26, 29, 34, 35, 39, 42, 66, 72, 73, 75, 77, 78, 83, 85, 86, 93, 96, 101, 104, 108, 124, 130, 131, 146

DFS Depth-First Search. ix, 36–38

DPN Dataflow Process Network. 7, 19

DSE Design Space Exploration. vi, vii, xi, xxviii–xxxii, 25, 26, 59, 60, 94, 119, 122, 123, 127–130, 132–150

DSP Digital Signal Processor. 12, 26, 32, 119

DVFS Dynamic Voltage and Frequency Scaling. 141, 144

EDF Earliest Deadline First. vi, 23, 25, 30, 89, 91, 92, 150

ET Execution Time. xxxi, 7, 17, 21, 24, 27, 33, 34, 41, 42, 52, 78, 96, 106–108, 112, 113, 115–117, 119, 122–124, 129, 130, 132, 141, 144, 151

FIFO First In First Out. 14–16, 18

FP Fixed Priority. 23, 25, 28–30, 92

FPGA Field-Programmable Gate Array. 12, 13, 27, 28, 119, 145

fps Frame Per Second. 44, 139, 140, 142, 143

FSM Finite State Machine. 7, 21, 22, 27

GPU Graphics Processing Unit. 3, 10, 26, 119

GUI Graphical User Interface. 28, 30–35, 127

HSDF Homogeneous Synchronous Data Flow. 15, 42

IBSDF Interfaced-Based Synchronous Data Flow. 20, 29

II Initiation Interval. vi, 17, 24, 25, 97, 98, 102–104, 111, 118, 132–137, 141–143

ILP Integer Linear Programming. 25, 26, 78, 82, 89, 92, 93, 97, 99, 100, 118, 145

KPN Kahn Process Network. 7, 9, 19–21, 27, 44, 145

LLC Last Level Cache. 3, 11

LP Linear Programming. 25, 119, 131

MCDA Multi-Criteria Decision Analysis. 148

MILP Mixed Integer Linear Programming. 146

MIMD Multiple Instruction Multiple Data. 10, 11

MoA Model of Architecture. 10, 11, 31

MoC Model of Computation. xxviii, xxx, 2–4, 6, 8–10, 13, 18, 31, 44, 45, 50, 53, 57–59, 102, 110, 122, 123, 145, 146, 150, 152

MPSoC Multi-Processor System-on-Chip. xxvii, xxviii, 3, 151

NoC Network on Chip. 12

NUMA Non Uniform Memory Access. 11

OS Operating System. 3, 4, 6, 8, 10

PE Processing Element. vi, vii, xxvii, xxix, xxxii, 2–5, 9, 15–18, 20, 22–26, 28, 32, 33, 41, 42, 44, 45, 47–49, 52, 54, 59, 63–70, 72, 73, 75–77, 79–87, 89–91, 93, 96, 99, 102, 103, 105, 106, 111, 112, 114–119, 122, 131, 137, 139, 141–144, 150, 151

PIMM Parameterized Interfaced Meta-Model. xxviii, xxx, 21, 31, 36, 41, 123

PISDF Parameterized Interfaced Synchronous Data Flow. v, vi, xi, xxix, xxx, xxxii, 28, 29, 31, 33–36, 39–41, 122–127, 131, 132, 134, 139, 147

PREESM Parallel and Real-time Embedded Executives Scheduling Method. v, xxx, xxxi, 2, 25, 28, 30–34, 36, 39–41, 53, 54, 56, 62, 69, 82, 86, 89, 94, 109–113, 116, 118, 120, 122–127, 130–132, 142, 146–149, 151

QoS Quality of Service. xxvii, xxix, xxx, xxxii, 2, 27, 122, 128–130, 134, 147, 149–151

