Systematic Design Space Exploration of Dynamic Dataflow Programs for Multi-core Platforms

THÈSE N^{O} 7607 (2017)

PRÉSENTÉE LE 17 MARS 2017
À LA FACULTÉ DES SCIENCES ET TECHNIQUES DE L'INGÉNIEUR
GROUPE SCI STI MM
PROGRAMME DOCTORAL EN GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Małgorzata Maria MICHALSKA

acceptée sur proposition du jury:

Prof. C. Dehollain, présidente du jury
Dr M. Mattavelli, directeur de thèse
Prof. J. Castrillon, rapporteur
Dr L. Morel, rapporteur
Prof. A. Burg, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2017

Those who sow with tears
will reap with songs of joy.
(Psalm 126:5)

To my husband Mirosław for reminding me
about the things that really matter …

Acknowledgements

The work described in this dissertation would not have been accomplished without the precious support of many people. I would like to express my sense of gratitude to some of them in particular.

I would like to thank Dr. MER Marco Mattavelli for his constant supervision of my work. The relationship between a student and a supervisor includes some elements of a lead, guidance, advice, help and criticism. It is the constructive criticism that I especially appreciate and that I will take with me for future challenges.

I deeply appreciate the collaboration with Prof. Nicolas Zufferey from University of Geneva. I am thankful for all the discussions and common publications providing valuable contributions to my thesis. For such smaller or bigger contributions, I would like to thank everyone I had a pleasure to collaborate with.

I would like to thank all past and current members of SCI-STI-MM lab. Especially, I would like to thank my colleagues from the former ELG 138 office: Dr. Junaid Ahmad, Dr. Endri Bezati and Dr. Simone Casale Brunet. I appreciate your time, patience and understanding.

I am thankful to my family and friends supporting me during these intensive years of work. Thank you for believing in me, even in my own moments of doubt.

Finally, I would like to thank the Swiss National Science Foundation for founding my research.

Lausanne, 27th February 2017 Małgorzata Michalska

Abstract

The limitations of clock frequency and power dissipation of deep sub-micron CMOS technology have led to the development of massively parallel computing platforms. They consist of dozens or hundreds of processing units and offer a high degree of parallelism. Taking advantage of that parallelism and transforming it into high program performances requires the usage of appropriate parallel programming models and paradigms. Currently, a common practice is to develop parallel applications using methods evolving directly from sequential programming models. However, they lack the abstractions to properly express the concurrency of the processes. An alternative approach is to implement dataflow applications, where the algorithms are described in terms of streams and operators thus their parallelism is directly exposed. Since algorithms are described in an abstract way, they can be easily ported to different types of platforms. Several dataflow models of computation ($MoCs$) have been formalized so far. They differ in terms of their expressiveness (ability to handle dynamic behavior) and complexity of analysis. So far, most of the research efforts have focused on the simpler cases of static dataflow $MoCs$, where many analyses are possible at compile-time and several optimization problems are greatly simplified. At the same time, for the most expressive and the most difficult to analyze dynamic dataflow (DDF), there is still a dearth of tools supporting a systematic and automated analysis minimizing the programming efforts of the designer. The objective of this Thesis is to provide a complete framework to analyze, evaluate and refactor DDF applications expressed using the $RVC-CAL$ language. The methodology relies on a systematic design space exploration (DSE) examining different design alternatives in order to optimize the chosen objective function while satisfying the constraints. The research contributions start from a rigorous DSE problem formulation. This provides a basis for the definition of a complete and novel analysis methodology enabling systematic performance improvements of DDF applications. Different stages of the methodology include exploration heuristics, performance estimation and identification of refactoring directions. All of the stages are implemented as appropriate software tools. The contributions are substantiated by several experiments performed with complex dynamic applications on different types of physical platforms.

Key words: dynamic dataflow, design space exploration, performance estimation, variable space search, heterogeneous platforms, RVC-CAL

Résumé

Les limitations de la technologie CMOS sous-micron profonde en termes de fréquence d'horloge et de dissipation de puissance ont conduit au développement de plates-formes de calcul massivement parallèle. Elles se composent de dizaines voire de centaines d'unités de traitement et offrent un haut degré de parallélisme. Tirer parti de ce parallélisme et le convertir en performances élevées nécessite l'utilisation de modèles et de paradigmes de programmation parallèle appropriés. Actuellement, une pratique courante consiste à développer des applications parallèles en utilisant des méthodes dérivant directement de modèles de programmation séquentielle. Cependant, celles-ci manquent d'abstractions permettant d'exprimer correctement la concurrence des processus. Une autre approche consiste à implémenter des applications dites de flux de données, dans lesquelles les algorithmes sont décrits en termes de flux et d'opérateurs, leur parallélisme étant ainsi directement exposé. Puisque les algorithmes sont décrits de façon abstraite, ils peuvent être facilement portés vers différents types de plates-formes. À ce jour, plusieurs modèles de calcul de flux de données ont été formalisés. Ils diffèrent en termes d'expressivité (capacité à gérer le comportement dynamique) et de complexité de l'analyse. Jusqu'à présent, la plupart des efforts de recherche se sont concentrés sur les cas de modèles statiques, plus simples, dans lesquels de nombreuses analyses sont possibles à la compilation et plusieurs problèmes d'optimisation sont grandement simplifiés. Or, pour les flux de données dynamiques, qui sont plus expressifs et plus difficiles à analyser, il existe toujours une pénurie d'outils supportant une analyse systématique et automatisée minimisant les efforts de programmation du développeur. L'objectif de cette thèse est de fournir un cadre complet pour analyser, évaluer et refactoriser les applications de flux de données dynamiques exprimées dans le langage RVC-CAL. La méthodologie s'appuie sur une exploration systématique de l'espace de design, examinant différentes alternatives de design, afin d'optimiser la fonction objectif choisie tout en satisfaisant les contraintes. Les contributions à la recherche partent d'une formulation rigoureuse des problèmes d'exploration de design. Celle-ci fournit une base pour la définition d'une méthodologie d'analyse complète et novatrice permettant d'améliorer systématiquement les performances des applications dynamiques. Les étapes de la méthodologie incluent l'application d'heuristiques d'exploration, l'estimation des performances et l'identification de directions de refactorisation. Toutes les étapes sont implémentées sous forme d'outils logiciel appropriés. Les contributions sont étayées par plusieurs expériences réalisées avec des applications dynamiques complexes sur différents types de plates-formes physiques.

Mots clefs : flux de données dynamiques, exploration de l'espace de design, estimation des performances, espaces variables de recherche, plates-formes hétérogènes, RVC-CAL

Contents

Acknowledgements . . . i
Abstract (English/Français) . . . ii
List of figures . . . xi
List of tables . . . xv
Abbreviations . . . xix
1 Introduction . . . 1
  1.1 Parallel systems development . . . 1
  1.2 Motivations and problem statement . . . 3
    1.2.1 Design exploration . . . 4
    1.2.2 Dataflow programming . . . 6
    1.2.3 Problem statement . . . 8
  1.3 Research contributions . . . 9
  1.4 Thesis organization . . . 10
2 Dataflow programming . . . 13
  2.1 Dataflow programs . . . 13
    2.1.1 Kahn process network . . . 14
    2.1.2 Dataflow process network . . . 15
    2.1.3 Actor transition systems . . . 16
    2.1.4 Dataflow programs comparison . . . 17
    2.1.5 Dataflow concurrency . . . 18
  2.2 Dataflow classes . . . 18
    2.2.1 Static dataflow programs . . . 19
    2.2.2 Cyclo-static dataflow programs . . . 21
    2.2.3 Dynamic extensions to static dataflow programs . . . 22
    2.2.4 Dynamic dataflow programs . . . 25
  2.3 CAL Actor Language . . . 25
    2.3.1 CAL program . . . 26
    2.3.2 CAL syntax . . . 28
    2.3.3 Example . . . 32
    2.3.4 Code generation . . . 34
    2.3.5 RVC-CAL . . . 35

2.4 Summary and conclusions . . . 36
3 State-of-the-art of dataflow-oriented analysis frameworks . . . 39
3.1 Frameworks . . . 39
3.2 Features . . . 41
3.3 Exploration heuristics . . . 41
3.4 RVC-CAL frameworks . . . 48
3.5 Design flow for dataflow programs . . . 50
3.6 Contributions to the state-of-the-art . . . 51
3.7 Summary and conclusions . . . 51
4 Program execution modeling . . . 55
4.1 Execution Trace Graph . . . 55
4.1.1 Firings . . . 56
4.1.2 Dependencies . . . 56
4.1.3 Example . . . 60
4.2 Properties . . . 61
4.2.1 Topological order . . . 63
4.2.2 Configuration-related dependencies . . . 63
4.2.3 Capturing the dynamic behavior . . . 65
4.2.4 Potential parallelism . . . 67
4.3 Conclusions . . . 68
5 Architecture modeling . . . 69
5.1 Abstract-to-timed translation . . . 69
5.2 Transport Triggered Architecture . . . 71
5.3 Intel 86x64 . . . 73
5.3.1 Processing weights . . . 74
5.3.2 Scheduling weights . . . 76
5.3.3 Communication weights . . . 77
5.4 Profiling accuracy . . . 79
5.5 Conclusions . . . 79
6 Design space exploration problem . . . 81
6.1 Related work . . . 81
6.2 Underlying optimization problems . . . 83
6.2.1 Partitioning . . . 83
6.2.2 Scheduling . . . 84
6.2.3 Buffer dimensioning . . . 84
6.3 Target platforms . . . 85
6.3.1 Homogeneous platforms . . . 85
6.3.2 Heterogeneous platforms . . . 85
6.4 Formulation of the design space exploration problem . . . 87

6.5 Problem instance sizes . . . 89
6.6 Conclusions . . . 91
7 Exploration of multiple multidimensional design spaces . . . 93
7.1 Related work . . . 93
7.1.1 Design space exploration variants . . . 93
7.1.2 Bottlenecks in design space exploration . . . 95
7.1.3 Variable space . . . 96
7.2 Multidimensional design space definition . . . 97
7.3 Space-to-space transition . . . 98
7.4 Design space quality . . . 99
7.5 Design space complexity . . . 99
7.6 Optimization scenarios . . . 101
7.7 Variable Space Search . . . 102
7.8 Conclusions . . . 103
8 Heuristics for design space exploration . . . 105
8.1 Partitioning . . . 106
8.1.1 Related work . . . 107
8.1.2 Greedy constructive procedures . . . 108
8.1.3 Descent local search procedures . . . 111
8.1.4 Tabu search . . . 113
8.1.5 Advanced tabu search . . . 115
8.1.6 Tabu search with iterative communication cost profiling . . . 115
8.2 Buffer dimensioning . . . 116
8.2.1 Related work . . . 117
8.2.2 Notion of partitioning and scheduling configurations . . . 118
8.2.3 Minimal and maximal buffer size estimation . . . 119
8.2.4 Bottom-up optimization procedures . . . 120
8.2.5 Top-down optimization procedure . . . 121
8.3 Scheduling . . . 122
8.3.1 Related work . . . 124
8.3.2 Intra-actor and intra-partition scheduling policies . . . 125
8.3.3 Scheduling cost . . . 127
8.4 Conclusions . . . 129
9 Performance estimation . . . 131
9.1 Related work . . . 131
9.2 Trace processor tool . . . 134
9.2.1 Atomic models . . . 135
9.2.2 Atomic models interaction . . . 136
9.2.3 Execution properties . . . 138

9.3 Critical path analysis . . . 140
    9.3.1 Algorithm . . . 140
9.4 Impact analysis . . . 141
9.5 Conclusions . . . 141
10 Experimental results . . . 147
10.1 Experimental set-up . . . 147
    10.1.1 JPEG decoder . . . 149
    10.1.2 MPEG4-SP decoder . . . 149
    10.1.3 HEVC decoder . . . 150
    10.1.4 Target platforms . . . 150
10.2 Partitioning: experiments on Transport Triggered Architecture . . . 152
    10.2.1 Methodology of experiments . . . 153
    10.2.2 Parameters tuning . . . 154
    10.2.3 Greedy and descent local search procedures . . . 154
    10.2.4 Tabu search . . . 155
    10.2.5 Discussion . . . 159
10.3 Partitioning: experiments on Intel 86x64 platforms . . . 160
    10.3.1 Methodology of experiments . . . 160
    10.3.2 Heuristics comparison . . . 161
    10.3.3 Discussion . . . 162
10.4 Buffer dimensioning . . . 162
    10.4.1 Bottom-up: throughput optimization with constrained resources . . . 163
    10.4.2 Top-down: resources minimization with constrained throughput . . . 163
    10.4.3 Multidimensional vs single-dimension exploration . . . 165
    10.4.4 Discussion . . . 165
10.5 Scheduling . . . 169
    10.5.1 Performance potential . . . 169
    10.5.2 Scheduling cost . . . 172
    10.5.3 Discussion . . . 173
10.6 Performance estimation . . . 173
    10.6.1 Estimation accuracy on TTA platform . . . 173
    10.6.2 Estimation accuracy on Intel 86x64 platforms . . . 175
    10.6.3 Estimation accuracy across different platforms . . . 177
    10.6.4 Discussion . . . 179
10.7 Variable Space Search . . . 180
    10.7.1 Design spaces . . . 180
    10.7.2 Critical path and impact analysis . . . 183
    10.7.3 Solutions in multiple spaces . . . 188
    10.7.4 Discussion . . . 191

10.8 Conclusions 197
11 Conclusions 199
11.1 Achievements of the Thesis 200
11.2 Work extensions 203
11.2.1 Improvements 203
11.2.2 Open problems 205
Bibliography 226
Curriculum Vitae 227

List of Figures

2.1 Dataflow *MoCs* classes. 18

2.2 A dataflow graph with two actors, a_i and a_j, connected through the buffer b_n. $p_{i,n}$ defines the number of tokens produced on b_n during each firing of a_i. $c_{j,n}$ defines the number of tokens consumed from b_n during each firing of a_j. . . . 19

2.3 Example of a dataflow graph. 20

2.4 *CAL* network and actors structure. 26

2.5 Action execution model 28

2.6 Basic dataflow program. 32

2.7 Dataflow *MoCs* comparison. 37

3.1 Heterogeneous system development design flow for *CAL* dataflow programs. 52

4.1 Example: Gantt chart. 60

4.2 *ETG* obtained after the execution of the *CAL* program from Section 2.3.3. The set of firings $\mathbb{S}$ (dependencies $\mathbb{D}$) is summarized in Table 2.3 (4.2), respectively. 61

4.3 *ETGs* of the *CAL* network depicted in Fig. 2.6. Dashed lines represent additional edges that model a particular partitioning, scheduling and buffer dimensioning configuration, as defined in different variants in Table 4.3. . . . 66

5.1 System development design flow: profiling. 70

5.2 Transport Triggered Architecture model. 72

5.3 *TTA* time-stamp placement: pseudocode. 73

5.4 Intel processing weights clock-cycles measurement: pseudocode. 75

5.5 *TTA* profiling data of a single action: collected timing data of 1600 firings. . . 76

5.6 Intel profiling data of a single action: collected timing data of 1600 firings. . . 76

5.7 Intel scheduling weights clock-cycles measurement: pseudocode. 80

6.1 The partitioning, scheduling and buffer dimensioning subproblems. 83

6.2 Homogeneous platform. 86

6.3 Heterogeneous platform. 88

6.4 Dataflow program representations. 90

7.1 DSE example: MPEG4-SP decoder on Intel i7-3770. 101

8.1 System development design flow: design space exploration. 106
8.2 Simple network with the assigned partitioning, scheduling and buffer dimensioning configurations. 119
8.3 Gantt charts for the execution of the network from Fig. 8.2 for the two buffer size configurations. 119
8.4 Scheduling policy - illustration. 126
8.5 Sample partition with the actors to be considered by the scheduler. 128

9.1 System development design flow: performance estimation. 132
9.2 Schematic illustration of the system components and connections. 135
9.3 Atomic actor model: state transition. 143
9.4 Atomic actor partition model: state transition. 144
9.5 Components interaction and exchanged signals. 144
9.6 Communication procedure between an actor and a buffer. 145
9.7 System development design flow: bottleneck and impact analysis. 146

10.1 Extended system development design flow for CAL dataflow programs. 148
10.2 JPEG decoder: dataflow network. 149
10.3 Dataflow networks. 151
10.4 Partitioning heuristics - TTA - experimental workflow. 153
10.5 Partitioning heuristics - Intel - experimental workflow. 160
10.6 Bottom-up buffer dimensioning heuristic (MPEG4-SP Parallel decoder). 164
10.7 Bottom-up buffer dimensioning heuristic (HEVC decoder, 7 cores). 165
10.8 Top-down buffer dimensioning heuristic. 166
10.9 MDS vs design B-subspace vs infinite buffer. 167
10.10 TTA platform execution. 171
10.11 Performance estimation validation - experimental workflow. 174
10.12 TTA clock-cycles comparison. 174
10.13 JPEG, machine M2 (discrepancy as % value). 176
10.14 MPEG-4 SP decoder estimation results (discrepancy as % value). 176
10.15 Estimation accuracy: different platforms. 178
10.16 Design space D_0. 188
10.17 Design space D_1. 189
10.18 Design space D_2. 189
10.19 Design space D_3. 189
10.20 Design space D_4. 190
10.21 Design space D_{11}. 190

10.22 Design space D_{12} 190
10.23 Throughput improvement summary (1). 192
10.24 Throughput improvement summary (2). 193
10.25 Throughput and resources in different design spaces: summary (1). 194
10.26 Throughput and resources in different design spaces: summary (2). 195

List of Tables

2.1 KPN, DPN and ATS programs: comparison. 17
2.2 CAL lexical tokens. 29
2.3 Firings of the CAL program described in Section 2.3.3. 34

3.1 Analysis frameworks: features summary, part 1. 42
3.2 Analysis frameworks: features summary, part 2. 43
3.3 Analysis frameworks: features summary, part 3. 44

4.1 Dependencies: types, directions, parameters and additional attributes. 59
4.2 Dependencies set $\mathbb{D}$ of the execution trace graph depicted in Figure 4.2. . . . 62
4.3 Summary of the design points considered for the program network in Figure 2.6. The actors: Source, Medium and Sink are denoted with Sr, Md, Sk, respectively. 64
4.4 Firings sequence of the CAL actor `Separator` (defined in Listing 2.2), when two input sequences are available in its input port `I`: $I_1 = \{1, 2, -3, -4\}$ and $I_2 = \{-5, -6, 0, -7\}$, respectively. 65

5.1 Statistical information obtained for a single action on the TTA and Intel platforms. 75
5.2 Sample communication cost data. 78

8.1 WB partitioning algorithm: sample flow. 108
8.2 BP partitioning algorithm: AW and APW settings. 110
8.3 BP partitioning algorithm: ACP settings. 110
8.4 BP partitioning algorithm: sample flow. 110
8.5 Conditions numbers updates. 128

10.1 Configurations of Intel platforms. 152
10.2 Speed-up: Greedy constructive procedures. 155
10.3 DLS Speed-up: balanced start. 155
10.4 DLS Speed-up: random start. 155
10.5 Speed-up: $N^{(B)}$ with balanced start. 156
10.6 Speed-up: $N^{(B)}$ with random start. 156

10.7 Speed-up: $N^{(I)}$ with balanced start. 157
10.8 Speed-up: $N^{(I)}$ with random start. 157
10.9 Speed-up: $N^{(CF)}$ with balanced start. 157
10.10 Speed-up: $N^{(CF)}$ with random start. 157
10.11 Speed-up: $N^{(R)}$ with balanced start. 157
10.12 Speed-up: $N^{(R)}$ with random start. 157
10.13 PTS and JTS speed-up: balanced start. 158
10.14 PTS and JTS speed-up: random start. 158
10.15 Improvement summary. 158
10.16 Averaged time of the final improvement. 160
10.17 Platform execution speed-up: SOA approach. 161
10.18 Platform execution speed-up: BP with additional optimization procedures. . 161
10.19 Platform execution speed-up: tabu search with an iterative re-profiling procedure. 161
10.20 MPEG4-SP Parallel: performance differences [FPS]. 163
10.21 HEVC: performance differences [FPS]. 163
10.22 Estimated speed-ups: balanced partitioning configurations. 169
10.23 Estimated speed-ups: random partitioning configurations. 170
10.24 Execution times *NnP*. 172
10.25 Execution times *RR*. 172
10.26 Conditions checked. 172
10.27 Conditions failed. 172
10.28 Execution time change vs the initial configuration (4 units). 175
10.29 Execution time change vs the initial configuration (5 units). 175
10.30 MPEG-4 SP Serial: estimation with the communication cost, discrepancy. . . 177
10.31 HEVC: summary of the considered design spaces. 181
10.32 HEVC: summary of the improvement achieved in various design spaces. . . . 182
10.33 Design space D_0. 184
10.34 Design space D_1. 184
10.35 Design space D_2. 184
10.36 Design space D_3. 185
10.37 Design space D_4. 185
10.38 Design space D_5. 185
10.39 Design space D_6. 186
10.40 Design space D_7. 186
10.41 Design space D_8. 186
10.42 Design space D_9. 187
10.43 Design space D_{10}. 187
10.44 Design space D_{11}. 187

10.45 Design space D_{12} . . . 188

Abbrevations

AAA/M	Algorithm Architecture Adequation Matching Methodology
ACP	Average Common Predecessors
APO	Average Partitioning Occupancy
APW	Average Preceding Workload
AST	Abstract Syntax Tree
ATS	Actor Transition Systems
BDF	Boolean Dataflow
BP	Balanced Pipeline
BPDF	Boolean Parametric Dataflow
CAL	Cal Actor Language
CFDLS	Communication Frequency Descent Local Search
CFG	Control Flow Graph
CNnP	Critical Non-Preemptive
COW	Critical Outgoings Workload
CP	Critical Path
CSDF	Cyclo-Static Dataflow
DAG	Directed Acyclic Graph
DDF	Dynamic Dataflow
DEVS	Discrete Event System

DLS	Descent Local Search
DSE	Design Space Exploration
ECO	Earliest Critical Outgoings
EET	Estimated Execution Time
EMF	Eclipse Modeling Framework
ETG	Execution Trace Graph
FCFS	First-Come First-Served
FNL	Functional unit Network Language
FSM	Finite State Machine
HW	Hardware
IANnP	Intra-Actor Non-Preemptive
IAP	Intra-Actor Preemptive
IASP	Intra-Actor Scheduling Policy
IDF	Integer Dataflow
IDLS	Idle Descent Local Search
IPSP	Intra-Partition Scheduling Policy
IR	Intermediate Representation
JTS	Joint Tabu Search
KPN	Kahn Process Network
LTS	Labeled Transition Systems
LUB	Lowest Upper Bound
MCDF	Mode-controlled Dataflow
MCDM	Multi-Criteria Decision Making
MDE	Model Driven Engineering
MDS	Multidimensional Design Space

MMPN	Multiprocessor Mappings of Process Networks
MoC	Model of Computation
MOEA	Multi-Objective Evolutionary Algorithms
MPSoC	Multiprocessor System-on-chip
NnP	Non-Preemptive
NnP/P	Non-Preemptive / Preemptive swapped
NUMA	Non Uniform Memory Access
ORCC	Open RVC-CAL Compiler
PAPS	Periodic Admissible Parallel Schedule
PASS	Periodic Admissible Sequential Schedule
PCP	Partial Critical Path
PE	Performance Estimation
PiMM	Parameterized and Interfaced dataflow Meta-Model
PSDF	Parametrized Synchronous Dataflow
PTS	Probabilistic Tabu Search
RR	Round Robin
RTL	Register Transfer Language
SADF	Scenario-Aware Dataflow
SDF	Synchronous Dataflow
SDO	Standard Deviation of Occupancy
SoC	System on Chip
SPDF	Schedulable Parametric Dataflow
SPEO	Strength Pareto Evolutionary Algorithm
SW	Software
TETG	Timed Execution Trace Graph

TP	Trace Processor
TPDF	Transaction Parametrized Dataflow
TS	Tabu Search
VLIW	Very Long Instruction Word
VSS	Variable Space Search
WB	Workload Balance
XDF	XML Data Format
XML	eXtensible Markup Language

1 Introduction

Due to the broad availability of many- and multi-core platforms, there is an increasing interest in developing applications taking advantage of the offered parallelism. Given the scale of the massively parallel platforms consisting of dozens or hundreds of processing units, different programming languages have been developed and compete in order to ensure scalability, productivity and reusability, and to meet design and performance constraints. One of the paradigms to be used in conjunction with such parallel platforms is dataflow programming. Dataflow programs respond to the increasing demands of designing highly parallel applications expressed at a high level of abstraction. Multiple complex applications (*i.e.*, in the field of media and signal processing) display dynamic behavior that does not fit into the static restrictions. Such applications can be expressed using a dynamic dataflow subclass. In order to make the development process of such dynamic applications maximally efficient, this research work aims at providing a complete methodology for design space exploration, analysis and refactoring of dynamic dataflow applications. It relies on the modeling of a dynamic program and an architecture that enables the portability of a design to different parallel platforms. Furthermore, it allows a systematic exploration of different design alternatives so that multiple objective functions and/or design constraints can be satisfied. Finally, it identifies the refactoring directions efficiently guiding the designer during the entire development process.

1.1 Parallel systems development

Since its very first release in the 1970s, a single processor has been constantly sped-up by various means, such as an increase of the clock frequency, exploitation of the instruction-level parallelism [1] and also an increase of the cache and pipeline size. However, the process of accelerating a single core is constrained as a consequence of the clock frequency and the power dissipation limitations of deep sub-micron CMOS technology. Coming close to the practical limits of a single core started an alternative path of manufacturing processing platforms consisting of multiple cores. Such multi-core platforms, currently ubiquitous, have opened a new chapter in the field of efficient application design and brought both, interesting opportunities and significant challenges.

An obvious advantage of the emerging many- and multi-core platforms is the level of parallelism related to the number of available processing units. For instance, the *Epiphany* from Adapteva [2] consists of 64 cores and the MPPA-256 chip from Kalray [3] of 256 cores. Among the challenges related to the usage of many- and multi-core platforms, one problem can be referred to as the *memory wall*. It is related to the gap between the speed of processors and the speed of memory accesses enforcing a higher memory bandwidth along with an increase of the number of cores. Another problem is the interconnections between the processors which are constantly increasing in number. Furthermore, the more cores are used, the more it is questioned if they are used efficiently so that the power consumption is commensurate with the actual usage. Finally, with an increased number of cores, the synchronization of memory accesses is getting more and more challenging. Some of these issues have been tackled at the level of platform design. A good example is the creation of a hierarchical memory architecture providing different bandwidths and speeds of access for the cores depending on their relative location. This kind of architecture is currently used in several embedded platforms, for instance in the, mentioned earlier, Kalray's MPPA-256.

The emerging field of many- and multi-core platforms introduces a requirement of defining programming methods capable of handling a massively parallel execution while minimizing the additional programming effort of the designers. In the case of single core platforms, the focus of the application developers is on writing a correct program, whereas the compiler generates the code. Without appropriate support on many/multi-core platforms, the developers often have to perform additional work assisting the compilation process in order to make the generated code efficient. This work can include a decomposition of the application into tasks, deciding where and when these tasks should be executed or even hand-tuning the code in order to gain performance by exploiting some specific hardware features. Performing these tasks explicitly for a given platform decreases the potential reusability of the code, introduces some low-level issues, such as synchronization, data locality management and race conditions, and prevents porting the code to other platforms.

In this context, an arising question is if the challenges, such as decomposition of an application into parallel running tasks, design of the synchronization and communication mechanisms and exploitation of the available parallelism can be taken care of by the abstractions provided by the programming paradigm. Traditional programming languages ($i.e.$, C, C++, Java) do not reflect the inherently parallel nature of the applications and the underlying many/multi-core architectures. They have been designed for single-core systems with unified memories and rely on a sequential control flow, procedures and recursion lacking a high-level abstraction to capture and express the parallelism. In order to adapt them for many/multi-core architectures with distributed memories, several partial solutions have been invented. They are based on extensions added to the sequential languages [4] by means of macros, annotations and message passing libraries (*i.e.*, PVM [5], MPI [6], OpenMP [7]). However, common drawbacks of the approaches based on threads, that is, sequential processes sharing the memory, are: non-determinism, susceptibility to hidden bugs and error-prone modifications. Furthermore, using thread-based methods causes difficulties fully exploiting the application parallelism and maintaining portability, since such implementations do not scale automatically. Instead, any difference in the structure of the target platform requires modifications directly on the algorithmic side of the program [8]. The implementations built with threads are also not analyzable and consist of several non-analyzable components, such as pointers.

1.2 Motivations and problem statement

It can be stated that parallel implementations evolving from sequential techniques have an important disadvantage in the connection between the behavioral description of a program and the target architecture. Not only does it make a program difficult to maintain when the structure of the target platform changes (*i.e.*, additional cores are added), but also it becomes a critical problem when the considered platform is heterogeneous, that is, it consists of both, hardware and software components. A common practice is to make *a priori* an assignment of different parts of the design to different architectural components. Such an approach prevents an efficient exploration of the design alternatives and requires a complete rewriting of entire parts of the design if the assumed assignment does not satisfy the design constraints. Hence, the main requirements for a flexible design of parallel applications can be summarized as follows:

- **Design abstraction**: already at the early stages of development, the designer must decide about the level of abstraction which should be used. Due to the diverse nature of the platforms, different levels of abstractions are possible, depending on the required amount of details and the constraints. In any case, behavioral descriptions should be able to seamlessly express both: sequential and parallel computation paradigms;
- **Modularity**: if a design is modular, the functionality of the system is split into components that communicate with each other and hence divide the functionality of the overall application. The design abstraction should support modularity as a data and task parallelism;
- **Composability**: if a software system is composable, it is formed of several independent and recombinant components. These components can be assembled in various combinations to satisfy specific design requirements. The design abstraction used to describe

a system should operate on such composable components;

- **Reusability**: the design abstraction and the modularity of a program should enable reusing the components among different designs. In this way target-dependent abstractions are avoided and different systems can be described using a set of common components.

1.2.1 Design exploration

Given a semantically correct sequential program, the opportunities for its exploration are very limited, because the only possibility is to identify some independent portions of the code and change their order of execution so that the unnecessary memory copies are reduced. In contrast, in the case of parallel applications running on many/multi-core platforms, there are plenty of possible configurations to be applied, such as: decomposition of an application into tasks, assigning the tasks to the processing units (statically or dynamically) and defining their execution order. Each of these configurations can lead to different metrics on the performance, power consumption, resource utilization etc.

Design space exploration (*DSE*) can be described as a process of exploration and evaluation of different design alternatives, also referred to as design configurations or design points. These alternatives are applied as some settings to the design and are not related to the modifications of the algorithmic parts. The objective of the exploration is to find such a configuration (or a set of configurations) that satisfies the given constraints and optimizes the value(s) of the objective function(s).

An exploration of the design is especially important in the early stages of system development. Finding high-quality design alternatives has two important implications. First, it allows an evaluation of the design in terms of compliance with the specified constraints and objective functions. Second, if supported by an identification of some directions, it can point to the currently infeasible or not achievable design points which may become achievable, when the necessary modifications are applied to the design. Considering some large and complex designs in the context of massively parallel platforms, performing a manual exploration is inefficient and error-prone, due to a huge number of different alternatives. Hence, different state-of-the-art *DSE* methodologies make use of some common functionalities, such as:

- **Prototyping**: a design is validated and tested before the final implementation using a generated set of prototypes. In this case, the cost and the time required for the final implementation is reduced and the impact of the design decisions in the implementation process can be highlighted;

- **Optimization**: feasible design configurations are explored in order to satisfy the design constraints. If these are not satisfied, the design requires modification;

- **System integration**: it requires a working assembly and a configuration of the components. *DSE* should result in a set of feasible assembly configurations.

Performing these tasks requires providing a formal method supported with an appropriate computer-aided framework which can accomplish different stages of the exploration with regards to the specification requirements in a systematic way. Although many structured *DSE* methodologies exist, they are characterized by a common practice of designing application-specific architectures at a detailed level, which can limit the number of design points to be explored. In consequence, it limits the freedom of defining trade-offs between the performance, resource utilization and programmability. In contrast, providing a *general DSE* methodology should take into consideration the following components:

- **Application and architecture models**: both components should be represented according to some rules. First, the models should be formal, so that the analysis and exploration can be performed in an automated way. Second, in order to keep the methodology retargetable, the two models should be independent. Finally, they must allow to capture and express the necessary constraints and objective functions;
- **Exploration techniques**: since for the large design spaces a manual or random exploration is highly inefficient, the methodology should provide a set of automated techniques for discovering potential high-quality design points. Due to the large number of points, these techniques should allow navigation between different design points and narrow the space to the promising regions. It is also important to ensure that the exploration can be performed in a reasonable time;
- **Refactoring directions**: if the design points established during the exploration do not meet the requirements (*i.e.*, do not satisfy the constraints), the designer should be provided with a set of refactoring directions indicating possible improvements that can be applied to the design in order to resolve the factors leading to an unsatisfactory quality of the design. These factors are often referred to as *design* or *performance bottlenecks*. When identified during the exploration, they prevent the *DSE* methodology from being a *black box* to the designer. Instead, one is made aware of the narrowing factors occurring in the design and the opportunities for resolving them;
- **Performance estimation**: it is used to directly evaluate different design points without requiring execution of the program on a physical platform. In consequence, no partial implementations of the design are necessary in the process of exploration. It is important to ensure that the performance estimation is accurate enough to correctly evaluate all design alternatives.

1.2.2 Dataflow programming

The dataflow programming paradigm, which can be expressed by different models of computation ($MoCs$), is an alternative solution to programming methods evolving directly from sequential approaches, when program implementations on many- and multi-core systems, or, in particular, heterogeneous parallel platforms are considered. Dataflow programs, in general, are composed of, possibly hierarchical, networks of communicating computational kernels, called **actors**. Actors are connected by directed, lossless, order-preserving point-to-point communication channels, called **buffers**. Hence, the flow of data between the actors is explicit, because they are not allowed to exchange data differently than by exchanging atomic data packets, called **tokens**. The internal parallelism of an application, related to the actors, is directly exposed, since they are not allowed to share state. In consequence, the decisions about assigning dataflow actors to different software or hardware components can be freely made. The strengths of dataflow programs can be briefly summarized as: parallelism scalability, modularity, composability and portability.

Parallelism scalability

Scalability of a computer application has two common meanings. First, it is an ability to function well when its size or volume changes. This change can be related to both, the system itself and the context ($i.e.$, the platform the system is operating on). Second, it is not only the ability to function well in the rescaled situation, but eventually to take an advantage from it and improve the performance. Scalability of a dataflow program corresponds to these two meanings, because the explicit concurrency of actors leads to a parallel composition mechanism. For instance, when the parallelism of a target platform increases, a dataflow program can be always mapped to the available processing units yielding a correct behavior. Furthermore, its performance can improve along with the increase of the platform parallelism, up to the maximal potential parallelism expressed in a dataflow design.

Modularity

Dataflow actors are encapsulated, so that they do not share state or variables. In consequence, it is ensured that changing one actor does not impact others and a high potential parallelism of actors is provided. The functionality of a program can be separated into independent, interchangeable modules (actors), such that each of them contains everything necessary to process a given aspect of the desired functionality. Different modules are reusable between different designs.

Composability

Modularity of dataflow actors leads to defining programs as composable systems formed of independent components that can be freely assembled to provide a required functionality. Unlike for the case of thread-based implementations, when different parts of a program (components) are assembled, the occurrence of races is eliminated and no synchronization mechanisms need to be developed. Furthermore, different configurations can be applied to a program design, depending on the considered platform. For instance, dataflow actors can be assigned to the available processing units always yielding a correct behavior of the overall design, without introducing any unpredictable behavior.

Portability

Portability of dataflow programs implies that different components are not only reusable between different designs, but also between different target platforms. For instance, the same program network can be executed on platforms with different levels of parallelism ($i.e.$, different numbers of processing units), only by specifying the assignment of actors to the available processing units. Furthermore, having a single representation of a program it is possible to generate and reuse the code on different targets, including software (SW) and hardware (HW) elements.

Properties

Dataflow programs can be expressed using different classes bringing different complexity when it comes to their analysis. For the subclass of *static* dataflow programs, it is possible to perform the analysis at compile-time. This leads to establishing a static schedule, exact bounds on the buffer sizes, exact prediction of throughput and latency etc. This class can be, however, insufficient when complex designs, such as signal processing applications, need to be expressed. An extreme opposite is the subclass of *dynamic* dataflow programs. They allow changing rates of token production/consumption and a data-dependent behavior. This expressiveness and flexibility comes, however, at the cost of more difficult analyzability which is not possible at compile-time.

Dataflow $MoCs$ possess several valuable properties which perfectly respond to the described requirements for a flexible system design. Transmitting these attractive features into efficient implementations on the emerging many- and multi-core systems requires, however, dealing with several challenges. The *composability* property implies that the program components can work in several configurations. These configurations result in different qualities of the implementation in terms of, for instance, performance. The open decisions for the designer, when porting an abstract description of a dataflow program onto a target platform, include the following:

- **Partitioning**: it specifies the assignment of dataflow components to the processing units. Depending on the platform, different numbers of processing units can be available, but the number of dataflow components often exceeds this number. The other commonly used terms for *partitioning* are *binding* and *mapping* in the space domain;
- **Scheduling**: if multiple dataflow components are partitioned to one processing unit, they are, in general, not allowed to be executed in parallel. Hence, for each processing unit a specific order of execution must be established. The term *scheduling* is sometimes also referred to as *sequencing* and *mapping* in the temporal domain. Depending on the considered dataflow *MoC* and the internal nature of the components, there might exist a static execution order. Otherwise, the order must be established dynamically at run-time;
- **Buffer dimensioning**: although according to general specifications dataflow components communicate over infinite buffers, when executed on a real platform each buffer must be assigned a finite size. Depending on the considered *MoC*, these sizes might be necessary only to guarantee an execution without deadlocks or can influence the achievable throughput.

These design decisions can be made in various combinations corresponding to the design points in the *DSE* procedure. Each point can lead to different metrics related to data throughput, energy consumption, memory utilization, latency, and so on. These metrics can be directly taken as specifications of the constraints and/or objective functions to optimize [9].

1.2.3 Problem statement

Dataflow programs possess the features necessary for a flexible design of parallel applications running on various many/multi-core platforms. The performances of the implementations depend on several configurations, including partitioning, scheduling and buffer dimensioning, which result from the design space exploration process. Hence, the problem considered in this dissertation can be stated as follows.

Thesis: *Transferring the features of dataflow programs into efficient implementations satisfying the design constraints and optimizing the values of the objective functions is subject to locating high-quality configurations in the available design space and identifying refactoring directions revealing new promising regions in the space.*

To support this statement, this dissertation provides the following contributions:

1. A rigorous design space exploration problem formulation for heterogeneous platforms;
2. A set of heuristics making the exploration process effective and efficient;
3. A high-accuracy performance estimation tool driving the exploration process;
4. A methodology of analysis and optimization of dataflow programs according to various constraints and objective functions.

1.3 Research contributions

This dissertation addresses the problem of design space exploration of dynamic dataflow applications and provides a systematic analysis methodology consisting of multiple stages. It follows the discussed general DSE approach overcoming the architecture dependency of the commonly used approaches. Hence, it can be applied to different types of many-, multi-core and heterogeneous platforms without any intervention in the methodology and allows a flexibility of choice in terms of the objective function and/or design constraints. Furthermore, it considers the most expressive, but also the most difficult to analyze, dynamic dataflow programs with all their implications. Nevertheless, the less expressive dataflow $MoCs$ are still encompassed. The main contributions can be summarized as:

1. **Design space exploration problem formulation** [10]: the considered problem, consisting of partitioning, scheduling and buffer dimensioning for dynamic dataflow programs executed on homo- and heterogeneous platforms is thoroughly described and discussed at a level of detail not considered in the literature so far. The problem is formalized in terms of decision variables, objective functions and constraints. As the problem formulation is analyzed and exploited, the possible design optimization objectives are also identified;
2. **Variable Space Search methodology**: following the novel problem formulation, the concepts of design points and design spaces are introduced and expressed so that the multidimensionality of the problem is properly captured. Next, employing the concept of design space exploration, identification of program bottlenecks and determining the refactoring directions, a complete methodology of analysis and improvement of dataflow applications is defined. The methodology relies on the concept of Variable Space Search introduced originally for the graph coloring problem. It can be used in different scenarios, without limiting the choices of trade-offs between the performance and resource utilization;
3. **Definition of a dynamic dataflow program execution model for DSE** [11, 12, 13, 14]: using the available tools for collecting the profiling information of an execution, the

appropriate notion of time is retrieved and processed in order to be injected into an abstract model of a dynamic execution expressed as an execution trace. In this way, a real execution on a target platform is modeled. The timing information is kept separately from the abstract execution, hence a single model can be exploited for different types of architectures ensuring portability;

4. **Design space exploration heuristics**: various heuristic approaches corresponding to the introduced design space exploration problem formulation are proposed. Each of the heuristics relies on a generic model of dynamic execution expressed as a graph, which is being post-processed;

    - **Partitioning** [13, 15, 16, 17, 18]: the algorithms include greedy heuristics, descent local search methods and tabu search with different types of neighborhoods and advanced variants;
    - **Buffer dimensioning** [19]: the algorithms represent two approaches: bottom-up and top-down which can be applied in different optimization scenarios. Nevertheless, in both cases the main objective is to enable finding a trade-off between the performance and resource utilization;
    - **Scheduling** [20, 21]: several dynamic scheduling policies aiming at establishing the most efficient order of execution inside each processing unit are defined and analyzed with regards to performance potential and scheduling cost. A figure of merit for the cost of the scheduling policy is also introduced.

5. **Performance estimation** [12, 19, 22, 23]: a highly accurate performance estimation *SW* tool is provided. It enables the analysis of the design points on different types of platforms. The analysis includes the estimation of the execution time expressed in clock-cycles and extraction of metrics which are used by the design space exploration heuristics. Similar to the proposed *DSE* heuristics, it relies on a graph-based representation of a dynamic execution. The same representation, when supplied with appropriate timing information, is used for different types of platforms. An algorithm for performing an analysis of the bottlenecks of the program is also implemented on top of the performance estimation module.

1.4 Thesis organization

Chapter 2 is an overview of the main concepts related to dataflow programming. Different paradigms and classes including static, cyclo-static, dynamic extensions of static and, finally, dynamic programs are compared and discussed. Furthermore, an introduction to *CAL* language is provided along with a set of examples. Finally, the process of code generation and the $RVC-CAL$ compiler are explained. **Chapter 3** comprises the state-of-the-art of dataflow-oriented analysis frameworks. Different frameworks are compared and discussed with regards to the supported models of computation, objectives, features and available DSE heuristics. A special emphasis is placed on the frameworks intended for applications expressed using $RVC-CAL$. After presenting an overview of the related works, a dataflow design flow is introduced and discussed. **Chapter 4** describes the concept of an execution trace graph used as an abstract model of a dynamic execution. The formal definition and properties are described using simple examples. The emphasis is on the genericness of the model when referred to different configurations (design points) and the opportunities of employing the model in the design space exploration process. The challenges related to the modeling of the dynamic behavior of the actors are also discussed. **Chapter 5** focuses on the requirements related to obtaining accurate timing information for an execution trace graph. Consequently, the profiling methodologies to generate appropriate weights (related to processing, scheduling and communication) are discussed for two types of target platforms: Transport Triggered Architecture and Intel 86x64. Following different properties of these platforms, the profiling challenges are discussed in conjunction with the procedures necessary for each type of architecture. **Chapter 6** presents the design space exploration problem formulation regarding partitioning, scheduling and buffer dimensioning. The formulation is preceded by an overview of the formulations commonly used in the field of parallel programming and multi-core systems. The general formulation is then referred to two cases of target platforms: homogeneous and heterogeneous architectures which introduce more precise specifications and/or additional constraints. Towards the end of the Chapter, the problem instance sizes are also illustrated. **Chapter 7** handles the concept of design space exploration performed in different spaces, which was originally proposed for the graph coloring problem. The related work discusses also different possible formulations and objectives of exploration, as well as the importance of bottleneck identification. Then, the concepts of design points and design spaces are introduced using an appropriate notation and capturing the multidimensionality of the problem. An example demonstrates the complexity of the considered design spaces. Finally, the Variable Space Search algorithm is defined with regards to different possible optimization criteria. **Chapter 8** defines various heuristics to be used during the exploration in order to find high-quality solutions for each of the considered subproblems (partitioning, scheduling, buffer dimensioning). Regarding each subproblem, an overview of related work is provided and several heuristics of different complexity are introduced. For the case of scheduling it is discussed how this subproblem differs from the others and a figure of merit to express the cost of a scheduling policy is introduced. **Chapter 9** describes the software tool for performance estimation. The related work considers different general approaches to performance estimation and the achievable level of accuracy reported in the literature. Then, the construction of the tool is thoroughly described and a list of the tracked execution properties is provided. An algorithm for calculating the critical path and performing the impact analysis leading to the identification of bottlenecks, which is built on top of the tool, is also presented. **Chapter 10** reports the experimental results performed with regards to different components of this research work. They include: the verification of partitioning heuristics on Transport Triggered Architecture and Intel 86x64, the experiments with buffer dimensioning and scheduling, the analysis of accuracy for the performance estimation tool and the validation of the proposed Variable Space Search methodology using the recent dataflow implementation of the HEVC decoder. **Chapter 11** concludes the dissertation, summarizes the accomplished task and briefly discusses further improvements and open problems identified during the realization of this research work.

2 Dataflow programming

Dataflow programming was first introduced in 1974. In principle, it is a paradigm where the programs are expressed as directed graphs of streams and operators. Such programs are currently in use in multiple fields, such as: signal and video processing, telecommunications, health care, transportation, retail, science, security, emergency response and finance. Under a general term "dataflow programming", various models of computation ($MoCs$) have been developed independently by some research communities. The two most commonly used classes are: Synchronous Dataflow (SDF) and Cyclo-Static Dataflow ($CSDF$), introduced in 1987 and 1995, respectively. Since these classes do not allow expressing dynamic applications, several extensions capable of handling some dynamic behavior have been defined. Expressing fully dynamic applications is possible using Dynamic Dataflow (DDF). This Chapter is an overview of dataflow programming, including the definition of a dataflow program, different $MoCs$ and classes, which are examined in terms of their properties related to the expressiveness and analyzability. Next, the Cal Actor Language (CAL) is discussed, regarding the syntax, semantics and different $MoCs$ that it can represent.

2.1 Dataflow programs

A dataflow program is defined as a directed graph where the vertices are operators, called actors, and the edges are streams. In general, stream graphs might be cyclic, but some systems only support acyclic graphs. Dataflow programs implement streams as FIFO (first-in, first-out) queues, called buffers, with sometimes limited capacity. Conceptually, streams are infinite sequences of atomic data items, called tokens, and each actor consumes data items from incoming streams and produces data items on outgoing streams. A token is the atomic unit of communication in a dataflow program. One of the main properties of dataflow programs is their data-driven semantic, because it is the availability of tokens that enables an actor. One of the principal strengths of dataflow programs is that they do not impose unnecessary sequencing constraints between the actors, hence the implemented algorithms are not overspecified. Instead, only a partial order is specified and the sequencing constraints are imposed only by data dependence. Since the actors can run concurrently, dataflow programs inherently expose the application parallelism [24, 25].

An overview of different dataflow $MoCs$ includes: the Kahn process networks (KPN) [26] that represent the underpinning representation for dataflow graphs, the closely related to KPN Dataflow process networks (DPN) [27] and the Actor transition system [28] that extends DPN with the notion of atomic steps, priorities and actor internal variables.

2.1.1 Kahn process network

A KPN is a network of processes that can communicate only through unidirectional and unbounded buffers. Each buffer carries a possibly infinite sequence of tokens. Using the notation formalized in [27], each sequence of tokens is denoted as $X = [x_1, x_2, x_3, \ldots]$. A token is considered to be an atomic data object written (produced) and read (consumed) exactly once. The process of writing to the buffers is non-blocking, hence it always succeeds immediately. Reading from the buffers is blocking in the sense that if a process attempts to read a token from a buffer and the data is not available, it stalls (waits) until the buffer has sufficient tokens to satisfy the consumer. It is not possible to test the presence of input tokens in advance.

Kahn process

Let S_p denote a set of p-tuples of sequences as in $X = \left\{X_1, X_2, \ldots, X_p\right\} \in S^p$. A Kahn process is then defined as a mapping from a set of input sequences to a set of output sequences such as:

$$F : S^p \rightarrow S^q \tag{2.1}$$

The KPN process F has an event semantic instead of state semantics as in some other domains such as continuous time. Moreover, the only technical restriction is that F must be a continuous mapping function.

Monotonicity and continuity

Considering a prefix ordering of sequences, the sequence X precedes the sequence Y (written $X \sqsubseteq Y$) if X is a prefix of (is equal to) Y . For example, if $X = [x_1, x_2]$ and $Y = [y_1, y_2, y_3]$ then $X \sqsubseteq Y$ and it is common to say that X approximates Y , since it provides partial information about Y. An empty sequence, denoted as $\perp$ is a prefix of any other sequence. An increasing chain (possibly infinite) of sequences is defined as $\chi = \{X_0, X_1, \ldots\}$ where $X_1 \sqsubseteq X_2 \sqsubseteq \ldots$. Such an increasing chain of sequences has one or more upper bounds Y, where $X_i \sqsubseteq Y$ for all $X_i \in \chi$.

The least upper bound (LUB) $\sqcup\chi$ is an upper bound such that for any other upper bound Y, $\sqcup\chi \sqsubseteq Y$. The LUB may be an infinite sequence.

Given a functional process F and an increasing chain of sets of sequences χ, as defined in Equation 2.1, F maps χ into another set of sequences that may or may not be an increasing chain. Let $\sqcup\chi$ denote the LUB of the increasing chain χ. Then F is said to be Scott-continuous [29] if for all such chains χ, $\sqcup F(\chi)$ exists and:

$$F(\sqcup\chi) = \sqcup F(\chi) \tag{2.2}$$

Networks of Scott-continuous processes have a more intuitive property called monotonicity. This property can be thought of as a form of causality that does not invoke time, so that *future input* concerns only *future output*. A process F is said to be monotonic if:

$$X \sqsubseteq Y \Rightarrow F(X) \sqsubseteq F(Y) \tag{2.3}$$

A continuous process is monotonic. However, a monotonic process may be noncontinuous A key consequence of this property is that a process can be computed iteratively [30]. This means that given a prefix of the final input sequences, it may be possible to compute a part of the output sequences. In other words, a monotonic process is non-strict: its inputs need not be complete before it can begin computation. In addition, a continuous process will not wait forever before producing an output (it will not wait for the completion of an infinite input sequence). Networks of monotonic processes are determinate.

2.1.2 Dataflow process network

Dataflow process networks ($DPNs$) are formally a special case of $KPNs$, where the computational blocks are called actors. Analogous to a KPN process, actors can communicate only through unidirectional and unbounded buffers which can carry possibly infinite sequences of tokens and writing to the buffers is non-blocking. In contrast, reading from buffers is non-blocking in the sense that an actor can first test for the presence of input tokens. If there are not enough input tokens, then the read returns immediately and the actor does not need to be stalled. This property introduces non-determinism, without forcing the actors to be non-deterministic.

Actor with firings

DPN networks are a special case of KPN networks where each process consists of repeated firings of an actor [31]. An actor firing can be defined as an indivisible (atomic) quantum of computation. The firings can be described as functions, and their invocation is controlled by a set of firing rules. Sequences of firings define a continuous Kahn process as the least-fixed-point of an appropriately constructed functional mapping, hence DPN can be formally established as a special case of KPN [32].

An actor with m inputs and n outputs is defined as a tuple (f, R), where:

- $f : S^m \rightarrow S^n$ is a function called the firing function;
- $R \subseteq S^m$ is a set of finite sequences called the firing rules;
- $f(r_i)$ is finite for all $r_i \in R$.
- no two distinct $r_i r_j \in R$ are joinable, in the sense that they do not have a LUB.

The Kahn process F, as defined previously, based on the actor (f, R) has to be interpreted as the least-fixed-point function of the functional $\phi : (S^m \rightarrow S^n) \rightarrow (S^n \rightarrow S^m)$ defined such as:

$$(\phi(F))(s) = \begin{cases} f(r) \oplus F(s') & \text{if there exist } s \in R \text{ such that } s = r \oplus s' \text{ and } s \sqsubseteq s' \\ \bot & \text{otherwise} \end{cases} \tag{2.4}$$

where $\oplus$ represents the concatenation operator and $(S^m \rightarrow S^n)$ is the set of functional mappings of S^m to S^n. It is possible to demonstrate that ϕ is, both, a continuous and monotonic function. In contrast, the firing function f does not need to be continuous, or even monotonic. It merely needs to be a function, and its value must be finite for each of the firing rules [32].

2.1.3 Actor transition systems

The Actor transition system (ATS) [28] describes actors in terms of labeled transition systems (LTS). The ATS extends the notion of an actor with firings by introducing the notions of atomic step, internal state, and priority. In an ATS, a step makes a transition from one state to another. An actor maintains and updates its internal variables: these are not sequences of tokens, but simple internal values that cannot be shared among actors. Hence, the state of an actor depends on the value (state) of its internal variables, and not just on the sequence of tokens it has received. Moreover, the notion of priority allows actors to ascertain and react to the absence of tokens. Hence, actors become more versatile and appropriate to express DDF programs. On the other hand, they become harder to analyze as undesired non-determinism can be introduced to a dataflow application.

Let Σ denote a non-empty actor state space, u the space of tokens that can be exchanged between actors and U^n a finite and partially-ordered sequence of n tokens over u. An n-to-m **actor** is an LTS $(\sigma_0, \tau, \succ)$ where:

- $\sigma_0 \in \Sigma$ is the actor initial state;
- $\tau \subset \Sigma \times U^n \times U^m \times \Sigma$ defines the transition relation;
- $\succ \subset \tau \times \tau$ defines a strict partial order over τ.

Any $(\sigma, s, s', \sigma') \in \tau$ is called a transition, where $\sigma \in \Sigma$ is its source state, $s \in S^n$ its input tuple, $\sigma' \in \Sigma$ its destination state and $s' \in U^m$ its output tuple. It must be noted that $\succ$ is a non-reflexive, anti-symmetric, transitive and partial-order relation on τ, also called its priority relation. An equivalent and more compact notation for the transition (σ, s, s', σ') is $\sigma \xrightarrow{s \rightarrow s'} \sigma'$. As for any *LTS*, in the *ATS* each transition can be labeled and referred to as an **action** λ such as:

$$\lambda : \sigma \xrightarrow{s \rightarrow s'} \sigma' \tag{2.5}$$

In summary, a step makes a transition from one state to another, each transition can be labeled as an action and the execution of a step is defined as a firing, in which tokens may be consumed and produced, and the internal variables may be updated.

2.1.4 Dataflow programs comparison

The most important properties of different dataflow programs discussed in the previous Sections are summarized in Table 2.1. In general, a transition from *KPN*, through *DPN*, up to *ATS* can be identified. *DPN* is a special case of *KPN*, where the presence of input tokens can be tested and considered when invoking a firing function. Furthermore, the program execution is described as a set of repetitive firings of actors. *ATS* goes further by introducing the notion of an atomic step and making its execution dependent also on the priorities and values of internal variables.

Property	KPN	DPN	ATS
reading from input FIFOs	blocking	non-blocking	non-blocking
writing to output FIFOs	non-blocking	non-blocking	non-blocking
computational blocks	processes	actors	actors
priorities	no	no	yes
internal state variables	no	no	yes
program execution	input/output sequence mapping	repetitive actor firings	atomic steps (firings)
firing function invocation control	-	input tokens	input tokens, state variables, priorities

Table 2.1 – *KPN*, *DPN* and *ATS* programs: comparison.

2.1.5 Dataflow concurrency

The emergence of massively parallel architectures, along with the difficulties to program these architectures, makes the dataflow paradigm a more appealing alternative to an imperative paradigm [33, 34, 35, 36, 37, 38, 39]. The main advantages of this paradigm are related to the ability of expressing concurrency without complex synchronization mechanisms. This is made possible by the internal representation of the program as a network of processing blocks that only communicate through communication channels. In fact, blocks are independent and do not produce any side-effects. This removes the potential concurrency issues that could arise when the programmer is asked to manually manage the synchronization between parallel computations [40, 8]. Moreover, this paradigm explicitly exposes all the natural parallelism of a program [33, 40].

2.2 Dataflow classes

Since the representation of a dataflow program does not over-constrain the order of operations, a scheduler of the program has the freedom it needs to adequately exploit the available parallelism in order to maximize the re-use or simply reduce the limited hardware resources available on the target platform. Figure 2.1 illustrates some of the dataflow *MoCs* classes. The respective actor behavior that can be represented for each of them is discussed in this Section.

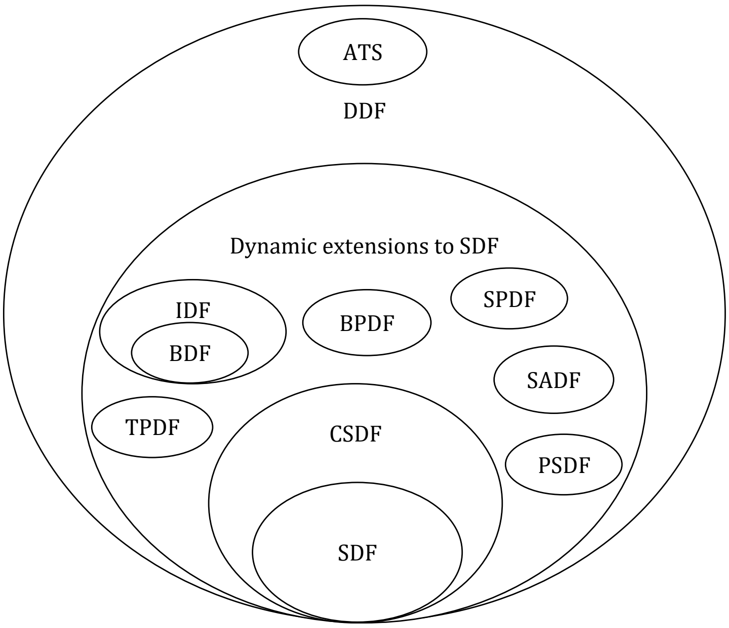


Figure 2.1 – Dataflow *MoCs* classes.

2.2.1 Static dataflow programs

Static dataflow (SDF), sometimes also called *synchronous dataflow*, is a special class of dataflow $MoCs$ where the number of tokens consumed and produced by each actor is fixed and known at compile-time. Repeated firings of the same actor respects the same behavior. This is the least expressive class of dataflow programs, but it is also the one that can be analyzed in the easiest way. In fact, its main advantage is its total predictability at compile-time, with respect to scheduling, memory consumption, and execution termination.

Static scheduling

In order to build a static schedule, the compiler should construct a single cycle of a periodic schedule. The first step is then to evaluate how many invocations of each actor should be included in each cycle. This can be established easily using the number of produced and consumed tokens for each actor firing. As depicted in Figure 2.2, the number of tokens consumed at each firing by the $i-th$ actor from the $n-th$ buffer is denoted by $c_{i,n} \in \mathbb{N}$, the number of tokens produced at each firing by the $i-th$ actor on the $n-th$ buffer is denoted by $p_{i,n} \in \mathbb{N}$, and the number of times the $i-th$ actor is invoked ($i.e.$, repeated) in each cycle of the iterated schedule is denoted by $r_i \in \mathbb{N}$. Hence, in order to have a feasible periodic schedule, it must be ensured that for each $n-th$ buffer of the dataflow graph the following condition is satisfied:

$$p_{i,n}\, r_i = c_{j,n}\, r_j \tag{2.6}$$

In other words, this equation ensures that in each cycle of the iterated schedule, the number of tokens produced on each buffer is equal to the number of tokens consumed on that buffer. Indeed, the first step to finding a schedule for an SDF graph is to solve a set of Equations (2.6) for the unknown r_i. Since for SDF programs the number of consumed and produced tokens for each actor firing is fixed and known at compile-time, the set of equations can be concisely written by constructing a topological matrix Γ. The entry $[\Gamma]_{i,n}$ contains the integer $p_{i,n}$ when

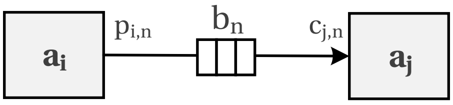


Figure 2.2 – A dataflow graph with two actors, a_i and a_j, connected through the buffer b_n. $p_{i,n}$ defines the number of tokens produced on b_n during each firing of a_i. $c_{j,n}$ defines the number of tokens consumed from b_n during each firing of a_j.

the $i-th$ actor produces $p_{i,n}$ tokens on the $n-th$ buffer, and the integer $c_{i,n}$ when the $i-th$ actor consumes $c_{i,n}$ tokens from the $n-th$ buffer. In general, this matrix does not need to be square. For example, a dataflow graph shown in Figure 2.3 has the following topological matrix:

$$\Gamma = \begin{bmatrix} p_{A,1} & -c_{B,1} & 0 & 0 & 0 \\ p_{A,4} & 0 & 0 & -c_{D,4} & 0 \\ 0 & p_{B,2} & -c_{C,2} & 0 & 0 \\ 0 & 0 & p_{C,3} & 0 & -c_{E,3} \\ 0 & 0 & 0 & p_{D,5} & -c_{E,5} \end{bmatrix} \tag{2.7}$$

The system of equations to be solved can be formulated such as:

$$\Gamma \overrightarrow{r} = \overrightarrow{0} \tag{2.8}$$

where $\overrightarrow{r}$ is the repetition vector containing the r_i value for each $i-th$ actor, and $\overrightarrow{0}$ is a zero-vector. Equation (2.8) is usually referred to as the balance equation of a dataflow program. A special case is when an actor has a connection to itself ($i.e.$, a self-loop). In this situation only one entry in Γ describes this buffer. This entry gives the net difference between the amount of tokens produced on this buffer and the amount of tokens consumed from this buffer each time the actor is executed. For a correctly constructed graph this difference needs to be zero. Hence, the entry describing a self-loop should be zero [41].

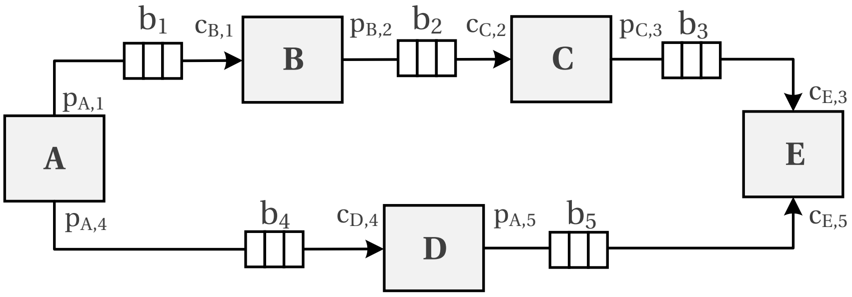


Figure 2.3 – Example of a dataflow graph.

Existence of an admissible schedule

An admissible sequential schedule ϕ_s is defined as a non-empty ordered list of actors such that if the actors are executed in the sequence given by ϕ_s, then the number of tokens stored in each buffer will remain non-negative and bounded. Each actor must appear in ϕ_s at least once.

A periodic admissible sequential schedule ($PASS$) is infinite. In [41] it has been demonstrated that, for any connected SDF graph, a necessary condition to be able to construct a $PASS$ is that the rank of Γ should be:

$$rank(\Gamma) = s - 1 \tag{2.9}$$

where s is the number of actors in the graph. In other words, the null space of Γ should have a dimension of 1. Furthermore, it is shown in [41] that when the rank is correct, a repetition vector $\overrightarrow{r}$ that contains only integers and relies on this null space always exists. This vector defines how many times each actor should be invoked in one period of a $PASS$. In other words, the rank of the topology matrix indicates a sample rate consistent with the graph. SDF graphs that have a topology matrix such that $rank(\Gamma) = s$ are said to be defective: any schedule for this graph will result either in a deadlock or unbounded buffer size configuration.

The use of a $PASS$ scheduler requires using a single processing unit implementation: this does not exploit the parallelism advantages of a dataflow application. Clearly, if a feasible schedule for a single processing unit can be generated, then a feasible schedule for a multiprocessor system can be also generated. In that case the objective is to find a periodic admissible parallel schedule ($PAPS$) defined as a set of lists $\Psi = \{\psi_i\ ,\ i = 1, \ldots, M\}$ where M is the number of processing units, and ψ_i specifies a periodic schedule for the $i - th$ processing unit. If a single processing unit is targeted, some reasonable scheduling objectives might include minimization of data or program memory requirements. For the case of multiprocessor targets, the common objectives are the maximization of the throughput or the minimization of the flow-time [41, 42, 43].

2.2.2 Cyclo-static dataflow programs

Cyclo-Static Dataflow ($CSDF$) generalizes the $SDFMoC$ by defining the firing rules which get changed cyclically. It must be noted that $CSDF$ extends SDF with the notion of state, while maintaining the same compile-time properties concerning scheduling and memory consumption. $CSDF$ programs allow the number of tokens consumed and produced by an actor to vary from one firing to the next according to a cyclic pattern. Unlike the scalar consumption and production parameters for SDF, in $CSDF$ programs $c_{i,n}$ and $p_{i,n}$ are integer vectors both defined as $\overrightarrow{\gamma}_{i,n}$. Since these patterns are periodic and predictable, it is still possible to statically construct periodic schedules using techniques based on those developed for SDF. The state can be represented as an additional argument to the firing rules and firing functions, hence, it is modeled as a self-loop [44, 45].

Static scheduling

The topological matrix entries are defined such as:

$$[\Gamma]_{i,j} = t_{i,j}\,\frac{\sigma_{i,j}}{d_{i,j}} \tag{2.10}$$

where $d_{i,j} = dim(\overrightarrow{\gamma_{i,j}})$ is the length or period of the token production/consumption pattern for the $i-th$ buffer connected to the $j-th$ actor. If there is no connection, then $d_{i,j} = 1$. The $j-th$ actor fires in a cycle with period $t_j = lcm(d_{i,j}, \forall i)$, which is the least common multiple of the consumption and production periods for all the buffers connected to that actor. Finally, $\sigma_{i,j}$ is the sum of the elements in $\overrightarrow{\gamma}_{i,j}$. As for the case of SDF, it is also possible for the $CSDF$ programs to solve the balance equation (2.8) and verify the existence of an admissible schedule. However, in $CSDF$ programs the repetition vector $\overrightarrow{r}$ does not represent the number of actor firings, but the number of cycles. In this case, the number of firings of each $i-th$ actor is defined as $r_i\ t_i$.

2.2.3 Dynamic extensions to static dataflow programs

In order to extend the expressiveness of the static dataflow $MoCs$, several extensions capable of handling some dynamic behavior have been introduced. They can be classified in two categories: the ones that allow the graph to change the topology at run-time and the ones that allow the amount of data exchanged between actors to change at run-time. The $MoCs$ from the first group (such as Boolean Dataflow and Integer Dataflow) introduce specialized actors that can change the topology of the graph at run-time using some parameters. The second group relies on the usage of parameters to control the amount of data communicated between the actors. This Section presents some of the models from both groups.

Boolean Dataflow and Integer Dataflow

Boolean Dataflow (BDF) belongs to the models focusing on altering the graph topology at run-time. It was originally introduced in [46] as an extension of SDF adding an "if-then-else" functionality. This functionality is provided by two special actors: *switch* and *select.* The first one has a single data input and two data outputs. It receives boolean input tokens at a boolean control input that enables the selection of an output. In the same way, the *select* actor consisting of two inputs and one output makes the choice of an input. A BDF graph is analyzed just like an SDF graph, except for the *switch* and *select* actors. Analyzing these actors requires calculating the rates related to the proportion of true tokens on their input boolean streams. Integer Dataflow (IDF) is an extension to BDF proposed in [47]. It replaces the boolean streams with integer streams so that the ports can be selected over many, not just two ports. Both models slightly increase the expressiveness of *SDF*, but do not allow making changes in the production/consumption rates.

Parametrized Synchronous Dataflow

Parametrized Synchronous Dataflow (*PSDF*) [48] allows arbitrary attributes of a dataflow graph to be parametrized. Each parameter is associated with a set of admissible values to be taken at any given time. The attributes can be scalar or vector attributes of individual actors, edges or graphs. This dataflow representation consists of three cooperating dataflow graphs referred to as the *body* graph, the *subinit* graph and the *init* graph. The body graph typically represents the functional core of the implemented algorithm, whereas the sub-init and init graphs are dedicated to managing its parameters. Changes to the body graph parameters occurring according to the parameters computed by the init and sub-init graphs cannot occur at arbitrary points in time. Instead, the body graph executes uninterrupted through an iteration, where the notion of iteration can be specified by the user. A combination of cooperating body, init, and subinit graphs is referred to as a *PSDF* specification. These specifications can be abstracted as *PSDF* actors in higher level *PSDF* graphs, hence they can be hierarchically integrated. *PSDF* does not allow changes in the topology of the graph.

Scenario-Aware Dataflow

Scenario-Aware Dataflow (*SADF*) is a modification to the original *SDF* model by means of system scenarios [49]. It introduces a special type of actor, called a detector, and enables using parameters as port rates. The role of the detectors is to detect the current scenario the application operates on and apply a change to the port rates accordingly. Detectors are assigned to non-overlapping sets of actors, so that each actor is controlled by exactly one detector using a control link. When an actor fires, it first reads a token from the control link that configures the values of its parameters, and then waits until it has sufficient tokens on its input edges. The set of possible scenarios is finite and known at compile-time. A scenario is defined by a set of values, one for each parametrized rate. Since all scenarios are known at compile-time, *SADF* is analyzed by considering all possible *SDF* graphs that result from each scenario. *SADF* resembles *CSDF* in the sense that it uses a fixed set of possible rates on each port. The difference is that it does not impose any ordering at compile-time. Unlike other models using the parametric rates, *SADF* does not require a parametric analysis as all configurations can be analyzed separately as *SDF* at compile-time. However, this approach can become expensive, when the number of scenarios is large. Hence, it remains reasonable when the number of scenarios is limited and manageable by a human. The dynamic changes can take place only in between the iterations, but include both dynamic rates and dynamic topology changes.

Schedulable Parametric Dataflow

Schedulable Parametric Dataflow ($SPDF$) is a MoC enabling dynamic changes of the rates of an actor within an iteration of the graph [50]. It uses symbolic rates which can be the products of positive integers or symbolic variables (parameters). The variable values are set by the special actors of the graph, called *modifiers*. Actors that have parameters on their port rates or at their solutions are called *users* of a parameter. The parameter values are produced by the modifiers and propagated towards all the users through an auxiliary network. Modifiers and users have their respective writing and reading periods, indicating the number of times an actor should fire before producing/consuming a new value for a parametric rate. The writing periods are annotated for each modifier and the reading periods are calculated by analyzing the graph. Some writing periods, $i.e.$, the ones causing inconsistency, are not allowed. Comparing to other parametric models, $SPDF$ provides the maximum flexibility in terms of changing of the parameter values. However, the increased expressiveness makes the scheduling problem very challenging, because the data dependencies are parametric and can change at any time during the execution, unlike for other parametric models, where a schedule can be established at the beginning of an iteration. Changes of the topology of the graph are not allowed.

Boolean Parametric Dataflow

Boolean Parametric Dataflow ($BPDF$) [51] is a model combining integer and boolean parameters. It allows expressing dynamic rates and the activation/deactivation of communication channels. Similarly to other parametric models, the input/output ports are labeled with consumption/production rates that can be parametric. Integer parameters can change at runtime between two iterations. Moreover, the edges can be annotated with boolean parameters allowed to change also within an iteration. Hence, both types of changes: production/consumption rates and graph topology are allowed. $BPDF$ is mostly considered for executions on $STHORM$ many-core chip from $STMicroelectronics$ [52], for which a scheduling algorithm for $BPDF$ graphs exists [53].

Transaction Parametrized Dataflow

Transaction Parametrized Dataflow ($TPDF$) [54] is a recently defined MoC extending $CSDF$ with parametric rates and a new type of control actor, channel and port. Hence, it aims at enabling dynamic changes of the graph topology and time constraints semantics. It has been designed to be statically analyzable ($i.e.$, in terms of deadlock and boundedness properties), while avoiding the restrictions of decidable dataflow models mentioned earlier. The dynamic behaviors can be viewed as a collection of different behaviors, called cases, occurring in certain unknown patterns. Each case is considered to be static by itself and predictable in performance. $TPDF$ can be considered similar to $BPDF$ with an extension to impose real-time constraints.

2.2.4 Dynamic dataflow programs

The $MoCs$ discussed in the previous Sections can be considered adequate for representing parts of many algorithms. However, they are rarely sufficient for expressing entire complex programs requiring consideration of data-dependent iterations, conditionals and recursion. For example, a functionality that contains conditional execution of dataflow subsystems or actors with dynamically-varying production and consumption rates cannot be expressed in decidable dataflow models [55, 56]. The dynamic dataflow (DDF) MoC defines actors with a number of produced and consumed tokens that is not statically specified. In a DDF program, an actor may have both firing rules and firing functions that are data-dependent. In other words, the token production and consumption rates can vary according to the program input sequence.

The increased modeling flexibility and expressiveness power make DDF programs much harder to analyze. Due to their Turing-complete nature, many analysis problems may become undecidable [55]. For example, DDF analysis techniques may succeed in guaranteeing a bounded buffer size execution and deadlock avoidance only for a significant subset of specifications ($e.g.$, input streams in signal processing systems) [57, 58, 59]. Similarly, DDF scheduling is generally a run-time operation. However, some or all of the scheduling decisions can be predicted at compile-time by either describing the program with a more restricted programming model or by analyzing the program to find if any parts of it can be described in a more restricted way [60, 61, 62, 63].

2.3 CAL Actor Language

The Cal Actor Language (CAL) [64] is a language that provides useful abstractions for dataflow programming based on actors. CAL directly captures the features of ATS actors adding the notion of atomic action firings, also called *steps*. Figure 2.4 illustrates the basic concepts of a CAL program. This is a dataflow network composed of a set of *actors* and a set of first-in first-out ($FIFO$) *buffers*. Each CAL actor is then defined by a set of *input ports*, a set of *output ports*, a set of *actions*, and a set of *internal variables*. The language also includes the possibility of defining an explicit *finite state machine* (FSM). The FSM captures the actor state behavior and drives the action selection according to its particular state, to the availability of input tokens and to the value of the tokens evaluated by other language operators called *guards*. Each action may capture only a part of the firing rule of the actor together with the part of the firing function that pertains to the input/state combinations enabled by that partial rule defined by the *FSM*. An action is enabled according to its input patterns and guards expressions. Input patterns are defined by the amount of data that are required in the input sequences, whereas guards are boolean expressions on the current state and/or on input sequences that need to be satisfied for enabling the execution of an action.

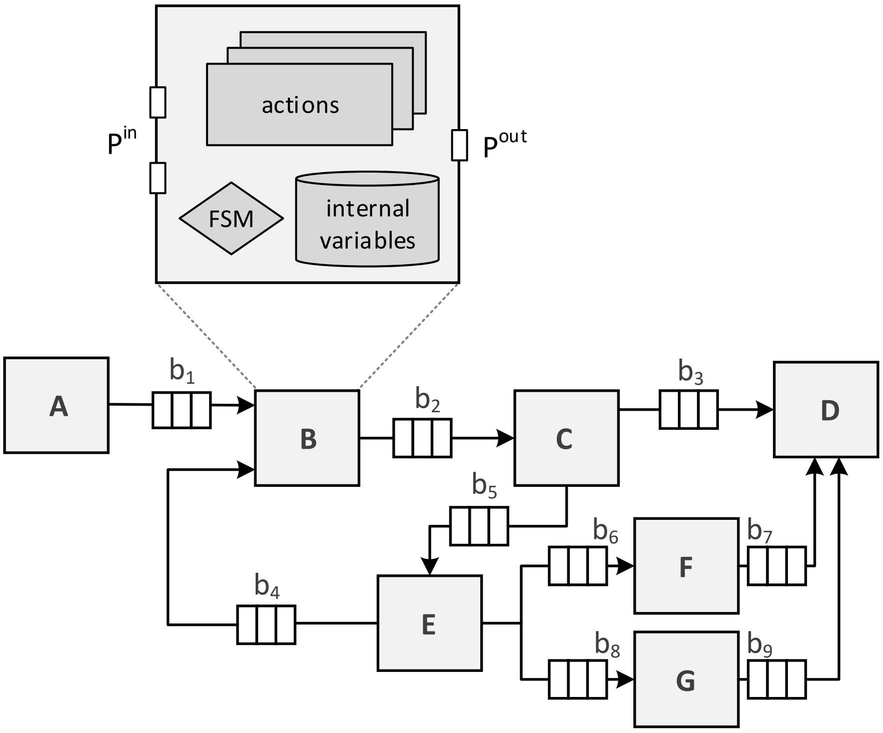


Figure 2.4 – *CAL* network and actors structure.

2.3.1 CAL program

A *CAL* program network N is defined as a tuple (K, A, B) where:

- $K = \{\kappa_1, \kappa_2, \dots \kappa_{n_\kappa}\}$ is a finite set of actor-classes;
- $A = \{a_1, a_2, \dots, a_{n_A}\}$ is a finite set of actors;
- $B = \{b_1, b_2, \dots, b_{n_B}\}$ is a finite set of buffers.

A *CAL* actor-class κ defines the program code template and the implementation behaviors of the actors (*i.e.*, the *CAL* source code). Different actors can be the instances of the same class, however each actor corresponds to a different object with its own internal states that cannot be shared.

A CAL actor a is defined as a tuple $(\kappa, P^{in}, P^{out}, \Lambda, \mathcal{V}, \text{FSM})$ where:

- κ is the actor-class;
- $P^{in} = \{p_1^{in}, p_2^{in}, \ldots, p_{n_I}^{in}\}$ is the finite set of input ports;
- $P^{out} = \{p_1^{out}, p_2^{out}, \ldots, p_{n_O}^{out}\}$ is the finite set of output ports;
- $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_{n_\Lambda}\}$ is the finite set of actions;
- $\mathcal{V} = \{v_1, v_2, \ldots, v_{n_V}\}$ is the finite set of internal variables;
- FSM is the internal finite state machine.

A CAL buffer b is defined as a tuple (a_s, p_s, a_t, p_t) where:

- $a_s \in A$ is the source actor ($i.e.$ the one that produces the tokens);
- $p_s \in P_{a_s}^{out}$ is the output port of the source actor;
- $a_t \in A$ is the target actor ($i.e.$ the one that consumes the tokens from the buffer);
- $p_t \in P_{a_t}^{in}$ is the input port of the target actor.

It is important to note that each input port can be connected at most to one buffer. On the other hand, multiple buffers can be connected to one output port. In order to execute an action, the following stages (summarized in Figure 2.5) are performed serially:

- **Wait for tokens** Q_{br}: the firing is waiting until all the required input tokens are available from the corresponding buffers;
- **Consume input tokens** Q_r: the firing is consuming the input tokens;
- **Action execution** Q_e: the firing performs the execution of its algorithmic part;
- **Wait for space** Q_{bw}: the firing is waiting until all the required output tokens can be accommodated in the corresponding buffers;
- **Write output tokens** Q_w: the firing is producing the output tokens.

The transition conditions are the following:

- **hasTokens**: the number of required input tokens is available in each corresponding input buffer;

- **hasSpace**: the number of output token space that is available in each corresponding output buffer.

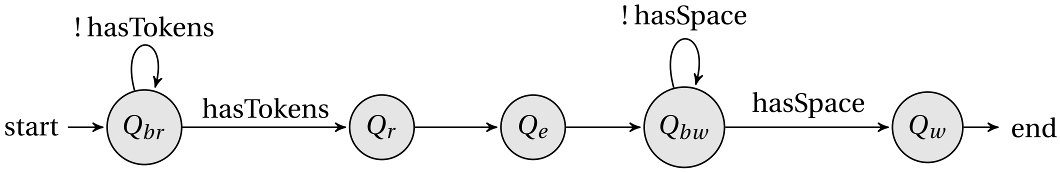


Figure 2.5 – Action execution model .

2.3.2 CAL syntax

In this Section, the syntax and the semantics of *CAL* are illustrated through simple examples. For more details the reader is referred to [64].

Lexical tokens

Lexical tokens are intended to make the user understand the functionality provided by any programming language. A lexical token is a string of indivisible characters known as *lexemes*. The *CAL* lexical tokens, summarized in Table 2.2, can be described as follows:

- **Keywords** are a special type of identifier, which is already reserved in a programming language by default. Hence, these keywords can never be used as identifiers in the code. Some of these keywords are `action`, `actor`, `begin`, `else`, `if`, `while`, `true` and `false`.

- **Operators** usually represent mathematical, logical or algebraic operations. Operators are written as strings of characters such as `!`, `%`, `^`, `&`, `*`, /, +, −, =, <, >, ?, `~` and `|`.

- **Delimiters** are used to indicate the start or the end of a syntactical element in the *CAL* code. The following elements are used as delimiters: `(`, `)`, `[`, `]`, `{` and `}`.

- **Comments** in *CAL* are the same as in languages like Java and C/C++. Single-line comments start with `//` and multiple-line comments start with `/*` and end with `*/`.

Keywords	`action`, `actor`, `procedure`, `function`, `begin`, `if`, `else`, `end`, `foreach`, `while`, `do`, `procedure`, `in`, `list`, `int`, `uint`, `float`, `bool`, `true`, `false`
Operators	!, %, ^, &, ⋆, /, +, −, =, <, >, ?, ˜, |
Delimiters	(,), [,], {, }, ==>, −>, :=
Comments	//, /⋆ ... ⋆/

Table 2.2 – CAL lexical tokens.

Actions, input patterns and output patterns

A very simple actor that can be described using *CAL* is the `Multiplier` actor defined in Listing 2.1. This actor consumes a token from its input port and produces a token to its output port. The actor's header is defined in line 1. The header contains the following information: (1) the actor name; (2) a list of parameters contained inside the `()` construct (empty, in this case); (3) the declaration of the input and output ports. The input ports are those in front of the ==> and the output ports are those after it. In this case, the input and output port sets are defined as $P^{in}_{\texttt{Multiplier}} = \{\texttt{I}\}$ and $P^{out}_{\texttt{Multiplier}} = \{\texttt{O}\}$, respectively. For each parameter and port, the data type is specified before the name (in this case all defined with an `int` data type). In Listing 2.1, the actor contains only one *action*, labeled as `multiply` as defined in line 3. In this case, the action set is defined as $\lambda_{\texttt{Multiplier}} = \{\texttt{multiply}\}$. Action `multiply` demonstrates how to specify token consumption and production. The part in front of the ==> (which defines the **input patterns**) specifies how many tokens are to be consumed, from which ports, and how these tokens are called in the rest of the action. In this case, there is one input pattern: `I:[val]`. This pattern indicates that one token is to be read (*i.e.*, consumed) from the input port `I`, and that this token is to be called `val` in the rest of the action. Such an input pattern also defines a condition that must be satisfied for this action to fire: if the required token is not present, this action will not be executed. Therefore, input patterns are responsible for the following:

- They define the number of tokens (for each port) that will be consumed when the action is executed (fired);
- They declare the variable symbols that are used within the action to refer to the tokens consumed by an action firing;
- They define a firing condition for the action, *i.e.*, a condition that must be satisfied for the action to be able to fire.

The **output patterns** of an action are the ones defined after the ==> construct. They define the number and values of the output tokens that will be produced on each output port by each firing of the action. In this case, the output pattern `O:[2 * val]` says that exactly one token will be generated at output port `O` and its value is `2 * val`. It is worth emphasizing that although syntactically the use of `val` in the input pattern `I:[a]` looks the same as the one in the output expression `O:[2 * val]`, their meanings are very different. In the input pattern the name `val` is declared: in other words, it is introduced as the name of the token that is consumed whenever the action is fired. By contrast, the occurrence of `val` in the output expression uses that name.

Listing 2.1 – Multiplier.cal

```
actor Multiplier() int I ==> int O :

    multiply: action I:[val] ==> O:[2 * val] end

end
```

Guards

So far, the only firing condition considered for the actions was the presence of a sufficient number of tokens to consume, according to their input patterns. However, in many cases, it is possible to specify additional criteria that need to be satisfied for an action to be fired. These are, for instance, conditions that depend on the values of the tokens, the actor internal variables, or both. These conditions can be specified using *guards*, as for example in the `Separator` actor, defined in Listing 2.2. This actor defines one input port `I`, two output ports `O1` and `O2`, and two actions `A` and `B`. These actions require the availability of one token in `I`, however their selection depends on the value of the input token `val` read from `I`, as defined in lines 4 and 7, respectively. In this example, if `val >= 0`, then action `A` is selected, otherwise action `B` is selected.

Listing 2.2 – Separator.cal

```
1  actor Separator() int I ==> int O1, int O2 :
2
3      A: action I:[val] ==> O1:[val]
4      guard val >= 0 end
5
6      B: action I:[val] ==> O2:[val]
7      guard val <  0 end
8
9  end
```

Actor parameters and internal variables

Using *CAL*, it is possible to define a set of actor parameters. They can be used when the same actor definition is used more than once in the same program. For example, the actor defined in Listing 2.3 (`ParametrizedSource`) uses the parameter `maxId`. This parameter, defined in line 1, is used as a guard condition by the (only) action `create` as defined in line 7. This actor also defines the internal variable `id` that is used and updated during each firing of the action as described in line 9.

Listing 2.3 – ParametrizedSource.cal

```
1  actor ParametrizedSource(int maxId) ==> int O :
2
3      int id := 0;
4
5      create: action ==> O:[id]
6      guard
7          id < maxId
8      do
9          id := id + 1;
10     end
11
12 end
```

Priorities and State Machines

In the `SwapInput` actor, reported in Listing 2.4, a finite state machine schedule is used to force the action sequence to switch between the two actions `A` and `B`. The schedule statement introduces two states `stateA` and `stateB`. In contrast, in the `PriorityInput` actor, reported in Listing 2.5, the selection of an action to fire is not only determined by the availability of tokens, but also depends on the priority statement.

Listing 2.4 – SwapInput.cal

```
actor SwapInput() T In1, T In2 ==> T O :

    A: action In1:[val] ==> O:[val] end

    B: action In2:[val] ==> O:[val] end

    schedule fsm stateA:
        stateA(A) --> B;
        stateB(B) --> A;
    end

end
```

Listing 2.5 – PriorityInput.cal

```
actor PriorityInput() T In1, T In2 ==> T O :

    A: action In1:[val] ==> O:[val] end

    B: action In2:[val] ==> O:[val] end

    priority
        A > B
    end

end
```

2.3.3 Example

In principle, *CAL* programs are structured as networks of interconnected actors. Figure 2.6 depicts a *CAL* program composed of 3 actors: `Source`, `Medium` and `Sink`, and 2 buffers: b_1 and b_2. Two different representations are supported for defining the *CAL* network structure: the first one is based on a functional programming language called Functional unit Network Language (*FNL*) and the second one is based on eXtensible Markup Language (*XML*) known as *XML* Dataflow Format (*XDF*).

As an example, the *FNL* and *XDF* network representations illustrated in Listings 2.6 and 2.7, respectively, both define a *CAL* program where the `Source` actor instantiates the actor class of `ParametrizedSource` defined in Listing 2.3, the `Medium` actor instantiates the `Multiplier` actor-class defined in Listing 2.1, and the `Sink` actor instantiates the `Disposer` actor-class defined in Listing 2.8. In this particular example, the `Source` actor instantiates its actor-class using the parameter `maxId=3`. Execution of this program in a single-core processing unit with an unlimited buffer size configuration (*i.e.*, it is always possible to produce tokens in a buffer) yields the corresponding action firings summarized in Table 2.3.

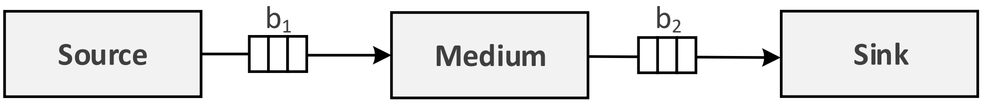


Figure 2.6 – Basic dataflow program.

Listing 2.6 – ProgramNetwork.nl

```
network ProgramNetwork () ==> :

entities

    Source = ParametrizedSource(maxId = 3);
    Medium   = Multiplier();
    Sink = Disposer();

structure

    Source.O --> Medium.I
    Medium.O --> Sink.I

end
```

Listing 2.7 – ProgramNetwork.xdf

```
<?xml version="1.0" encoding="UTF-8"?>
<xdf name="ProgramNetwork">
  <instance id="Source">
    <class name="ParametrizedSource"/>
    <parameter name="maxId">
      <expr kind="literal" literal-kind="integer" value="3"/>
    </parameter>
  </instance>
  <instance id="Medium">
    <class name="Multiplier"/>
  </instance>
  <instance id="Sink">
    <class name="Disposer"/>
  </instance>
  <connection src="Source" src-port="O" dst="Medium"   dst-port="I"/>
  <connection src="Medium"   src-port="O" dst="Sink" dst-port="I"/>
</xdf>
```

Listing 2.8 – Disposer.cal

```
actor Disposer() int I ==>  :

    dispose: action I:[val] ==> end

end
```

Firing	Actor	Actor-class	Action
s_1 s_2 s_3	`Source`	`ParametrizedSource`	`create`
s_4 s_5 s_6	`Medium`	`Multiplier`	`multiply`
s_7 s_8 s_9	`Sink`	`Disposer`	`dispose`

Table 2.3 – Firings of the CAL program described in Section 2.3.3.

2.3.4 Code generation

The portability of dataflow programs onto different HW and SW platforms is provided by a compiler infrastructure capable of generating a low-level representation from the high-level program description. As illustrated later in Figure 3.1, the compiler infrastructure is an essential part to enable an effective implementation and DSE of a dataflow program. In this Section, some basic components related to compilation and code generation are illustrated.

Abstract syntax tree

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of the source code. Each node of the tree denotes a construct occurring in the source code. The syntax is abstract in the sense that it does not represent every detail appearing in the real syntax. An AST is usually the result of the syntax analysis phase of a compiler or an interpreter. It often serves as an intermediate representation of the program through several stages that the compiler requires and has a strong impact on the final output of the compiler. After verifying the syntax, the AST serves as the base for code generation. The AST is used to generate the intermediate representation for the code generation or interpretation.

Intermediate representation

Intermediate representation (IR) is a representation of a program part-way between the input source and output target code. A well-structured IR does not depend on the input source code nor the target architecture. Hence, it maximizes its ability to be re-used in a retargetable compiler.

Control flow graph

The control flow graph (CFG) is a graph-based representation of the program control flow, which is generally used for making analyses from the IR representation of an input program [65]. The CFG of a function is a connected, directed graph where the set of nodes represents the sequences of program instructions and the set of directed edges ($i.e.$, ordered pairs of nodes) represents the control flow. More precisely, a node represents a basic block which is a maximal sequence of consecutive statements with a single entry point, a single exit point, and no internal branches.

2.3.5 RVC-CAL

CAL language has been explicitly designed in order to be fully analyzable and thus to support different forms of code analysis. Such an opportunity makes it possible to look for a variety of optimization techniques that can be applied before and during the synthesis from a dataflow program to the implementation code. A subset of the more general CAL language, called $RVC-CAL$, has been standardized by the ISO/IEC SC29WG11 committee also known as $MPEG$ [66, 67, 68, 69]. This subset restricts the data-types, operators, and features that can be used when describing a CAL actor. $RVC-CAL$ is used within the $MPEG$ community as a reference software language for the specification of the $MPEG$ video-coding technology in the form of a library of components ($i.e.$, the actors) that are configured and instantiated into networks to generate standard $MPEG$ video decoders ($e.g.$, MPEG4-SP, AVC, HEVC).