RAM Random Access Memory. 3, 11, 32

S-LAM System-Level Architecture Model. 11, 31–33, 41, 147

SAT Satisfiability. 25

SDF Synchronous Data Flow. v–vii, xi, xxviii–xxxii, 2, 8, 9, 13–22, 24, 26–31, 39–42, 44–59, 62–70, 72, 73, 75–79, 84–88, 91–94, 96–98, 101–104, 107–115, 117–119, 122, 123, 129–131, 133, 144–146, 149–152

SIFT Scale Invariant Feature Transform. v–vii, xi, xxx, 45, 46, 48, 49, 53, 55–57, 59, 60, 102, 111, 124–126, 131, 139, 140, 142–144, 149, 150

SIMD Single Instruction Multiple Data. 10, 12

SLAM Simultaneous Localization And Mapping. 62

SMT Simultaneous Multi-Threading. 11

SMT Satisfiability Modulo Theories. 25, 26

SPIDER Synchronous Parameterized and Interfaced Dataflow Embedded Runtime. 28, 125, 151

SRSDF Single-Rate Synchronous Data Flow. v, vi, 15–18, 20, 22, 24, 26, 28, 31, 39, 40, 42, 77, 78, 83, 86, 92, 93, 107, 117, 118, 130–132, 145

SSA Single Static Assignment. 9, 35, 125

TIOA Timed Input Output Automata. 7

UML Unified Modeling Language. 145

VLSI Very Large Scale Integration. 117

WCET Worst Case Execution Time. vi, 11, 42, 63–66, 72, 73, 75, 82–84, 86, 88, 90

Edition notice

This thesis has been modified after the PhD defense in order to address some requests made by the jury members (as specified in the document on the next page). The present version includes all the requested modifications. In particular, Sections 1.4, 3.4.3, 4.1.3 and 4.1.4 have been entirely written after the PhD defense.

Versions of this document:

- *September* 30$^{\text{th}}$*, 2020* original version sent to the reviewers;
- *February* 22$^{\text{nd}}$*, 2021* (major update) version with all corrections asked by the reviewers, and approved by the president of the jury;
- *July* 22$^{\text{nd}}$*, 2021* (minor update) version with darker colors for printing and revised Section 4.1.4;
- *September* 1$^{\text{st}}$*, 2021* (minor update) version with list of versions and certificate of corrections.

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Modélisation, Ordonnancement, Pipelinage et Configuration de Graphes Synchrones de Flux de Données sous contrainte de Cadence

Nom Prénom de l'auteur : HONORAT ALEXANDRE

Membres du jury :
- Madame PAGETTI Claire
- Monsieur SUTER Frédéric
- Monsieur GIRAULT Alain
- Monsieur LILIUS Johan
- Monsieur NEZAN Jean-François
- Monsieur DESNOS Karol

Président du jury : Alain Girault

Date de la soutenance : 27 Novembre 2020

Reproduction de la these soutenue

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

Fait à Rennes, le 27 Novembre 2020

Signature du président de jury

Le Directeur,

M'hamed DRISSI

ATTESTATION DE CORRECTIONS DE MANUSCRIT DE THESE*

**A fournir, lorsque le jury a indiqué que le manuscrit de thèse pouvait être reproduit après corrections souhaitées (dans un délai de 3 mois). Cette attestation est à remettre avec le manuscrit au moment du dépôt à la bibliothèque.*

Titre de la thèse : Modélisation, Ordonnancement, Pipelinage et Configuration de Graphes Synchrones de Flux de Données sous contrainte de Cadence

Nom et prénom de l'auteur : Alexandre Honorat

Date et lieu de soutenance : 27 Novembre 2020 à Rennes

Nom et prénom du directeur de thèse : Jean-François Nezan

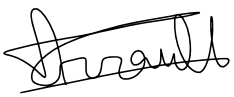
Membres du jury : Claire Pagetti, Frédéric Suter, Alain Girault, Johan Lilius, Jean-François Nezan, Karol Desnos

Président du jury : Alain Girault

Membre désigné pour l'attestation de corrections : Alain Girault

Je soussigné, Alain Girault, désigné par le jury de soutenance dans le cadre de cette thèse, atteste que les corrections souhaitées par le jury ont été prise en compte dans l'exemplaire final du manuscrit de thèse.