The $RVC-CAL$ compiler infrastructure used in this work is called open $RVC-CAL$ compiler infrastructure ($ORCC$) [70, 71, 72]. It provides the necessary tools for the design, simulation and code generation of different targets for $RVC-CAL$ programs. During the compilation flow, the $RVC-CAL$ program is translated into a code intermediate representation (IR). The IR is built using a model-driven engineering (MDE) meta-model. More precisely, it makes use of the MDE technologies available on the Eclipse IDE [73], such as the Eclipse modeling framework (EMF) [74, 75], Xtext [76] and Xtend [77]. The $ORCC$ compilation flow can be summarized as follows:

- **Front-end**: the $RVC-CAL$ code is parsed and translated into an AST. The AST is successively transformed into an IR. At this stage the semantic validation, the type inference and the expression evaluation are performed;
- **Core**: a meta-model of the IR is created and serialized. The serialization enables incremental compilations and analysis;
- **Interpreter**: the IR can be directly interpreted from its meta-model generated by the back-end. The code interpretation is type-accurate and it permits a first high-level and

behavioral verification of the program;

- **Back-end**: target-specific optimizations (*i.e.*, *IR* to *IR* transformations) are performed before the low-level code generation. Successively, the *IR* is translated into a general purpose programming language (*e.g.*, C/C++, Java) or to a register transfer language (*RTL*) (*e.g.*, VHDL, Verilog).

2.4 Summary and conclusions

This Chapter presented the principles of the dataflow programming paradigm which allows developing portable and composable applications. It also discussed different classes of dataflow programs starting from static and cyclo-static programs, through *MoCs* offering various extensions to the static models and hence capable of handling some dynamic behavior, up to entirely dynamic applications. Each class differs in terms of *expressiveness*, *facility of developing efficient implementations* and *analyzability*. An attempt of assessing different classes in these terms is presented in Figure 2.7. In general, it can be stated that the least expressive *MoCs*, such as *SDF* and *CSDF*, are easier to analyze and develop efficient implementations. Along with an increase of the expressiveness, the analyzability and the facility of developing efficient implementations decrease. The models offering dynamic extensions to static models discussed in Section 2.2.3 are difficult to order according to the expressiveness, since they offer different opportunities to model some aspects of a dynamic behavior. Some of them (*i.e.*, *TPDF*) try to establish a trade-off between the high expressiveness and analyzability [54]. It can be stated, that the most expressive *DDF* is also the most difficult to implement efficiently and analyze, but it enables an implementation of some algorithms that cannot be expressed using different *MoCs*. This observation highlights the necessity of developing efficient analysis methodologies supporting the design of *DDF* applications. Since these methodologies are the objective of this work, the *CAL* language capable of expressing an entirely dynamic application and the process of its compilation and code generation have been discussed towards the end of the Chapter.

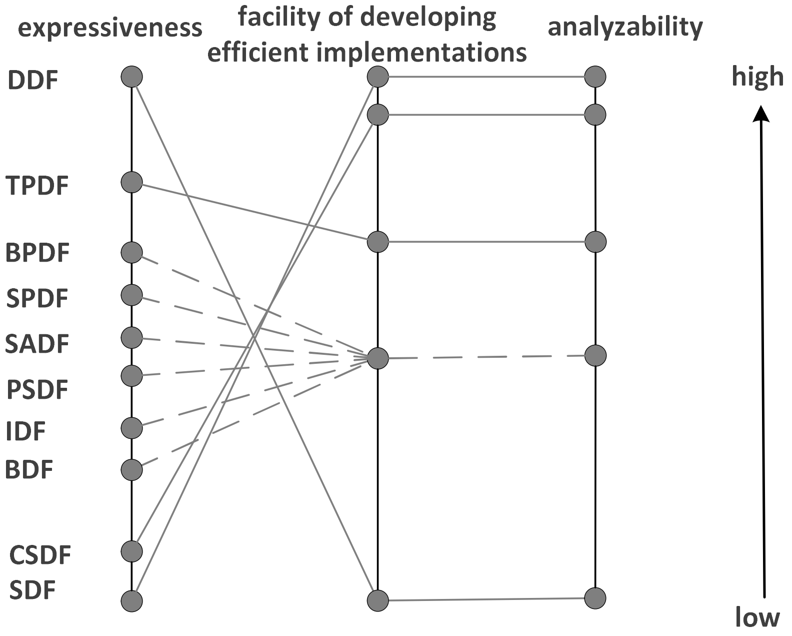


Figure 2.7 – Dataflow *MoCs* comparison.

3 State-of-the-art of dataflow-oriented analysis frameworks

This Chapter summarizes the state-of-the art of frameworks aiming at the analysis of parallel programs running on many/multi-core platforms and, in particular, heterogeneous platforms. The frameworks considered in this summary rely on the concept of dataflow directly ($i.e.$, using specifically dataflow programming languages) or indirectly ($i.e.$, using an application model corresponding to the actor-oriented concept of dataflow programs). The objective of this overview is to examine the analysis tools available for different $MoCs$, their features and limitations. As a second part, a separate Section is dedicated to the frameworks considering $RVC-CAL$ programs. Following the common stages present in various frameworks, in the last part, the dataflow design flow considered in this work is presented and discussed.

3.1 Frameworks

Daedalus allows an analysis of KPN programs by means of rapid system-level architectural exploration, high-level synthesis, programming and prototyping. It was first introduced in 2007 and the most recent version was released in 2012 [78, 79, 80].

MAPS: MPSoC Application Programming Studio, introduced in 2008, also targets KPN programs. Its main functionalities include design space exploration and performance estimation in order to provide fast and functional design validation. The framework is currently maintained and being transfered into commercial tools [81, 82, 83, 84, 85].

Mescal does not limit the set of supported $MoCs$ and considers any combination which is natural for the application domain. It can be used to design heterogeneous application-specific, programmable multiprocessors. It provides an abstraction path from micro-architectures to application-architecture mappings. It was introduced in 2002 and actively developed until 2005 [86, 87].

Metropolis also supports various $MoCs$, due to meta-modeling with precise semantics. In allows a description and refinement at different levels of abstraction. The integrated functionalities include: modeling, simulation, synthesis and verification. It was introduced in 2003 and the most recent version was released in 2008 [88, 89].

PeaCE is oriented at $SPDF$ programs (Section 2.2.3). It offers a co-design flow from functional simulation to system analysis. During the entire design process it uses the features of formal models. The releases of the framework date to 2003-2006 [90, 91].

PREESM: Parallel and Real-time Embedded Executives Scheduling Method is a framework offering rapid prototyping of SDF applications in order to optimize the throughput. It allows an automatic generation of functional code for heterogeneous multi-core embedded systems. Since its introduction in 2009, it has been continuously under active development [92, 93, 94].

Ptolemy analyses programs expressed as hierarchical combinations of different $MoCs$ with a high level of abstraction. It offers a component-based modeling of heterogeneous platforms and design space exploration with third party environments. Since the first release in 2003, the framework is still maintained [95, 96].

SDF^3 considers SDF and $CSDF$ applications. It offers a model analysis and simulation without generation of an executable prototype of the application. The releases of the framework date to 2007-2014 [97, 98].

Sesame, introduced in 2006, is another KPN-oriented framework. Its main functionality is identification of a suitable and efficient *MPSoC* platform architecture. It evaluates the application, the architecture and the mapping between them. It has recently migrated to Eclipse RDF4J [99, 100].

Space Codesign considers programs expressed as SystemC. It is a co-simulation environment for user-written SystemC modules making calls to real-time operating system kernels. It was released in 2008 and actively maintained until 2015 [101, 102].

SPADE: Stream Processing Application Declarative Engine, introduced in 2008, considers programs expressed as System S [103] which is a large-scale, distributed data-stream processing middleware. It offers rapid application development and code generation framework to create optimized applications that run natively on the Stream Processing Core (SPC) [104, 105].

SynDEx, introduced in 2010, supports various $MoCs$. It is a computer-aided-design software aiming at mapping an algorithm to an architecture. The objective of the design space exploration is the application throughput [106].

SystemCoDesigner, introduced in 2008, considers programs expressed as SysteMoC which is a high-level language built on top of SystemC. It allows a hardware-software *SoC* generation with automatic design space exploration techniques [107, 108, 109].

3.2 Features

The comparisons made in this Section include different properties and functionalities offered by the frameworks. The overview summarizes the chosen frameworks and considers the way the applications and architectures are modeled and handled, available features and additional requirements, such as the usage of external tools. The analysis is presented in Tables 3.1-3.3.

3.3 Exploration heuristics

The design space exploration problem tackled by the frameworks is usually related to evaluation and exploration of different design alternatives. These design alternatives are related either to the parameters of the platform (*i.e.*, when different types of architectural components are considered in the model) or to the configurations applied to a program when it is ported onto a platform (*i.e.*, partitioning of the program components onto the processing elements). However, even when the exploration of available architectural options is tackled, it implies defining an assignment of the program components to the processing elements of the platform. In some cases, the exploration process is supported by various methods to establish a *critical path* of the design and/or to identify the bottlenecks of the execution. Different approaches to the *DSE* in terms of the formulation of the problem, the applied set of steps, along with the available heuristics are summarized for the chosen frameworks throughout this Section.

MAPS

The *DSE* flow consists of several phases related to mapping and scheduling. At each stage, various heuristics are available. In general, they put an emphasis on the advantages of lightweight heuristics over evolutionary methods, and aim at satisfying the specified constraints. The following stages are defined:

- **Pre-Scheduling**: this is devoted to a specification of finite buffer sizes. It is assumed that appropriate sizes must be found, so that no deadlocks occur. Such a configuration can be established using two heuristics; *Simulated Execution* relies on observing channel utilization for different inputs. *Traffic ratio* relies on allocating memory to every channel proportionally to the traffic on this channel;
- **Scheduling**: the supported policies are all data-driven. The first heuristic relies on the idea of *computing a time slot*, so that context switches before potential channel

Framework	application handling	architecture handling	features	additional
Daedalus	applications are modeled using a C/C++ imperative specification and converted to KPN with KPNgen-tool [110]	multimedia *MPSoC* are considered as target architectures	-	*DSE* is performed with Sesame framework
MAPS	applications are modeled as execution traces, where dependencies between different firings are related to tokens exchange and internal variables	simplified view of the target *MPSoC* (list of processing elements and communication primitives modeled by a cost function)	• exploration of different configuration options determining the efficiency of a program; • performance estimation using trace replay module: a discrete-event simulator that schedules the segments of the execution trace (Chapter 6 of [111]); • several heuristics for mapping, scheduling, synchronization and search algorithm; • composability analysis for the purpose of executing simultaneously multiple applications on a given platform without interference [82]; .	integrated with High-Level Virtual Platform simulator (HVP) [112]
Mescal	-	a micro-architecture description includes the memory subsystem based on an architecture description language	-	-

Table 3.1 – Analysis frameworks: features summary, part 1.

Framework	application handling	architecture handling	features	additional
Metropolis	applications are modeled as a set of processes that communicate through media	architectural building blocks are represented by performance models where the events are annotated with the costs of interest	• a third network correlates the two models by synchronizing the events between them; • non-deterministic behavior and constraints can be considered; • possible translation into a Petri net specification and interprocess communication removal; .	-
PeaCE	3 models provided: computation tasks, control tasks and a task model to describe interactions	-	-	based on Ptolemy
PREESM	application representation as parametrized and interfaced dataflow meta-model $PiMM$ [113]	high-level architecture description using system-level architecture model $S-LAM$	• based on the concept of algorithm architecture adequation matching methodology (AAA/M) [114]; • a scenario determines the set of parameters specifying the deployment conditions and constraints; .	-
Ptolemy	tokens are used as underlying communication mechanism regulating how the actors fire	-	-	DSE performed with external tools ($i.e.$, PeaCE)
Sesame	application model describing functional behavior	architecture model describing the available resources and their performance constraints	• trace-driven simulation; • intermediate mapping layer for scheduling and event-refinement purposes; .	-

Table 3.2 – Analysis frameworks: features summary, part 2.

Framework	application handling	architecture handling	features	additional
Space Codesign	-	-	• 1st layer: application specification and verification; • 2nd layer: hardware/-software partitioning; • 3rd layer: emulation of a more sophisticated architecture model using a cycle-accurate simulation; .	-
SPADE	an intermediate language for the composition of parallel and distributed dataflow graphs	a toolkit of type-generic, built-in stream processing operators supporting scalar and vectorized processing	• a compiler automatically mapping applications into appropriately-sized execution units; • communication overhead minimization and parallelism exploitation; .	-
SynDEx	algorithm graph	architecture graph and system constraints	• algorithm architecture adequation matching methodology (AAA/M); • distribution and scheduling of the algorithm manually or automatically with optimization heuristics, based on multi-periodic distributed real-time scheduling analyses; .	-
System CoDesigner	high-level model translated into behavioral SystemC	-	• performance estimation using performance models generated automatically from the SystemC behavioral model; • HW and SW synthesis; • DSE using state-of-the-art multi-objective algorithms; .	HW synthesis delegated to a commercial tool: Forte Cynthesizer [115];

Table 3.3 – Analysis frameworks: features summary, part 3.

writes are avoided. The second heuristic considers the *process importance*, which can be implemented in multiple ways, including, for instance, topology, output rate, execution weight *etc*;

- **Mapping phase**: a static assignment of the processes to the processing elements is considered. The available strategies are: *Computation balancing, Affinity, Output rate balancing, Simulated mapping*;
- **Post-scheduling phase**: this consists of making final adjustments to the schedule descriptors (*i.e.*, fine tuning of buffer sizes).

PeaCE

The *DSE* is considered as a two-steps process. The communication architecture, including the memory system, is explored after the processing components are selected and the *HW/SW* partitioning decisions are made. Global feedback forms an iterative *DSE* loop. The loop is applied only to dataflow tasks that are computationally intensive, whereas the processing elements to execute control tasks are determined manually. An iteration of the co-synthesis loop (the first inner loop of the proposed flow) solves three subproblems: selecting appropriate processing elements, mapping function blocks to the selected processing elements, and evaluating the estimated performance or examining schedulability to check whether the given time constraints are met.

Since the considered design space of architectural solutions is very wide, it is traversed in an iterative fashion. First, a subspace of architecture candidates is explored quickly to build a reduced set of design points to be carefully examined. Within a given set of architecture candidates, all design points are visited by varying the priorities and conditions. Design points with a performance difference of less than 10% compared to the best one (the value of 10% comes from the accuracy of the used estimation method) are collected. The second step applies trace-driven simulation to the design points selected from the first step. It accurately evaluates the performances in the reduced space and determines the best point. The process continues as long as the iterations bring an improvement to the solution. The third step generates the next set of architecture candidates relying on the architecture of the best design point and applying small modifications to it. It results in a set of Pareto-optimal design points (that is, not dominating in terms of all optimization criteria). Although this heuristic does not explore the entire design space, its main objective is to prune the design space aggressively and arrive quickly to a high-quality solution.

During the process of simulation performed for different candidates, a performance profiler is used. The execution time and the number of executions of different tasks are recorded. From this information, the performance bottleneck of the implemented code is identified.

PREESM

The process of rapid prototyping consists of exploring the design space of a target system in order to minimize its cost and guarantee the respect of different constraints. The most common ones are: latency, throughput, memory, and energy consumption. Other constraints may exist, such as jitter or signal simultaneity.

The problems of mapping and scheduling are considered jointly by different optimization algorithms, ranging from simple to evolutionary methods. The problem of buffer dimensioning is related to a bounded memory execution guaranteeing a deadlock-free execution. Due to the supported MoC, it is possible to statically distribute the tasks. A static scheduling algorithm is usually described as a monolithic process carrying two distinct functionalities: choosing the core to execute a specific function and evaluating the set of the generated solutions. The implemented scheduling algorithms include:

- *list scheduling*: the tasks are scheduled in the order dictated by a list constructed from estimating a critical path. Once made, a mapping choice is never modified. List scheduling is used as a starting point for other refinement algorithms;
- *FAST algorithm*: it is used as a refinement of the list scheduling solutions by utilizing probabilistic hops. It changes the mapping choices of randomly chosen tasks keeping the best latency found, until stopped by the user;
- *genetic algorithm*: is coded as a refinement of the FAST algorithm, since the n best solutions found by *FAST* are used as the base population for the genetic algorithm.

A necessity to analyze the impact of application and architecture bottlenecks on the system performance is also emphasized. The high-level architecture description facilitates studying the bottlenecks arising on the platform side.

Sesame

In the context of this framework, the mapping is related to an intermediate layer between the application and the architecture. Hence, defining a mapping is necessary in order to evaluate a candidate architecture. The mapping problem is defined as Multiprocessor Mappings of Process Networks ($MMPN$), which is defined as: $min f(x) = (f_1(x), f_2(x), f_3(x))$, subject to $g_i(x), i \in \{1, \ldots, n\}, x \in X_f$, where:

- f_1 is the maximum processing time;
- f_2 is the total power consumption;

- f_3 is the total cost of the system.

The functions g_i are the constraints and $x \in X_f$ are the decision variables. They represent decisions, *i.e.*, which processes are mapped onto which processors or which processors are used in a particular architecture. The constraints make sure that the decision variables are valid, *i.e.*, result in a feasible solution. The optimization goal is to identify a set of solutions which are superior to all other solutions when all three objective functions are minimized. This is accomplished by a Strength Pareto Evolutionary Algorithm (*SPEO*) that finds a set of approximated Pareto-optimal mapping solutions. Scheduling of the processes can be performed in a static, semi-static or dynamic manner. A mapping configuration contains also an assignment of buffers (with limited sizes) that are parameterized and dependent on the architecture. The assignment of buffers is performed in a *safe* way, that is, guaranteeing a deadlock-free execution. The performance numbers provided by the framework are intended to inspire the designer to improve the architecture, restructure/adapt the application, or modify the mapping of the application.

The output of system simulations in Sesame provides the designer with performance estimates of the system(s) under investigation together with statistical information such as utilization of architectural components (idle/busy times), the contention in a system (*e.g.*, network contention), profiling information (time spent in different executions), critical path analysis, and average bandwidth between architecture components. Such results allow an early evaluation of different design choices, identifying trends in the systems' behavior and revealing performance bottlenecks early in the design cycle.

SystemCoDesigner

The process of automatic *DSE* consists of finding optimal or near optimal solutions in terms of throughput, latency or required chip size by allocating processors, memories, buses, and hardware accelerators, and binding the actors and channels to the resources. As the first step, the particular instance of the system synthesis problem is formalized by providing a so-called architecture template specifying the architectural components and their interconnections. From this set, the automatic *DSE* has to select a subset in order to form an implementation. The target architecture allows for hardware only, software only, and mixed hardware/software designs of an application.

Next, a formal model serving as an input to the *DSE* is built. It consists of (a) the application, (b) the architecture template and (c) the mapping constraints. For *each* possible mapping, the execution times of an actor in a specific binding are annotated. Then, the exploration is performed using Multi-Objective Evolutionary Algorithms (*MOEA*) [116]. The problem addressed by *MOEA* can be stated as follows:

- The architecture is modeled as a graph representing possible interconnected hardware resources;
- The application is modeled as a graph describing the behavior of the system (vertices describe tasks, directed edges - dependencies). Data–dependent tasks have to be executed on the same or adjacent resources to ensure correct communication;
- The set of mapping edges indicates whether a specific task can be executed on a hardware resource;
- The allocation set α is the set of hardware resources;
- The binding β determines on which allocated resource each task is executed. For each task from the problem graph exactly one mapping has to be used.

Due to the data dependencies, a binding can be infeasible. A binding is called feasible if it guarantees that the data communications imposed by the problem graph can be established by the allocated resources. Furthermore, a feasible allocation is an allocation α that allows at least one feasible binding β. The task of design space exploration is formulated as a multi–objective optimization problem: minimize $f(\alpha, \beta)$, subject to: (1) α is a feasible allocation, (2) β is a feasible binding. Unlike for the case of a single-objective optimization problems, in multi–objective optimization problems, the feasible set is only partially ordered and, thus, there is generally not only one global optimum, but a set of Pareto–optimal solutions. The $MOEA$ does not make any assumption about the objective function.

3.4 RVC-CAL frameworks

In summary, the $MoCs$ considered by the aforementioned frameworks can be grouped as follows: static dataflow, dynamic dataflow expressed as KPN, combined multiple $MoCs$, SystemC and its derivatives. All of these $MoCs$ differ from the concepts behind the $RVC-CAL$ programs. First, the methods designed for the analysis of static $MoCs$ cannot be applied directly to the dynamic ones. Second, $RVC-CAL$ captures the features of ATS actors which differ in some details from the concept of KPN (as discussed in Sections 2.1.1 and 2.1.3). Hence, these methods also cannot be applied, although many similarities can be identified. This Section summarizes two analysis frameworks available for $RVC-CAL$: **CAL Design Suite** [37, 117, 118] and **COMPA** [119].

CAL Design Suite

CAL Design Suite (released 2010-2013) aims at exploration and optimization of the design space of $RVC-CAL$ programs. It constitutes the first functional attempt to define a complete design flow for multi-core and heterogeneous platforms [34]. The analyses are based on an execution trace, where dependencies between different firings are related to tokens exchanges. It formalizes a basic architecture model for heterogeneous platforms. The analysis of the programs is static and relies on the usage of weighting operators.

The *DSE* problem is separated into two phases: assigning actors to processors and then sequencing the actions. The aim of the exploration is to find such a partitioning and scheduling configuration that leads to an efficient implementation. The complex solution space is split into two orthogonal spaces: the permutation space of the actors on the processors and the space specifying precedence among actions. It is assumed that usually the number of actors is very small in comparison to the number of nodes. Thus, a search for the partitioning of actors can be considered as sufficiently comprehensive to an examination of *all* possibly efficient partitioning configurations. Several heuristics are implemented for this purpose:

- Round-robin load balancing: this relies on the technique aiming at distributing the computations of the actors across the processing units;
- Simulated annealing load balancing: the load balancing technique is kept as a basis, but a simulated annealing approach [120] taking into account communications costs is adopted;
- Causation trace scheduling: minimization of the makespan (completion date of the execution) based on the causation trace (execution trace) using a simulated annealing approach is used;
- Static regions scheduling: an alternative approach, aiming at considering the non-negligible scheduling overhead, consists of extracting static regions and at computing their schedule at compile-time;

Another part of the exploration and analysis is the identification of the *critical path* of the design and the actions with the largest contribution to the critical path so that the bottlenecks of the design are efficiently identified.

COMPA

COMPA is a framework intended for analysis and optimization of $RVC-CAL$ programs and was introduced in 2011. The exploration process aims at finding different trade-offs regarding the parallelism, communication cost and memory size. These trade-offs are modeled as source-to-source transformations. The exploration is based on the static analysis of the source code.

3.5 Design flow for dataflow programs

Although the frameworks presented in the previous Sections differ significantly in terms of the considered *MoC*, *DSE* objectives and the available functionalities, many similarities can be identified. For instance, several frameworks emphasize the importance of independent application and architecture models, so that the portability of the design is ensured. In many cases, the flow provided by the frameworks consists of some analyses of the application and/or architecture, prototyping, simulation and code generation. The analysis is often supported by means of *DSE* heuristics and performance estimation. All of these stages can be directly translated into a complete system development design flow.

Such a design flow, as introduced in [121, 122], is illustrated in Figure 3.1. The program behavior is separated from the architecture model and expressed using the *CAL* dataflow programming language. The architecture represents the target platform the program is to be implemented on and is characterized by the available resources and, if applicable, constraints. The design flow consists of the following elements:

1. **Compiler infrastructure**: the source code of a *CAL* program is transformed into an equivalent intermediate representation. At this stage the compiler allows performing a verification about whether the program behavior is correct using directly the intermediate representation, without prototyping or creating a partial implementation of the design;

2. **Profiling and analysis**: the exploration of different design alternatives is performed with regards to the constraints and objective functions. At this stage it is also possible to perform various analyses, *i.e.* profiling, bottlenecks;

3. **Refactoring directions**: if the design point established during the exploration satisfies the requirements (constraints, objective functions), it is used to drive the compiler infrastructure through a set of compiler directives. In the opposite case, the set of refactoring directions is identified and provided as a feedback to the designer. The refactoring directions can include, for instance, the parts of the program identified as the main bottlenecks for the high-quality design point found during the exploration. Resolving these bottlenecks is then subject to a programming effort;

4. **Performance estimation**: the execution times for different design points are estimated without requiring any partial implementation of the program. The estimation is done according to the abstract model of the program execution and the model of the architecture providing the appropriate timing information. According to the estimation results, the number of design points considered for further exploration can be narrowed;

5. **Code generation**: the CAL representation of a program is transformed into a low-level code representation. An appropriate SW/HW code is generated according to the mapping of the program to the target architecture;

6. **Synthesis or compilation**: the SW (HW) code is compiled (synthesized), respectively, and the executables are obtained;

7. **Implementation**: if the constraints are satisfied and the expected values of the objective functions are achieved, the design is implemented on the appropriate software and/or hardware architecture. If both architecture types are present, the interfaces provided by the architecture are automatically integrated into the design.

3.6 Contributions to the state-of-the-art

So far, most of the research efforts regarding the analysis and design space exploration of dataflow programs target the simple cases of static dataflow $MoCs$, where the spectrum of algorithms for which the implementations are possible is very limited. In contrast, the methodologies described in this dissertation consider fully dynamic dataflow programs, but remain valid also for other, less expressive variants. An implementation and analysis of dynamic programs is much more difficult, however still possible, when appropriate programmatic approaches are used, as discussed and demonstrated in different Chapters. An important novelty of the dissertation comparing to the state-of-the-art methodologies is the concept of design exploration in the context of a multidimensional space, consisting of various configurations that impact each other in the process of exploration and all together lead to a certain performance of a program.

3.7 Summary and conclusions

This Chapter examined some of the analysis frameworks available for parallel programs and, in particular, dataflow programs. The review has been performed at three layers. First, several frameworks have been described in terms of the MoC, the main properties and functionalities (*e.g.*, the assumed objective of DSE). Second, the chosen interesting features of the frameworks have been listed. The choice of the frameworks considered in this summary and the listed features has been made according to the relevance to the work described in this Thesis. Finally, among the frameworks offering some DSE methodologies, the way the DSE problem is formulated and handled has been presented and/or the available heuristics have been briefly described. In the second part, the frameworks targeting $RVC-CAL$ programs have been discussed separately.

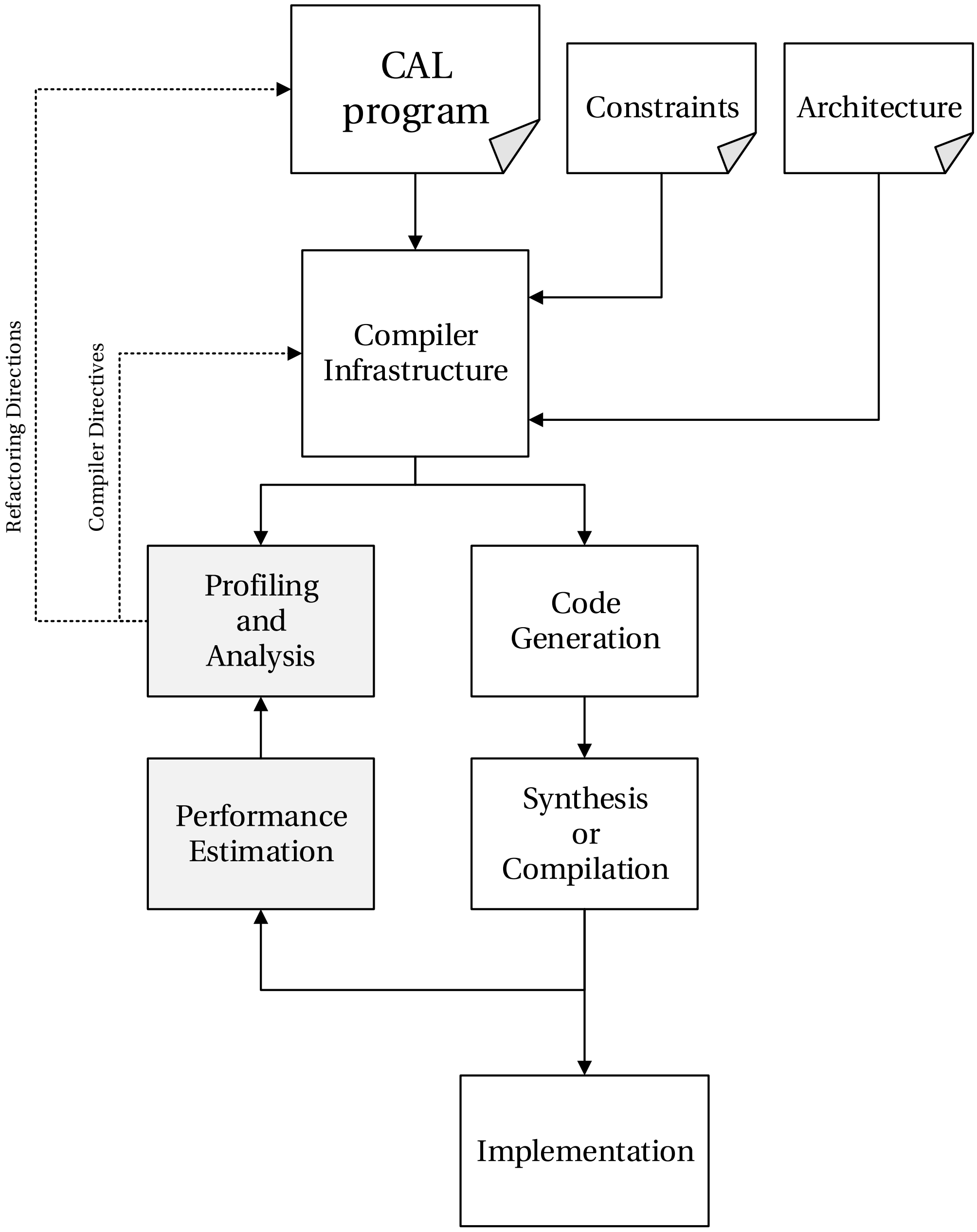


Figure 3.1 – Heterogeneous system development design flow for *CAL* dataflow programs.

Despite the differences occurring at different stages of the analysis, some common features can be identified among the frameworks. They include: independent application and architecture models, verification, design space exploration, performance estimation, code generation *etc*. These common stages can be translated into a complete dataflow design flow, presented and discussed in the last part.

Comparing different frameworks, an important observation is that many rely one way or another on the concept of an execution trace. A trace, expressed as a directed graph, represents the execution and acts as a basic model used in the exploration process. Adding accurate timing information related to an execution on a target platform (*i.e.*, obtained by profiling) provides rich performance metrics. The details of the construction of an execution trace which is used in this work and the way the appropriate timing information is obtained, are discussed in the two following Chapters.

4 Program execution modeling

Following the discussion about dataflow *MoCs* in Chapter 2 and, in particular, the properties of firings which constitute the program execution, this Chapter describes the construction of an *execution trace graph* (ETG). This graph-based representation demonstrates the correlation between the firings and models the execution behavior. This representation, originally introduced and thoroughly discussed in Chapter 5 of [121], is used as a basic model for the *DSE* methodologies described in this work.

4.1 Execution Trace Graph

Execution trace graph (ETG) is a directed acyclic graph (DAG) where each node represents a single action firing and each directed arc is either a data or a logical dependency between two different action firings [37, 57, 82, 123]. As described in Chapter 2, when an action is fired it can consume a finite number of input tokens, produce a finite number of output tokens, and modify the actor's internal variables. Hence, the dependencies arising between different firings can be observed. For example, if an action consumes some tokens during a firing, then it must rely on the execution of the action that produces these tokens. The same can be stated if an action, in the processing part, makes use of some of the internal actor variables that have been previously modified or used by another action. There are several types of dependencies that can be identified and used to characterize the execution of a dataflow program. They are discussed in detail in Section 4.1.2.

Formally, an ETG is defined as a $DAG(\mathbb{S},\mathbb{D})$, where:

- $\mathbb{S}$ is the set of single action firings, defining the nodes of the graph;

- $\mathbb{D} = \mathbb{S} \times \mathbb{S}$ is the set of dependencies, defining the directed edges of the graph.

Defining dependencies between action firings establishes precedence orders. If firing $s_2 \in \mathbb{S}$ depends on firing $s_1 \in \mathbb{S}$, then s_1 has to be executed and completed before s_2 can start. The dependency is then defined as $(s_1, s_2) \in \mathbb{D}$. The transitive hull of the dependencies is the precedence relation $\leq$. Thus, $\mathbb{S}$ can be defined as a partially ordered space $(\mathbb{S}, \leq)$ and the precedence constraint between s_1 and s_2 can be expressed as $s_1 \prec s_2$. It is assumed that the number of firings in $\mathbb{S}$ and the number of dependencies in $\mathbb{D}$ are finite and denoted with $|\mathbb{S}| < \infty$, $|\mathbb{D}| < \infty$, respectively.

4.1.1 Firings

Each $s_i \in \mathbb{S}$ represents a single action firing occurring during the execution of a dataflow program. This means that if an action is fired n times, then n nodes in $\mathbb{S}$ are used to represent each single firing of this action.

A single action firing $s \in \mathbb{S}$ is formally defined as a 3-tuple $s(a, \lambda, \eta)$, where:

- $a \in A$ is the actor;
- $\lambda \in \Lambda$ is the action;
- $\eta \in \mathbb{N}$ is the action execution index, that identifies two different firings of the same action during the entire program execution.

4.1.2 Dependencies

Each $(s_i, s_j) \in \mathbb{D}$ represents a dependency between two executed actions s_i and s_j, such that $s_i \neq s_j$. Several types of dependencies can be defined during the execution of a dataflow program. These are: *internal variable, finite state machine, guard, port* and *tokens* (as summarized in Table 4.1). As illustrated in the following, each can be defined by a subtype and enhanced with some profiling parameters useful for the future *DSE* analysis. Hence, more than one dependency of different types can be defined between each couple s_i, s_j.

A dependency $(s_i, s_j) \in \mathbb{D}$ is formally defined as a 5-tuple (s_i, s_j, μ, d), where:

- $s_i \in S$ is the source action firing;
- $s_j \in S$ is the target action firing;
- μ is the dependency type;
- d is the dependency direction. The direction can be: *read/read, read/write, write/read, write/write, enable, disable* or *undefined.*

The incoming dependencies set of a firing s_i is defined as:

$$\delta(s_i)^-_E = \{(s_n, s_m) : \forall(s_n, s_m) \in \mathbb{D}, s_m = s_i\} \tag{4.1}$$

The set of firings which are the sources of the incoming dependencies of s_i is the set of *predecessors*, and is denoted as:

$$\delta(s_i)^-_{\mathbb{S}} = \{s_j : \exists(s_j, s_i) \in \mathbb{D}\} \tag{4.2}$$

Firings that do not have any predecessors are called sources of the ETG. The set of sources is defined as:

$$\mathbb{S}_{\emptyset^-} = \{s_i : \delta(s_i)^-_S = \emptyset\} \tag{4.3}$$

Similarly, the set of outgoing dependencies of a firing s_i is defined as:

$$\delta(s_i)^+_E = \{(s_n, s_m) : \forall(s_n, s_m) \in \mathbb{D}, s_n = s_i\} \tag{4.4}$$

The set of firings which are the targets of the outgoing dependencies of s_i is called the set of *successors*, and is denoted as:

$$\delta(s_i)^+_{\mathbb{S}} = \{s_j : \exists(s_i, s_j) \in \mathbb{D}\} \tag{4.5}$$

Firings that do not have any successors are called sinks of the ETG. The set of sinks is defined as:

$$\mathbb{S}_{\emptyset^+} = \{s_i : \delta(s_i)^+_S = \emptyset\} \tag{4.6}$$

Internal variable

An internal variable dependency $(s_i, s_j) \in \mathbb{D}$ occurs when two actions of the same actor share the same internal variable $v \in V$. More precisely, four different directions can be defined:

- **write/read**: when an action firing s_j reads the internal variable v without an intervening write operation and s_i is the last action firing, previous to s_j, that wrote to v;
- **write/write**: when an action firing s_j has an intervening write operation to the internal variable v and s_i is the last action firing, previous to s_j, that wrote to v.
- **read/read**: when both action firings s_i and s_j read the internal variable v without an intervening write operation and s_i is the last action firing, previous to s_j, that read from v.

- **write/write**: when both action firings s_i and s_j wrote to the internal variable v and s_i is the last action firing, previous to s_j, that wrote to v.

Only the *write/read* is a data dependency. By contrast, the *read/write*, *read/read* and *write/write* express only a memory utilization precedence between the two firings. The parameter that can be stored in this type of dependency is the variable v that the dependency is related to. Additional attributes, which need to be obtained by profiling, are the initial and final values of this variable. The set of dependencies of this type is denoted as $\mathbb{D}_v \subseteq \mathbb{D}$.

Finite state machine

An internal state machine dependency $(s_i, s_j) \in \mathbb{D}$ connects two executed actions belonging to the same actor and related via its internal state scheduler. In other words, a dependency of this type occurs when the execution of action firings s_i and s_j is driven by the actor internal FSM and s_i is the last action firing, previous to s_j, scheduled by the FSM. The set of dependencies of this type is denoted as $\mathbb{D}_f \subseteq \mathbb{D}$.

Guard

A guard dependency $(s_i, s_j) \in \mathbb{D}$ occurs when an action firing s_i modifies the value of the guard conditioning the action firing s_j. The guard condition, which can be described as a combination of state variable and token value, can be *enabled* or *disabled* by s_i through the modification of its variables or the production of particular token values. For this type of dependency, two different directions can be defined:

- **enable**: when the modification of an internal variable or the production of a token performed by s_i makes the action firing s_j executable ($i.e.$, enabled);

- **disable**: when the modification of an internal variable or the production of a token performed by s_i makes the action firing s_j non-executable ($i.e.$, disabled).

The parameters that can be stored are the guard identifier the dependency is related to and the appearance order according to which this guard was enabled or disabled. The set of dependencies of this type is denoted as $\mathbb{D}_g \subseteq \mathbb{D}$. It must be noted that in some design cases, uncovering these dependencies might have the side effect of making the trace dependent on both the buffer dimensioning and the scheduling configuration used during the program execution. A more detailed discussion about this dependency and its influence on the modeling of a dynamic execution is presented in Section 5.3.6 of [121].

Port

A port dependency $(s_i, s_j) \in \mathbb{D}$ connects two action firings of the same actor that share an input or an output port p. It defines in which order the tokens must be consumed or produced from/to this port. More precisely, two different directions can be defined:

- **read/read**: when both action firings s_i and s_j retrieved some tokens from the input port p and s_i is the last action firing, previous to s_j, that retrieved at least one token from p;
- **write/write**: when both action firings s_i and s_j sent some tokens to the output port p and s_i is the last action firing, previous to s_j, that sent at least one token to p.

The parameter that can be stored in this type of dependency is the port p (input or output) that the dependency is related to. The set of dependencies of this type is denoted as $\mathbb{D}_p \subseteq \mathbb{D}$.

Tokens

A tokens dependency $(s_i, s_j) \in \mathbb{D}$ connects an action firing that produces some tokens to the one that consumes at least one of them. In such cases, these actions may belong to different actors or they may be parts of the same actor (*i.e.*, in the case of a direct feedback loop). The parameters that can be stored in this type of dependency are the number of tokens that the firing s_j consumed from the tokens produced by the producer firing s_i. Additional attributes, to be retrieved by profiling, are the token values. The set of dependencies of this type is denoted as $\mathbb{D}_t \subseteq \mathbb{D}$.

	Name	**Direction**	**Parameters**	**Additional attributes**
$\mathbb{D}_v$	internal variable	read/read write/write read/write write/write	variable id	initial value final value
$\mathbb{D}_f$	finite state machine			
$\mathbb{D}_g$	guard	enable disable	guard id appearance order	
$\mathbb{D}_p$	port	read/read write/write	port id	
$\mathbb{D}_t$	tokens		output port id number of tokens	token values

Table 4.1 – Dependencies: types, directions, parameters and additional attributes.

4.1.3 Example

The main structure of ETG is illustrated using the example of the dataflow program presented earlier in Section 2.3.3. The firing set $\mathbb{S}$ contains 9 action firings $s = \{s_1, s_2, \ldots, s_9\}$ (summarized in Table 2.3). The firing set $\mathbb{S}$ can be divided into 3 subsets, one for each actor of the network, $\mathbb{S}_{Sr} = \{s_1, s_2, s_3\}$, $\mathbb{S}_{Md} = \{s_4, s_5, s_6\}$ and $\mathbb{S}_{Sk} = \{s_7, s_8, s_9\}$, such that $\mathbb{S} = \mathbb{S}_{Sr} \cup \mathbb{S}_{Md} \cup \mathbb{S}_{Sk}$ and $\mathbb{S}_{a_n} \cap \mathbb{S}_{a_m} = \emptyset$ for each couple of actors $a_m \neq a_n$. Sets $\mathbb{S}_{Sr}$, $\mathbb{S}_{Md}$ and $\mathbb{S}_{Sk}$ contain the firings of `Source`, `Medium` and `Sink` respectively. The dependencies set $\mathbb{D}$ contains 16 dependencies $\mathbb{D} = \{e_1, e_2, \ldots, e_{16}\}$ (summarized in Table 4.1).

Now, the sequence of firings for this example is used in order to highlight how the ETG (depicted in Figure 4.2) is constructed. Let's assume a partitioning configuration consisting of a single partition (all actors are assigned to the same processing element), a predefined and static scheduling configuration (order of execution: $\{Sr, Sr, Sr, Md, Md, Md, Sk, Sk, Sk\}$) and a buffer dimensioning configuration where both buffers are assigned the same size of 512 tokens. Figure 4.1 depicts the Gantt chart of the execution in this particular set of configurations, where each action firing takes exactly 1 (abstract) clock-cycle to perform its execution.

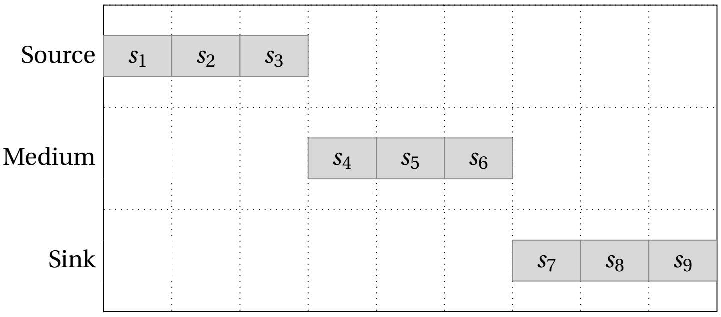


Figure 4.1 – Example: Gantt chart.

At time $t = 0$, the `Source` actor fires the action `create` (denoted as firing s_1). During this execution, s_1 updates the internal actor variable `id` from $\texttt{id}_i = 0$ to $\texttt{id}_f = 1$. The firing terminates with writing an output token $\tau_1 = 1$ to the output port `O`. At time $t = 1$, again the `Source` actor fires the action `create` (denoted as firing s_2). Also s_2 updates the internal actor variable `id` from$\texttt{id}_i = 1$ to $\texttt{id}_f = 2$. The firing terminates with writing an output token $\tau_1 = 2$ to the output port `O`. During the execution of firing s_1, the internal state variable $\texttt{counter}_i$ has the value previously written by the firing s_1. This implies defining an internal variable dependency between s_1 and s_2, denoted as e_1. Since both firings wrote to this variable, the dependency direction is *write/write*. Moreover, both s_1 and s_2 wrote a token to the same output port. This implies defining a port dependency, with direction *write/write*, denoted as e_2. The same happens at time $t = 2$, when the same action is fired for the 3rd time in a row. This new firing is denoted with s_3. Also in this case an internal variable and a token dependency can be

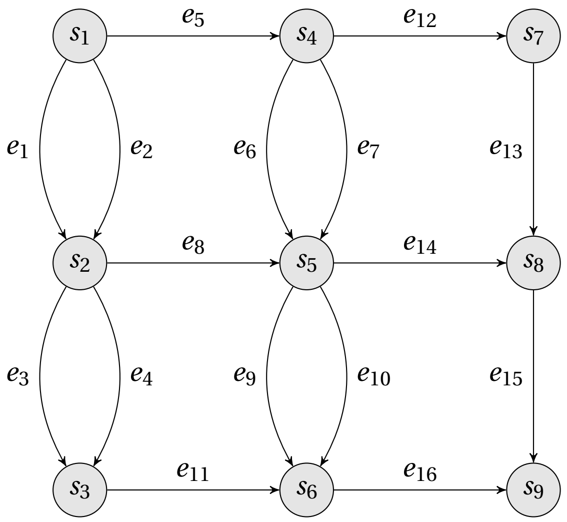


Figure 4.2 – ETG obtained after the execution of the CAL program from Section 2.3.3. The set of firings $\mathbb{S}$ (dependencies $\mathbb{D}$) is summarized in Table 2.3 (4.2), respectively.

defined with the previous step s_2 (e_3 and e_4, respectively).

At time $t = 3$, the `Medium` actor fires the action `multiply` (denoted as firing s_4). During this execution, s_4 consumes the token τ_1 from its input port `I` and produces an output token τ_4 to its output port `O`. Since the input token τ_1 was previously produced by the firing s_1, a token dependency between s_1 and s_4 can be defined (denoted as e_5). At time $t = 4$, the `Medium` actor fires the action `multiply` again (denoted as firing s_5). Also this firing read the token τ_2 from the input port `I` and wrote the token τ_5 to the output port `O`. Since τ_2 was previously produced by s_2, a new token dependency can be defined (denoted as e_8). Furthermore, since the firing s_5 read and wrote tokens from and to the same ports as the firing s_4, two new port dependencies can be defined. They are denoted as e_6 and e_7 with directions *read/read* and *write/write*, respectively. The execution of the entire program continues until time $t = 9$ and the described considerations can be used in order to build the remaining dependencies of the ETG.

4.2 Properties

This Section summarizes the main properties of an ETG. The objective is to demonstrate how these properties can be successfully exploited when exploring the design space of a program.