A Grenoble, Le 25/02/y2021,



Signature du membre désigné pour l'attestation des corrections

version of September 1st, 2021
This document has been typeset with LaTeX.
If you find any error or typo[1] , feel free to contact me.
My email address is indicated on my website:
https://ahonorat.github.io

[1]Please provide its error type and precise at what level it has affected you, sorted by U type.

DOCTORAT BRETAGNE LOIRE MATHSTIC

INSA RENNES

Titre : Modélisation, Ordonnancement, Pipelinage et Configuration de Graphes Synchrones de Flux de Données sous Contrainte de Cadence

Mots-clés : SDF, modélisation, ordonnancement, pipelinage, DSE

Résumé : Les Systèmes Multi-Processeurs Intégrés sur Puce (MPSoC) sont maintenant embarqués dans de plus en plus d'appareils, par exemple sur des caméras intelligentes qui peuvent exécuter des applications de traitement d'image en temps réel. La conception d'applications exploitant entièrement les capacités d'un MPSoC est difficile ; elle demande de prendre en compte plusieurs contraintes telles qu'une consommation maximale d'énergie pour préserver la batterie et une fréquence d'images minimale pour assurer une bonne qualité vidéo. Grâce à de rapides heuristiques d'analyse et de synthèse, cette thèse adopte une approche globale, de la modélisation à la configuration, au problème de conception. Pour ce faire, l'application à concevoir est d'abord modélisée indépendamment de n'importe quel MPSoC. L'application est ensuite automatiquement configurée pour un MPSoC grâce à un logiciel d'analyse et de synthèse. Les modèles utilisés dans cette thèse découlent du Modèle de Calcul (MoC) Synchrone de Flux de Données (SDF), et le logiciel d'analyse et de synthèse est PREESM.

Dans cette thèse, trois aspects de la conception ont été abordés : la modélisation de boucles itératives, l'ordonnancement sous contraintes temps réel, et le pipelinage de tâches. Une quatrième contribution allie ces trois aspects, il s'agit d'un algorithme d'Exploration de l'Espace de Conception (DSE) prenant en considération des contraintes de cadence, de latence et d'énergie. Cette DSE permet de configurer automatiquement les paramètres d'une application de telle sorte que toutes les contraintes soient respectées. Toutes les contributions ont été implantées dans le logiciel PREESM.

Title: Modeling, Scheduling, Pipelining and Configuration of Synchronous Dataflow Graphs with Throughput Constraints

Keywords: SDF, modeling, scheduling, pipelining, DSE

Abstract: Multi-Processors System-on-Chip (MPSoC) are now embedded in more and more devices, for example, in smart cameras which can run image processing applications in real-time. Designing applications that fully exploit the capacity of an MPSoC is a complex task that requires addressing multiple constraints such as maximum energy consumption to preserve the battery and minimal frame rate to ensure a good video quality. This thesis adopts a global approach to the design problem, from modeling to tuning, thanks to fast analysis and synthesis heuristics. To do so, the designed application is first modeled independently from any MPSoC. The application is later automatically tuned and mapped on a MPSoC thanks to an analysis and synthesis framework. The models used in this thesis derive from the Synchronous DataFlow (SDF) Model of Computation (MoC), while the analysis and synthesis framework is PREESM.

In this thesis, three aspects of the design process have been addressed, all at the software level: the modeling of iterative loops, the scheduling of real-time constraints, and the pipelining of tasks. A fourth contribution combines all these aspects: a Design Space Exploration (DSE) algorithm taking into account throughput, latency, and energy constraints. This DSE makes it possible to automatically tune the parameters of an application so that all constraints are met. All the contributions have been implemented and evaluated in the PREESM framework.