(s_i, s_j)	**Source**	**Target**	**Kind**	**Direction**	**Parameter**	**Attribute**
e_1	s_1	s_2	Variable	Write/Write	variable=`id`	initial=1 final=2
e_2	s_1	s_2	Port	Write/Write	port=`O`	
e_3	s_2	s_3	Variable	Write/Write	variable=`id`	initial=2 final=3
e_4	s_2	s_3	Port	Write/Write	port=`O`	
e_5	s_1	s_4	Token	-	count=1 source-Port=`I` source-Port=`O`	value=1
e_6	s_4	s_5	Port	Read/Read	port=`I`	
e_7	s_4	s_5	Port	Write/Write	port=`O`	
e_8	s_2	s_5	Token	-	count=1 source-Port=`I` source-Port=`O`	value=2
e_9	s_5	s_6	Port	Read/Read		
e_{10}	s_5	s_6	Port	Write/Write	port=`O`	
e_{11}	s_3	s_6	Token	-	count=1 source-Port=`I` source-Port=`O`	value=3
e_{12}	s_4	s_7	Token	-	count=1 source-Port=`I` source-Port=`O`	value=-1
e_{13}	s_7	s_8	Port	Read/Read	port=`I`	
e_{14}	s_7	s_8	Token	-	count=1 source-Port=`I` source-Port=`O`	value=-2
e_{15}	s_5	s_8	Port	Read/Read	port=`I`	
e_{16}	s_8	s_9	Token	-	count=1 source-Port=`I` source-Port=`O`	value=-3

Table 4.2 – Dependencies set $\mathbb{D}$ of the execution trace graph depicted in Figure 4.2.

4.2.1 Topological order

Since ETG is considered to be a $DAG(\mathbb{S},\mathbb{D})$, it is possible to define a partial order on the firing set $\mathbb{S}$. This topological order can be defined with a mapping function $l : \mathbb{S} \rightarrow \mathbb{N}$ such that:

$$s_i \leq s_j \Rightarrow l(s_i) < l(s_j) \tag{4.7}$$

It must be noted that a DAG can have different valid topological orders. In other words, given two valid topological mapping functions l_1 and l_2, it is possible that $l(s)_1 \neq l(s)_2$. As demonstrated towards the end of this Section, an ETG can express the maximum parallelism (potential parallelism) of the program. This property is strictly related to the fact that a DAG generally admits several valid topological orders.

4.2.2 Configuration-related dependencies

Representation of a program execution in the form of an ETG is, in principle, independent of the applied partitioning, scheduling and buffer dimensioning configurations. Hence, it is a generic model with the features allowing exploration of different design configurations. A problem that can be considered in this case is how to annotate the notion of specific configuration in such an independent generic model. This problem has been studied considering partitioning and scheduling for the purpose of automatic analysis and synthesis of dataflow programs, for example in [124]. This Section illustrates how different design configurations can be represented in the ETG by introducing further dependencies to the graph. This property is demonstrated using the same example of a dataflow program described in Section 4.1.3 and the set of design points summarized in Table 4.3. These dependencies result from the *combination* of all configurations related to a design point. It is important to remark, that these considerations are valid, when the considered actors are deterministic, that is, their execution is not time dependent.

Example (1)

Let's consider the 2 design points x_2 and x_4, as defined in Table 4.3. These two design points differ only by the partitioning configuration. In x_2 all the actors are assigned to one partition, unlike for x_4 where two partitions are defined. The scheduling and buffer dimensioning configurations are identical in both cases. For x_4, the firing set $\mathbb{S}$ has been obtained with the following order $\mathbb{S}(x_4) = \{s_1, s_4, s_2, s_7, s_5, s_3, s_8, s_6, s_9\}$. Considering the relationships between the firings, the set of dependencies $\mathbb{D}(x_4)$, as well as $\mathbb{D}(x_2)$ is the same as the original set $\mathbb{D}$, considered in Section 4.1.3. The corresponding ETG is also the one depicted earlier in Figure 4.2. The design configurations lead to a specific order of firings. The edges related to this order are introduced to the graph, as depicted in Figure 4.3d. These additional edges are depicted

Design point x_i	**Partitioning P**	**(static) Scheduling S**	**Buffer dimensioning B**
x_1	$P_1^1 = \{Sr, Sk, Md\}$	$S_1^1 = \{Sr, Sr, Sr, Sk, Sk, Sk, Md, Md, Md\}$	$B_1^1 = 3$ $B_2^1 = 3$
x_2	$P_1^2 = \{Sr, Sk, Md\}$	$S_1^2 = \{Sr, Md, Sk, Sr, Md, Sk, Sr, Md, Sk\}$	$B_1^2 = 3$ $B_2^2 = 3$
x_3	$P_1^3 = \{Sr, Md\}$ $P_2^3 = \{Sk\}$	$S_1^3 = \{Sr, Md, Sr, Md, Sr, Md\}$ $S_2^3 = \{Sk, Sk, Sk\}$	$B_1^3 = 1$ $B_2^3 = 1$
x_4	$P_1^4 = \{Sr, Md\}$ $P_2^4 = \{Sk\}$	$S_1^4 = \{Sr, Md, Sr, Md, Sr, Md\}$ $S_2^4 = \{Sk, Sk, Sk\}$	$B_1^4 = 3$ $B_2^4 = 3$
x_5	$P_1^5 = \{Sr, Md\}$ $P_2^5 = \{Sk\}$	$S_1^5 = \{Sr, Sr, Sr, Md, Md, Md\}$ $S_2^5 = \{Sk, Sk, Sk\}$	$B_1^5 = 1$ $B_2^5 = 1$
x_6	$P_1^6 = \{Sr, Md\}$ $P_2^6 = \{Sk\}$	$S_1^6 = \{Sr, Sr, Sr, Md, Md, Md\}$ $S_2^6 = \{Sk, Sk, Sk\}$	$B_1^6 = 3$ $B_2^6 = 3$

Table 4.3 – Summary of the design points considered for the program network in Figure 2.6. The actors: Source, Medium and Sink are denoted with Sr, Md, Sk, respectively.

with dashed arrows, so that they are not confused with the configuration-independent edges. They lead to the following partial ordered set $\mathbb{S}(x_4) = \{s_1 < s_4 < s_2 \leq s_7 < s_5 < s_3 \leq s_8 < s_6 < s_9\}$. It can be observed that when the dependencies are satisfied, firings of actors mapped on P_1^4 can be executed in parallel to firings mapped on P_2^4.

Example (2)

The same considerations can be applied to the design points which differ from each other only by the scheduling configuration. For instance, this is the case for the design points x_1 and x_2. In these cases, the following partial orders can be considered: $\mathbb{S}(x_1) = \{s_1 < s_2 < s_3 < s_4 < s_5 < s_6 < s_7 < s_8 < s_9\}$ and $\mathbb{S}(x_2) = \{s_1 < s_4 < s_7 < s_2 < s_5 < s_8 < s_3 < s_6 < s_9\}$. These orders lead to introducing additional edges to the graph, as depicted with dashed arrows in Figures 4.3a and 4.3b, respectively.

Example (3)

This example illustrates the additional dependencies introduced for two design points differing only by the buffer dimensioning configuration. Let's consider the 2 design points x_3 and x_4. In x_3 the buffer dimensioning configuration is defined as $B_1^3 = B_2^3 = 1$, opposite to x_4 where the buffer dimensioning configuration is defined as $B_1^3 = B_2^3 = 3$. The partitioning and scheduling configurations of x_3 and x_4 are the same. In these two cases notice that the partial orders are the same in both cases and $\mathbb{S}(x_3) = \mathbb{S}(x_4)\{s_1 < s_4 < s_2 \leq s_7 < s_5 < s_3 \leq s_8 < s_6 < s_9\}$. In consequence, the resulting ETG has exactly the same set of additional dependencies.

Example (4)

Although the difference in the buffer dimensioning configuration is not directly reflected in the *ETG* (*i.e.*, the resulting sets of additional dependencies are identical, as in the previous example), it must be emphasized that the buffer dimensioning determines the feasibility of the applied scheduling configuration. For instance, let's consider the design points x_5 and x_6 which result from the points x_3 and x_4 by changing only the scheduling configuration. The point x_6 represents a feasible design point, whereas the point x_5 is not feasible, because the specified scheduling configuration cannot be realized due to the buffer restrictions.

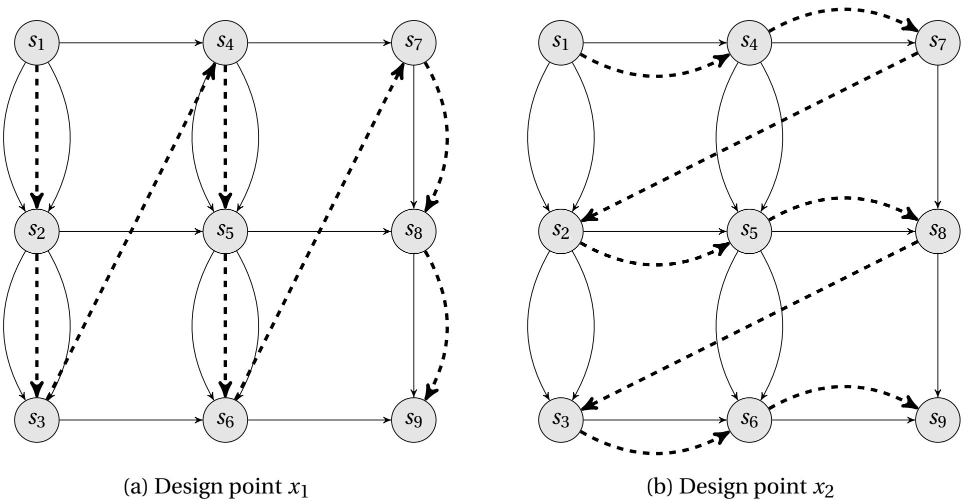
4.2.3 Capturing the dynamic behavior

The considered dataflow *MoC* assumes that the actors can be characterized by a dynamic behavior. For instance, they can be data-dependent. This data dependence can be illustrated with a simple example. Let's consider the *CAL* actor `Separator` defined in Listing 2.2. It consists of 2 actions: `A` and `B`, respectively. The firing conditions of both actions imply that one input token should be available in the input port `I`. However, action `A` is executable only if the value of the token is `val` ≥ 0 and action `B` if `val` < 0. Let's suppose that 2 input sequences are available in the input port `I`: $I_1 = \{1, 2, -3, -4\}$ and $I_2 = \{-5, -6, 0, -7\}$, respectively. Hence, the firing sequence $\mathbb{S} = \{s_1, s_2, s_3, s_4\}$ of this actor defines different action firings, as illustrated in Table 4.4. Changing the firing sequence can lead also to some changes in the dependencies set $\mathbb{D}$. Hence, the model such as *ETG* (in terms of firings $\mathbb{S}$ and dependencies $\mathbb{D}$) is valid for a given input data.

Firing	**Action**	
	I_1	I_2
s_1	A	B
s_2	A	B
s_3	B	A
s_4	B	B

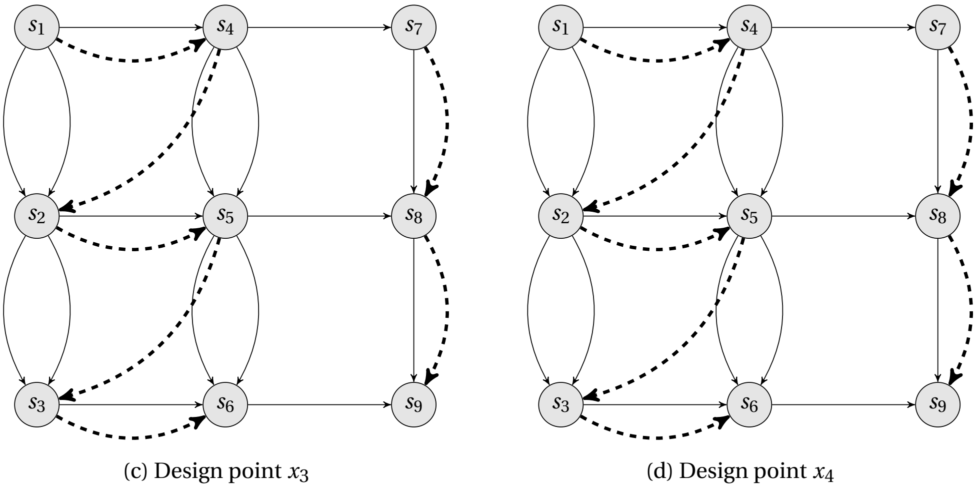
Table 4.4 – Firings sequence of the *CAL* actor `Separator` (defined in Listing 2.2), when two input sequences are available in its input port `I`: $I_1 = \{1, 2, -3, -4\}$ and $I_2 = \{-5, -6, 0, -7\}$, respectively.

Abstracting from this simple example and moving towards real size applications, it is essential to consider the *ETG* always in conjunction with a given input stimulus. Such a stimulus should be statistically meaningful so that the entire dynamic behavior of an application is captured. For instance, it should excite all parts of the application. Finding a high-quality design point for a given input stimulus does not guarantee that this particular point will remain high-quality



(a) Design point x_1

(b) Design point x_2



(c) Design point x_3

(d) Design point x_4

Figure 4.3 – *ETGs* of the *CAL* network depicted in Fig. 2.6. Dashed lines represent additional edges that model a particular partitioning, scheduling and buffer dimensioning configuration, as defined in different variants in Table 4.3.

for another set. Hence, analyzing a dynamic application and generating representative $ETGs$ should be based on a *set* of statistically meaningful data sequences (input stimuli) providing a syntax for the exploration.

4.2.4 Potential parallelism

The term *potential parallelism* is popularly linked to the law defined by Amdahl (often referred to as Amdahl's law or Amdahl's argument) [125]. In principle, it expresses the theoretical reduction in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. In the context of this work, the potential parallelism can be defined as the maximal achievable speed-up of a dataflow program versus a fully serial ($i.e.$, mono-core) execution. This maximal speed-up is related to an execution in *optimal* conditions, that is, when all the actors can work in parallel and the buffer size is unbounded. Hence, it corresponds to a maximally parallel execution of a program.

The information about the potential parallelism (in the context of a given input stimulus) is carried by the ETG. As described previously, the precedence relations between the firings are imposed only by the precedences related to the order of processing the data. For example, a token dependency defines that the firing that consumes given tokens can only be executed after the execution of the firing that produced these tokens. The same applies to the other types of dependencies. As such, the dependencies set $\mathbb{D}$ defines only a minimal information based on the data processing ($i.e.$, tokens, internal variables) and resource utilization ($i.e.$, ports, guards) that should be respected in order to obtain a correct program execution. The constraints imposed by the design configurations can only be modeled by introducing additional edges, as discussed in detail in Section 4.2.2.

The ETG without additional edges corresponds to an execution of the program using a fully-parallel configuration ($i.e.$, where each partition contains only one actor) and unbounded buffer sizes, because the scheduling choices are not constrained in any way. Hence, no additional dependencies (like the dashed lines in Figure 4.3) are present. Such a model corresponds to a fully *parallel* execution and is denoted as ETG_p. Another model is related to a fully *serial* execution and is denoted as ETG_s. It corresponds to the situations depicted in Figures 4.3a and 4.3b, when all actors are partitioned to the same unit and executed one after the other. Assuming that the graphs ETG_p and ETG_s have their respective values of makespans $k(ETG_p)$ and $k(ETG_s)$, the potential parallelism carried by the ETG can be defined as:

$$\mathbb{P} = \frac{k(ETG_p)}{k(ETG_s)} \tag{4.8}$$

The information about the potential parallelism is highly valuable from the perspective of DSE because it defines an *upper bound* on the performance achievable by the design points consisting of different sets of configurations. In contrast, considering the ETG with additional edges related to the configurations (as depicted in the examples in Figure 4.3) allows evaluating the parallelism of a given design point. Comparing these two values (potential parallelism and the parallelism of a given design point) provides an important indication about the quality of the design point with regards to the theoretical best-quality point available in a given design space. On the other hand, if the upper bound of the performance determined by the potential parallelism is not sufficient ($i.e.$, it does not satisfy the constraints), it might be necessary to perform the modifications of the program leading to a new design space with a different value of potential parallelism. This concept is discussed in detail in Section 7.7 and illustrated with experimental results in Section 10.7.

4.3 Conclusions

This Chapter presented the concept of execution trace graph which is used as an abstract model of a dynamic execution. The properties that make it an appropriate generic model for design space exploration to be performed on different platforms have been discussed. Special attention has been paid to the notion of dynamic behavior of the actors and to the information about the potential parallelism of an application that is carried by the ETG.

5 Architecture modeling

This Chapter describes the underlying profiling methodologies which are used in order to provide an abstract model of execution with the notion of time when referred to a given target platform. This process in encompassed in the "Profiling and analysis" stage of the design flow discussed in Section 3.5, as illustrated in Figure 5.1. The ETG is generated for a given CAL program and an input stimulus. The profiling consists of executing a CAL program on the platform in order to obtain the timing information injected into the ETG. This timing information can be retrieved by reading the values of the clock-cycles from hardware counters of the processor ([126]). Depending on the considered platform and its properties, different information can be extracted with a varying level of accuracy. The Chapter describes the profiling methodologies used for profiling of the two main types of platforms used in this work.

5.1 Abstract-to-timed translation

Time information is added to an ETG by defining for each firing and each dependency a corresponding time value. In this way, the abstract definition of the program execution is translated into the timed execution on a given platform. For this purpose, the ETG is transformed to a weighted graph which is a special type of labeled graph where labels are considered to be the, always positive in this case, numbers called **weights**. The **timed execution trace graph** ($TETG$) is formally defined extending the notation of the ETG as a $DAG(\mathbb{S}, \mathbb{D}, \Psi_{\mathbb{S}}, \Psi_{\mathbb{D}})$ where:

- $\Psi_{\mathbb{S}} : \mathbb{S} \rightarrow \mathbb{R}^+$ is the firings weight mapping function.
- $\Psi_{\mathbb{D}} : \mathbb{D} \rightarrow \mathbb{R}^+$ is the dependencies weight mapping function.

In other words, for each firing $s_i \in \mathbb{S}$ is assigned a time value called *firing weight* and defined

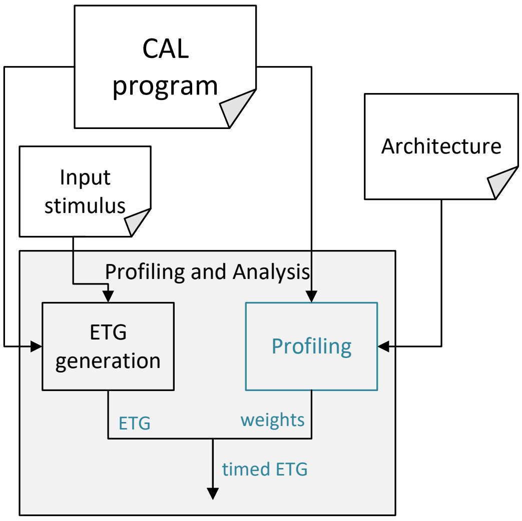


Figure 5.1 – System development design flow: profiling.

as $w(s_i) \geq 0$. Similarly, the *dependency weight* $w(s_i, s_j) \geq 0$ is defined for each dependency $(s_i, s_j) \in \mathbb{D}$.

Firing weight

A weight related to an execution of the firing models the entire time required for an execution of the action firing s_i. It consists of the following elements:

- the time spent on reading the input tokens;
- the execution of the algorithm;
- the time spent on writing the output tokens.

Dependency weight

A weight related to the dependency models the time required to make the dependency $(s_i, s_j) \in \mathbb{D}$ available to the target firing step s_j after the execution of the firing s_i has been completely performed. Just like the firing weight, it may also depend on the applied set of configurations. For different types of dependencies, this weight may model different factors. For example, if $(s_i, s_j) \in \mathbb{D}_t$ is a token dependency, then $w(s_i, s_j)$ can model the time required by the buffer to receive and make the corresponding tokens available. The same considerations can be made for state variable dependencies $(s_i, s_j) \in \mathbb{D}_v$ where the token is now a state variable and the buffer is a local memory region. Considering the FSM dependencies, the weight can model the time required for selecting the action by the internal actor scheduler. Furthermore, when introducing the configuration-related dependencies to model the scheduling configuration, the weights of these dependencies model the time required to select the actor by the partition scheduler.

Processing, scheduling and communication weights on the platform

Considering the profiling and modeling opportunities with limited memory resources, the practical model of a platform consists of 3 types of weights. The processing weights (also referred to as *action weights*) correspond to the time spent in the algorithmic part of a given action, calculated in terms of statistical properties (average, maximum and minimum values) among different firings of this action. The scheduling weights are related to the action selection by the internal actor scheduler. They are profiled and modeled differently for the two considered platforms. The communication weights are related to the process of reading/writing of the tokens over a given buffer. The weights are assigned to each buffer and calculated according to the information about the memory level that served the memory access and the latency related to this process.

5.2 Transport Triggered Architecture

The first platform is built as an array of *Transport Triggered Architecture (TTA)* processors (Fig. 5.2) [127]. It resembles the *Very Long Instruction Word (VLIW)* architecture with the internal datapaths of the processors exposed in the instruction set [128]. The program description consists only of the operand transfers between the computational resources. A *TTA* processor is made of functional units connected by input and output sockets to an interconnection network consisting of buses.

TTA architecture has several strengths: it enables intruction-level parallelism and reduces the registered file traffic [129]. The run-time hardware is simple and economical [130]. It allows also configuring the processors in several ways [131]. The platform contains a simple instruction memory without caches. Furthermore, it is a multiprocessor platform with no significant inter-processor communication penalty. The execution time of a program can be measured cycle-accurately and different runs provide exactly the same values.

The profiling of the processing times of an application is characterized with negligible overheads and with independence from the applied partitioning configuration. In fact, the profiling methodology operates on actors executed in isolation, that is, one actor at a time on a single processor core. The profiling is applied only once and then its results are explored in various

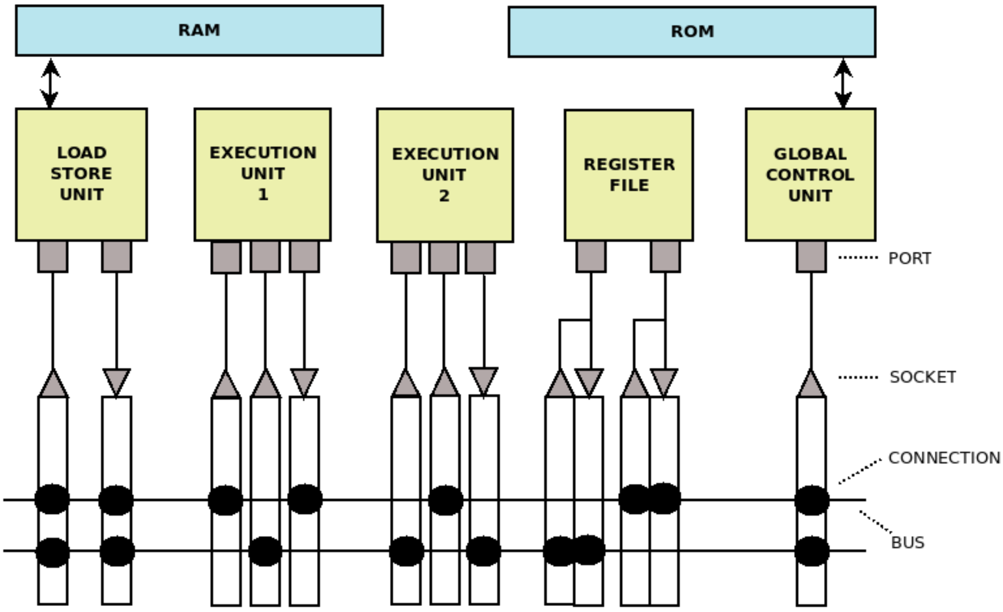


Figure 5.2 – Transport Triggered Architecture model.

configurations. It is a valuable property of the *TTA* architecture compared to profiling of other platforms, where the results usually depend on the partitioning configuration and may turn out to be invalid when other configurations are considered [132]. For these reasons, it is possible to build a simple model for this platform.

The profiling is performed on a cycle-accurate simulator [133], where the processor core is equipped with a special *time-stamp* hardware operation creating a record to an external file every time the operation is executed. The measurement is minimally intrusive, as it executes in one clock-cycle on a processor that is capable of executing multiple operations every clock-cycle. Figure 5.3 illustrates the placement of time-stamps. The clock-cycles elapsed between the `STAMP_10` and `STAMP_11` (`STAMP_20` and `STAMP_21`, respectively) correspond to the processing time of `action x` (`action y`, respectively). The clock-cycles between `STAMP_0` and `STAMP_1` correspond to the overall time spent inside an actor, excluding the maintenance of the buffers. By subtracting the processing times of all actions from this value, it is possible to calculate an overall intra-actor scheduling overhead. Since the information about scheduling overhead is not provided at the level of actions (for instance, the structure and complexity of the *FSM* of an actor is not taken into account), the overall value is distributed among the actions depending on their frequency of execution and incorporated into the action weights.

```
void actorA_scheduler() {
   STAMP_0();
   switch (fsm_state) {
      case state_1:
         if(action_x_guard() == true) {
            STAMP_10();
            action_x_body();
            STAMP_11();
         }
       case state_2:
          if(action_y_guard() == true) {
             STAMP_20();
             action_y_body();
             STAMP_21();
          }
    }
    STAMP_1();

    fifo_maintenance_calls();
 }
```

Figure 5.3 – TTA time-stamp placement: pseudocode.

5.3 Intel 86x64

Profiling of Intel 86x64 (further referred to as Intel) platforms is a much more challenging task than profiling the simpler *TTA* architecture for several reasons. First, the measured clock-cycles (CPU cycles) may vary depending on the availability of the required data/instructions in the data/instruction caches. Second, neglecting the communication cost will not be irrelevant for a good accuracy of the results. Moreover, it is not possible to measure the communication cost in a partitioning-independent way, because the results obtained for one configuration will not be valid for the other [132]. Hence, the actors cannot be executed in isolation (as for the case of *TTA*), but rather in a given configuration (partitioning, scheduling, buffer dimensioning).

Finally, execution of any process which is not written as a kernel module does not guarantee the exclusive ownership of the processor [134], which means that any interrupts may affect the accuracy of the results. In order to minimize this effect, the profiling is performed under the minimal Linux system with no unnecessary processes running and the hyper-threading and turbo-mode *(DVFS)* turned off from BIOS/UEFI.

The Intel-based PCs, which are the target platforms for several experiments discussed later in Chapter 10, are considered to consist of identical processors. In most cases, the profiling is performed for the mono-core configuration. The measurements of the clock-cycles consumed for the execution (or scheduling) of each action are performed via the benchmarking functions

`RDTSC_tick()` and `RDTSC_tock()` to start and stop the benchmarking, respectively.

Most Intel processors have a per-core time-stamp counter register and using the `RDTSC` and `RDTSCP` instructions (which read this register), Intel CPUs allow developers to keep track of every CPU cycle. Although the main utility of the `CPUID` instruction is to load the processor information into the registers, it is used along with `RDTSC` instruction to serialize the execution with no effect on program flow and guarantee that the benchmarking functions do not execute out of order. Separate intrinsic executions of `CPUID` and `RDTSC`, however, often result in large variances when used to benchmark the same piece of code. To solve this issue, Intel provides `RDTSCP` instruction, which performs both operations (reading the time-stamp counter register and loading the processor's info) in an intrinsic atomic instruction.

These two benchmarking functions are implemented using the in-line volatile assembly instructions which invoke the intrinsic instructions `RDTSC`, `RDTSCP`, and `CPUID`. Despite the aforementioned obstacles, it has been demonstrated in prior works that the profiling performed via these instructions is quite stable [135]. In order to support a systematic and automated benchmarking of the whole dataflow program on Intel platforms, the profiling utility is integrated in the *CAL* to C code generation of *ORCC* [70].

5.3.1 Processing weights

The action weights are calculated based on the measurements of the clock-cycles elapsed between `RDTSC_tick()` and `RDTSC_tock()` and consider the computational part of each action, excluding the scheduler and the management of buffers. The location of the `RDTSC_tick()` and `RDTSC_tock()` calls are depicted in Fig. 5.4.

The number of clock-cycles elapsed between the `RDTSC_tick()` and `RDTSC_tock()` are stored as a weight of a firing. The weights for each firing of a given action are stored as a list. The post-processing stage filters out the outliers, that is, the weights with extraordinary values that can result from the, mentioned earlier, interrupts occurring during the execution. After the filtering, the new values of μ and σ^2 are calculated. The final processing weight for each action corresponds to the calculated mean value. The filtering is performed according to the threshold. Any values exceeding the threshold are removed from the list of firings. The threshold is calculated according to the mean value (μ) and variance (σ^2), calculated for all profiled firings, using the following formula:

$$threshold = \mu + 2\sqrt{\sigma^2} \tag{5.1}$$

The weights remain dependent on the used configurations. All configurations (partitioning, scheduling, buffer dimensioning) impact the results, because they influence the availability

```
void actorA_action_x_body() {
   int i;

   RDTSC_tick();
   // Computational part of action’s body
   i = tokens_InputPort1[(index_InputPort1
    + (0)) % SIZE_InputPort1];
   tokens_OutputPort1[(index_OutputPort1
    + (0)) % SIZE_OutputPort1] = i;
    RDTSC_tock();

    // Update ports indices
    index_InputPort1 += 1;
    index_OutputPort1 += 1;
    rate_InputPort1 += 1;
 }
```

Figure 5.4 – Intel processing weights clock-cycles measurement: pseudocode.

of data in the caches throughout the execution. Additionally, the partitioning configuration impacts also the communication cost related to token exchange between the actors. Hence, the results remain the most reliable only for the set of configurations that were originally used for profiling. In order to use the results of profiling obtained for one configuration to explore other configurations, preliminary experiments demonstrated that the most accurate results can be obtained if the profiling is performed for a mono-core execution and relatively big buffers.

Figures 5.5 and 5.6 illustrate well the differences between the profiling data obtained for TTA and Intel platforms (using a realistic example of, described later in Section10.1.2, MPEG4-SP decoder) and justify the necessity of applying the filtering for the case of an Intel platform. Then, Table 5.1 summarizes the statistical values collected during the profiling in both cases and after applying the filtering.

Platform	**Samples**	**Average**	**Min**	**Max**	**Variance**
TTA	1600	549.73	547	567	24.37
Intel (Original)	1600	666.89	156	21224	5.13×10^6
Intel (Filtered)	1528	188.39	156	776	4.61×10^3

Table 5.1 – Statistical information obtained for a single action on the TTA and Intel platforms.

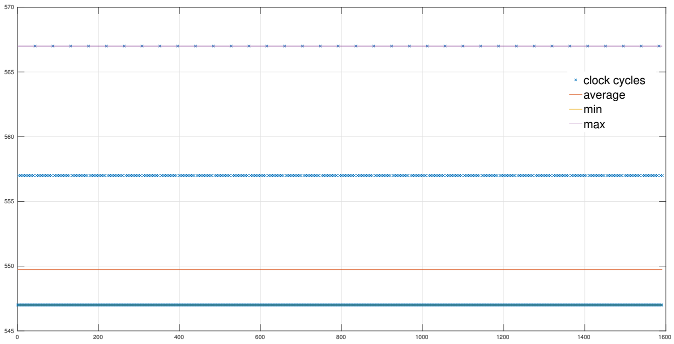
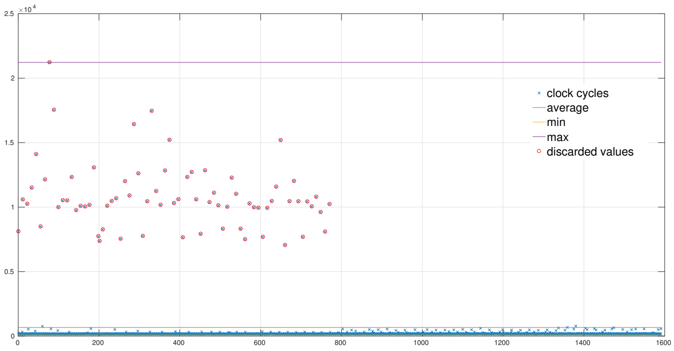
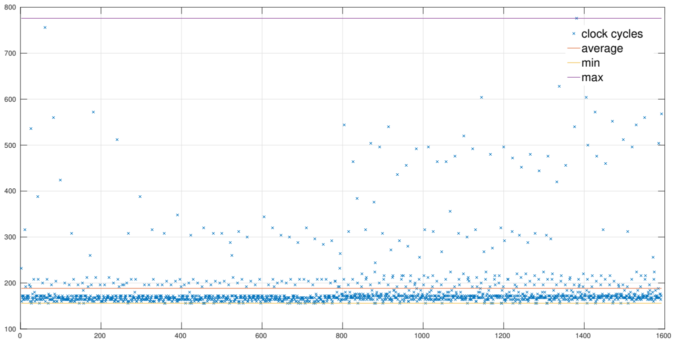


Figure 5.5 – TTA profiling data of a single action: collected timing data of 1600 firings.



(a) Unfiltered data. Red circles highlight the outliers (values higher than the $treshold$)



(b) Filtered data

Figure 5.6 – Intel profiling data of a single action: collected timing data of 1600 firings.

5.3.2 Scheduling weights

The measurement of the clock-cycles related to the scheduling is performed using the same instructions, averaging and filtering mechanism. The `RDTSC_tick()` and `RDTSC_tock()` calls related to the profiling of scheduling cost are depicted in Fig. 5.7. For the case of scheduling weights, apart from the action executed, it is also important to record the action which was executed before. Hence, the values of scheduling weights are always linked to a certain *transition* (source - target action set) and stored in an appropriate map. The filtering is applied within each transition. If a previously executed action was an action of the same actor, its name is stored in a map. An empty name means that the action was entered "from outside". This happens for the very first execution of a given actor or if the last executed action in a given partition belonged to another actor.

It must be emphasized that the scheduling weights obtained this way are related only to the internal scheduler inside each actor and do not include the cost of the scheduler inside a partition. The values of the weights obtained for each transition do not depend on the applied partitioning, scheduling nor buffer dimensioning configurations. The applied configurations (scheduling and buffer dimensioning), however, influence the occurrence of certain transitions. For instance, the set of possible transitions which are profiled can differ for *small* and *big* buffer sizes. In consequence, profiling one configuration and exploiting the results for exploration of other configurations means that certain transitions might not be profiled, hence they are not present in the resulting weights. Preliminary experiments demonstrated that the set of common transitions (overlapping between different configurations) is quite large, however the presence of some differentiating subsets cannot be eliminated.

5.3.3 Communication weights

The generation of communication weights is performed using the *numap* library [136]. This low-level memory profiling library is intended to be used for memory profiling in centralized shared memory systems. It was initially designed for profiling of the memory usage on Non Uniform Memory Access ($NUMA$) architectures and supports many micro-architectures from Intel. The implementation is portable, because according to the set of supported architectures, the correct hardware event to be used for memory read/write sampling is selected. The functions of the library count memory requests, generate memory samples and provide access to them for analyzing memory behavior of applications. The profiling can be performed for multiple $NUMA$ nodes and multiple threads.

Since different samples are available ($i.e.$, shared memory variables, global and local variables), the samples related to the buffers are extracted. Different runs of profiling are required for reading and writing data and for each operation different information is available. For each buffer present in the profiling report the following information about *reading* is provided:

- memory level that served the access (L1, L2, L3, LFB, Local RAM, Uncached Memory);
- the number of accesses for each memory level;
- the average latency related to serving the access;

- whether a memory *hit* or *miss* occurred.

In contrast, the information related to *writing* is limited to:

- whether a memory *hit* or *miss* occurred in level L1;
- the number of occurrences for each event.

The communication weight related to the *reading* operation for each buffer is calculated as a weighted arithmetical mean for the set of recorded events (memory level + hit/miss), where the values correspond to the average latency and the weight to a percentage of occurrences of certain events in the entire set of events, as illustrated in Equation 5.2. Table 5.2 presents a realistic example of the values obtained in the report. The calculation of the communication weight for this example is demonstrated in Equation 5.3. The communication weight related to the *writing* operation is calculated in the same way, based on just 2 samples (L1 hit, L1 miss) where some constants are specified as latencies. The value of these constants have been specified as maximum values (L1 hit, not-L1 hit, respectively) occurring for the reading events.

$$w_{jj'} = \frac{\sum_{i=1}^{n} l_i na_i}{\sum_{i=1}^{n} na_i} \tag{5.2}$$

Memory level	Number of accesses (na)	Average latency (l)	event
L1	1114	10.61	hit
L2	4	19.50	hit
L3	0	0.00	hit
LFB	6	137.08	hit
Local RAM	1	597.00	hit
Uncached memory	0.00	22	hit

Table 5.2 – Sample communication cost data.

$$w_{jj'} = \frac{1114 * 10.61 + 4 * 19.50 + 6 * 137.08 + 1 * 597}{1125} = 11.84 \tag{5.3}$$

At this stage it must be emphasized that profiling with *numap* succeeds according to a certain sampling rate (Chapter 5 of [132]). Depending on that rate (and, obviously, on the length of the input stimulus used for profiling), some buffers can be represented with more samples than others, hence, the generated communication weight is more accurate. It is also possible that for some buffers the communication weight cannot be generated because of an insufficient representation of this buffer in the samples used to generate the report.

5.4 Profiling accuracy

An important problem related to profiling is the accuracy of the retrieved information which translates directly into the accuracy of the performance estimation methodology or the design space exploration heuristics. In general, modeling the architecture relying on the profiling and, in particular, on the limited set of measurements coming from the platform (*i.e.*, the weight calculated according to the statistical properties) can be burdened with some errors. These general uncertainties of profiling result from multiple factors, such as: varying execution times resulting from interrupts, counting some instructions multiple times or intractable optimizations of the compiler [137].

Since different platforms provide different levels of accuracy, what is the acceptable level of discrepancy, for instance, in the performance estimation methodology based on profiling can be debated. For the purpose of design space exploration it can be assumed that the requirement is to ensure such a level of accuracy that permits correct evaluation of different design points and allows performing the moves in the considered design space efficiently and in a systematic way.

5.5 Conclusions

This Chapter presented a way to provide an abstract model of execution with the timing information, so that the execution essentials can be captured enabling a reliable comparison of different design points. The two discussed platforms are characterized with different properties, which contribute to the level of accuracy of the provided models. A *TTA* platform allows, in general, the creation of a very accurate model in a simple way, since the profiling results do not depend on the partitioning configuration and the communication cost can be neglected. For the case of Intel platforms, obtaining an accurate model is more difficult, since the partitioning-dependent communication cost must be introduced to ensure enough accuracy. Furthermore, the interrupts require applying additional filtering techniques.

```
void actorA_scheduler() {

lastSelectedAction = OUTSIDE;
RDTSC_tick();

// jump to FSM state
switch(FSM_state) {
case my_state_state_1:
goto l_state_2;
case my_state_state_2:
goto l_state_3;
}

// FSM transitions
l_state_2:
if (isSchedulable_action_x) {
RDTSC_tock();
currentSelectedAction = action_x;

// execute the action
action_x_body();

lastSelectedAction = currentSelectedAction;
RDTSC_tick();
}
else if (isSchedulable_action_y) {
RDTSC_tock();
currentSelectedAction = action_y;

// execute the action
action_y_body();

lastSelectedAction = currentSelectedAction;
RDTSC_tick();
}

l_state_3:
if (isSchedulable_action_z) {
RDTSC_tock();
currentSelectedAction = action_z;

// execute the action
action_z_body();

lastSelectedAction = currentSelectedAction;
RDTSC_tick();
}
}
```

Figure 5.7 – Intel scheduling weights clock-cycles measurement: pseudocode.

6 Design space exploration problem

Before attempting to solve the problem of design space exploration and optimization of dynamic dataflow programs, it is important to clarify what are the exact requirements, properties, decision variables and constraints. To the best of the author's knowledge, such a rigorous formulation of the design space exploration problem is still missing in the dataflow-related literature. Moreover, the available formulations mainly target only the SDF computation model which is characterized by several limitations, as discussed in Chapter 2. The current formulations respond also only partially to the demands of DDF, because when considering the partitioning and scheduling problems, they usually do not take into account the buffer dimensioning problem, nor its influence on the size of the partitioning and scheduling design space. Considering these subproblems, this Chapter presents a detailed formulation of the design space exploration (DSE) problem which follows precisely the demands of dynamic dataflow applications. The problem is formulated in terms of the decision variables, objective functions and constraints. It also considers how the general problem is specified ($i.e.$, extended with additional constraints) when an execution on a given type of platform is considered. An example illustrating the problem instance sizes is also included.

6.1 Related work

In general, the problem of partitioning and scheduling of parallel programs has been already extensively studied in the literature in its numerous variants, associated terminology, optimization functions and algorithms used to find close-to-optimal solutions [138]. However, the specific partitioning and scheduling problem (considering also the buffer dimensioning) that is faced when dealing with dynamic dataflow programs on heterogeneous architectures has not been yet clearly formulated in the literature.

The problem of the partitioning of standard- and multi-constraint graphs and directed acyclic graphs (*DAG*) are discussed, for instance, in [139]. The employed objective function is the minimization of the numbers of edges with endpoint vertices belonging to different subsets and does not explicitly consider the makespan (total time) of an application execution. This variant is partially handled in [140], where the contributing *weights*, considered constant, are added to the edges. Several strongly simplifying assumptions are also presented in this formulation, such as full connectivity of available processors, contention-free communication and homogeneity of the processors.

An interesting approach is to handle both: the program and the target architecture as graphs that need to be embedded in each other [141] or to extend the *DAG* definition by a description of the dependencies between the tasks [142] or the delays assigned to the edges [143]. However, both of these approaches assume that the dependencies occur at the same level, where the partitioning and scheduling are performed. For instance, the precedence constraints occur directly between the partitioned/scheduled objects (or actors).

To explicitly approach the partitioning problem, when dealing with heterogeneous platforms, a formalism assigning different processing costs to the tasks executed as software or implemented in a hardware component [144] and defining a heterogeneous multi-core system model [145] can be considered. In both cases, however, it is not taken into account that different processor families (*i.e.*, software or hardware) may imply some eligibility constraints (*i.e.*, some tasks cannot be performed on certain machines and, also, the communication between them can be more complex and constrained). Using a similar formalism on the application and on the architecture model, the partitioning and scheduling problem could be formulated as an assignment and execution of tasks that respect the given deadlines [146]. However, such an approach is difficult to apply in the dataflow domain, where the firings are not time-aligned and task deadlines do not exist.

Summarizing various formulation variants available in literature, there are two important novelties of the rigorous problem formulation provided in this Chapter. First, it includes also the problem of buffer dimensioning, which is considered equally important to be solved together with partitioning and scheduling in order to perform an efficient design space exploration. Furthermore, it does not impose any particular order or priority for finding the solutions to the aforementioned subproblems. Second, since it considers the most expressive (dynamic) dataflow programs, it describes the program execution in the most detailed way, that is, using action firings to express the dependencies between different dataflow components and constraints. Thanks to such an exhaustive description, it is possible to model and thoroughly analyze a given program.

6.2 Underlying optimization problems

The design space exploration problem consists of partitioning, scheduling and buffer dimensioning. These subproblems are illustrated in Fig. 6.1. The solutions (configurations) applied to these subproblems lead to defining a fixed execution order of the firings that compose the execution and hence restrict the topological order in the execution trace graph by introducing the configuration-related dependencies, as described in Section 4.2.2.

Defining such an execution order for the case of an ATS program is much more complex than for the case of KPN or DPN programs. For a KPN program the execution order is defined directly for the set of processes composing the program [84]. For a DPN program it is a sequence of actor firings the execution order has to be defined for [147]. The case of ATS programs requires defining the execution order for a sequence of atomic steps, where each step consists of an execution of a firing function controlled not only by the availability of input tokens, but also by state variables and priorities, as summarized in Section 2.1.4.

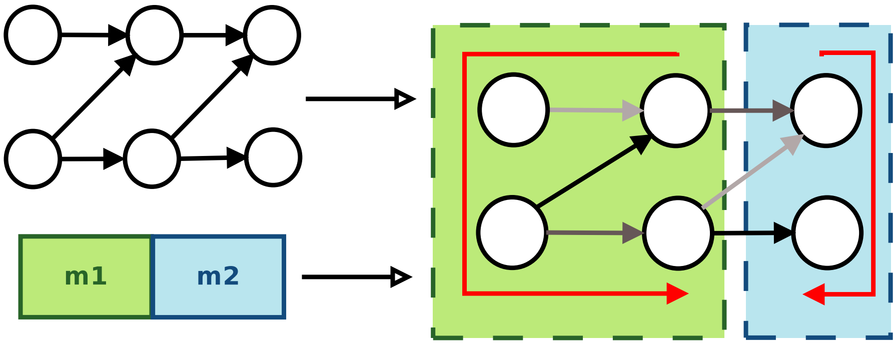


Figure 6.1 – The partitioning, scheduling and buffer dimensioning subproblems.

6.2.1 Partitioning

Using the terminology coming from the *production field*, the problem is to assign n jobs (corresponding to the action firings) to m parallel machines (also referred to as partitions or processing units). Each job j has an associated processing time (or an *action weight*) p_j and it belongs to a group (or an actor) g_j. There are l possible groups, denoted as $g_1, g_2, \ldots, g_k$. If $g_2 = 3$, it means that job 2 belongs to group g_3. Each group g_j can be divided into subgroups, where all jobs have the same processing times and thus can be identified with different executions of the same action. Between some pairs $\{j, j'\}$ of incompatible jobs (*i.e.*, with $g_j \neq g_{j'}$) is associated a communication time $w_{jj'}$. It is subject to a fixed quantity $q_{jj'}$ of information (or the number of tokens) that needs to be transferred. The size of this data is fixed for any subgroup.

The partitioning problem can be represented by an acyclic directed graph $G = (V, A)$, with the vertices (or nodes) set V and the arcs set A. Each vertex or node j represents a job and each arc (j, j') (between two nodes of the same group or not) represents a precedence constraint. With each arc (j, j') such that $g_j \neq g'_j$ is associated a communication time (or a *communication weight*) $w_{jj'}$. With each arc (j, j') such that $g_j = g_{j'}$, no weight or cost is associated. It can be observed that the relative order of execution of the nodes belonging to the same group is quite constrained ($i.e.$, the decision space is very restricted for such arcs) and is, in fact, imposed by the input stimulus used to build the graph. Finally, a *group* constraint can be defined, which implies that all jobs belonging to the same group have to be processed on the same machine. In other words, all executions of a particular actor must be partitioned to the same machine.

6.2.2 Scheduling

Typically, only one job can be executed at a time on one machine. Hence, for each assignment of jobs to the machines, the order of execution ($i.e.$, the sequencing) of jobs $S = \{j, j', \ldots\}$ must be decided in each machine. In practice, the sequencing of jobs within one group is quite constrained. Therefore, it can be reasonably assumed that there is almost no impact on the resulting makespan if some permutations are performed in the sequencing of each group. In this case, the scheduling problem can be limited to choosing at each step a group for which a node should be executed. The eligibility for execution for each node is determined by the availability of the necessary input tokens and spaces in the outgoing buffers. Depending on the internal nature of each actor ($i.e.$, static or dynamic) and on the underlying structure of its nodes, an optimal static order may or may not exist. In the situation where there are several jobs available for an execution on one machine, the selection of one of them is very sensitive to the solution's quality.

The sequencing must take into account two constraints. The *precedence* constraint (j, j') means that the job j (plus the associated communication time) must be completed before the job j' is allowed to start at the time point $T_{start}(j')$. The *setup* constraint requires that for each existing arc (j, j') involving nodes from different groups, a setup (or communication) time $w_{jj'}$ is occurring. In contrast with the job scheduling literature [148], one can observe that a setup time also occurs if the involved jobs are assigned to two different machines.

6.2.3 Buffer dimensioning

Each communication channel i (buffer in the network) that the information (tokens) is being transmitted through is bounded by B_i, so that a configuration $B(b_i) = B_i$ is specified. More precisely, the sum of the $q_{jj'}$'s along any arc assigned to a particular buffer i cannot exceed B_i. This size must be taken into consideration when evaluating the execution eligibility of a job j, because if the necessary space is not available, this job cannot be executed (*i.e.*, $j_S \neq j$). The optimal size should be set independently for each buffer in the network so that the delays of job executions arising from unavailability of the space in the buffers are minimized.

6.3 Target platforms

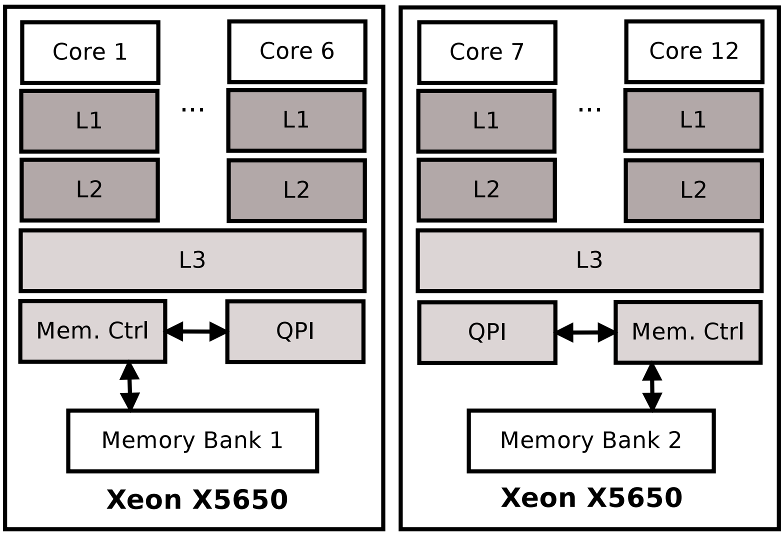
6.3.1 Homogeneous platforms

In the case of a homogeneous platform, each job j must be performed on any of the m parallel identical processing units (machines). The associated processing time p_j (*action weight* on the platform) is thus the same on each processing unit. A group r_i is assigned to each machine i. This definition remains consistent with the construction of the *Non-Uniform-Memory-Access* (*NUMA*) architectures, where the processing units (cores) are grouped and connected to different memory banks [149], as depicted in Fig. 6.2a for the case of Intel Xeon X5650 (example of a *NUMA* architecture) considered in this work as a homogeneous platform.

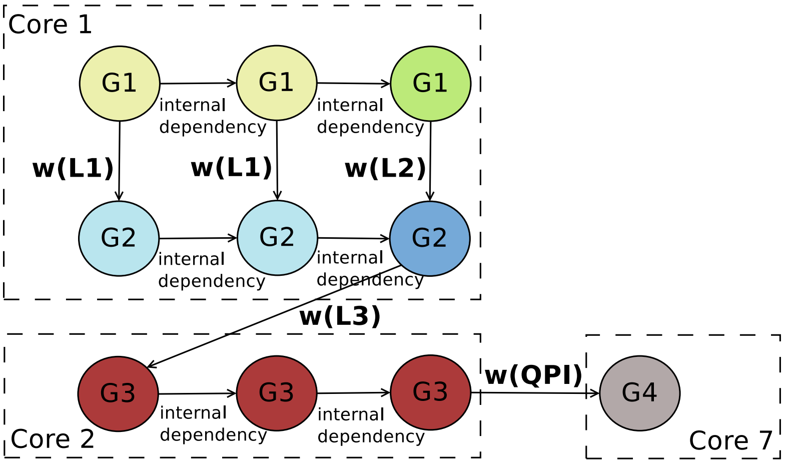
The communication time $w_{jj'}$ required for transferring a given number of tokens consists of the product of two elements: the (previously described) quantity $q_{jj'}$ and the variable time $w_{jj'}(h(j), h(j'))$ needed to transfer a single unit of information (where $h(j)$ denotes the machine the job j is performed on). The value of $w_{jj'}$ can belong to one of the three cases: (1) j and j' are scheduled on the same machine (communication via the L1-L3 caches or the local memory); (2) j and j' are scheduled on machines of the same group (L3 cache or local memory); (3) j and j' are scheduled on machines from different groups (remote memory) [132]. It can be assumed that case (3) will always introduce much higher values of $w_{jj'}$ than cases (1) and (2), whereas case (2) is likely to have a higher latency than case (1). This will depend, however, on the communication demands of the actors in a specific partitioning configuration. Figure 6.2b presents a sample assignment of four groups (actors) to a set of homogeneous machines corresponding to the mentioned earlier example of the *NUMA* architecture.

6.3.2 Heterogeneous platforms

Heterogeneous platforms imply considering different families of processing units. In each family, all processing units (machines) are identical, but the processing units of one family are not necessarily faster than the processing units of another. Typically, there are two families: HW (for hardware) and SW (for software). If only family SW is considered, the extended problem is reduced to the problem of homogeneous machines described previously. Otherwise, a job j has the same processing time on all the SW processing units (denoted $p_j(SW)$), and another processing time on all the HW processing units (denoted $p_j(HW)$). The actors



(a) Platform example: Intel Xeon X5650.



(b) Sample assignment.

Figure 6.2 – Homogeneous platform.

assigned to the HW processing units work in parallel. Thus, the scheduling problem for a HW subset of machines is eliminated without having any impact on the makespan. Figure 6.3a illustrates the basic construction of Xilinx Zynq 7000, which is an example of a heterogeneous platform. The two *ARM* machines, along with the associated memories (L1, L2, *DRAM*) belong to the SW family, whereas the *AXI Masters* component denotes the set of HW processing units. Handling heterogeneous platforms introduces different figures of merit for the communication time. The $w_{jj'}$'s are all equal to 1 (small value) if the involved groups are assigned to HW (assuming internal communications for hardware modules). The $w_{jj'}$'s are represented with different levels of values (depending on the assignment to the processing units, similar to the homogeneous platform case) if the involved groups are assigned to SW. If one group is executed on HW and the other on SW, $w_{jj'}$ depends on the amount of information to be sent.

The constraints introduced in this case are mostly subject to the HW family. First of all, an *eligibility* constraint occurs, meaning that a HW processing unit cannot perform all the jobs ($i.e.$, there is a set $d(i)$ of unsupported operations for each processing unit i), for instance the floating point operations. Furthermore, there is a *capacity* constraint due to the limited memory size of the HW family. Each group g (buffer b) has a memory requirement of $mem(g)$ ($mem(b)$), respectively. The sum of $mem(g)$ and $mem(b)$ for all groups and buffers assigned to the HW processing units cannot exceed a given limit. Finally, the *buffer mapping* constraint implies that the number of possible connection paths cnp between SW and HW (*HP* components in the aforementioned example) is fixed to HP_{max}. The maximal number of buffers that can be mapped to one connection path is fixed to B_{max}.

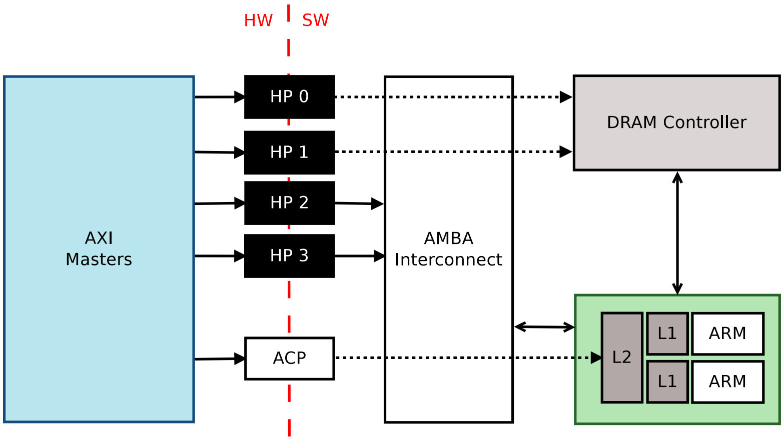
Figure 6.3b presents a sample assignment of four groups (actors) to the, mentioned previously, example of a heterogeneous platform.

6.4 Formulation of the design space exploration problem

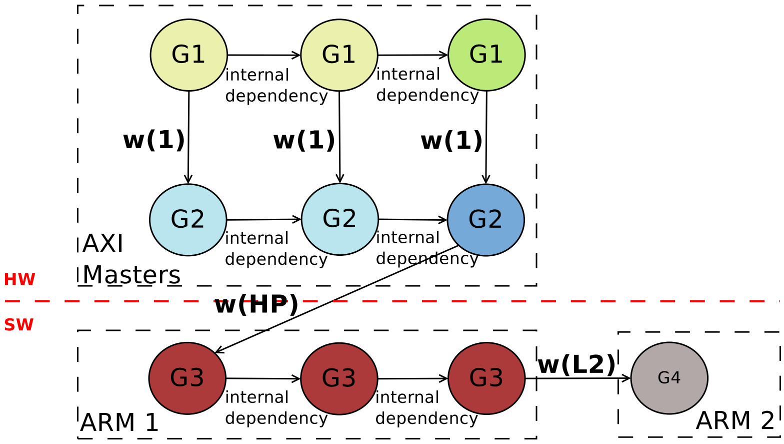
The properties of the underlying (partitioning, scheduling and buffer dimensioning) problems can be summarized with a formulation provided below. One of the possible objective functions is the makespan, which corresponds to the completion time T_{end} of the last performed job (denoted as j_{last}).

Decision variables: $\forall j : P(j) = h_i$, $\forall h_i : S = \{j, j', \ldots\}$, $\forall b_i : B(b_i) = B_i$ (each job is assigned to a machine, each machine has an execution order of the jobs, each buffer has a finite size)

Objective function (example): $\min(T_{end}(j_{last}))$ (minimization of the completion time of the last performed job)



(a) Platform example: Xilinx Zynq 7000.



(b) Sample assignment.

Figure 6.3 – Heterogeneous platform.

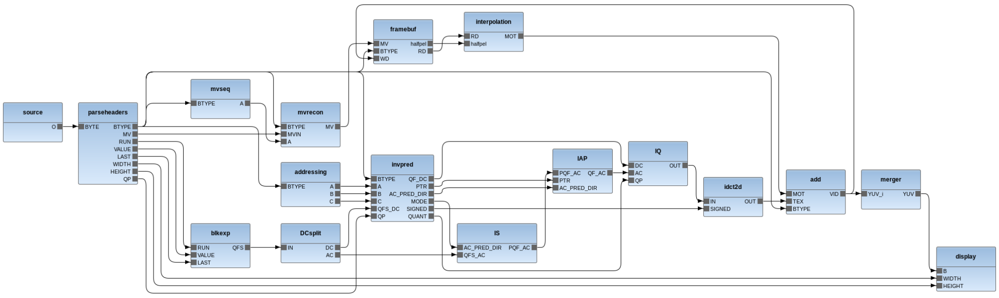
Constraints:

- $g_j = g_{j'} \Rightarrow P(j) = P(j')$ (*group*: firings belonging to the same actor must be partitioned to the same processing unit)
- $j \prec j' \Rightarrow T_{start}(j') \geq T_{end}(j)$ (*setup*, *communication*: the succeeding job can be executed only after the termination of the preceding job, including the communication time)
- $j_S = j \Rightarrow \sum_{q_{b_i}} + tokens(j) \leq B_i$ (*buffer capacity*: a job can be scheduled only if there is a sufficient space in its outgoing buffers)
- $P(j) = h_i \Rightarrow j \notin d(i)$ (*eligibility*: a job cannot be assigned to the processing units that does not support all of its operations)
- $\sum_{g \in HW} mem(g) + \sum_{b \in HW} mem(b) \leq size(HW)$ (*capacity*: the sum of the memory requirements of the actors and the buffers partitioned to hardware cannot exceed the limit)
- $\sum_{cnp \in HP} \leq HP_{max}$, $\sum_{b \in cnp} \leq B_{max}$ (*buffer mapping*: the number of paths between the HW and SW components and the number of buffers that can be mapped to one connection are limited)

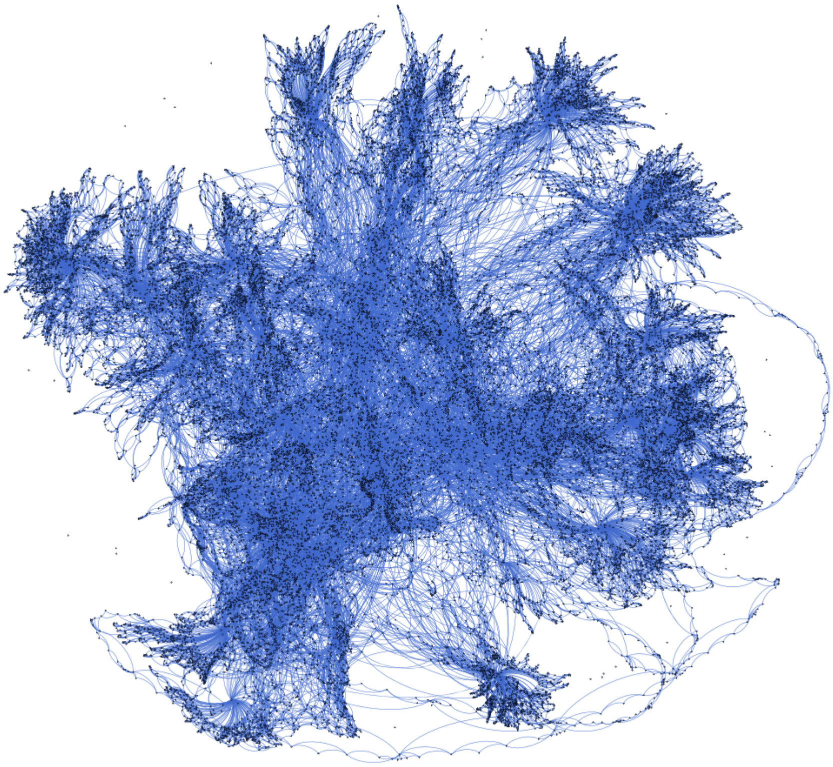
6.5 Problem instance sizes

Taking into consideration the typical instances of the problem, it is possible to define some practical boundaries on the size of the input data. The small instances start from: $n \in [200,000;500,000], m \in [2,20], l \in [5,40]$, whereas the large instances can range up to $n \approx 1,000,000,000, m \approx 500, l \approx 500$. Such instance sizes are huge when compared to the complexity of problems found in the production literature [148] and the graph partitioning literature [150], which makes it somewhat difficult to use heuristics directly taken from these fields to design a solution method for the dataflow design space exploration problem.

Comparing the classical problem formulation (where the relationships between the actors, not between the action firings are analyzed) with the problem formulation introduced in this Chapter, a huge difference can be demonstrated in the problem instance sizes, but also in the level of detail that is considered in both cases. Taking an example of a real dataflow application (MPEG-4 SP video decoder [151]), Fig. 6.4a illustrates the network consisting of actors and buffers, which is a graph to be partitioned using the classical problem formulation. It consists of 17 nodes and 38 arcs. In comparison, Fig. 6.4b presents the rendered *ETG* generated for this application using an input sequence consisting of only a few frames. Taking into consideration all of its firings and the dependencies between them, it consists of 176,649 nodes and 1,609,543 arcs.



(a) Program to be partitioned: MPEG-4 SP decoder.



(b) *ETG* generated for the program network.

Figure 6.4 – Dataflow program representations.

6.6 Conclusions

This Chapter formulated the design space exploration problem with regards to the decision variables, objective functions and constraints. It discussed in detail the underlying partitioning, scheduling and buffer dimensioning problems. It considered the problem for the case of homogeneous and heterogeneous architectures and discussed typical problem instance sizes. Analyzing the provided formulation and the illustration of the instance sizes, the complexity of the design space exploration problem can be realized. First, each of the subproblems, even when tackled separately, is NP-complete. Nevertheless, they have to be considered jointly in the process of exploration, since defining a solution to one of them, constraints the space available for the others. Second, as illustrated with an example, the number of the design points to be explored is huge and expands along with the increase of the design and architecture complexity. Finally, depending on the architecture, several constraints must be taken into account, that make the exploration process even more troublesome.

7 Exploration of multiple multidimensional design spaces

Following the definition of the design space exploration problem presented previously, this Chapter describes the concepts of design points, design spaces and transitions between them. It introduces a formulation capable of capturing the multidimensionality of the DSE problem and discusses possible optimization scenarios. An example demonstrates the complexity of the design space of DDF programs which has a direct impact on the efficiency of the solution methods that can be developed. The formal definitions of design spaces and transitions between them is comprised in the Variable Space Search (VSS) methodology enabling systematic improvements of a program. The novel contributions of this Chapter are preceded by an overview of related works discussing some similar aspects to the proposed VSS methodology.

7.1 Related work

Design space exploration of parallel programs ($i.e.$, streaming applications, microprocessors) is a problem widely described in the literature. Depending on the type of application, the considered *MoC*, the target platform and the objectives, the problem can be formulated according to different decision variables, constraints and objective functions. Nevertheless, in most cases, it is possible to identify some common challenges, such as the exponential number of configurations in the number of design variables and a nonlinear interaction between them [152]. Furthermore, multiple objective functions might often conflict with each other [153]. The overview presented in this Section is complementary to the discussion of dataflow-oriented frameworks, presented earlier in Chapter 3 and, in particular, in Section 3.3.

7.1.1 Design space exploration variants

The exploration aims at finding a configuration optimizing a desired objective function, where a configuration consists of a set of parameters. The number of parameters determines the number of dimensions in the design space. The parameters may be related to the program configurations (*i.e.*, mapping of processing kernels to the available hardware/software elements, dimensioning the buffers between the kernels), but also to the architecture parameters, such as custom datapath designs, cache sizes and instruction sets [154]. The exploration can be performed according to different optimization criteria. In [155], a search is described for an assignment of program components to hardware and software components, so that a trade-off between the performance and the code size is achieved. In [156], complexity of the implementation is introduced as a design variable, and a trade-off between the performance and the complexity is searched for. In this case, the design space consists of different/similar designs and their fitness to the aforementioned optimization criteria is measured. The work discussed in [157] considers a mapping of *KPN* processes, where binary decision variables are introduced to represent the mapping of the *KPN* nodes and edges to the processing- and memory elements, respectively. The optimization considers three criteria: performance, power consumption and cost. The same optimization criteria are considered in [158], where a solution method for the mapping decision problem represented as a multi-objective combinatorial problem is proposed. A performance and power trade-off is an objective of the exploration described in [159]. In that work, the design space results from adjusting parameter values for a fixed application mapped onto the *SoC* architecture. The exploration is performed using the Y-chart consisting of the architecture, applications and performance measures.

An interesting formulation of the *DSE* problem can be found in [160, 161]. The set of configurations that must be specified consists of an allocation of the architecture components, a binding of the processes to the components and defining a scheduling. It is assumed that *all* solutions can be described and explored, but some of them are infeasible. Hence, the exploration aims at establishing the feasible solutions and returns a set of solutions of different quality with regards to different design objectives. The exploration is performed using Multi-Objective Evolutionary Algorithms (*MOEAs*). Since the size of the design space is usually huge, an exhaustive exploration is possible only for very small instances of the problem. For any larger instances, it must be decided how the design space should be explored. One possible option is to perform a random sampling, which can provide an unbiased view of the space [156]. In contrast, using heuristics instead of a random sampling may reduce the size of the space by rejecting unsatisfactory solutions followed by identifying solutions which are best in terms of certain objective functions [162]. Further techniques can be also developed to reduce the size of the space and hence improve the exploration efficiency (*e.g.*, dedicated filtering techniques, as proposed in [163]). The aforementioned work ([159]) points to an opportunity for reducing the size of the explored space by identifying dependencies between the parameters. Hence, the exploration can be performed only for a subset of parameters, according to the created dependency model. Another approach proposed in [154] is to decompose the space into subproblems. Due to such branching, different problem blocks can be solved independently from each other.

An important aspect of design space exploration is a performance estimation which should allow correctly evaluating different solutions and enabling performing moves (*i.e.*, minor structured modifications) from one solution to another. Since such an evaluation is often time consuming, an estimation can be based on two different models, as described in [164]. One model is more accurate, but time consuming, whereas the other is quicker and simpler, but less accurate. This type of trade-off between accuracy and speed is well-known in other fields, especially if simulation is required to accurately evaluate a solution (*e.g.*, [165]). Another idea is to use a visualization of the space to help locate the optimal points [166]. It is, however, limited only to three dimensions.

Apart from establishing a, hopefully, close-to-optimal solution, especially for the case of multi-criteria objective functions, it might be important to provide a *set* of high-quality solutions. In [159], a set of Pareto-optimal configurations is explored, where each solution is better than the others according to at least one criterion. On the other hand, the Multi-Criteria Decision Making (*MCDM*) introduced in [164] performs a ranking of solutions rather than choosing the best one.

Summarizing different ideas of reducing the size of the space or making the exploration process more efficient, it has to be concluded that the discussed approaches cannot be really adapted to the design space exploration problem discussed in this work. The considered configurations remain tightly connected and a setting applied to one of them may strongly affect the exploration opportunities of the others. Hence, the targeted problem remains in any case NP-complete and multidimensional.

7.1.2 Bottlenecks in design space exploration

The design space exploration process can be supported by means of bottleneck analysis. According to [167], the bottlenecks of an application are defined as the factors identified to affect the length of the critical path, where different factors can have different impacts, leading to a ranking of bottleneck factors. In that work, the considered *DSE* problem consists of specifying the microprocessor parameters in two stages. First, the most impacting parameters are identified and second, search algorithms are used to find the, hopefully, optimal design point. The critical path analysis is performed based on a dependency graph and a cycle-accurate simulator. The information about the most impacting bottlenecks is used in order to drive the optimization algorithms. It is also verified that a bottleneck analysis can be particularly useful for optimization algorithms operating on a non-completely random basis (*e.g.*, tabu search, simulated annealing).

Another bottleneck-based approach for design space exploration is discussed in [168], where media processing systems represented as *SDF* graphs are considered. The exploration process consists of finding the mappings of the application onto the components of the architecture, and then dimensioning the architecture in order to achieve a final trade-off between the throughput and the resource utilization. Bottlenecks are defined as resources required to be increased in order to improve the application throughput. In principle, the flow discussed in that work has some significant similarities with the methodology described in this Chapter. In a similar way, possible design configurations are explored in order to find a set of high-quality solutions, then the bottlenecks are analyzed and, possibly, optimized in order to enable further improvement of the throughput. The important difference is that [168] relates bottlenecks to the architecture dimensions that are increased.

In the methodology introduced in this work, the definition of bottlenecks is wider, because they can be related to both, design and platform. For example, a bottleneck related to the design is a long sequential processing part, whereas a bottleneck related to the platform is the bandwidth between the partitions. The bottlenecks are identified more precisely in Section 7.7 for the design cases analyzed in the experiments, after having defined the optimization scenarios. In the analysis process of an application, the emphasis is put on the bottlenecks related to the parts of program implementations. In consequence, it leads to a throughput improvement resulting from optimizations inside the program implementation with an unchanged configuration of the target platform. Furthermore, the considered design space is much more complex compared to the referenced work, because it also includes the scheduling and buffer dimensioning, as required for the case of dynamic dataflow programs.

7.1.3 Variable space

The concept of a search performed in variable spaces has been already successfully applied to the NP-complete problem of graph coloring [169]. In that case, the main idea is to consider several search spaces, with various solution representations, neighborhood structures and objective functions, and move to another space if the search is blocked at a local optimum in a given space. Different spaces are defined based on the formulations of the problem differing, for instance, in terms of the constraints that can be relaxed in one space, but satisfied in another one. Every time a transition between the spaces is performed, a high-quality solution established in one space is translated into a corresponding solution in the new space. The flow proposed in this work uses the concept of multiple search spaces in order to find a high-quality design point in global terms, that is, among all visited design spaces.

To the best of the author's knowledge, this idea has not been yet properly studied nor formalized for the purpose of design space exploration of dynamic dataflow programs or even, more generally speaking, parallel system implementations. However, among the related works, some hints indicating the possible advantages of such an approach can be found. For instance, the work described in [170] points out that creating a new search space can improve the effectiveness of the search, because it can increase the likelihood that the search will arrive at a *correct* path. Similarly, the results of [171] demonstrate that the search space should be expanded along the variables that are most likely to positively impact the design objectives. It is also emphasized that a requirement of defining a new design space might result from the changeable design objectives being constantly updated during the exploration.

7.2 Multidimensional design space definition

Considering the three underlying problems described in the previous Chapter, let a design point X in the design space be a 3-tuple $X = (P^X(m^X), S^X, B^X)$, where $P^X(m^X)$ refers to a partitioning configuration with m^X machines, S^X is a vector referring to the scheduling configuration on each machine, and B^X is a vector where each component refers to the size of its associated buffer. Vectors S^X and B^X have the following structures:

- $S^X = (S_1^X, S_2^X, \ldots, S_{m^X}^X)$, where S_i^X refers to the scheduling configuration on machine i;
- $B^X = (B_1^X, B_2^X, \ldots, B_n^X)$, where n is the number of buffers (which is the same for any design point in a given design space) and B_i^X refers the size of buffer i.

A *point-to-point transition* (or a *move*) in a given design space is a modification of any of the components describing a design point. In the case of vectors S and B, the modification involves changing at least one element. Given the two design points X and Y, it can be stated that: $X \neq Y$ if $P^X(m^X) \neq P^Y(m^Y)$ or $S^X \neq S^Y$ or $B^X \neq B^Y$. The structural difference between X and Y can be denoted as $\Delta(X, Y)$. It is defined as:
$\Delta(X,Y) = \{(P : P^X(m^X) \rightarrow P^Y(m^Y); S_i : S_i^X \rightarrow S_i^Y, \forall\, i \in \{1, \ldots, \max(m^X, m^Y)\}; B_j : B_j^X \rightarrow B_j^Y, \forall\, j \in \{1, \ldots, n\})\}$.
Every move has an associated difference value of the objective function, denoted as $\Delta f(X,Y) = f(Y) - f(X)$. If a transition involves a modification of only one component (*i.e.*, either partitioning, scheduling or buffer dimensioning), the design points belonging to this transition can be called *neighbor design points*.

The notation $B_i^X \nearrow B_j^X = A$ ($S_i^X \nearrow S_j^X = SC$) means that all buffers (machines) between i and j have the same size A (the same scheduling configuration SC), respectively. Consider the following example:

- $X = (P = pconf1(3); S_1^X = sconf1, S_2^X \nearrow S_3^X = sconf2; B_1^X \nearrow B_2^X = 2, B_3^X \nearrow B_{20}^X = 1024)$; $f(X) = 1000$
- $Y = (P = pconf2(8); S_1^Y \nearrow S_4^Y = sconf1, S_5^Y \nearrow S_8^Y = sconf3; B_1^Y = 4, B_2^Y = 2, B_3^Y \nearrow B_{20}^Y = 2048)$; $f(Y) = 100$

The two points X and Y differ in the partitioning configuration, which is spanned on different numbers of machines (3 and 8, respectively). Hence, the length of the vector S is different in both cases. For machines 2 and 3, the scheduling configuration changes from $sconf2$ to $sconf1$, whereas the others appear only in the design point Y. As for the buffers, only buffer 2 remains unchanged, whereas the others change. These two points have also a different value of the objective function. All these differences can be expressed as:

- $\Delta(X,Y) = (P : pconf1(3) \rightarrow pconf2(8); S_2 \nearrow S_3 : sconf2 \rightarrow sconf1, S_4 : 0 \rightarrow sconf1, S_5 \nearrow S_8 : 0 \rightarrow sconf3; B_1 : 2 \rightarrow 4, B_3 : 1024 \rightarrow 2048)$
- $\Delta(Y,X) = (P : pconf2(8) \rightarrow pconf1(3); S_2 \nearrow S_3 : sconf1 \rightarrow sconf2, S_4 : sconf1 \rightarrow 0, S_5 \nearrow S_8 : sconf3 \rightarrow 0; B_1 : 4 \rightarrow 2, B_3 \nearrow B_20 : 2048 \rightarrow 1024)$
- $\Delta f(X,Y) = 100 - 1000 = -900 = -\Delta f(Y,X)$

7.3 Space-to-space transition

Let $D = \{X^1,\ldots,X^p\}$ be a design space containing all the possible design points. This space can be described as a *Multidimensional Design Space (MDS)*. Let $\mathscr{F} = \{f(X^1),\ldots,f(X^p)\}$ be the set of values (according to the considered objective function f) corresponding to the design points. With each D are associated three properties denoted as $\mathscr{P}$, $\mathscr{S}$ and n^D. $\mathscr{P}$ ($\mathscr{S}$) is the set of partitioning (scheduling) configurations, respectively, that are possible in D, and n^D is the number of buffers in D.

A *space-to-space transition* (also called a *move*) results only from applying refactoring operations to the analyzed dataflow program. The refactoring might involve modifications of two types: (1) insertion/removal of an actor and/or a buffer to/from the network; (2) modification of the internal structure of an actor leading to variation of the processing weight of an action, or the number of dependencies, or the set of dependencies between the actions.

Modifications of type (1) lead to a completely new design space with an empty set $X(COM)$ of common design points. Modifications of type (2) may result in a design space with a non-empty set $X(COM)$, some design points that are removed (they are put in a set $X(OUT)$), and some others that are added (they are put in a set $X(IN)$). When generating design space D_2 from design space D_1, the set of added (removed) design points contains the solutions that are feasible (infeasible) in D_2 and infeasible (feasible) in D_1, respectively. The structural difference between D_1 and D_2 is defined as:
$\Delta(D_1,D_2) = \{X(COM),X(OUT),X(IN);\mathscr{P}(COM),$
$\mathscr{P}(OUT),\mathscr{P}(IN);\mathscr{S}(COM),\mathscr{S}(OUT),\mathscr{S}(IN);n : n^{D_1} \rightarrow n^{D_2}\}$.
The difference between the values of the objective function for the overlapping design points

can be denoted as:
$\Delta\mathscr{F}(D_1, D_2) = \{\Delta f(X_i(D_1), X_i(D_2)), \forall\, i \in X(COM)\}$.
Depending on the type of refactoring applied to the program, the resulting set $X(COM)$ might be empty (*i.e.*, all design points are new). The same holds for $\mathscr{P}(COM)$ and $\mathscr{S}(COM)$. If $X(COM) \neq \emptyset$, the value of each design point in $X(COM)$ may change.

7.4 Design space quality

The most important indication about the *quality* of the design space D_i is the quality of the best solution $X^*(D_i)$ in terms of the objective function f and/or the satisfaction of the constraints, if any. If different design spaces provide solutions of a comparable quality, a volume $V(D_i)$ (average objective function variation) of different design spaces is compared. A volume can be calculated for all design points generated by certain heuristics or, for huge numbers of design points, for a fraction of points obtained by sampling. The number of design points used to calculate the volume can be different in different spaces.

Let $X^*(D_i)$ denote the best-found solution in D_i (according to f). Its value is denoted by $f(X^*, D_i)$. The following indicators can be further considered when measuring the quality of D_i:

- $T(D_i)$: computing time required to find $X^*(D_i)$;
- $C(D_i)$: total number of solutions evaluated during the exploration of D_i;
- $I(D_i)$: proportion of improving moves (when moving from a current solution to a neighboring solution within the considered local search framework) during the exploration of D_i.

A design space can be explored by appropriate (meta)heuristics, which can target any of the underlying optimization problems. High-quality (meta)heuristics are essential in order to find a competitive design point in a given design space. If the values of $X^*(D_i)$, $T(D_i)$, $C(D_i)$ and $I(D_i)$ are not satisfied, and the constraints (if any) are violated, the idea is to generate a new design space, explore it, and hopefully find better solutions.

7.5 Design space complexity

The following example demonstrates a simple design space, where the considered design points consist of the 3, mentioned earlier, configurations. They are realistic examples of feasible design points generated for MPEG4-SP decoder executed on Intel i7-3770 platform (4 cores). The values of the makespan for each design point have been calculated using a highly accurate performance estimation methodology, which is described in detail in Chapter 9. For each of the 3 configurations, two variants are considered.

Partitioning:

- $P^X(2) = RM$, where all actors (jobs) are randomly distributed among the two available machines;
- $P^X(2) = BD$, where the actors (jobs) are distributed among the two machines, so that the total processing time of each machine is as close to equal as possible.

Scheduling:

- $S_1^X \nearrow S_2^X = NnP$, where on each machine, jobs are executed according to the $Non-Preemptive$ scheduling policy, that is, each actor is executed as many times in a row as possible;
- $S_1^X \nearrow S_2^X = RR$, where on each machine, jobs are executed according to the $Round-Robin$ scheduling policy, that is, after a successful execution of an actor, another actor is chosen for execution. Both policies are discussed in more detail in Section 8.3.2.

Buffer dimensioning:

- $B^X = B_{16k}$, where the buffer dimensioning configuration is defined as $B_1^X \nearrow B_{65}^X = 16,384$;
- $B^X = B_{\min}$, which corresponds to a feasible, deadlock-free buffer configuration with $B_{total}(X)$ close to minimal. The configuration is: $B_{\min} = \{B_1^X \nearrow B_4^X = 1, B_5^X \nearrow B_6^X = 2, B_7^X \nearrow B_8^X = 4, B_9^X \nearrow B_{14}^X = 8, B_{15}^X \nearrow B_{19}^X = 16, B_{20}^X \nearrow B_{26}^X = 32, B_{27}^X \nearrow B_{34}^X = 64, B_{35}^X \nearrow B_{38}^X = 128, B_{39}^X \nearrow B_{43}^X = 256, B_{44}^X \nearrow B_{65}^X = 512\}$.

Figure 7.1 depicts a complete space of possible solutions for this example, where the connections between the design points represent the neighbor design points. Even though it is an extremely simplified design space ($i.e.$, only two variants are considered for each configuration) it illustrates the complexity of the design space exploration problem, as formulated in Chapter 6. For instance, assuming $f = k$, it can be observed that the best point (G) can be reached from multiple paths, where each path results from some point-to-point transitions between the neighboring points. This illustrates that different dimensions of the space can be explored in multiple orders. Furthermore, the extreme design points (here points F and G) can be equal with respect to some configurations. In this particular case, the structural difference is $\Delta(G,F) = (P : BD \rightarrow RM; S_1 \nearrow S_2 : NP \rightarrow RR)$ and the buffer size configuration remains the same for both. Finally, design points of a similar quality can consist of different configurations. For example: $\Delta(D,H) = (S_1 \nearrow S_2 : NP \rightarrow RR; B_1 \nearrow B_{65} : B_{\min} \rightarrow B_{16k})$. This transition corresponds to $\Delta f(D,H) = -7164$, which makes a difference of only 0.06%.

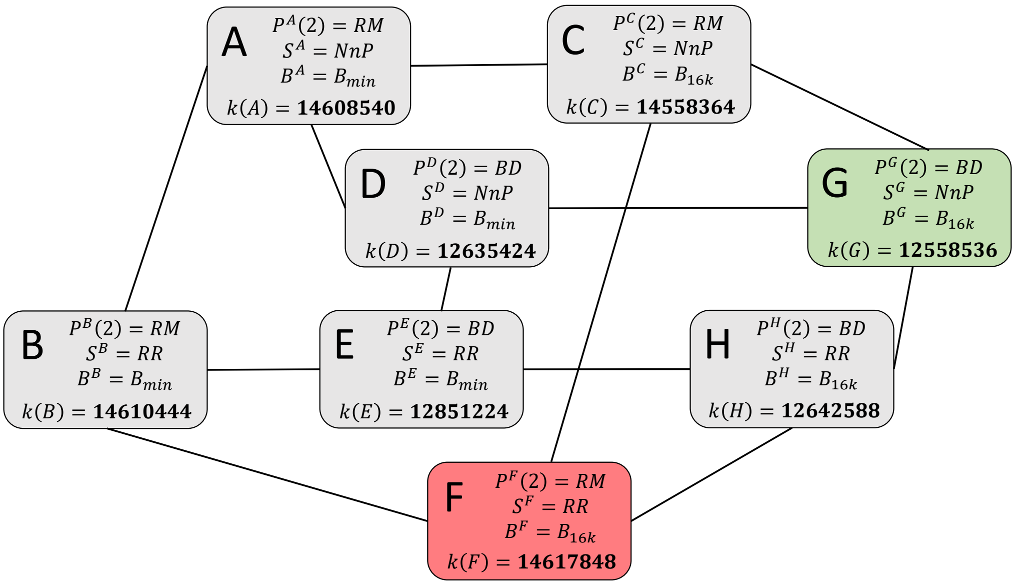


Figure 7.1 – *DSE* example: MPEG4-SP decoder on Intel i7-3770.

7.6 Optimization scenarios

It is possible to define some constraints depending on the solutions being searched for. For a given design point X, the number of machines m^X (the total buffer size $B_{total}(X)$ and the value of the makespan $k(X)$) can be upper-bounded by U^m (U^b and U^k), respectively. The optimization problem is to find the best (according to f) design point X^* among the visited design spaces. The following optimization scenarios can be considered.

1. (S1) Minimize the makespan f^k with upper bounds U^m and U^b on the number of machines and on the total buffer size, respectively.
2. (S2) Minimize the number f^m of machines with upper bounds U^k and U^b on the makespan and on the total buffer size, respectively.
3. (S3) Minimize the total buffer size f^b and the number f^m of machines with an upper

bound U^k on the makespan.

Let g^m (g^b and g^k) be the penalty function associated with the violation of the upper bound U^m (U^b and U^k), respectively. The values of these functions are calculated as follows: $g^m(X) = \max(m^X - U^m; 0)$; $g^b = \max(B_{total}(X) - U^b; 0)$; $g^k = \max(k(X) - U^k; 0)$. The optimization problem consists of minimizing $F = f + \alpha \cdot g$, where g is a penalty function and $\alpha \geq 0$ is a weighting parameter. If $\alpha > 0$, the constraints violations are penalized. Increasing (decreasing) α augments (reduces) the importance given to the penalty function and the search is likely to better focus on feasible (competitive) solutions, respectively. The tuning of α is the decision of the program designer who might change it during the exploration, depending on the obtained results. Using the proposed new notation, the above-mentioned optimization scenarios can now be formulated as below. Note that if two components appear in either f or g, they are normalized in order to give them the same importance.

1. (S1) $f = f^k$ and $g = g^m + g^b$.
2. (S2) $f = f^m$ and $g = g^k + g^b$.
3. (S3) $f = f^b + f^m$ and $g = g^k$.

7.7 Variable Space Search

A *bottleneck* of a program is defined as a part of its implementation which must be modified (*i.e.*, optimized or parallelized) in order to reduce the makespan for a given design point, when an execution on a given target platform is considered. Bottlenecks can be caused by both design and platform factors. A *design bottleneck factor* corresponds to a long sequential part of the program (*i.e.*, resulting from insufficient potential parallelism of the design), whereas a *platform bottleneck factor* may be related to the bandwidth between the partitions, limited cache sizes *etc*. A critical path of the design (*CP*) is defined as the longest time-weighted sequence of events from the start of the program to its termination and can be evaluated using multiple algorithms as described in [121]. Different bottleneck factors are reflected in the *CP*, hence the *CP* is a measure of the execution representation. The refactorization of the program leading to the generation of a new design space should be based on the analysis and, consecutively, resolving the bottlenecks. Originally proposed in [169], the process of design space exploration performed in different design spaces generated by the refactoring process, as discussed earlier in Section 7.1.3, can be described as a *Variable Space Search (VSS)*, and is presented in Algorithm 1. It must be noted that the same concept can be applied to different optimization scenarios, according to the specified constraints and objective function(s).

Algorithm 1: Variable Space Search (VSS)

```
Data: Input: $D_0$; Output: $X^*(D_i)$ (with $i > 0$)
Initialization:
    • In the given design space $D_0$, find $X^*(D_0)$ with a solution method.
    • Set Continue = true.
    • Set $i = 1$.
while Continue = true do
    • Generate a new design space:
        1. Analyze the bottlenecks of $X^*(D_{i-1})$.
        2. In order to reduce $g[X^*(D_{i-1})]$, refactor the program (programming effort, code optimization) to build $D_i$.
    • Generate an initial solution in $D_i$ by updating $X^*(D_{i-1})$.
    • Design Space Exploration (DSE): use (meta)heuristics to find $X^*(D_i)$.
    • Set Continue = false if one of the following conditions is encountered:
        1. $g[X^*(D_{i-1})] = 0$ and $g[X^*(D_i)] > 0$
        2. $g[X^*(D_{i-1})] = 0$ and $g[X^*(D_i)] = 0$ and $[f(X^*(D_i)) > f(X^*(D_{i-1})) + \epsilon]$
        3. $g[X^*(D_i)] > g[X^*(D_{i-1})] > 0$ and $[f(X^*(D_i)) > f(X^*(D_{i-1})) + \epsilon]$
        4. Set $i = U^i$ (where $U^i$ is an upper bound on $i$)
    • Set $i = i + 1$.
end
```

7.8 Conclusions

This Chapter introduced the concept of a multidimensional design space and an appropriate formulation to capture its complexity, quality and describe the moves performed in the space. The formulation follows directly the *DSE* problem definition presented in Chapter 6. Furthermore, the concept of variable design spaces and the ways they are generated has been introduced. This concept has been used in a novel algorithm for analysis and improvement of dynamic dataflow applications based on the idea of Variable Space Search known from the graph coloring field. In the case of the dataflow application of the problem, it relies on the design space exploration and analysis of design bottlenecks. The algorithm can be used according to different optimization scenarios related to performance and resource utilization.

8 Heuristics for design space exploration

This Chapter describes several design space exploration heuristics aiming at partitioning, scheduling and buffer dimensioning which compose the functionalities of *Turnus co-design framework*. They belong to the "Profiling and Analysis" stage of the design flow discussed in Section 3.5, as illustrated in Figure 8.1. The usage of such heuristics is necessary in order to make the exploration process as efficient as possible. In fact, following the problem formulation presented in Chapter 6, it can be stated that the considered design space is huge, constrained and multidimensional, where different dimensions strongly influence each other. In this case, it is hard to imagine performing the exploration randomly or manually and it is necessary to create methods capable of approaching the high-quality solutions easily and in a systematic way.

An efficient exploration of the multidimensional design space has two important applications. First, exploration of feasible regions leads to determining a, hopefully, close-to-optimal set of configurations according to the desired objective function. Second, it enables the identification of unreachable regions of the design space that could become reachable by applying refactorization stages to the considered design. For instance, a different implementation of an algorithm might be required for obtaining higher performances if its current exposed parallelism is lower than the potential parallelism offered by the processing platform.

The considered *DSE* problem consists of multiple subproblems, where each one is considered NP-complete [121, 10]. Hence, it is only possible to develop heuristic approaches. First, different partitioning approaches of different complexity are described. Second, the two approaches to buffer dimensioning are presented. Finally, a few dynamic scheduling policies are proposed along with a figure of merit to evaluate the scheduling cost.

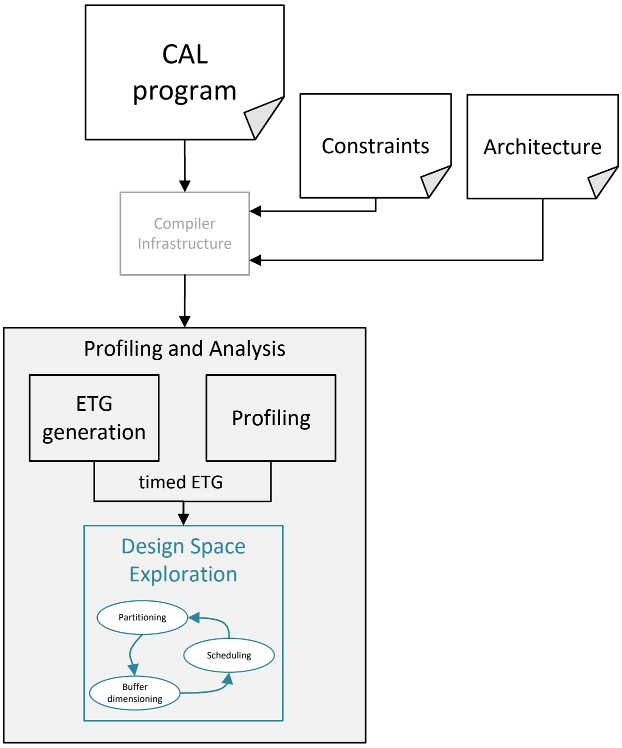


Figure 8.1 – System development design flow: design space exploration.

8.1 Partitioning

An important property of a dataflow program is the composability of its components (actors). Hence, a program can be executed on different types of platforms (*i.e.*, with different numbers of processing units) without any interference in the implementation. A link between a given program and a target platform in terms of an assignment of dataflow actors to the available units is taken care of by specifying a partitioning configuration. Finding such a configuration so that a given objective function is satisfied has been proven to be NP-complete even for the case of only two processors [172]. According to the commonly used terminology, the partitioning can be defined as a mapping of an application in the spatial domain and is also known as binding [173]. This Section describes different heuristics aimed at finding a high-quality partitioning configuration. The heuristics are ordered according to the complexity, starting from simple constructive heuristics, through local search methods, up to metaheuristics employing some learning features [18].

8.1.1 Related work

Due to the NP-completeness of the partitioning problem, for realistic instances it is only possible to develop methods providing close-to-optimal solutions [174]. They can be obtained by applying constructive heuristics, where a solution is generated from scratch by sequentially adding components to the current partial solution according to some criteria until the solution is complete [175]. Another possibility is metaheuristics, formally defined as iterative generation processes which guide a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space [176]. Metaheuristics (*e.g.*, simulated annealing, tabu search, variable neighborhood search, guided local search) can usually lead to solutions of higher quality, but in general they require much longer computing times [177, 178].

There are several examples of the approaches based on metaheuristics used for partitioning or, more generally, for the design space exploration of dataflow programs. In [179], simulated annealing is employed for estimating the bounds of the partitioning program. Various optimization stages (including the selection of a target architecture, partitioning, scheduling and design space exploration) are applied in [160] in order to identify feasible solutions. The optimizations are performed using an evolutionary algorithm. Multi-objective evolutionary algorithms used for performing an automatic design space exploration are also an objective of the work discussed in [161]. An interesting transition from simple heuristics to advanced metaheuristics (such as genetic algorithms) is also described in [93], where more advanced methods act as a refinement to the less advanced ones.

The partitioning (mapping) heuristics being part of the frameworks described earlier in Chapter 3 (*i.e.*, *MAPS* [84], *Sesame* [180], *PREESM* [93]) have been designed explicitly for the purpose of dataflow partitioning, which is a specific instance of a graph partitioning problem. The research field of graph partitioning is, indeed, thoroughly covered by different algorithms proposed in the literature ([181]) as well as some software packages, such as METIS ([182]) or SCOTCH ([183]). Such general purpose partitioning algorithms cannot be, however, easily applied for the case of dataflow programs, since they are not aware of the semantics related to the elements of a dataflow graph. An attempt at applying the, mentioned earlier, METIS, for the purpose of a run-time actor mapping of dataflow programs has been made in [184]. This approach explores the results of profiling and extracts some optimization criteria (*i.e.*, the connectivity between the actors). It is, however, difficult to evaluate the obtained solutions in terms of being close-to-optimal or not, or point to possible optimizations in the design, since no execution model is provided. The considered partitioning graph is the program network itself and not the execution trace which can provide some elements and measures of the execution properties of the dataflow program. Furthermore, such a combinatorial approach, which might operate quite effectively for small instances of the problem, cannot be successfully applied to the exploration of design problems of a larger size.

8.1.2 Greedy constructive procedures

In order to construct a solution, the greedy procedures require specifying only the target number of processors. The solution generation succeeds in a negligible time frame, since no performance estimation needs to be performed.

Workload Balance (WB)

The concept of balancing the workload in order to minimize the bottlenecks of the program and hence maximize the throughput has been already successfully employed for partitioning purposes of systems of different types [185]. Inspired by such approaches, the very first constructive heuristic has been designed. The algorithm starts from calculating the total workload of each group throughout the program execution. It is expressed as the sum of the p_j's for all jobs (firings) belonging to one group (actor) g. The actors are then sorted decreasingly by the sum of weights (workload) $wl(g) = \sum_{j \in g} p_j$. The partitioning decision is based on the sum of workloads of actors partitioned already in one processor ρ: $wl(\rho) = \sum_{g \in \rho} wl(g)$. The next actor on the list is always partitioned on the processor with the smallest sum of workloads $wl(\rho)$. In this way, a balance of the total workload of each partition should be achieved and the workload of the most occupied processor is likely to be minimized.

In order to illustrate the flow of the algorithm, the sample network depicted earlier in Figure 2.4 is used. The sample file containing processing weights for the actions of the actors is presented in Listing 8.1. For the considered set of weights, Table 8.1 presents all the steps of the algorithm, assuming that a partitioning on 2 machines is to be established. For each step, it is indicated which actor is selected for an assignment, its total workload, the target partition it is chosen to be assigned to (ρ_1 or ρ_2, in this case) and the value of $wl(\rho)$ after every assignment. Notice that the resulting values of $wl(\rho)$ are very close to each other (410 and 406, respectively).

Step id	actor/group	$wl(g)$	target ρ	$wl(\rho)$
1	D	270	ρ_1	270
2	F	180	ρ_2	180
3	G	120	ρ_2	300
4	B	110	ρ_1	380
5	C	60	ρ_2	360
6	A	50	ρ_2	410
7	E	26	ρ_1	406

Table 8.1 – WB partitioning algorithm: sample flow.

Listing 8.1 – Sample (processing) weights file for the program from Figure 2.4

```xml
<?xml version="1.0" ?>
<network name="Sample_Network">
    <actor id="A">
        <action id="a" clockcycles="10" firings="5"/>
    </actor>
    <actor id="B">
        <action id="b1" clockcycles="20" firings="4"/>
        <action id="b2" clockcycles="15" firings="2"/>
    </actor>
    <actor id="C">
        <action id="c" clockcycles="10" firings="6"/>
    </actor>
    <actor id="D">
        <action id="d1" clockcycles="50" firings="3"/>
        <action id="d2" clockcycles="40" firings="3"/>
    </actor>
    <actor id="E">
        <action id="e1" clockcycles="5" firings="2"/>
        <action id="e2" clockcycles="4" firings="4"/>
    </actor>
    <actor id="F">
        <action id="f" clockcycles="30" firings="6"/>
    </actor>
    <actor id="G">
        <action id="g" clockcycles="20" firings="6"/>
    </actor>
</network>
```

Balanced Pipeline (BP)

The algorithm starts from giving each actor a dedicated processor. Next, the processors are iteratively reduced and the members of the least occupied processor are attached to the remaining processors. The optimization criteria of the algorithm include equalizing the average preceding workload *(APW)* between the partitions and maximizing the number of common predecessors *(ACP)* for each partition. *APW* is defined as the maximal sum of weights of the jobs belonging to the actors (groups) that precede the given actor in the network in terms of topological order: $max \sum p_{j \in g_j} g_j \prec G$. The *ACP* number is evaluated for each pair of actors and denotes the number of actors appearing on the topological list of predecessors. An actor is also considered to be its own predecessor. In addition, the list of predecessors must consider the cycles between the actors, if they appear. The idea behind employing the aforementioned criteria is to join the units where the overall *APW* is small with those with a big *APW* so that the actors which are about to fire at the similar time during the execution do not block each other. An additional criterion favors a high *ACP* value between actors inside one unit, as most likely there is a *pipeline* between them that would disable their parallel execution anyway.

The flow of this partitioning algorithm is illustrated using the same example (the program network and the weights) as for the WB algorithm. Again, partitioning on 2 processing units is considered. First, the settings for the algorithm are presented. Table 8.2 summarizes the calculated values of AW and APW for each actor, and Table 8.3 presents the values of ACP for each pair of actors. Table 8.4 illustrates the steps of the algorithm. In each step it is indicated what is the initial partitioning configuration, what is the value of APW for each partition (put in brackets) and which move is chosen to be performed. Notice that, unlike for the previous algorithm, in this case the created partitions have close values of the preceding workload, instead of the workload. The resulting configuration is also completely different compared to the one established by the WB algorithm.

actor/group	AW	APW
A	50	0
B	110	330
C	60	186
D	270	546
E	26	220
F	180	246
G	120	246

Table 8.2 – BP partitioning algorithm: AW and APW settings.

	A	B	C	D	E	F	G
A	-	1	1	1	1	1	1
B	1	-	4	4	4	4	4
C	1	4	-	4	4	4	4
D	1	4	4	-	4	5	5
E	1	4	4	4	-	4	4
F	1	4	4	5	4	-	5
G	1	4	4	5	4	5	-

Table 8.3 – BP partitioning algorithm: ACP settings.

Step id	Partitioning configuration	Chosen connection
0	$\{D\} = 546, \{B\} = 330, \{F\} = 246, \{G\} = 246, \{E\} = 220, \{C\} = 186, \{A\} = 0$	$\{A\} \rightarrow \{D\}$
1	$\{A, D\} = 273, \{B\} = 330, \{F\} = 246, \{G\} = 246, \{E\} = 220, \{C\} = 186$	$\{C\} \rightarrow \{B\}$
2	$\{A, D\} = 273, \{B, C\} = 258, \{F\} = 246, \{G\} = 246, \{E\} = 220$	$\{E\} \rightarrow \{B, C\}$
3	$\{A, D\} = 273, \{F\} = 246, \{G\} = 246, \{B, C, E\} = 245$	$\{B, C, E\} \rightarrow \{F\}$
4	$\{A, D\} = 273, \{G\} = 246, \{B, C, E, F\} = 245$	$\{B, C, E, F\} \rightarrow \{G\}$
5	$\{A, D\} = 273, \{B, C, E, F, G\} = 245$	-

Table 8.4 – BP partitioning algorithm: sample flow.

The algorithm can operate in two modes. If the number of partitions is fixed, the algorithm proceeds until the given number is reached. Otherwise, the number of processing units must be established. For that purpose, two additional parameters are introduced: (a) the **Average Partitioning Occupancy** (*APO*), calculated as an average value of the processing time of each unit expressed in percent; (b) the **Standard Deviation of Occupancy** (*SDO*), calculated as a statistical standard deviation for the *APOs* of the units. These parameters are calculated during the performance estimation. Preliminary experiments and observations lead to characterization of the balanced workload of a partitioning configuration with a high value of average occupancy and, at the same time, a low value of standard deviation. With such a distribution of values, in the ideal case, all partitions should be equally and maximally occupied. Therefore, the ratio of *APO* to *SDO* is used as an evaluation of partitioning configuration. As the reduction procedure continues, this ratio quite naturally increases. If the opposite occurs, it usually means that a strong inequality of the workload among units is introduced. Hence, this determines the stop condition of the algorithm.

Once an initial partitioning configuration is established, a further optimization procedure can be applied, for instance one of the descent local search methods (idle time or communication volume minimization) described in the following Section. Alternatively, instead of using a performance estimation during the search, it is also possible to specify a fixed percentage of the most *idle* (most *communicative*, respectively) actors which will be moved to different processing units.

8.1.3 Descent local search procedures

As described in [186], a local search starts from an initial solution and then explores the solution space by moving from the current solution to a neighbor solution. A *neighbor* solution is usually obtained by making a slight modification of the current solution, called a *move*. The *neighborhood* $N(s)$ of a solution s is the set of solutions obtained from s by performing each possible move. In a *descent local search* (*DLS*), the best solution (according to the considered objective function f) of $s' \in N(s)$ is generated at each iteration. The main drawback of this method is that it stops in the first local optimum. Two *DLS* approaches are proposed: the *Idle DLS* and the *Communication Frequency DLS*.

Idle descent local search (IDLS)

Representing the program execution with an *ETG*, and simulating its execution for a given partitioning, scheduling and buffer dimensioning configuration using the performance estimation tool, provides important information related to the actor states throughout the execution. The following states may occur for an actor that is currently not processing and has not yet terminated:

- **Blocked reading** considers the situation where an actor has not yet received the required input tokens and therefore cannot be executed;
- **Blocked writing** takes into account the situation where the buffer an actor is expecting to write to is full, so it has to wait for the available space;
- **Idle** corresponds to the situation where although an actor has the necessary tokens and required space in the buffers, it cannot be fired because another actor is currently processing in the same processor (because, as previously mentioned, only one job can be executed on each processor at a time).

When looking for a partitioning and scheduling configuration yielding high performance, it is particularly important to minimize the occurrences of the idle state. In order to achieve that, in $IDLS$ all actors are sorted according to their idle times in decreasing order (*idle time list*). A newly created solution s is generated by moving a single actor to the most idle partition, where the *idleness* of a partition is defined as the overall time during the execution when none of its actors could be executed due to being blocked reading/writing or terminating. In each iteration, the possible moves are prioritized according to the position of the considered actor on the *idle time list*. A move is evaluated by estimating the makespan of the new solution. For the case of a successful move, the statistics on the idle times of the actors and the corresponding *idle time list* are regenerated. Since the moves are prioritized, there is a risk that if there is a move with a high priority that does not improve the solution, it will be unnecessarily repeated in each iteration. To prevent that from happening, a simple *release* mechanism is implemented: a (once unsuccessful) move, expressed as an actor-partition pair, may be repeated only if the content of the target partition has been modified by applying another move.

Communication frequency descent local search (CFDLS)

Another piece of information that can be extracted from the *ETG* is the number of token dependencies between the firings of different actors. Accumulating these numbers for all firings leads to the creation of an actor-actor communication frequency map. This map is independent from the partitioning configuration, but taking the partitioning into consideration, it can be easily transformed into an actor-partition map. Indeed, this map is taken as an optimization criterion by another local search. For each actor, the algorithm calculates the internal communication frequency (token exchange with actors partitioned to the same processor) and external communication frequency (token exchange with actors partitioned to different processors). Partitioning of actors may strongly influence the values of communication cost and therefore the makespan.

If for any actor, the external communication frequency with one processor exceeds the internal communication frequency, this actor-partition pair is considered as a move. The moves are prioritized according to the overall communication frequency of the actors and a *release* mechanism (similar to $IDLS$) is implemented. The move can be evaluated in two ways: by estimating the execution time of a new solution or by analyzing if the overall external communication frequency (calculated collectively for all partitions) has decreased.

8.1.4 Tabu search

Tabu search (TS), as introduced by Glover [187], is still among the most cited and used local search metaheuristics for combinatorial optimization problems. It avoids the problem of getting stuck in the first local optimum by making use of recent memory with a *tabu list*. More precisely, it forbids performing the reverse of the moves done during the last tab (parameter) iterations, where tab is called *tabu tenure*. At each iteration of TS, the neighbor solution s' is obtained from the current solution s by performing on the latter the best non-tabu move (ties are broken randomly). The process stops, for instance, when a time limit T (parameter) is reached. In most TS implementations, if the neighborhood size is too big, only a proportion is explored in each iteration. This proportion can be, for instance, a random sample involving e% (parameter) of the neighbor solutions.

TS has proven to have a good balance between intensification ($i.e.$, the capability to focus on specific regions of the solution space) and diversification ($i.e.$, the ability to visit diverse regions of the solution space). In addition, it has a good overall behavior according to the following measures [178]: (1) quality of the obtained results (according to a given objective function f that has to be optimized); (2) speed (time needed to get competitive results); (3) robustness (sensitivity to variations in data characteristics); (4) simplicity (facility of adaptation); and (5) flexibility (possibility to integrate properties of the considered problem). To adapt TS to the studied problem, the following elements have to be designed: the representation of any solution s, the neighborhood structure ($i.e.$, what is a move), the tabu list structure ($i.e.$, what type of information is forbidden), and a stopping criterion ($i.e.$, what is the most appropriate time limit).

Solution encoding and neighborhood structure

A solution for partitioning is represented as a map of actors and processors, where the number of processors is fixed. Each actor can be mapped to only one processor at the time, and each processor must be mapped to at least one actor. Hence, leaving empty processors is not allowed. The following basic types of moves are possible: (1) *REINSERT*: move an actor to another processor; (2) *SWAP* two actors belonging to two different processors. For the purpose of swapping, the term *complementary move* is introduced. Assume that a move $m(j, \rho_i, \rho_{i'})$

consists of relocating an actor j from a source partition ρ_i to a target partition $\rho_{i'}$. A move $m(j', \rho_{i'}, \rho_i)$ is complementary to $m(j, \rho_i, \rho_{i'})$ if it involves moving any actor j' from a source partition $\rho_{i'}$ to a target partition ρ_i. The neighborhood structures are generated by performing REINSERT and SWAP moves according to the four different criteria, presented below.

1. $N^{(B)}$ (for balancing):
    - *REINSERT*: choose randomly an actor from the most occupied processor and move it to the least occupied processor;
    - *SWAP*: choose randomly two actors in different partitions so that swapping the actors decreases the relative workload imbalance between the two partitions;
2. $N^{(I)}$ (for idle):
    - *REINSERT*: for each actor which has a bigger idle time than its processing time, find the most idle processor, different from the one currently mapped, where the definition of *idle* is as described for $IDLS$;
    - *SWAP*: generate a set of moves on the *REINSERT* basis, but allow actors to be moved to *any* partition except for the least idle one, then search for complementary pairs of moves;
3. $N^{(CF)}$ (for communication frequency):
    - *REINSERT*: check the internal and external communication frequency of each actor and consider the moves, as described for $CFDLS$;
    - *SWAP*: generate a set of moves on the *REINSERT* basis, then search for complementary pairs of moves;
4. $N^{(R)}$ (for random):
    - *REINSERT*: choose randomly an actor and move it to a different processor (randomly chosen);
    - *SWAP*: generate a set of moves on the *REINSERT* basis, then search for complementary pairs of moves.

Parameters

Any time an actor j is moved from a processor ρ to another processor, it is forbidden to put j back to ρ for tab iterations, where tab is an integer uniformly generated in interval $[a, b]$, and the values of parameters a and b are tuned to 5 and 15, basing on the preliminary experiments. Smaller values do not allow escape from local optima, whereas larger values do not allow intensification of the search around promising solutions. There are two other sensitive parameters that have to be tuned for TS, namely e (the proportion of neighbor solutions explored during each iteration) and T (the time limit). Reaching the time limit T results in immediate termination of the search and returning of the best solution ever found. Usually, T is set so that the improvement potential is poor ($i.e.$, the percentage of improvement is below a threshold during a pre-defined time interval) if the method is run for larger time limits. Next, the smaller is e, the more iterations are performed but the fewer neighbors are investigated in each iteration. A large value of e contributes to the intensification ability of the method (indeed, all the solutions around the current one are explored), whereas a small value plays a diversification role (indeed, no focus is put on the neighborhood of each solution). Finally, a small (large) value of tab strengthens the intensification (diversification) ability of the search, respectively.

8.1.5 Advanced tabu search

Since each of the used neighborhood structures relies on different properties, a more advanced version of the TS relies on a consolidation of all neighborhood structures. It is applied in two different variants:

- **Joint Tabu Search (JTS)**: at each iteration, the neighborhood structure includes moves obtained according to all types. Therefore, the used neighborhood structure is $N^{(J)} = N^{(B)} \cup N^{(I)} \cup N^{(CF)} \cup N^{(R)}$. This variant should have more flexibility, because it comprises various types of moves. The proportion of the set sizes for different types of moves can be freely tuned;
- **Probabilistic Tabu Search (PTS)**: at each iteration, the search assigns a probability to the selection of each neighborhood of the set $\{N^{(B)}, N^{(I)}, N^{(CF)}, N^{(R)}\}$. This probability is tuned based on the history of the search during the considered run. As a result, the search is guided by the success rate of each type of move (where a success corresponds to an improvement of the current solution).

8.1.6 Tabu search with iterative communication cost profiling

Tabu search and, in particular, its advanced variants are capable of finding high-quality solutions which much outperform the simpler greedy or DLS methods, as illustrated later by the experiments described in Section 10.2.4. It must be, however, noted, that this method strongly depends on the performance estimation in the sense that every decision about making a particular move or not is determined by the estimation-based evaluation. Hence, the estimation accuracy is the crucial factor leading to success or failure of the algorithm. Depending on the target platform, in some cases it is possible to perform the profiling only once and then use the results in order to explore the entire space of solutions. In other words, once obtained, the results remain of the same accuracy for all possible partitioning configurations. This is the case for the *TTA* platform, the aforementioned results are based on.

Such a situation is, however, quite abstract, when most of the commercially used platforms are considered. For the case of a $NUMA$-based *SW* platform, the communication cost changes from configuration to configuration. Hence, the more modifications are applied to the configuration originally used for profiling, the higher is the risk that the quality and accuracy of the generated solutions will diminish. This implies using more advanced methods to overcome this problem, but minimizing, whenever possible, the number of profiling runs. Hence, the original tabu search procedure has been extended by an additional *re-profiling* procedure. It consists of profiling of the communication cost and the generation of the communication-related weights, as described in Section 5.3.3. This procedure is not performed every time new moves are considered, but after a complete tabu search run, so that another run, with the updated weights, is performed. The procedure is presented in Alg. 2. The action weights (a_w) and communication weights (c_w) are considered separately. Applying this procedure makes the tabu search partitioning strategy useful also on $NUMA$-based platform, as confirmed by the results presented in Section 10.3.2.

Algorithm 2: NUMA re-profiling procedure.

```
Data: IN: $S^X, B^X$; OUT: $P^X$
$a_w$ = profileActions();
$P^X$ = generatePartitioning();
while $iteration < max$ do
    $c_w$ = profileCommunication($P^X, S^X, B^X$);
    $P^X$ = tabuSearch($P^X, S^X, B^X, a_w, c_w$);
end
```

8.2 Buffer dimensioning

According to the specifications of dataflow *MoCs*, the sizes of the interconnecting channels (buffers) constituting the network are considered unbounded [188]. However, when a program is executed on a given platform each buffer must be assigned with a finite size. A necessary constraint related to this process is to specify the buffer sizes so that the program can correctly execute without any deadlocks. A possible design objective can be to minimize the total buffer size in order to meet the platform memory constraints (*i.e.*, embedded-memory limitations of FPGAs). In the case of dynamic dataflow programs, the performance of an implementation can also strongly depend on the assigned buffer dimensions. The buffer size set should be determined with regards to, for instance, data dependencies and traffic on each buffer. The trade-off between the minimization of the total buffer size and the maximization of the program throughput constitutes an interesting optimization problem and this is the objective of the heuristic methodology described in this Section.

8.2.1 Related work

The problem of buffer dimensioning for the classes of dataflow programs denoted as static *(SDF)* and cyclo-static *(CSDF)* [189]) has been already extensively studied by the research community [190, 191, 192, 193, 194]. For these *MoCs*, a restricted set of dataflow actors and predictable patterns of reading/writing the tokens exist, which is not the case for dynamic dataflow programs (*DDF*) [55] where no static information can be extracted from the design.

The problem of buffer dimensioning for *DDF* has been considered in [195]. In this case, the objective was to minimize the total memory usage while reserving sufficient space for each data production without overwriting any live data and guaranteeing a satisfaction of real-time constraints. The approach, however, is applied to a specific *MoC*, namely Mode-controlled Dataflow (*MCDF*), which is a restricted form of dynamic dataflow that allows mode switching at runtime and static analysis of real-time constraints. Therefore, the approach is not generic enough to support the widest class of dynamic dataflow *MoC*.

Guaranteeing a deadlock-free execution is, obviously, an indispensable step in the design space exploration process. It separates the region of feasible solutions from the deadlock region, so that only feasible solutions are considered in the exploration process. This separation is emphasized in the, described previously in Section 3, *DSE* frameworks. When a deadlock-free configuration is guaranteed, all of them, however, focus on exploring the configurations related rather to partitioning and scheduling.

Establishing a deadlock-free configuration is not sufficient for the case of dynamic applications, because apart from satisfying the constraints (*i.e.*, limited memory resources), other objective functions (*i.e.*, throughput) must be also taken into consideration. This problem has been addressed in [196], where an off-line buffer sizing algorithm based on the rate constraints and on the dependency information gathered from profiling results has been proposed for *KPN* processes. In that work it is emphasized how small buffer size configurations, even when deadlock-free, can limit the execution effectiveness of the actor processes.

The work described in [197] describes a methodology of buffer dimensioning that aims at finding a trade-off between the buffer size B_{total} and the program throughput. It starts from an initial deadlock-free configuration and iteratively increases the sizes of certain buffers in order to improve the throughput. The heuristics described in this Section are based on a similar concept. An important difference is in the number of dimensions considered in the exploration. Whereas the referenced works targeting buffer dimensioning for the purpose of throughput improvement consider a fully-parallel execution (*i.e.*, hardware), the presented *bottom-up* and *top-down* heuristics model and explore a wider region of the design space including partitioning and feasible scheduling configurations (*i.e.*, when a subset of actors partitioned to the same processor is executed serially according to the scheduler). Hence, it provides high-quality results for any types of heterogeneous platforms (hardware-software co-design), where multiple subsets of actors can be executed either serially or in parallel.

8.2.2 Notion of partitioning and scheduling configurations

In the case of a parallel execution which disregards the partitioning and scheduling configurations, increasing the size of any buffer always leads to an increase of the performance (if a relevant *blocking instance* is removed) or the execution time remains unchanged (if an increase is not sufficient to remove a relevant *blocking instance*). The case of a partitioned execution, that is, when a given subset of actors is executed sequentially within one processing unit, is also affected by the presence of a scheduler. Depending on the scheduling policy within a processing unit, actors can be chosen for execution in a different order and the availability (or not) of necessary space can affect the feasibility of different schedules. Hence, it is possible that increasing the size of a buffer will lead to a *decrease* of performance, since the order of execution inside a given processing unit may become less favorable. This situation takes place quite often, since an increase of a given buffer affects the scheduling eligibility of *all* firings requiring writing to such buffer. In fact, only a fraction of them might be critical and executing a non-critical firing instead of a critical one might lead to the, mentioned earlier, drop of performance. So as to illustrate this problem, Fig. 8.2 presents a simple network consisting of a few actors assigned to two partitions. The scheduling policy assumes that an actor is executed as many times as possible and in *Actor Q* the *action q1* has a priority over *action q2*. Two scenarios are considered: (1) all buffers have an equal size of 1, (2) *buffer b1* has the size of 2, the others of 1. Fig. 8.3 presents the Gantt charts obtained in both cases. Notice that although in the second scenario the buffer size configuration is larger, the execution time has been extended by 2 units. At this stage it must be emphasized that the likelihood of this behavior of a network remains fully dependent on the scheduling policy and its sensitivity to the buffer sizes. For this reason, the moves cannot be performed *blindly* and after each iteration it is necessary to evaluate a move and revert it if a performance decrease has occurred. Furthermore, instead of picking up one buffer in each iteration, a *ranking* of buffers must be created and in case of a necessity to revert a move, the next buffer from the ranking is considered for an increase.

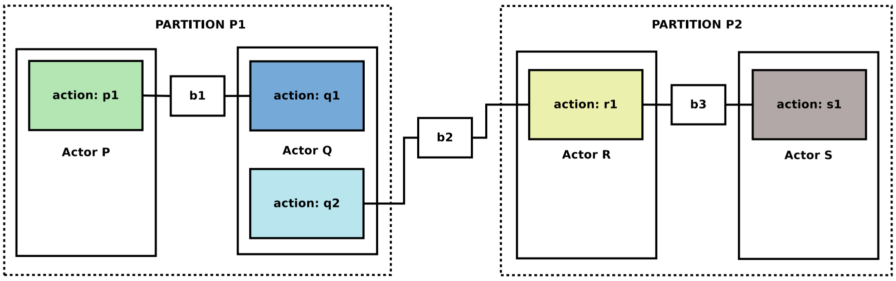


Figure 8.2 – Simple network with the assigned partitioning, scheduling and buffer dimensioning configurations.

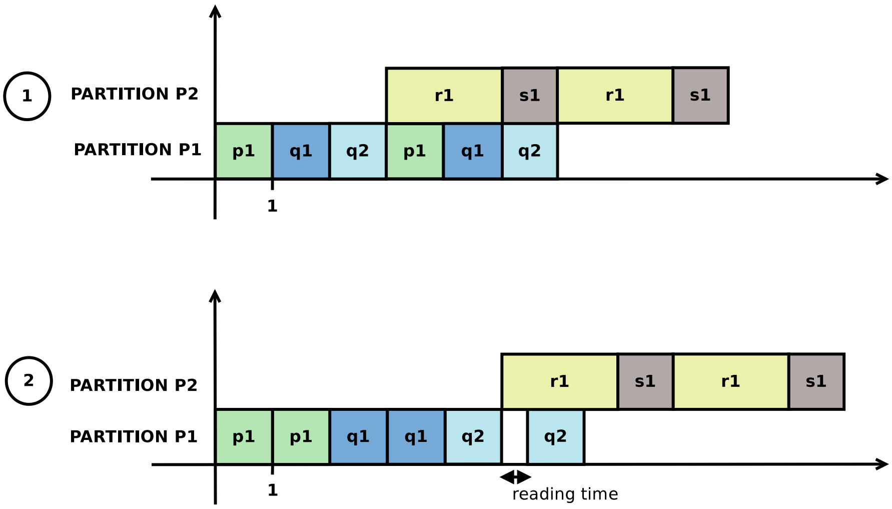


Figure 8.3 – Gantt charts for the execution of the network from Fig. 8.2 for the two buffer size configurations.

8.2.3 Minimal and maximal buffer size estimation

The deadlock-free buffer size configuration constitutes a border between the set of feasible and infeasible design points. This configuration, considered as close-to-minimal, is evaluated on the basis of *ETG*, as described in Section 8.4.3 of [121], relying on the approach originally introduced in [198].

The *maximal* buffer size configuration is established during the performance estimation. Given configurations of P and S are estimated with an approximation of infinite buffer sizes, corresponding to the maximum value of an integer. In these circumstances, no blocking of firings resulting from buffer size limitations occurs. For each buffer the maximal number of tokens present in this buffer at the same time is recorded. In this way, the original sizes equal to the maximum value of an integer are reduced to the sizes which are required in practice to prevent any blocking of tokens. This set of sizes constitutes the maximal buffer configuration.

8.2.4 Bottom-up optimization procedures

The starting point for the heuristic is a minimal buffer size configuration. Consecutively, in each iteration one buffer is chosen for an increase. The heuristic has two different variants and both are related to the analysis of the critical path of the design, as it has been defined in Section 7.7. Hence, it contains the firings contributing to the longest serial part of a program execution. If such a firing requires writing to an output buffer, this buffer is considered to be *critical*. Any *blocking instance*, that is, an insufficient space in the output buffer occurring for such buffer affects the total execution time and hence, the overall data throughput.

Heaviest blocking ranking

The first ranking, presented in Algorithm 3, looks for the *heaviest* blocking instance along the critical path. In this context, the *heaviness* of a blocking instance is measured by a multiplication of the number of tokens blocked (tk_B) and the time they remained blocked ($time_B$). For each buffer in the critical path (B_{cp}) a maximal *heaviness* throughout the execution is recorded and among different critical buffers the one with the largest corresponding *heaviness* is chosen. This ranking intends to remove the most impacting sources of delay in the execution. Having to revert a move implies considering the next buffer in the map.

Algorithm 3: Heaviest blocking ranking procedure.

```
Data: IN: $P^X, S^X, ETG$; OUT: $B^X$
$B^X$ = minConf($ETG$);
while iteration < max do
    map{$B_{cp}, max(tk_B * time_B)$} = cpAnalysis($P^X, S^X, B^X$);
    foreach $B_{cp}$ do
        $B_m = argmax$(map{$B_{cp}, max(tk_B * time_B)$});
        $B_m = B_m * 2$;
        if $time^* < time$ then
            break;
        end
        else
            $B_m = B_m/2$;
        end
    end
end
```

Criticality ratio ranking

The second ranking, presented in Algorithm 4, calculates the *ratio* between the critical blocking instances of a buffer (Bi_{cp}) and all blocking instances of this buffer (Bi) throughout the execution. Buffers with the highest ratio Bi_{cp}/Bi are first considered for an increase. This ranking intends to minimize the unnecessary increases for the firings which are not in the critical path. It must be emphasized that for the case of both rankings, the CP analysis has to be performed in every iteration, since changing even one buffer size in the network can modify the execution order and, consequently, the location of the CP.

Algorithm 4: Criticality ratio ranking procedure.

```
Data: IN: $P^X, S^X, ETG$; OUT: $B^X$
$B^X$ = minConf($ETG$);
while iteration < max do
    map{$B_{cp}, \langle Bi_{cp}, Bi \rangle$} = cpAnalysis($P^X, S^X, B^X$);
    foreach $B_{cp}$ do
        $B_m = argmax(Bi_{cp}/Bi)$;
        $B_m = B_m * 2$;
        if $time^* < time$ then
            break;
        end
        else
            $B_m = B_m/2$;
        end
    end
end
```

8.2.5 Top-down optimization procedure

In most cases, the solutions obtained with the previously described approach manage to improve the performance (compared to the close-to-minimal deadlock-free configuration) by only few percent. Hence, further challenges related to buffer dimensioning for a partitioned program executed in software can be identified. The bigger are the buffers, the more likely it is, in general, to increase the number of cache misses. This number, however, is not entirely proportional to B_{total} and depends strongly on the configurations of P and S. On the other hand, the smaller are the buffers, the more context switching related to executing different actors one after another takes place. Furthermore, once the makespan changes when a move from one design point to another is performed ($\Delta(X,Y) \neq \emptyset$, $\Delta f(X,Y) \neq 0$), it can be assumed that the critical path changes and the analysis performed for one design point is not valid for another one.

These observations lead to another buffer dimensioning heuristic, referred to as the *top-down* approach. This approach starts from a *maximal* buffer configuration. The heuristic performs a critical path analysis for this configuration. All buffers which are not critical are then considered for a reduction performed iteratively. A reduction is accepted if the total execution time remains unchanged. It can be reasonably assumed that if the length of the critical path (corresponding to the makespan k) is not affected, the critical path analysis results remain valid for the new configuration. Hence, unless the makespan is affected, there is no need to perform cp analysis, which reduces the memory requirements of the heuristic. Further steps reduce the least impacting critical buffer and, finally, allow a certain increase of execution time (if it is outstripped by the reduction of B_{total}), as presented in Algorithm 5.

The top-down buffer dimensioning approach generates a full spectrum of solutions ranging from relatively *large* buffers leading to good performances and *small* buffers with a remarkable performance drop compared to the first group. For solutions located in the interesting regions of the space, that is, providing good performances while keeping buffer sizes as close as possible to the initial deadlock-free configuration, it has been investigated if the solutions can be further improved by applying small modifications to some of the buffers. The investigated modifications included a reduction of the biggest buffers in the network and an increase of the smallest ones. The ideas behind these modifications are as follows: (1) reducing the size of the largest buffers (setting an *upper bound* on the buffer size) should reduce the number of cache misses, where they are most likely to occur, and significantly reduce B_{total}; (2) increasing the size of the smallest buffers (setting a *lower bound* on the buffer size) should reduce the number of context switches, where they are expected to occur most often, while keeping the increase of B_{total} negligible. The whole range of possible values of a lower and upper bound has been tested (1 - 262144), but on condition that none of the buffers is assigned with a smaller value than in the initial deadlock-free configuration. The best values are application- and partitioning configuration dependent. For the case of the HEVC decoder [199], the values of 64 and 16384 for the lower and upper bound, respectively, seem to be the best choices in terms of the throughput - buffer size trade-off.

8.3 Scheduling

The scheduling problem for static dataflow programs has been studied very well and a whole class of compile-time algorithms is proven valid [41]. In the case of dynamic dataflow programs the problem becomes much complicated, because it requires creating a reliable model of execution that could sufficiently cover and capture the entire application behavior, which depends on the input data. It can be stated that if the whole dynamic behavior of an application is properly captured for a given input sequence and a deadlock-free buffer configuration is applied, the scheduling problem for dataflow programs is always feasible, in comparison to

Algorithm 5: Top-down buffer dimensioning procedure.

```
Data: IN: $P^X, S^X, ETG$; OUT: $B^X$
$B^X$ = maxConf($ETG$);
$min$ = minConf($ETG$);
$critical\langle\rangle$ = cpAnalysis($P^X, S^X, B^X$);
foreach $B_i \notin critical$ do
    while $B_i \geq 2 \cdot min(i)$ do
        $B_i = B_i/2$;
        if $k' > k$ then
            $B_i = B_i \cdot 2$;
            break;
        end
    end
end
foreach $B_i \in critical$ do
    while $B_i \geq 2 \cdot min(i)$ do
        $B_i = B_i/2$;
        if $k' > k$ then
            $B_i = B_i \cdot 2$;
            break;
        end
    end
end
foreach $B_i$ do
    if $B_i \geq 2 \cdot min(i)$ then
        remaining.add($B_i$);
    end
end
while $iteration < max$ do
    $B_m = argmax(B_i - min(i))$;
    $B_m = B_m/2$;
    if $\frac{k'}{k} - 1 \geq 1 - \frac{B'_{total}}{B_{total}}$ then
        $B_i = B_i \cdot 2$;
        remaining.remove($B_i$);
    end
end
```

some other programming paradigms [200]. Under these circumstances the challenge becomes to find such a scheduling configuration that optimizes the desired objective. Defining such a configuration is a different problem than partitioning and buffer dimensioning. For these two problems, multiple heuristics can be designed that provide appropriate configurations ready to use for different input stimuli. For the case of scheduling, the objective is to define an order of executions for the set of firings (jobs) which are executed dynamically, so no fixed order can be specified. Instead, it is only possible to define certain *strategies*, referred to as *scheduling policies* which determine a rule or a set of rules used by the partition scheduler and the actor scheduler when selecting an actor / an action to execute. Since the actual decision is made at run-time, scheduling implies certain cost which depends on the properties and the complexity of the policy. Hence, when designing efficient scheduling policies, there are two aspects that should be taken into consideration: (1) the potential gain coming from an appropriate order of execution of the firings (*i.e.*, minimizing the time the firings are waiting for their predecessors), (2) the cost of the policy corresponding to, for instance, the number of conditions that need to be checked and the context switches occurring when the generated code is executed. This Section describes a few scheduling policies enforcing different orders of execution of firings and defines a figure of merit for the scheduling cost.

8.3.1 Related work

The general problems of partitioning and scheduling of parallel programs, as closely related challenges, have been widely described in the literature in numerous variants [138]. Whereas the partitioning is often referred to as mapping in the spatial domain, scheduling takes place in the temporal domain and is also called sequencing [173]. Literature positions often emphasize that the partitioning is performed at compile-time, whereas scheduling occurs at run-time and is subject to the satisfaction of firing rules, as well as to the scheduling policy for the sequential execution of actors inside each processor [201]. Although the partitioning and scheduling problems seem to rely and impact each other, much more attention has been paid so far to the partitioning problem. Several experiments suggest that finding a solution to the partitioning problem will dominate over the scheduling problem, because the quality of the partitioning configuration impacts the opportunities for scheduling the actors or, more generally speaking, the tasks efficiently [200, 202].

Since some dataflow models can be very general and therefore difficult to schedule efficiently, an interesting idea comes along with the concept of flow-shop scheduling [203]. The asynchronous dataflow models can be, in some cases, transformed into simpler synchronous ones, where the partitioning and scheduling can be applied directly to the actions. After the partitioning stage (which is the assignment of all actions to the processing units), the scheduling is performed first in the off-line phase (schedules are computed at compile-time), and then in the run-time phase when a dispatching mechanism selects a schedule for data processing [204].

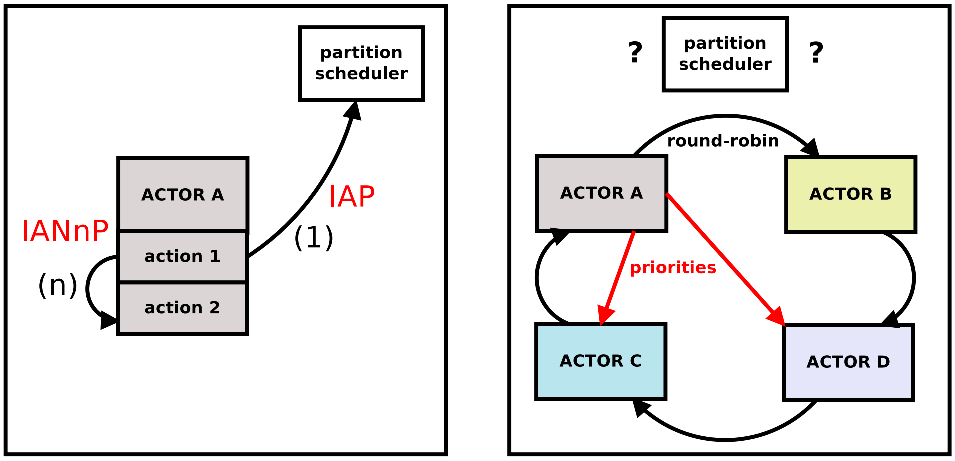
Another approach to simplifying the scheduling problem is to reduce the complexity of the network and control the desired level of granularity. This can be achieved by actor merging, which can be treated as a special transformation performed on the sets of actors [205]. Recent research shows that actor merging is possible even in the case of applications with data-dependent behavior and in the end can act quasi-statically [206]. This, however, does not solve the scheduling problem entirely, since even for a set of merged actors, if multiple merged actors are partitioned on one processor, a scheduling policy still needs to be defined.

8.3.2 Intra-actor and intra-partition scheduling policies

During the execution of a dataflow program, scheduling occurs at multiple levels. The lowest level is related to a selection of actions inside each actor. The order of execution of the actions is determined by guards, priorities and, obviously, the availability of input tokens and spaces in the outgoing buffers. What can be additionally driven at this level, is the allowed number of executions of certain actors in a row. This is referred to as the intra-actor scheduling policy ($IASP$). Two variants of $IASP$ are considered. The Intra-Actor Preemptive policy (IAP) means that an actor is allowed to execute only once and then it returns to the partition scheduler. The Intra-Actor Non-Preemptive policy ($IANnP$) means that once an actor is chosen by the partition scheduler, it is allowed to execute as many times as possible, that is, until none of its actions is eligible to execute. This approach is often mentioned in the literature as *FCFS* (first-come, first-served) scheduling, known also as Run-to-Completion [207]. The two variants of $IASP$ are depicted in Fig. 8.4a. The expression "preemptiveness" refers here to the change of the target actor after a successful firing and not to the interruption of a single task, which is, by nature, not allowed in dataflow programs.

The next level of scheduling occurs inside each partition. It is referred to as the Intra-Partition Scheduling Policy ($IPSP$). It is related to the decision of the scheduler to choose an actor to execute, as illustrated in Fig. 8.4b. Once an actor is chosen, it can execute on the basis of IAP or $IANnP$. A common approach involving executing different actors repetitively according to a given list is known as round-robin and is commonly used in operating systems as described in [207]. Another possibility is to extend the round-robin procedure by assigning certain numbers of *cells* to each task (actor, in this case) so that each of them can be executed a certain number of times before the round-robin procedure is continued. This approach is known as round-robin with credits [208]. An alternative to the round-robin procedure is to define certain priorities and use them when making a decision (priority scheduling [207]).

The following scheduling policies combine different $IASP$ and $IPSP$ approaches. They aim at



(a) Intra-actor.

(b) Intra-partition.

Figure 8.4 – Scheduling policy - illustration.

investigating how different strategies and types of priorities, if applicable, lead to differences in performance. The first group can be considered as direct implementations of the popular and commonly used scheduling techniques.

- **Non-Preemptive** (*NnP*): $IANnP$ scheduling is assumed for every actor; the partition scheduler chooses the next actor to execute on a round-robin basis;
- **Round Robin** (*RR*): IAP scheduling is assumed for every actor (opposite to the previous policy); the partition scheduler chooses the next actor to execute on a round-robin basis;
- **Non-Preemptive / Preemptive swapped** (*NnP/P*): the list of actors for each partition is sorted according to the *criticality*, which is represented as a percentage of executions of a certain actor belonging to the critical path; $IPSP$ iterates over this list on a round-robin basis; the most critical actor (among the remaining actors on the list) is executed as $IANnP$, the others as IAP; this approach can be considered as the, mentioned earlier, round-robin with credits with a binary choice of cell numbers: either equal to one or the number determined by the $IANnP$.

The second group of policies cannot be compared with the existing techniques because they exploit the information obtained at the level of action firings, not actors ($i.e.$, jobs, not groups). As a result, although only the actors can be chosen by the scheduler, the system of priorities changes from firing to firing throughout the execution.

- **Critical Non-Preemptive** (*CNnP*): the partition scheduler chooses the next actor to execute on a round-robin basis; if the next firing of a chosen actor is contained in the critical path, the actor is executed on the $IANnP$ basis and after a successful execution, its next firing is analysed for criticality; if an executed firing is not critical, the IAP is applied and the scheduler moves to the next actor; this results in a similar strategy to *NnP/P*, but the priorities are resolved independently for each action firing and only the actual critical firings are given the priority, not the actors as such;

- **Critical Outgoings Workload** (*COW*): all actors are executed as IAP; when making a choice of an actor to execute, its next firing is always considered; the eligible firings in different actors are compared according to the set of priorities: the highest priority goes to the firing which is critical, if critical firings are the next ones to be executed in multiple actors, the highest priority is given to the one which has outgoing dependencies in other partitions, and if the decision cannot be made based on the first two criteria, the firing with the highest weight is chosen;
- **Earliest Critical Outgoings** (*ECO*): all actors are executed as IAP; when making a choice of an actor to execute, its next firing is always considered; the eligible firings in different actors are compared according to the set of priorities: the highest priority goes to the firing with the earliest occurrence in the critical path, or, if no critical firing is currently available, to a firing with the highest number of outgoing dependencies in other partitions; for the cases which cannot be resolved according to these criteria, a round-robin choice is applied.

8.3.3 Scheduling cost

Since all scheduling policies (both $IASP$ and $IPSP$) are applied to a dynamic execution, they are related to making some decisions at run-time. Hence, the number of conditions and constraints considered when making a decision impacts the program performance. Let $cc(S^X)$ be the scheduling cost related to a given configuration of S. It consists of two components: $c(S^X) = \sum cc(S^X) + cf(S^X)$, where $cc(S^X)$ denotes the number of *conditions checked* and $cf(S^X)$ denotes the number of *conditions failed.* The conditions in this case are related to the input (availability of the input tokens) and output (availability of the spaces in output buffers).

The value of $cc(S^X)$ is calculated as:

$$cc(S^X) = \frac{\sum_i \frac{\sum cc_e(S_i^X)}{n_i}}{m^X} \tag{8.1}$$

Consequently, the value of $cf(S^X)$ is calculated as:

$$cf(S^X) = \frac{\sum_i \frac{\sum cf_e(S_i^X)}{n_i}}{m^X} \tag{8.2}$$

The value of $cc_e(S^X)$ ($cf_e(S^X)$) corresponds to the number of conditions checked (failed) which is elapsed between two consecutive successful firings, respectively. Let's consider the following example. Figure 8.5 illustrates a set of actors in a partition. For the next firing in each actor, it is indicated from how many input buffers it reads the tokens and to how many output buffers it writes the tokens. The buffers marked with red indicate that the tokens/spaces are not available in these buffers. Assuming that the last successful firing took place for actor C and the scheduling policy is RR, as defined earlier, Table 8.5 presents the process of updating the values of $cc_e(S_i^x)$ and $cf_e(S_i^x)$ when the scheduler makes the next attempt. In this case, actor C is the next one chosen for execution and the respective values of $cc_e(S_i^x)$ and $cf_e(S_i^x)$ are 8 and 2.

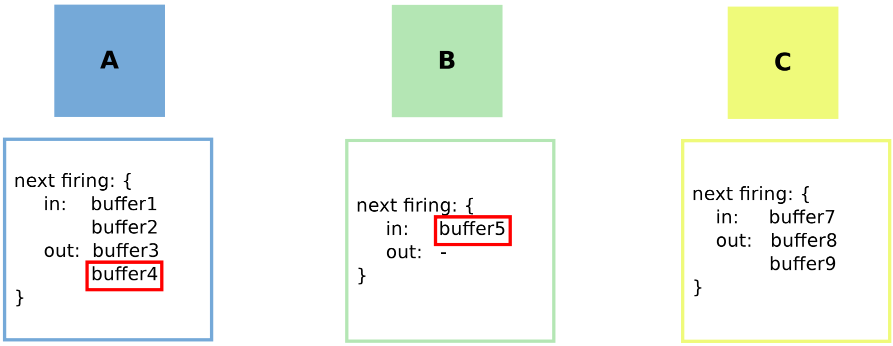


Figure 8.5 – Sample partition with the actors to be considered by the scheduler.

Attempt	actor	c. checked	c. failed	$cc_e(S_i^X)$	$cf_e(S_i^X)$
		last firing c_j			
1	A	4	1	4	1
2	B	1	1	5	2
3	C	3	0	8	2
		next firing c_{j+1}			

Table 8.5 – Conditions numbers updates.

This procedure is continued for every firing in the partition i. The sum of the elapsed conditions checked/failed is divided by the number of firings is this partition n_i. This value is a metric for a given partition i. In order to provide the metrics for the configuration S^X in total, the values obtained for each partition are summed and divided by the number of partitions m^X. As a result, the values of $cc(S^X)$ and $cf(S^X)$ express the numbers of conditions checked/failed during the execution, per firing, per partition.

The defined values should be considered as a *lower bound* on the number of conditions for two reasons. First, they consider a limited set of conditions ($i.e.$, no guard conditions considered). Second, the update of the values of $cc_e(S_i^x)$ and $cf_e(S_i^x)$ is performed only with certain attempts of the scheduler. The cases when the scheduler works *infinitely*, *i.e.*, when none of the actors in the partition can be executed and the scheduler keeps iterating over them without any success are difficult to model reasonably.

8.4 Conclusions

The heuristics described in this Chapter aim at design space exploration of dynamic dataflow programs. Each of them took one of the subproblems discussed in Chapter 6 and explored the design points in order to establish a high-quality solution. For each of the subproblems, that is, partitioning, scheduling and buffer dimensioning, several approaches of different complexity have been proposed. Naturally, they may lead to different qualities of the solutions and operate within different time requirements. In each case, the base for the heuristics are the rich performance metrics provided by a timed ETG.

In the first part, the partitioning problem was considered. Although it is a popular problem described in the literature, it can be observed that most of the general purpose approaches for graph partitioning are not applicable in this case, since they are not aware of the semantics at the edges of dataflow networks. Hence, specific heuristics must be developed instead. The partitioning algorithms introduced in this Chapter are grouped as greedy heuristics, descent local search methods and tabu search. Greedy heuristics usually provide a solution quickly, but only the search methods explore different variations of the solutions.

In the second part, the problem of finding a finite buffer size for the buffers in the network was considered. Analyzing the related work, it was observed that so far this problem has been tackled in a very limited way, that is, reducing the MoC to the models analyzable at compile-time or simplifying it to finding a deadlock-free configuration, disregarding the effect on the performance. In some works the impact on the throughput was considered, but not in the context of a multidimensional exploration. The buffer dimensioning heuristics introduced in this Chapter aim at finding a trade-off between the program performance and resource utilization, with regards to the specific partitioning and scheduling configurations. They can be applied directly to the optimization scenarios introduced in Section 7.6.

In the last part, the scheduling problem was considered. Different aspects of the problem differentiating it from the others were presented. An overview of related work discussed different approaches aimed at eliminating the scheduling problem or maximally reducing its impact. Since the dataflow applications are expected to have a dynamic and data-dependent behavior, it is only possible to define scheduling policies. Hence, several policies have been proposed. The run-time scheduling is subject to a cost related to establishing the schedule. Therefore, a figure of merit for the scheduling cost, relying on the number of conditions checked and failed throughout the execution was also introduced.

9 Performance estimation

The process of design space exploration consists essentially of performing moves from one design point to another. If the moves are properly driven, the whole exploration procedure becomes much more efficient and the final design points can be of a higher quality. An important role in driving the optimization heuristics is played by the performance estimation. First, it allows calculating the performance of a program on a given platform without having to physically execute the program on this platform. Second, it evaluates different design points in order to make a decision about whether to perform a certain move or not. Finally, if the performance estimation *simulates* the entire behavior of a program, it can allow extracting some execution properties to be used by the optimization algorithms or identify the most critical parts of the execution that should be considered for optimization. This Chapter presents a performance estimation tool, developed as a module within the *Turnus co-design framework*, serving as a basis for the VSS methodology described in Chapter 7 and the heuristics described in Chapter 8. It is one of the stages of the design flow discussed in Section 3.5, as illustrated in Figure 9.1.

9.1 Related work

An accurate performance estimation methodology is usually built upon two stages: appropriate modeling and its evaluation. The quality and the level of detail in the model determine the accuracy of the estimated results when referred to the actual execution. Usually, the most accurate results can be achieved when the model is very detailed. It implies, however, longer evaluation times. On the other hand, less detailed models, which can be evaluated in shorter times, provide a lower level of accuracy [209, 210, 211]. The performance estimation methodologies used in various dataflow-oriented DSE frameworks (if available) were presented earlier in Section 3.2. The overview provided in this Section aims at discussing more generally different approaches to performance estimation and prediction in the field of

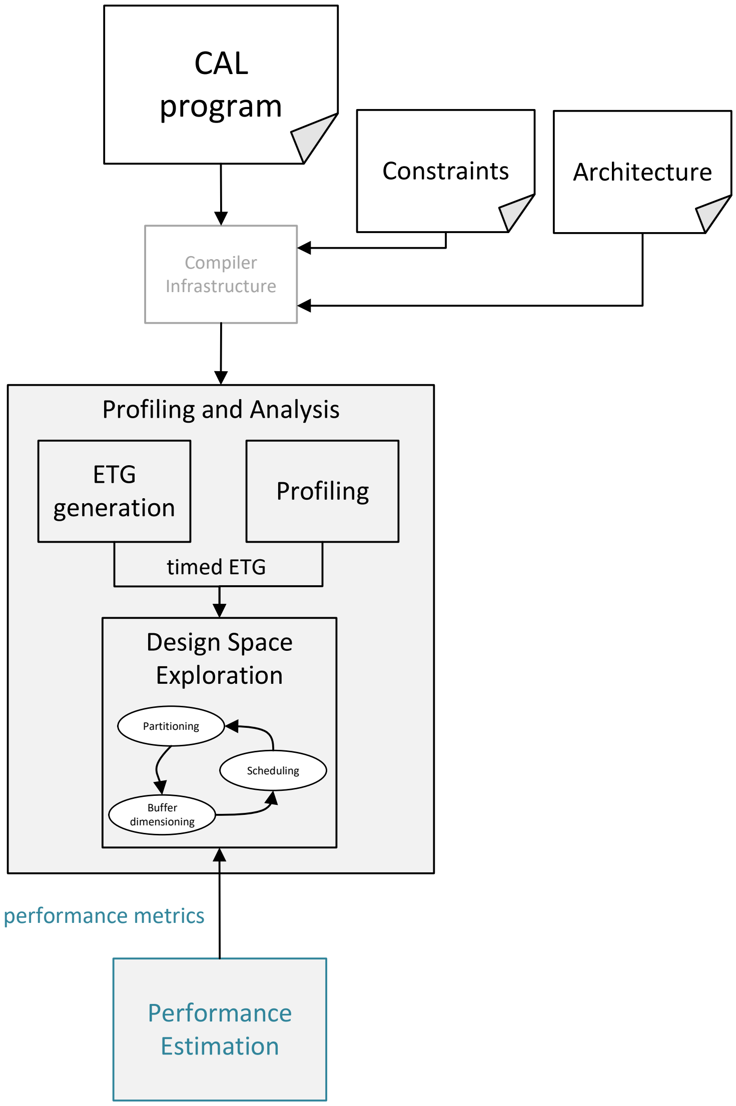


Figure 9.1 – System development design flow: performance estimation.

parallel programming and multi-core platforms, with an emphasis on some very recent works and the achievable level of accuracy.

The work discussed in [155] relies on the usage of abstract models of the target system and the structure of the program. They are represented as a finite state machine with some input and output events. The estimation is performed with regards to the execution time and the code size since a trade-off between these two properties is considered as the objective of DSE. It is emphasized that the main role of performance estimation is to reduce the exploration time of the considered design space. The achieved accuracy of estimation is around 20%.

Another approach, discussed in [212], employs analytical models in order to perform the DSE of pipelined MultiProcessor System-on-Chips ($MPSoC$), where the architecture favors the implementation of applications characterized as repetitive executions of some sub-kernels [213]. The models are used for an estimation of the execution time, latency and throughput, avoiding slow full-system cycle-accurate simulations of all the design points by extracting the latencies of individual processors. The two described methodologies include a single simulation of all the processor configurations and, on the other hand, multiple simulations of a subset of processors. It is stated that the considered design space consists of ca. 10^{12} to 10^{18} design points, hence a complete exploration is infeasible, as it would be expected to take years to complete. Narrowing the set of design points reduces the simulation times to several hours with the estimation accuracy between 12.95% and 18.67% (maximum absolute error). A similar concept of analytical models is used in [214], where the estimation relies on defining several figures of merit for the considered properties. They are based on certain equations using metrics measured on the platform. In this work, the estimation is applied to a different problem, namely the slowdown caused by multiple applications running simultaneously.

One of the methods employed for the purpose of performance estimation is source code analysis using the concept of elementary operations. The work described in [215] performs the profiling of different sets of operations and uses this information for estimation in heterogeneous $MPSoC$ achieving an estimation error around 6%. Similarly, [216] describes a methodology for source code profiling at the level of intermediate representation, where dataflow graphs are used to capture the dependencies between different operations. Another possibility to create a structural representation of operators is to use UML activity diagrams so that each operator is decomposed into operational units of different granularity. The operations are modeled with regards to the latency, power and area to form a pre-characterized operators library. It is stated that shifting the estimation level from the code to the model allows a fast DSE in the early design steps.

The objective of DSE described in [217] is to find the most efficient target System-on-Chip (SoC) for a given application. The core of the used performance estimation methodology is the classification and learning method called Regression Random Forest. Thanks to applying this learning technique, the number of analyzed configurations is reduced by learning the relationships between different design parameters. Using machine learning techniques for the purpose of performance prediction has recently become a popular topic of research. For instance, [218] and [219] use Artificial Neural Networks (ANN) in order to provide the mapping and/or scheduling heuristics with some reliable performance measures. In both cases, the ANN-based performance prediction is used to drive the heuristics and results in remarkably better-quality configurations. In [219], the process of retraining the artificial networks aims at analyzing the discrepancy between the real and predicted performance in order to apply an error-correction learning rule and hence, maximally reduce the prediction inaccuracy.

Among the possibilities to reduce the time required for a single run of performance estimation, one opportunity is to define two models with different levels of detail and accuracy. The work described in [164] introduces two estimation models: an accurate, but time consuming model and a simpler one, enabling a short estimation with a higher discrepancy. The complexity of these models depends on the number of parameters which are considered in the estimation. The models are used interchangeably in order to search for various trade-offs related to design implementations in hardware and software. The term *fidelity of estimation* is introduced and corresponds to the percentage of correctly predicted comparisons between different design points.

Finally, an interesting approach for a cross-platform performance and power consumption estimation is described in [220]. It employs a learning algorithm that synthesizes analytical proxy models that predict the performance and power of the workload in each program phase from performance statistics obtained through hardware counter measurements on the host. The objective of the investigation is to verify if a few example runs on a slow, detailed simulator (commonly available to software developers) and the corresponding runs on real hardware can give an insight into the correlation between the two [221]. The learning approach based on this concept is considered to provide over 97% prediction accuracy.

9.2 Trace processor tool

The tool is based on a discrete event system specification formalism *(DEVS)* [222]. A *DEVS* system is constructed as a set of atomic models described by their state transition, output- and time advance functions. The state transitions can be triggered by some internal and external events. One of the advantages of a *DEVS* model is the fact that the *confluent* events ($i.e.$, that introduce races or some unpredictable behavior) can be efficiently identified and resolved. The communication between the atomic models succeeds through the signals received (sent) as the port values that define the template argument for the types of objects accepted (produced) as input (output), respectively. Since the tool needs to model the complete behavior of a dataflow program, the following components building the system are included: an actor, a buffer, an actor partition and a buffer partition. The events driving the performance estimation are stored in the internal list of each actor that contains the firings of this particular actor extracted from the *ETG*. The relative order of execution within the firings of one actor is fixed, as determined by the input stimulus. Figure 9.2 illustrates the construction of a *DEVS* model for a sample dataflow program. In this case, the *Producer* and *Filter* actors belong to the same *PartitionA*, and the *Consumer* actor is a part of *PartitionB*. The two buffers *b1* and *b2* are assigned to one buffer partition *PartitionBF*.

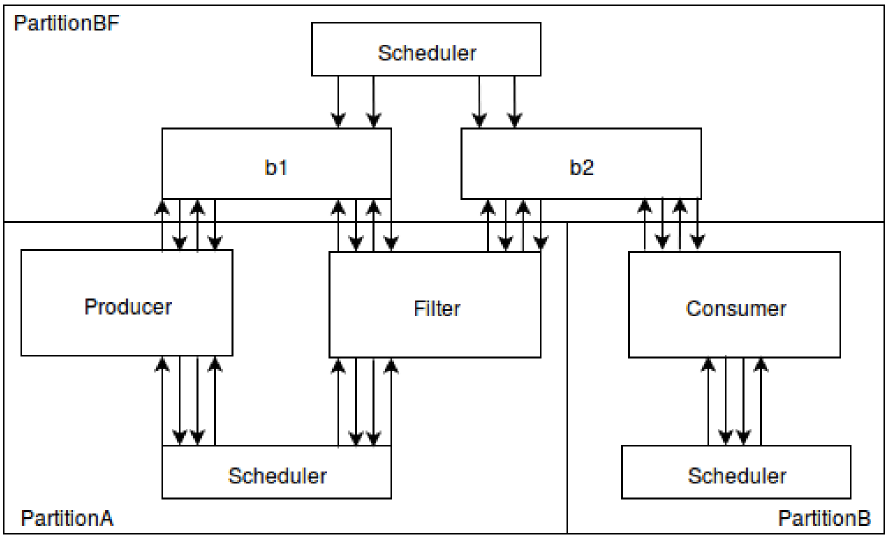


Figure 9.2 – Schematic illustration of the system components and connections.

9.2.1 Atomic models

- **Atomic actor** models a dataflow actor which executes the firings according to its internal list. The time advance function corresponds to the action weights obtained by profiling and assigning each firing. It defines the next update time (*i.e.*, state transition) of an actor model. The execution of each firing requires going through several states of an actor. A detailed transition considering all possible states of an actor is illustrated in Fig. 9.3. There are two procedures that can be separated from this transition and they are both triggered by a specific signal received from the actor partition. They include: (a) checking the schedulability, (b) executing a firing. Procedure (a) consists of checking the availability of the input tokens and the necessary spaces in the output buffers for the next firing on the list. If both, input and output conditions are satisfied, an actor moves to the schedulable state, where it awaits an enabling signal from its partition. Procedure (b) assumes that an actor is *schedulable* and the firing execution with all underlying procedures (selecting an action, reading the input, processing, writing the output tokens) can be performed. The *release buffers* state can be performed either before, or after the *processing* state, depending on the way the reading from the buffers is implemented in the generated code. The preliminary experiments demonstrated that supporting these two modes is crucial in order to provide a highly accurate model for the hardware, as well as the software implementations.

- **Atomic buffer** models a buffer connection between the two actors that is used to exchange the tokens. It is modeled as an asynchronous receiver and transmitter that can be enabled or disabled by the buffer partition. The time advance function corresponds to the communication weights related to reading and writing. Each buffer has a fixed size, specified as a configuration.

- **Atomic actor partition** corresponds to a partition of actors. Every time an actor ends a firing, it sends a notification signal to *all* actor partitions in the system. Then, each actor partition performs a few verification procedures: (a) determining if it is allowed to schedule another actor ($i.e.$, no actors currently are running or the parallel execution is supported), (b) if there are actors in the *schedulable* state to choose for an execution, (c) if there are actors that need to be checked for schedulability. Among the schedulable actors, an actor partition makes the choice of an actor to schedule based on the scheduling policy that can be freely defined for each partition. The scheduling policy may give equal chances for each actor to be chosen or use multiple priorities. When the choice is made, the target actor is notified with an appropriate enabling signal. In addition, each partition keeps track on its members that are running, schedulable, blocked at input or blocked at output. In some cases, this information allows reducing the actor - actor partition communication volume and leads to a higher efficiency of the estimation system. Fig. 9.4 presents the state transition for an actor partition.

- **Atomic buffer partition** models a buffer partition, which can enable and disable the transmitting and receiving functionality of a buffer. The construction of a buffer partition enables future extensions of the model in order to cover various types of input-output interfaces in heterogeneous architectures.

9.2.2 Atomic models interaction

In order to respect the token dependencies between the firings of different actors, communication between the models of the actors and the buffers is necessary. The partitioning and scheduling configurations enforce also actor - actor partition and buffer - buffer partition communication. For this reason, each actor contains a set of input and output ports. The types of information exchanged between the actors, buffers and partitions are summarized in Fig. 9.5. The information is sent (received) to dedicated ports. Regarding the actor - buffer communication, the following ports are defined:

- **IN SEND HAS TOKENS**: used for inquiring about the availability of tokens that should be consumed;
- **IN RECEIVE HAS TOKENS**: used for receiving a true/false signal depending on the availability of the requested tokens;
- **IN SEND ASK TOKENS**: used for sending a request for tokens to be consumed;
- **IN RECEIVE TOKENS**: used for receiving the input tokens from the input buffer;
- **OUT SEND HAS SPACE**: used for sending a space request to a buffer;
- **OUT RECEIVE HAS SPACE**: used for receiving a true/false signal depending on the availability of the requested space;
- **OUT SEND TOKENS**: used for sending the produced tokens to the output buffer;
- **OUT RECEIVE TOKENS RECEIVED**: used for receiving an acknowledgement from the output buffer when it accepts all tokens.

The ports of the buffers used for the communication with actors that write to (read from) a given transmitter (receiver) buffer are complementary to the ports defined for the actor model. Each actor port of type *receive (send)* has a corresponding port of type *send (receive)* in the input (output) buffer.

The next set of ports is related to the actor - partition communication. Most signals are exchanged only between an actor and the partition it is statically mapped to. The following port are defined:

- **PARTITION RECEIVE ASK SCHEDULABILITY**: used for receiving a request for checking the satisfaction of execution conditions;
- **PARTITION SEND SCHEDULABILITY**: used for sending a true/false signal depending on the satisfaction of the execution conditions and hence the ability to be scheduled *(schedulability)*;
- **PARTITION RECEIVE ENABLE**: used for receiving an enable signal which means that an actor has been chosen for an execution by the partition scheduler;
- **PARTITION SEND END OF FIRING**: used for sending a notification about the end of a current firing to all partitions in the system.

The ports inside an actor partition are complementary to the ones specified inside an actor regarding the actor - partition communication and include:

- **SEND ASK SCHEDULABILITY**: used for sending a request to check the schedulability, if it is necessary. An actor partition stores the information about all its members (*i.e.*, if they are currently running, schedulable or blocked) and sends this request only if the information about schedulability cannot be deduced;
- **RECEIVE SCHEDULABILITY**: used for receiving a true/false signal depending on the schedulability;
- **SEND ENABLE**: used for sending an enable signal to the actor chosen for execution, where the choice is made among all schedulable actors in a given partition based on the scheduling policy specified in the internal scheduler of each partition;
- **RECEIVE END OF FIRING**: used for receiving a notification about the end of a firing. Unlike for the case of other ports, which are coupled only with the actors belonging to this partition, each partition has a dedicated port for each actor in the system, so that it knows which actor sent the notification and whether it can be used for a deduction of actor schedulability.

The states of a buffer model include *RX enable*/*RX disable* as well as *TX enable*/*TX disable*. Using these states, a buffer model can be controlled by its partition scheduler for an asynchronous receiving and transmission functionality. The ports defined for this purpose are:

- **RECEIVE RX ENABLE**;
- **RECEIVE TX ENABLE**;

Figure 9.6 illustrates the communication between different ports of an actor and its input and output buffers in order to successfully execute a firing and reports how different signals from the buffers determine the state transitions of an actor, depicted earlier in Fig. 9.3.

9.2.3 Execution properties

The core functionality of the tool is the estimation of the execution time, expressed in clock-cycles, of a program on a given platform. The estimation process consists of executing the events inside each actor partition respecting the dependencies and constraints resulting from the scheduling policy specified for each partition and the bounded buffer sizes. Hence, all configurations are reflected in the estimated results. The execution is timed according to the values of time advance functions, and the total execution time corresponds to the value of the time advance function when the very last event (firing) in the system has terminated. The set of logging functions allows the extraction of various information at the level of action firings. Depending on the purpose (criticality analysis, partitioning heuristic *etc.*), a different set of logging functions can be injected. The properties tracked during the performance estimation may include:

- the exact time of each change of state (*i.e.*, reading, scheduling, processing *etc.*) of each actor;
- total processing time of an actor/action;
- total time of being *blocked reading* for an actor/action;
- total time of being *blocked writing* for an actor/action;
- total time of being *idle* for an actor/action (*idleness* corresponds to a situation when an actor is deliberately *schedulable*, but it cannot be fired because another actor in the same processor is currently processing or chosen by the partition scheduler and the scheduling policy does not allow a parallel execution);
- percentage of occupancy of each partition (corresponding to the time slots when any of the actors in the partition is processing);
- average occupancy of all partitions for a given partitioning configuration and the standard deviation of occupancy;
- participation of an actor/action in the critical path of the execution (*i.e.*, what is the workload corresponding to a given actor/action which participates to the overall critical path)
- participation of the critical workload of an actor/action in the total workload of a given actor/action;
- criticality of a firing (*i.e.*, if a particular firing participates to the critical path or not);
- dependencies in other partitions for an actor/action/firing;
- the number of input/output conditions checked during the execution (for an actor);
- the number of input/output conditions that failed (were not satisfied) during the execution (for an actor);
- the number of tokens blocked for each *blocked writing* occurrence (for a buffer);

- the time the tokens remained blocked for each *blocked writing* occurrence (for a buffer);
- the buffer, the number of tokens and the time of blocking for each *blocked writing* occurrence along the critical path;
- the maximal number of tokens present at the same time in each buffer.

9.3 Critical path analysis

The critical path of a weighted *DAG* is defined as the longest weighted path from a source node to a sink node of the graph. In consequence, the *CP* of a weighted *ETG* can be used to identify the actions that contribute the most to the overall execution time of a program and affects its throughput. Different approaches can be found in the literature that aim at finding either an exact or an approximated *CP* of the *ETG* [121, 223, 224, 225]. It must be emphasized that in the case of a *DAG*, the exact *CP* can be found in a linear time [226].

9.3.1 Algorithm

The *CP* of an *ETG* can be evaluated during the performance estimation. For each post-processed firing $s_i \in S$ (corresponding to an event in the *DEVS* system) a partial *CP (PCP)* value is defined. It contains the following parameters:

- *finishingTime* containing the finishing time of the associated firing;
- *weightsMap* data containing the sum of weights of an action that participated in the *PCP*;
- *firingsMap* data containing the sum of numbers of executions of an action participating in the *PCP*;

During the post-processing of *ETG*, the *PCP* is evaluated for each firing s_i by selecting its predecessor (s_k from the set of predecessors δ) that has the highest value of *finishingTime*:

$$PCP(s_j^*) = \max_{\text{finishingTime}} \{PCP(s_k) : s_k \in \delta(s_i)\} \tag{9.1}$$

In the case of multiple predecessors with equal values of *finishingTime*, the second comparison is made according to the weight of the *PCP*:

$$PCP(s_j^*) = \max_{\Sigma weightsMap} \{PCP(s_k) : s_k \in \delta(s_i)\} \tag{9.2}$$

Hence, the $PCP(s_i)$ is computed starting from $PCP(s_j^*)$ and adding to *weightsMap* the respective action weight associated with each s_i. Similarly, in the *firingsMap* the number of the executions of the action associated with each s_i is incremented. At the end of the post-processing, the *CP* can be evaluated considering the maximal *PCP* among all *ETG* firings. As a result, the *firingsMap* contains the number of firings of each action that is included in the *CP*. Every action that has at least one firing along the *CP* is considered to be *critical*. Similarly, the *weightsMap* contains the overall execution time along the *CP* of every action. Due to the construction of the *PCP* structure, the algorithm relies on a *propagation* and only a small portion of data (instead of the full *ETG*) is loaded to the memory at one time. Furthermore, as long as the design points from the same design space are considered, the *CP* analysis can be performed for the same *ETG*, independently from the configurations P^X, S^X, B^X. Different configurations have, however, an influence on the obtained results, for example, the biggest portion of the *CP* is usually related to the most occupied partition.

9.4 Impact analysis

As demonstrated in Chapter 10 Experimental results, ranking the actions according to their *CP* participation value is not sufficient to estimate the potential reduction of the *CP* (and the associated throughput increase) corresponding to an optimization of the algorithmic part of an action. This is caused by the possible presence of multiple parallel *CPs* in the *ETG*. If this is the case, the programming effort related to optimization might not be reflected in the quality of the newly created design space.

Hence, it is important to define a metric capable of pointing to the bottlenecks of a design more precisely. This can be done by estimating the *CP* length reduction (throughput increase) when the clock-cycles of the most critical actions are reduced. Algorithm 6 illustrates the impact analysis heuristic. First, the initial *CP* is evaluated. Successively, for each critical action ($i.e.$, the one that has at least one firing along the *CP*), the *CP* is evaluated by iteratively reducing the required execution clock-cycles. Consequently, it is possible to identify which actions should be optimized in order to increase the application throughput. Identifying such actions by means of bottleneck (critical path) and impact analysis connects two stages from the design flow discussed in Section 3.5. Being implemented on top of "Performance Estimation", it is an essential part of "Profiling and Analysis", since it provides the refactoring directions, as illustrated in Figure 9.7.

9.5 Conclusions

This Chapter presented a detailed overview of the performance estimation tool based on the $DEVS$ concept and developed as a module within the *Turnus co-design framework*. The tool relies on the ETG and the weights which can be assigned to the processing, scheduling and communication parts. It models a complete dynamic execution of the program and considers the configurations such as partitioning, scheduling and buffer dimensioning. Apart from estimating the execution time, the tool is capable of keeping track of various execution properties which are used by different DSE heuristics. The bottleneck and impact analysis for the purpose of evaluation of different design points is also built on top of the estimation module. As presented later in Section 10.6, the tool achieves a very high accuracy of estimation, compared to the state-of-the art approaches.

Algorithm 6: Impact analysis.

```
$CP \leftarrow$ computeCP();
$\Lambda^* \leftarrow$ criticalActions($CP$) ;
foreach $\lambda \in \Lambda^*$ do
    foreach $i \in [1,100] \subset N$ do
        $w(\lambda, i) \leftarrow w(\lambda) \times (i-100)/100$;
        $CP(\lambda, i) \leftarrow$ computeCP($w(\lambda, i)$);
    end
end
```

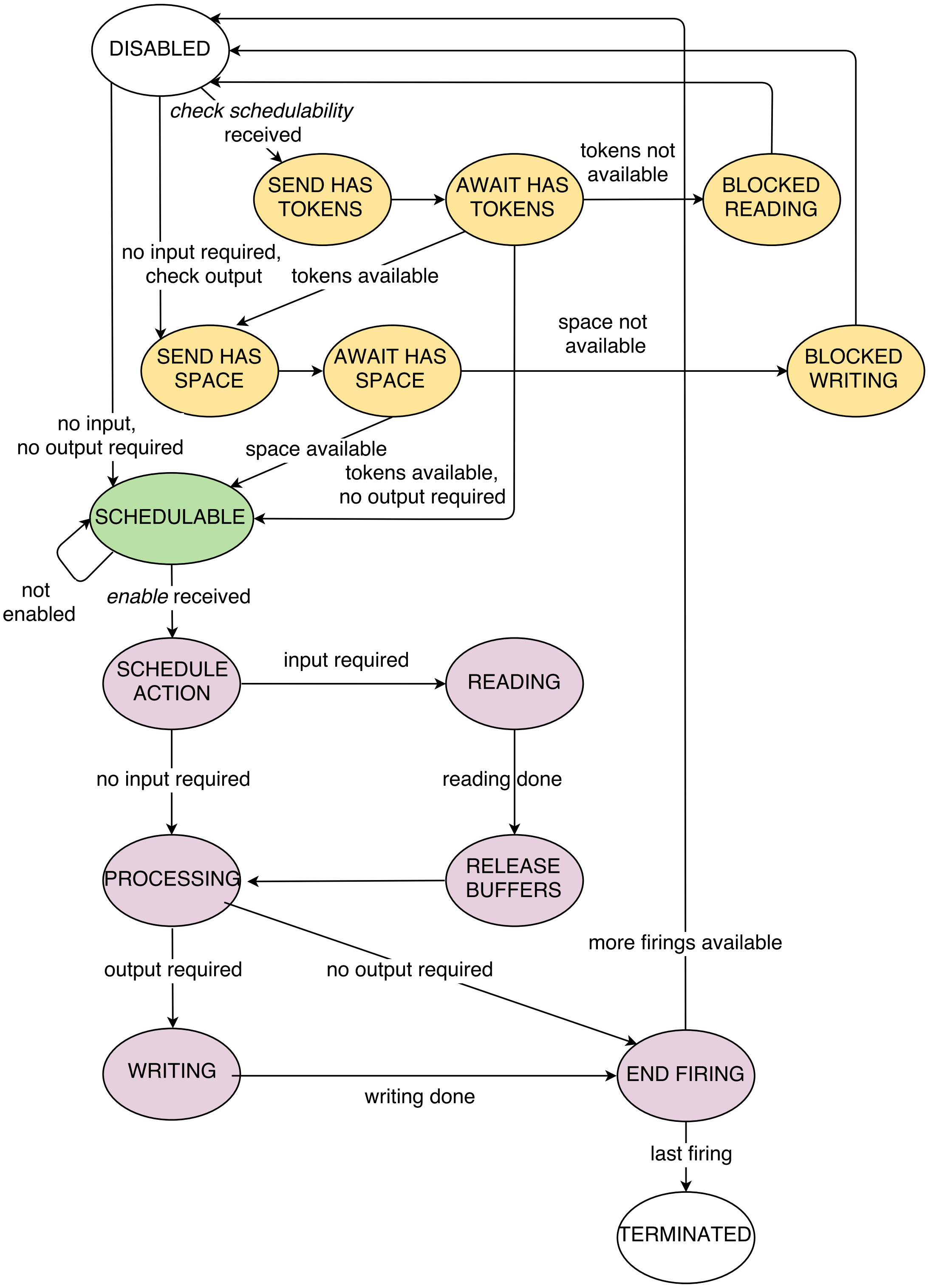


Figure 9.3 – Atomic actor model: state transition.

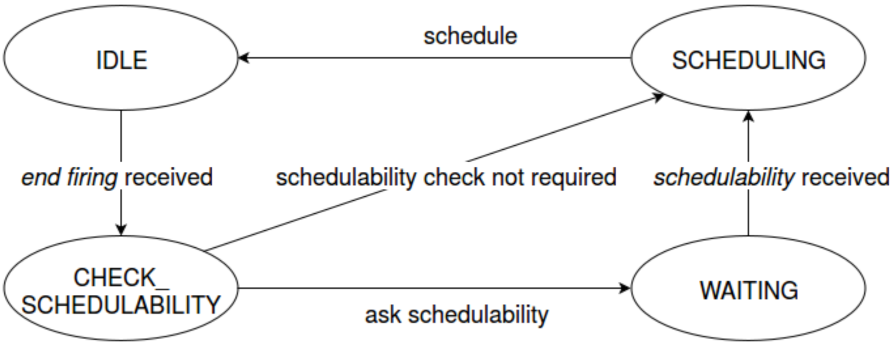


Figure 9.4 – Atomic actor partition model: state transition.

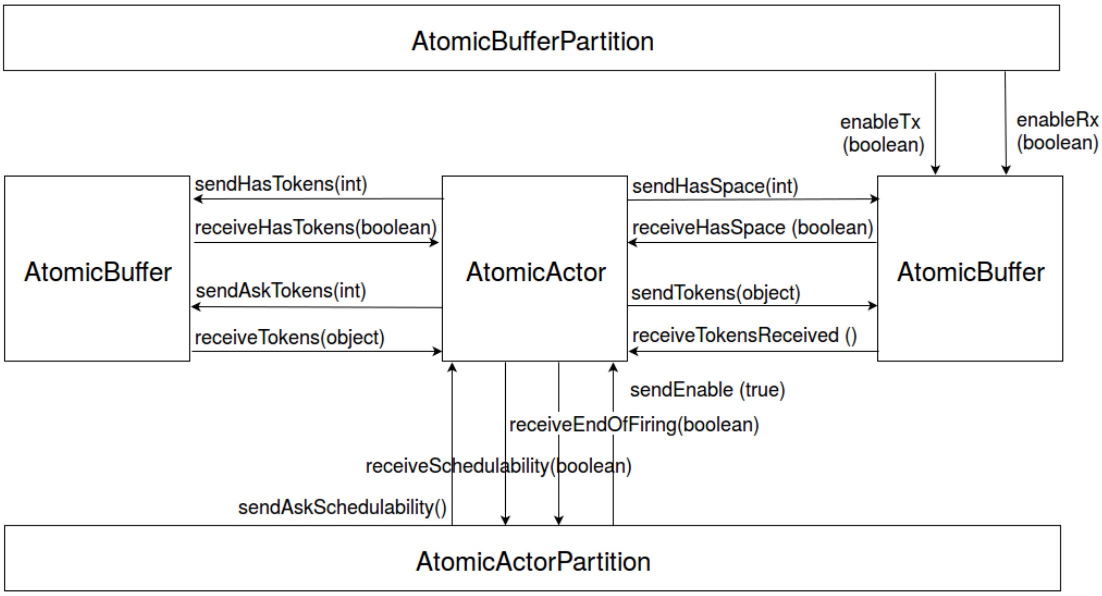


Figure 9.5 – Components interaction and exchanged signals.

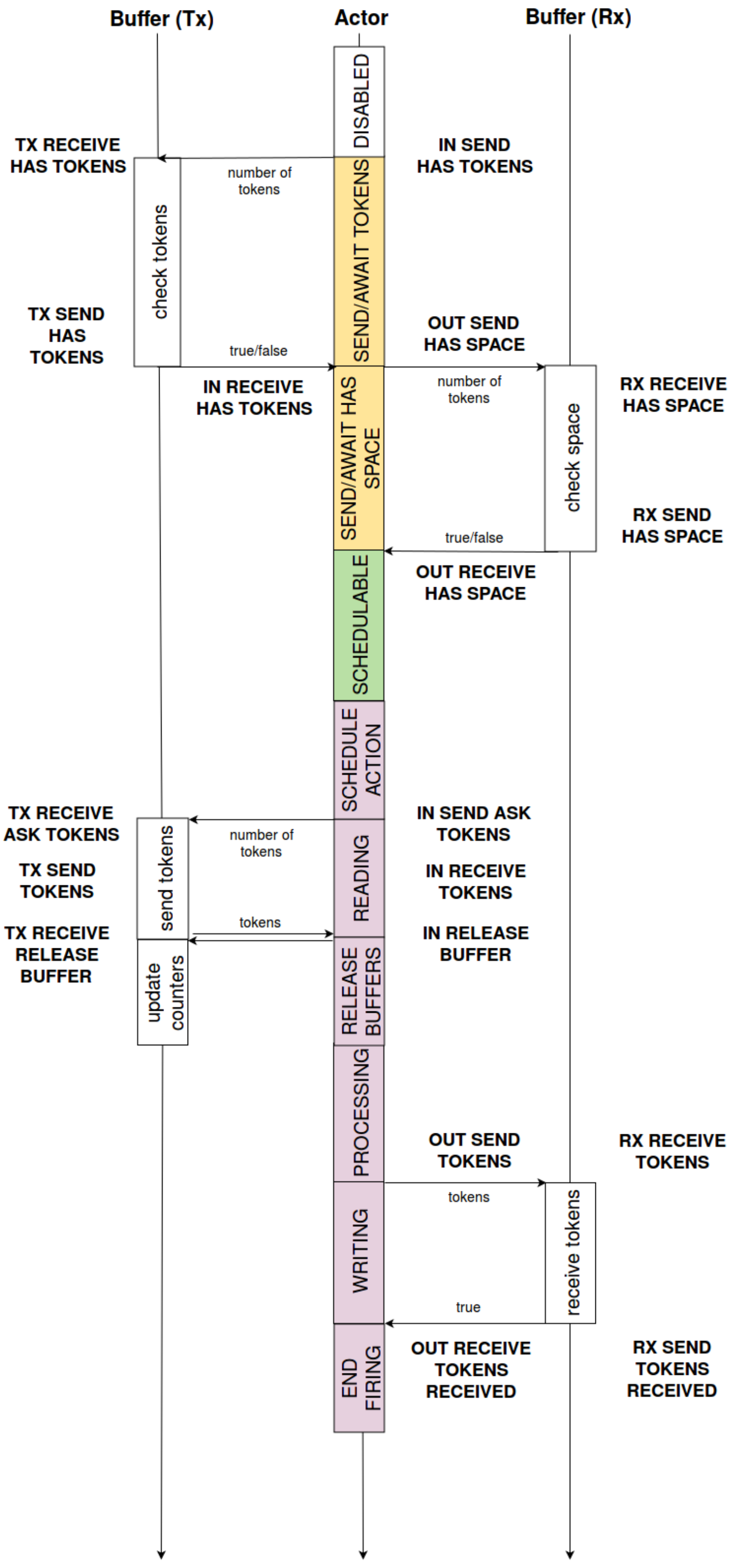


Figure 9.6 – Communication procedure between an actor and a buffer.

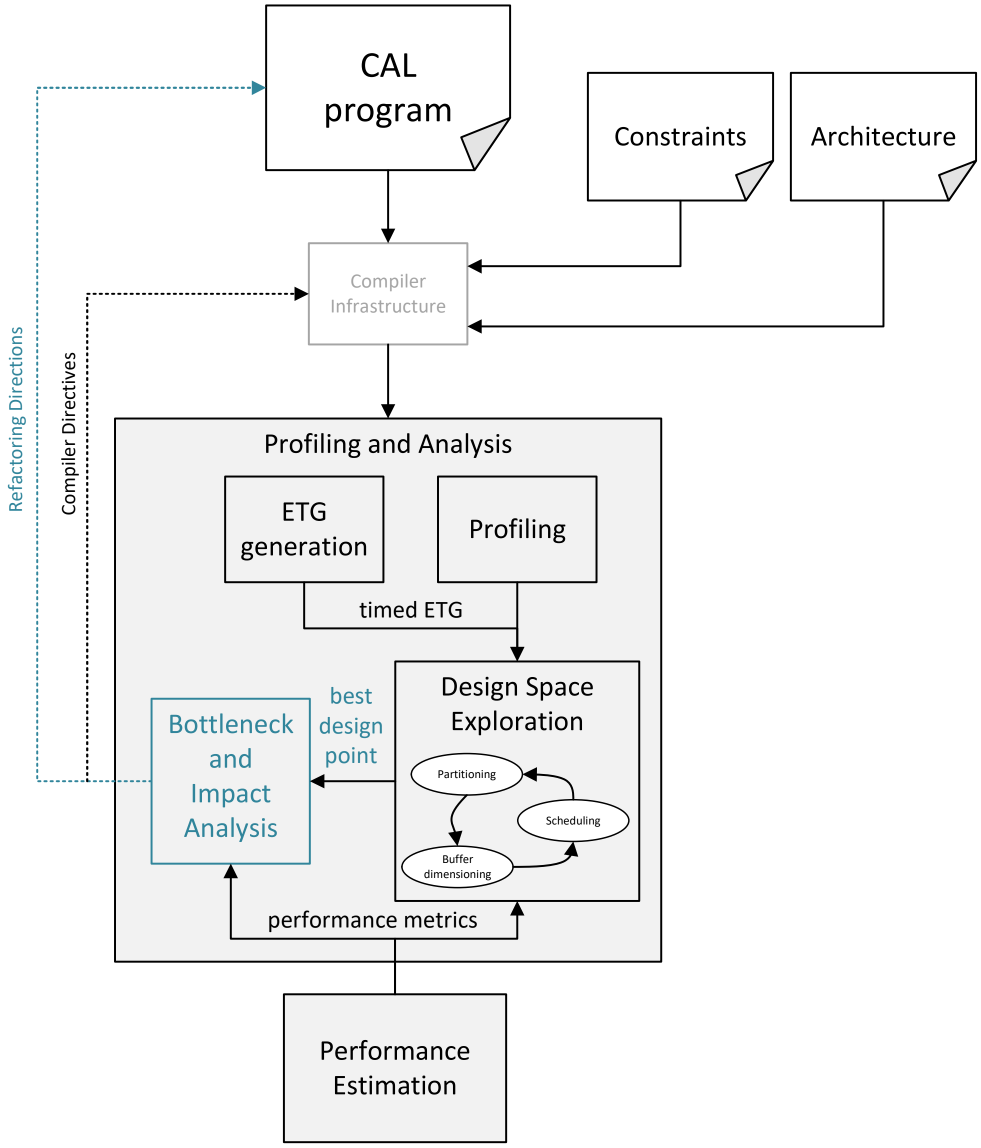


Figure 9.7 – System development design flow: bottleneck and impact analysis.

10 Experimental results

The heuristics and methodologies discussed in the previous Chapters, that is, profiling, partitioning, scheduling, buffer dimensioning, performance estimation and bottleneck analysis are encapsulated in different stages of the dataflow design flow discussed earlier in Section 3.5. An updated design flow emphasizing the newly formalized and developed stages is presented in Figure 10.1. This Chapter reports the experimental results conducted with regards to different components of that flow.

The process of design space exploration consists of finding appropriate partitioning, scheduling and buffer dimensioning configurations. These configurations, as discussed in Chapter 6 are closely related and lead to establishing a fixed execution order of the firings associated with a certain performance. In order to solve the overall problem, it is iterated over the subproblems. This iteration does not have to follow any particular order, hence at each stage it is essential to find a high-quality solution to a subproblem considering given solutions to the other two subproblems. Different parts of this Chapter describe experiments at different iteration stages.

The dataflow design flow corresponds directly to the VSS algorithm introduced in Chapter 7. Hence, in the next part of this Chapter, the algorithm is validated with a real design case comprising all previous partial results and methodologies.

10.1 Experimental set-up

This Section summarizes the set of designs and architectures used in the experiments. All applications have been implemented using the, described earlier, RVC-CAL formalism. They include: JPEG, MPEG4-SP and HEVC decoders. These applications are characterized by different complexities and various levels of dynamism occurring inside the dataflow actors. The platforms considered in the experiments are TTA and Intel 86x64.

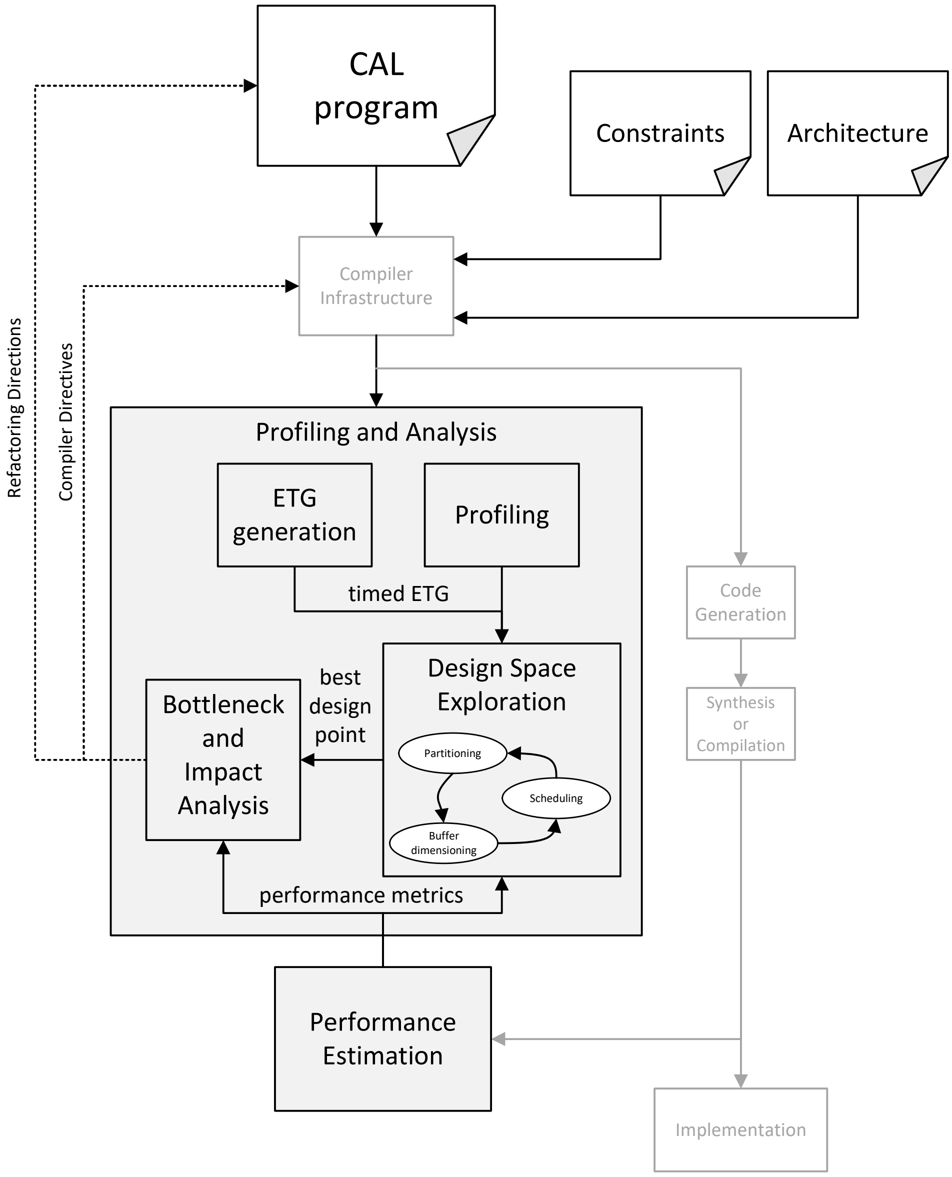


Figure 10.1 – Extended system development design flow for *CAL* dataflow programs.

10.1.1 JPEG decoder

The first application used in the experiments and described using the RVC-CAL formalism is a JPEG decoder [227]. The dataflow network for this design is depicted in Figure 10.2. It consists of 6 actors in total. The main functional components are: Parser, Huffman decoder, Inverse quantization (IQ) and Inverse discrete cosine transform ($IDCT$) block, respectively. The decoder takes a compressed 4:2:0 bit-stream as input and outputs a decoded image.

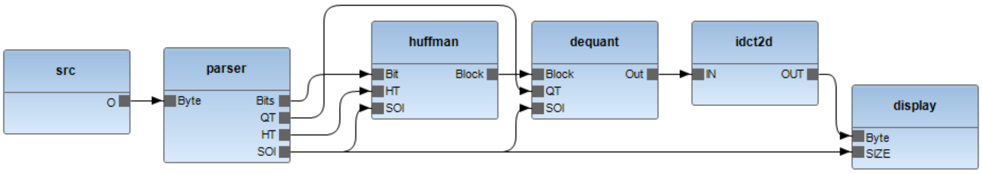


Figure 10.2 – JPEG decoder: dataflow network.

10.1.2 MPEG4-SP decoder

MPEG4-SP is an $RVC-CAL$ implementation of the full MPEG-4 4:2:0 Simple Profile decoder standard ISO/IEC 14496-2 [66, 151]. The main functional blocks include: Parser, Reconstruction block, 2-D inverse discrete cosine transform ($IDCT$) block and Motion compensator. All of these functional units are hierarchical compositions of actors in themselves. In the first place, the Parser analyzes the incoming bit-stream and extracts the data from it. Then, it feeds the data to the rest of the decoder depending on where it is required. The Parser is a single actor that is composed of 71 actions. Therefore, it is the most complex actor in the entire decoder. In the next step, the Reconstruction block performs the decoding that exploits the correlation of the pixels in neighboring blocks. The $IDCT$ is the most demanding actor in terms of resources, since it performs most of the computations of the decoder. Finally, the Motion compensator adds blocks selectively by issuing from the $IDCT$ the blocks taken from the previous frame. Consequently, the Motion compensator needs to store the entire previous frame of video data and access it with a certain degree of randomness.

An illustration of a dataflow network for the MPEG4-SP decoder (the variant with serial processing of Y, U and V components) has been already presented in Section 6.5. Figure 10.3a depicts the differences in the network when these components are processed in parallel. These two designs are referred to as MPEG4-SP Serial/Parallel, respectively. They differ significantly in complexity. The complete design of MPEG4-SP Serial consists of 17 actors, whereas MPEG4-SP Parallel of 34 actors. The decoder takes a compressed 4:2:0 bit-stream as input and outputs a decoded video sequence.

10.1.3 HEVC decoder

High Efficiency Video Coding (HEVC) standard [228] represents the state-of-the-art in video coding. Its compression performance is significantly improved compared to the previous, Advanced Video Coding (AVC) standard [229, 230], however, at the cost of higher complexity. Due to this complexity, efficient HEVC codec implementations are vital in video products approaching 4K (2160p) resolution. It was designed with the intention to define new parallelization tools [231] capable of taking advantage of execution on a multi-core architecture.

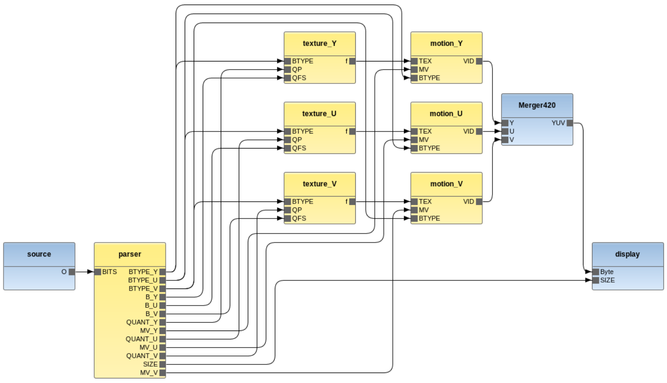
A dataflow implementation of an HEVC decoder [199] has been created according to the specifications standardized in the MPEG-RVC Framework [66] and in its basic form consists of 13 actors. Its functional units correspond to the algorithmic blocks of the HEVC standard decoder and include: bit-stream Parser, Motion Vector Prediction, Inter Prediction, Intra Prediction, $IDCT$, Reconstruct Coding Unit, Deblocking Filter, Sample Adaptive Offset Filter and Decoding Picture Buffer. Figure 10.3b illustrates the complete network with all components.

Due to the repetitive communication of some large data structures (like Decoding picture buffer) from one functional unit to another, an important factor improving the performance of the dataflow implementation of HEVC is the sharing of these big data structures among different functional units and making them read/write from/to these shared data structures. Note that this approach of sharing data-structures can only be used in the implementations targeting platforms supporting shared memory architectures. The experimental results showed that such a shared memory implementation does not affect the potential parallelism, but instead allows the functional units to work in a more synchronized way.

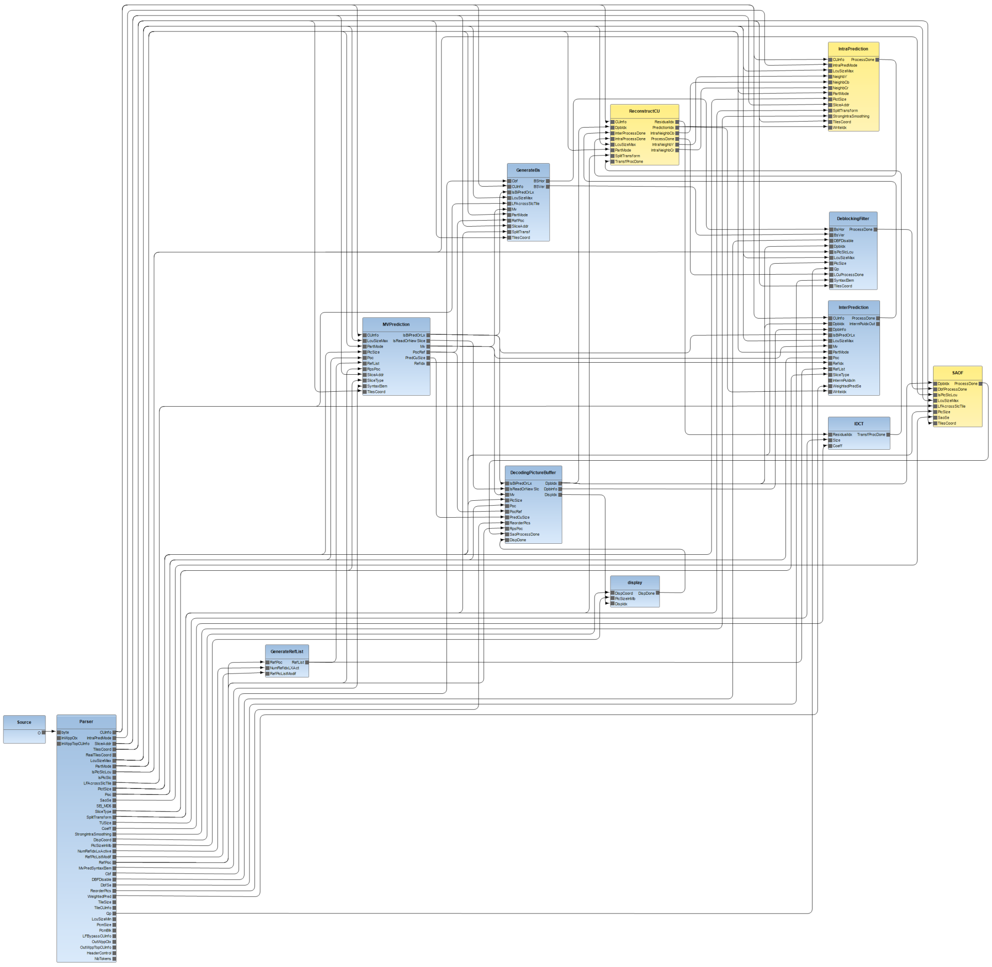
Another possible concept to apply in the implementation is a multi-parser configuration. According to this scheme, multiple parsing units (multiple instances of the Parser) can independently parse the bit-stream portions corresponding to decoding units (WPP-rows, tiles, slices) at the same time. Parsed data is then combined by a Merger into a single stream to be processed by the rest of the decoder.

10.1.4 Target platforms

The experiments have been performed on different platforms. One of the objectives was to demonstrate that the proposed model of execution, when supported by a profiling methodology, can be successfully used on different types of target architectures. Furthermore, depending on the properties of the platform and its notion of uncertainty, different behaviors can be modeled and the accuracy of the proposed methodologies can be verified. A large portion of the experiments have been performed on TTA using a cycle-accurate simulator [133]. These experiments aimed at the validation of the model and the proposed DSE heuristics in the circumstances where the platform can be easily modeled, profiled and its behavior is highly



(a) MPEG4-SP Parallel decoder.



(b) HEVC decoder.

Figure 10.3 – Dataflow networks.

deterministic. The next step was to move towards platforms which are more difficult to model. In this part, various Intel-based platforms were considered, which are characterized by different underlying micro-architectures, cores numbers and operating systems, as summarized in Table 10.1.

Table 10.1 – Configurations of Intel platforms.

Platform	Hardware and Operating System Details
Machine 1 (M1): 1 x 8 cores Desktop PC	– CPU: Intel(R) Core(TM) i7-5960X CPU @ 3.00GHz – Memory: 32GB RAM – OS: Ubuntu 15.04 (Linux 3.19.0-15-generic x86_64) – Compiler: gcc 4.9.2 (Ubuntu 4.9.2-10ubuntu13)
Machine 2 (M2): 1 x 4 cores Desktop PC	– CPU: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz – Memory: 8GB RAM – OS: Ubuntu 14.04.2 LTS (Linux 3.16.0-61-generic x86_64) – Compiler: gcc 4.8.4 (Ubuntu 4.8.4-2ubuntu1 14.04)
Machine 3 (M3): 2 x 10 cores Server	– CPU: Intel(R) Xeon(R) E5-2660 CPU @ 2.60GHz – Memory: 264GB RAM – OS: CentOS Linux 7.2.1511 (Linux 3.10.0-327.28.2.el7.x86_64) – Compiler: gcc 4.8.5 (Red Hat 4.8.5-4)

10.2 Partitioning: experiments on Transport Triggered Architecture

The focus of the experiments described in this Section was to validate the proposed partitioning heuristics, and hence explore the design space in terms of configurations of P, whereas the configurations of S and B are fixed. The dynamic scheduling policy used in the experiments for both, estimation and execution on the platform was *NnP*, following the definition in Section 8.3.2. The buffer size configuration assigned each buffer with a fixed size of 512 tokens. This size has been verified to keep the overall *blocked writing* time of the actors, as defined in Section 8.1.3, at a negligible level.

Finding appropriate dataflow programs for validating partitioning algorithms is not a trivial task. In fact, it is essential to perform the experiments with an application that, in principle, can provide a sufficient level of parallelism. If this condition is not satisfied, it is likely that *any* partitioning algorithm can result in a satisfactory performance. The test application was the MPEG-4 SP Parallel design and the target platform was an array of TTA processors. For this particular platform, MPEG4-SP Parallel provides the potential parallelism, as defined in Section 4.2.4, around 6.28.

10.2.1 Methodology of experiments

Most of the tools used for the experiments are the components of the *Turnus co-design framework* [16, 232, 233]. They include: the generation of an *ETG* for a given statistically meaningful input stimulus, the performance estimation tool exploiting the *ETG* and the results of the platform profiling and the generation of partitioning configurations using different algorithms. Complementary units in this workflow are the profiling of the *TTA* architecture and a *TTA* cycle-accurate simulator [133] that allows verification of the estimated results in terms of a real execution time obtained on the platform. The complete workflow is presented in Figure 10.4.

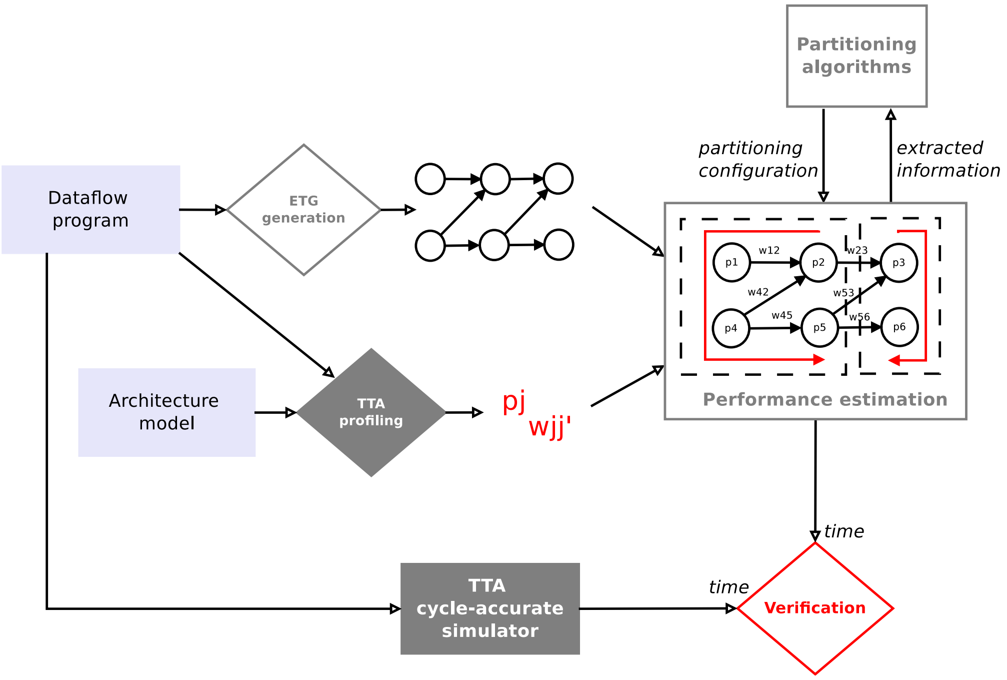


Figure 10.4 – Partitioning heuristics - TTA - experimental workflow.

The partitioning configurations have been generated using each of the described algorithms for the number of processors between 2 and 8. Considering the choice of application, 8 units should already approach its potential parallelism. For the local search methods that require specifying an initial solution, in each case, two sets have been tested: the random one and the one generated by the WB algorithm. Such a choice has been made in order to provide the algorithms with possibly good, as well as bad, initial configurations and also observe their sensitivity to the quality of an initial solution. The evaluation of the solutions generated by each algorithm has been accomplished by means of performance estimation calculating the total execution times in clock-cycles. Based on these values, the speed-up versus the mono-core execution has been calculated in each case. The presented results target the values of the speed-up in order to relate them easily to the potential parallelism of the application.

Finally, the results obtained by estimation have been verified by the platform executions. This verification is, however, handled in more detail later in Section 10.6.

10.2.2 Parameters tuning

As described in Section 8.1.4, apart from the length of the *tabu tenure*, TS is sensitive to two parameters that need to be properly tuned: the time limit T and the percentage e of explored neighbor solutions. For that purpose, 3 runs on a set of initial partitioning configurations have been performed for each: $N^{(B)}$, $N^{(I)}$, $N^{(CF)}$, and $N^{(R)}$. First, with a fixed value of $e = 0.5$, each TS variant was performed 3 times on the initial set of partitioning configurations. For each run, TS was stopped any time 5 minutes have elapsed without improving the best encountered solution (during the current run) by at least 1%. Parameter T has been set as the largest encountered stopping time (minus 5 minutes) among all these experiments.

Next, with the selected value of T, all TS variants were tested with different values of $e \in 0.2, 0.4, 0.6, 0.8$ in order to deduce the best value for each neighborhood type. The value of $e = 0.4$ has been chosen as the one providing the best average results among all instances in the test set. It has been also observed that if a method is performed several times on the same instance, it gets similar results. This indicates the robustness of the proposed approach.

Proper tuning of parameters is important in order to reliably compare all of the iterative methods. All stages of parameter tuning in the proposed methodology have been performed automatically. The time limit T tuned for TS has been used also as a time limit for the DLS methods. Additionally, since these methods tend to get quickly stuck in local optima, a *restarting* procedure has been implemented. If DLS finishes before consuming the given time limit, it is restarted with a new random solution. At the end, the best found solution (among the restarts) is returned.

10.2.3 Greedy and descent local search procedures

Table 10.2 contains the speed-up values obtained for partitioning configurations generated with the WB and BP algorithms along with the values estimated for a random set of configurations. Tables 10.3 and 10.4 contain the results obtained for the $IDLS$ and $CFDLS$ heuristics for the two sets of initial partitioning configurations.

Since the purpose of a greedy constructive method is to build a solution from scratch, an important property is the scalability of the performance. In this case, both algorithms scale, however the BP achieves a saturation already around 5 processors, unlike the WB that scales further. The maximal speed-up obtained for BP configurations is similar to the random configurations, but is achieved on a smaller number of processors (5 vs 8). Applying the $IDLS$

and *CFDLS* methods in all the cases improved the initial solution, but the improvement is greater for *CFDLS*. The quality of the solution provided by the *DLS* heuristics depends also strongly on the quality of the solution provided as a starting configuration.

Proc.	Workload Balance	Balanced Pipeline	Random
1	1.00	1.00	1.00
2	1.83	1.79	1.75
3	2.66	2.20	2.08
4	3.07	2.84	2.77
5	3.38	3.10	2.34
6	4.07	3.09	2.61
7	5.40	3.10	2.48
8	5.76	3.10	3.14

Table 10.2 – Speed-up: Greedy constructive procedures.

Proc.	IDLS	CFDLS
1	1.00	1.00
2	1.83	1.94
3	2.67	2.68
4	3.25	3.49
5	3.48	4.20
6	4.08	4.75
7	5.47	5.63
8	5.93	6.06

Table 10.3 – *DLS* Speed-up: balanced start.

Proc.	IDLS	CFDLS
1	1.00	1.00
2	1.77	1.95
3	2.09	2.32
4	2.78	3.19
5	2.35	4.27
6	2.95	3.98
7	3.02	3.93
8	3.85	4.23

Table 10.4 – *DLS* Speed-up: random start.

10.2.4 Tabu search

The first experiment aimed at confirming the most beneficial types of moves. It has been performed separately for each type of neighborhood structure. In the first execution, only REINSERT moves were allowed, whereas in the second one, SWAP moves were also included. SWAP moves were not considered alone, since they do not lead to any change of the initial size of each partition (resulting in a non-connected solution space). The results of this comparison for each neighborhood type are presented in Tables 10.5 - 10.12. Along with admitting the SWAP moves, a significant improvement has been brought only to the $N^{(B)}$. In fact, the performance of $N^{(B)}$ based on REINSERT only was very poor and a slight improvement was introduced only for certain initial configurations. It relies on the fact that the possible space of moves is very narrow in this case (only actors from the most occupied partition are taken into consideration) and the tabu list can be very restrictive. Since it also aims at balancing the workload, for a higher number of processors, it is not rare to encounter a solution where the heaviest bottleneck actor is placed alone on the processor. Due to the solution definition described in Section 8.1.4, the algorithm cannot proceed from that point. A relative balancing of the workload between the two partitions instead of an overall balancing seems to be a much more effective approach.

For the other neighborhood structures, allowing SWAP moves decreased the quality of the final solution in the vast majority of cases. This might be due to the fact that SWAP moves unnecessarily increased the set of neighbor solutions and reduce the diversification of the method. Comparing the neighborhood structures, there are some conclusions that can be made. First, $N^{(I)}$ outperforms the other variants, including $N^{(CF)}$. This observation is contrary to what has been previously observed for the DLS heuristics, where a search based on *communication frequency* outperformed the *idle* optimization. This confirms that determining a local optimization criterion is one challenge, whereas employing an appropriate exploration strategy ($i.e.$, the TS framework) is the another. Finally, the results obtained for $N^{(I)}$ and $N^{(CF)}$ also prove that a *guided* choice of moves outperforms random selection. In other words, the complete freedom of choice when choosing a move, as for $N^{(R)}$, does not necessarily lead to competitive solutions.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.85	1.94
3	2.67	2.77
4	3.21	3.48
5	4.04	4.32
6	4.83	4.85
7	5.40	5.62
8	5.76	6.19

Table 10.5 – Speed-up: $N^{(B)}$ with balanced start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.75	1.95
3	2.08	2.74
4	2.77	3.30
5	2.34	4.13
6	2.61	3.68
7	2.48	4.04
8	3.24	4.76

Table 10.6 – Speed-up: $N^{(B)}$ with random start.

The final part of the experiments with TS aimed at a comparison of its two advanced variants. Taking into consideration the previous observations, the analysis targeted the neighborhood $N^{(B)}$ based on the SWAP moves, and the neighborhoods $N^{(I)}$, $N^{(CF)}$ and $N^{(R)}$ based on REINSERT moves. Since different types provide different sizes of the neighborhood sets, such sizes have been equalized according to the averaged values. For this reason, another parameter, namely the *admission rate*, has been introduced for each neighborhood structure. Admission rate expresses the percentage of moves that is generated at each iteration. For $N^{(I)}$ and $N^{(CF)}$, a given percentage of moves is extracted according to the priorities ($i.e.$, most idle or most communicative actors, respectively). For $N^{(B)}$ and $N^{(R)}$, since there are no priorities, the solutions are extracted randomly. The values of admission rate have been tuned as follows: 0.9

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.92	1.92
3	2.75	2.80
4	3.61	3.46
5	4.45	4.23
6	4.92	4.88
7	5.81	5.66
8	6.28	6.18

Table 10.7 – Speed-up: $N^{(I)}$ with balanced start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.93	1.94
3	2.66	2.66
4	3.36	3.27
5	3.94	3.59
6	4.73	4.12
7	4.31	4.65
8	5.95	4.65

Table 10.8 – Speed-up: $N^{(I)}$ with random start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.88	1.93
3	2.79	2.78
4	3.49	3.56
5	4.30	4.29
6	4.95	4.97
7	5.79	5.74
8	6.26	6.18

Table 10.9 – Speed-up: $N^{(CF)}$ with balanced start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.87	1.88
3	2.65	2.68
4	3.33	3.29
5	4.23	4.36
6	4.94	3.85
7	4.62	4.36
8	4.66	4.23

Table 10.10 – Speed-up: $N^{(CF)}$ with random start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.92	1.94
3	2.70	2.74
4	3.47	3.38
5	4.26	4.11
6	4.91	4.76
7	5.64	5.57
8	6.27	6.18

Table 10.11 – Speed-up: $N^{(R)}$ with balanced start.

Proc.	REINSERT	SWAP
1	1.00	1.00
2	1.94	1.90
3	2.65	2.29
4	3.30	3.16
5	4.01	3.20
6	3.65	3.53
7	4.02	3.47
8	4.56	4.36

Table 10.12 – Speed-up: $N^{(R)}$ with random start.

for $N^{(I)}$, 0.48 for $N^{(CF)}$, 0.16 for $N^{(R)}$, and 0.08 for $N^{(B)}$.

Tables 10.13 and 10.14 contain the results of the analysis of the advanced variants of TS. In almost all cases, PTS performed better than JTS and provided the results that, considering the previously mentioned potential parallelism of an application, can be considered as close-to-optimal. PTS and JTS were also less sensitive to the quality of the initial configuration. In fact, in a few cases, a random initial solution leads to better results than a balanced initial configuration.

Proc.	PTS	JTS
1	1.00	1.00
2	1.93	1.96
3	2.83	2.80
4	3.58	3.57
5	4.44	4.37
6	5.13	5.03
7	5.81	5.72
8	6.22	6.22

Table 10.13 – PTS and JTS speed-up: balanced start.

Proc.	PTS	JTS
1	1.00	1.00
2	1.96	1.93
3	2.77	2.71
4	3.62	3.39
5	4.42	4.26
6	4.98	4.31
7	5.10	4.86
8	5.72	5.12

Table 10.14 – PTS and JTS speed-up: random start.

Finally, Table 10.15 summarizes the best solutions obtained with PTS as the number of processors is increased. In addition to the values of the execution times expressed in clock-cycles and the speed-up, the distance between the execution time and the length of the critical path expressed in % is also highlighted. This value indicates how far is a given solution from the potential parallelism of the application. The last column contains the value of the estimation discrepancy for this particular solution.

Proc.	Time	Speed-up	CP dist [%]	Diff [%]
1	36938764	1.00	528	4.06
2	18839134	1.96	220	12.64
3	13045976	2.83	122	7.35
4	10200033	3.62	73	3.62
5	8321995	4.44	41	14.73
6	7194547	5.13	22	13.8
7	6354158	5.81	8	14.28
8	5941632	6.22	1	11.96

Table 10.15 – Improvement summary.

10.2.5 Discussion

Comparing the results obtained for all implemented algorithms, the first observation is that according to the decreasing quality of the output solutions, the algorithms can be ordered as follows: advanced TS variants, TS, DLS, and the greedy constructive procedures. This ranking is consistent, as a more refined approach outperforms a simpler one. It highlights that the specific ingredients belonging to a more refined method are relevant. Additionally, finding a good partitioning configuration for a small number of processors ($i.e.$, 2 or 3) is relatively easy and the differences between the solutions provided by different algorithms are minor. For instance, for the case of two processors, the difference between the solutions provided by the best and the worst algorithm is less than 6%. With the increasing number of processors, the differences become more significant. For the case of the WB algorithm, the biggest difference of 30% with respect to PTS can be observed at around 5 processors, whereas for the BP algorithm, on 8 processors, the difference goes up to 100% (Tables 10.2 and 10.13).

The comparison of different variants of TS leads to the conclusion that the resulting solution benefits from varying the definition of the neighborhood. In fact, both JTS and PTS outperformed the variants where only one type of neighborhood was taken into consideration. Among the advanced variants, the success of PTS over JTS might rely on two factors: (1) using the history of local search, which allows an adaptation of the search to the properties of the test case, and (2) the much smaller size of the neighborhood in each iteration that contributes to a diversification of the search.

An important aspect that must be also taken into account for evaluating the algorithms is the time required for their completion. It includes the evaluation time for all considered solutions in all iterations, extraction of the optimization criteria and computation of new solutions. For DLS and TS, the upper-bound on the time is defined by the user. However, for each algorithm, it has been observed when the last improving move (before a termination at the specified point) was performed. The averaged values among different instances are summarized in Table 10.16. For the TS, a big difference is visible between $N^{(R)}$ and the other variants. In fact, it is the time elapsed for $N^{(R)}$ that enforces the time limit for all other algorithms, but in the case of this particular variant, it does not necessarily correspond to the quality of the final solution. A promising observation can be made for the advanced variants of TS, since PTS not only provides the best results, but it also succeeds in ca. 10% shorter time than JTS. In all cases, the most significant factor is the number of iterations performed, since the performance estimation and, at the same time, the extraction of the optimization criteria much outstrip the cost of computing a new solution.

Algorithm	Time
WB	N/A
BP	N/A
IDLS	73 min
CFDLS	58 min
$N^{(B)}$	13 min
$N^{(I)}$	44 min
$N^{(CF)}$	84 min
$N^{(R)}$	318 min
JTS	308 min
PTS	276 min

Table 10.16 – Averaged time of the final improvement.

10.3 Partitioning: experiments on Intel 86x64 platforms

Due to the properties of Intel platforms (uncertainty, not negligible communication cost $etc.$), a thorough and precise comparison of different partitioning heuristics, as was done for the case of TTA, would be less meaningful. Hence, the focus of the experiments described in this Section went to a comparison of the partitioning methodology based on the analysis of ETG (with injected weights obtained during the profiling) and a state-of-the-art approach for partitioning available in $ORCC$.

10.3.1 Methodology of experiments

The referenced approach is based on a run-time actor profiling combined with a general purpose graph partitioning library [184], as described earlier in Section 8.1.1. The initial comparison of the two approaches has been described in detail in [15]. The tested application was the MPEG4-SP Parallel decoder running on machine M2. The flow of the experiments is illustrated in Figure10.5.

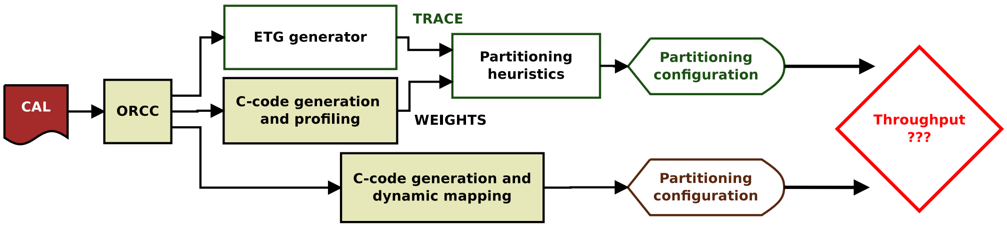

CAL
ORCC
ETG generator
C-code generation and profiling
TRACE
WEIGHTS
Partitioning heuristics
Partitioning configuration
C-code generation and dynamic mapping
Partitioning configuration
Throughput ???

Figure 10.5 – Partitioning heuristics - Intel - experimental workflow.

10.3.2 Heuristics comparison

The heuristics provided by the state-of-the-art approach are of different complexity. The simple variants include a round-robin placement (RR) and a strategy (WLB) which is, in principle, similar to the WB described in this work in Section 8.1.2. The other strategies employ different algorithms provided by the graph partitioning library. The values of speed-up compared to a mono-core configuration obtained for the *Foreman* sequence are summarized in Table 10.17. The heuristics that are compared in this approach are the basic procedure BP and two additional optimization procedures used in the variant, where the number of actors to be moved is fixed by the user. The values of speed-up for these cases are presented in Table 10.18. Finally, the PTS has been used, where the initial configurations (for 2, 3 and 4 processing units) were generated by the WB heuristic. As verified with the previous experiments on TTA, PTS comprises the advantages of different generators and is expected to provide the best results among different TS variants. Since for the case of Intel platforms the communication cost cannot be neglected without affecting the accuracy of estimation, the results presented in Table 10.19 rely on the iterative re-profiling procedure described in Section 8.1.6.

Proc.	MKCV	MKEC	MR	RR	WLB
1	1.00	1.00	1.00	1.00	1.00
2	1.86	1.83	1.84	1.36	1.81
3	2.22	2.43	2.44	1.65	2.23
4	2.14	2.26	2.14	1.63	2.26

Table 10.17 – Platform execution speed-up: SOA approach.

Proc.	BP	+ Idle. min.	+ Comm. vol.
1	1.00	1.00	1.00
2	1.62	1.84	1.89
3	2.48	2.49	2.01
4	2.11	2.28	1.93

Table 10.18 – Platform execution speed-up: BP with additional optimization procedures.

Proc.	PTS
1	1.00
2	1.84
3	2.50
4	2.95

Table 10.19 – Platform execution speed-up: tabu search with an iterative re-profiling procedure.

10.3.3 Discussion

It can be stated that already the first set of results (without considering the results of PTS) outperformed the referenced state-of-the-art approach. Furthermore, a relevant advantage of the proposed approach is the stability of the solution, as the analysis of ETG does not introduce any notion of randomness. In contrast, for the referenced strategies it could be easily noticed when running the same procedure several times for the identical input stimuli, that the resulting configurations can be close to the results of the proposed algorithm, as well as far less efficient than a simple RR placement (the results presented in Table 10.17 always contain the best result among several attempts). This stability can be considered as an advantage of the partitioning methodologies based on creating a model of execution, rather than only a run-time profiling.

The BP and its additional optimization procedures have some advantages, as well as some drawbacks. Apart from the, mentioned earlier, stability of the solution, the partitioning configurations are generated in a negligible time and, since the additional optimization procedures can be used without relying on the performance estimation, the accuracy of the performance estimation methodology does not affect the quality of the generated configurations. Hence, these methods can operate reasonably well in the situation, when the profiling data is less accurate. However, it is observed that the quality of the solution in different variants (basic procedure, idle time and communication volume minimization) vary a lot and although the new approach outperforms the referenced one, the best solution is located each time in a different variant. This is not the case for the PTS, which in any case generates the best results. The difference between the quality of the solution is visible especially for the case of partitioning spanned on 4 cores (Tables 10.18 and 10.19).

Some preliminary experiments included also applying the referenced state-of-the-art approach for the partitioning of a more complex design, such as the HEVC decoder. However, the quality of the multi-core configurations generated by different strategies was not much higher than for a mono-core execution. Hence, it was not useful in the process of VSS, results of which employing the algorithms validated in this Section, are reported later in Section 10.7.

10.4 Buffer dimensioning

This Section reports the experimental results performed for the bottom-up (constrained resources, throughput optimization) and top-down (constrained throughput, resource minimization) buffer dimensioning heuristic. The tested applications included the MPEG4-SP Parallel decoder (34 actors, 80 buffers) running on machine M2 and the HEVC decoder running on machine M1 (22 actors, 219 buffers). The sequences used in the experiments were a 30-frame QCIF *Foreman* bit-stream for MPEG4-SP Parallel and a 10-frame HD *BQ Terrace* bit-stream for HEVC. Based on the preliminary experiments it has been observed that high-quality configurations found for these sequences remain of high-quality also for the other sequences.

The first set of results (Tables 10.20 and 10.21) reports the differences in the throughput between a close-to-minimal buffer size configuration and an *extreme* configuration, where each buffer is assigned with a size equal to $2^{18} = 262144$. This configuration is considered as an approximation to an infinite buffer size. The corresponding values of B_{total} are indicated in the Tables. It is observed that the differences are remarkable for both designs and for different partitioning configurations. Hence, this justifies the necessity of optimizing combined throughput-memory objective functions.

Proc.	14k	21m
1	744	1005
2	1156	1613
3	1410	2240
4	1820	3138

Table 10.20 – MPEG4-SP Parallel: performance differences [FPS].

Proc.	93k	57m
1	41	53
2	42	79
3	51	103
4	52	121
5	57	123
6	54	130
7	53	127

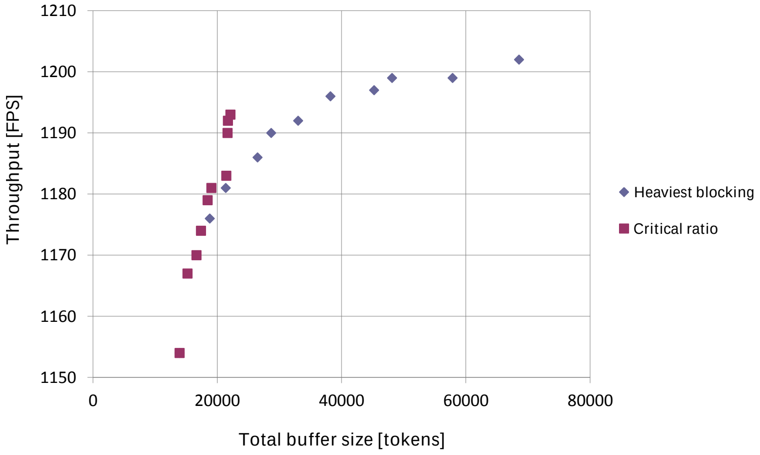
Table 10.21 – HEVC: performance differences [FPS].

10.4.1 Bottom-up: throughput optimization with constrained resources

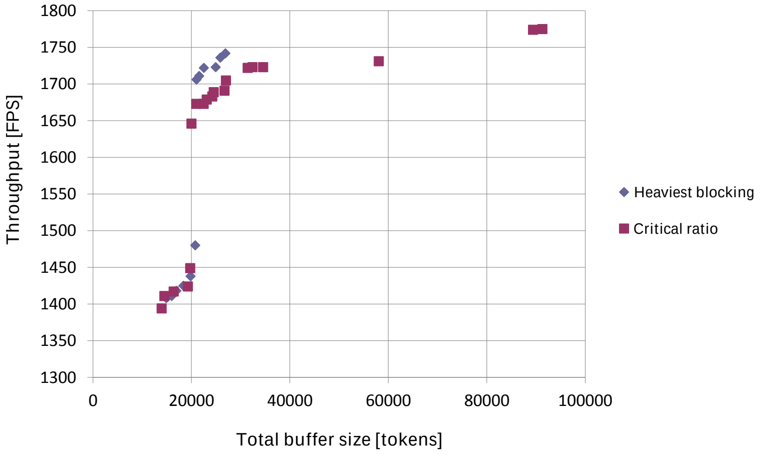
Figures 10.6-10.7 present the buffer size configurations generated in different iterations of the bottom-up heuristic in its two variants. Both variants were executed with the same upper bound on the number of iterations. The charts summarize the throughput with regards to the total buffer size of each configuration. For the case of the MPEG4-SP Parallel decoder, 4 processing units and the HEVC, 7 processing units it can be concluded that the *criticality ratio* ranking provides better results, because within the same number of iterations it leads to higher throughputs and smaller total buffer sizes. For the other two analyzed cases, it is not possible to make such a general statement, because one variant moves more towards higher throughputs, whereas the other one towards smaller total buffer sizes. It can be also observed that the partitioning configuration remains dominant over buffer size configuration, but for each partitioning configuration the buffer dimensioning improves the solution by 3-22%.

10.4.2 Top-down: resources minimization with constrained throughput

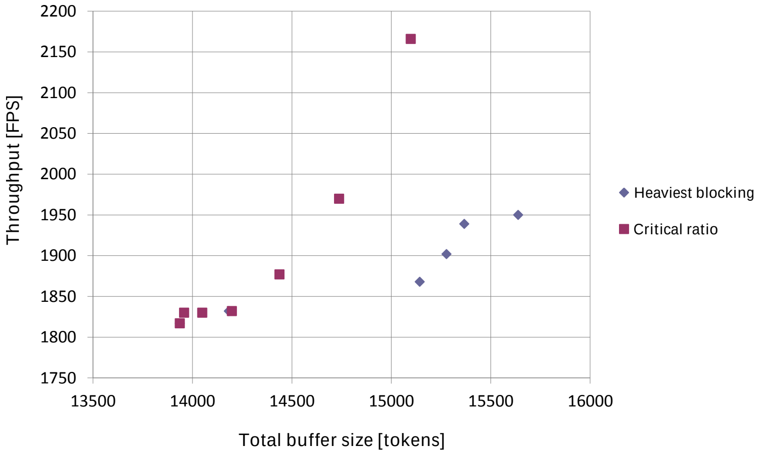
Figure 10.8 presents the results of buffer dimensioning performed with the top-down algorithm (Alg. 5) for different partitioning configurations in terms of the overall program throughput



(a) 2 cores partitioning.



(b) 3 cores partitioning.



(c) 4 cores partitioning.

Figure 10.6 – Bottom-up buffer dimensioning heuristic (MPEG4-SP Parallel decoder).

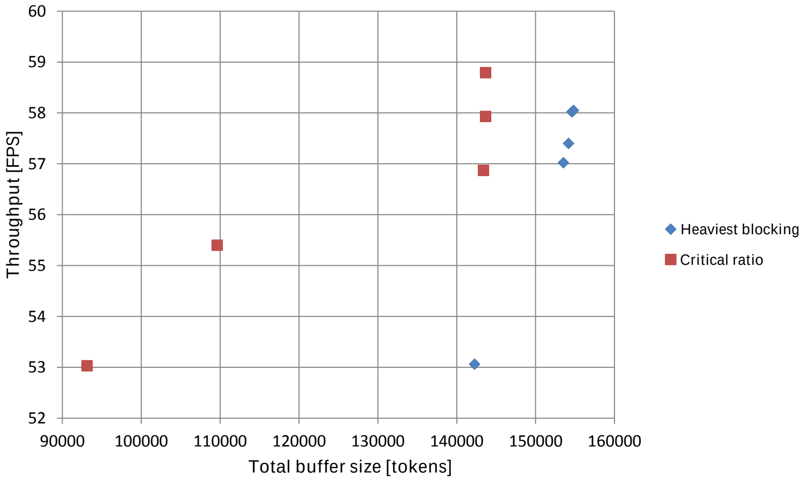


Figure 10.7 – Bottom-up buffer dimensioning heuristic (HEVC decoder, 7 cores).

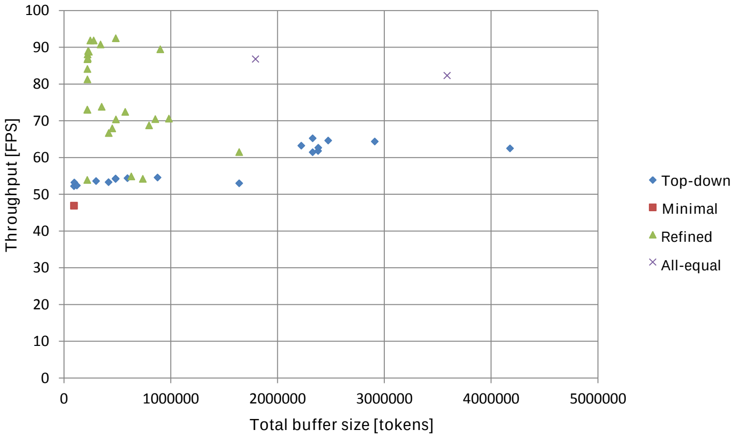
and the total buffer size B_{total}. The results have been generated for the HEVC decoder running on machine M1. Next to the spectrum of the solutions generated by the algorithm, the throughput for the minimal buffer size is indicated, as well as the throughput achieved for the configurations, where *all* buffers have the same size equal to 8192 and 16384, respectively. The points indicated as "Refined" correspond to the solutions obtained after assigning a lower and upper bound on the buffer size, as described in Sections 8.2.5. For the partitioning configuration on 7 cores, some results of the bottom-up algorithm are also indicated.

10.4.3 Multidimensional vs single-dimension exploration

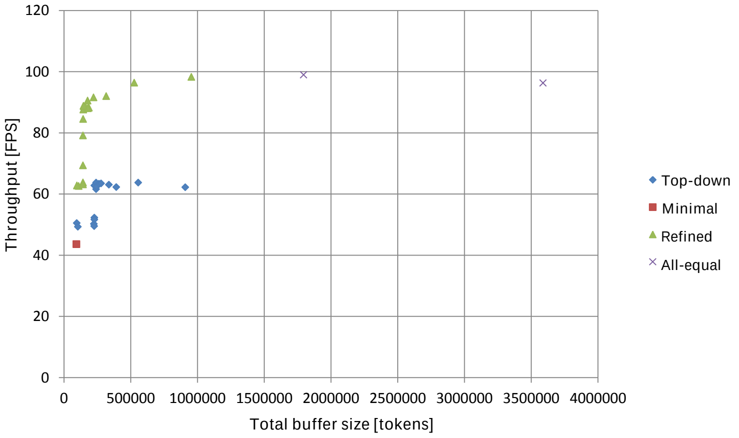
A design B-subspace is defined as the set of design points disregarding the P and S configurations, hence assuming a fully parallel execution. The two heuristics (bottom-up and top-down) can be used to explore the B-subspace as well as the multidimensional design space MDS. Let A (C) be the set of design points found in the B-subspace (MDS), respectively. Next, assign specific configurations of P and S to the design points of A, resulting in a set D of design points in MDS. Figures 10.9a and 10.9b illustrate the selected best solutions obtained with the two heuristics for the MPEG4-SP and HEVC designs, respectively. The improvement of the throughput and the increase of the buffer size have been calculated taking the initial deadlock-free configuration as the reference point.

10.4.4 Discussion

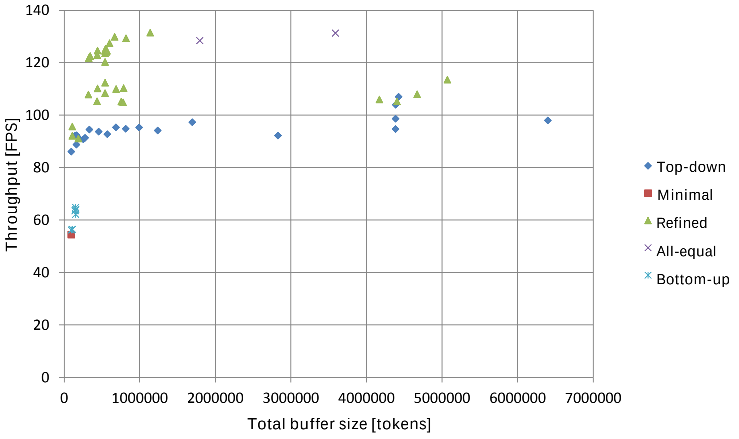
The very first observation of the results presented in the last Section proves the importance of buffer size optimization over providing only a deadlock-free configuration. In each con-



(a) HEVC, 2 cores partitioning.

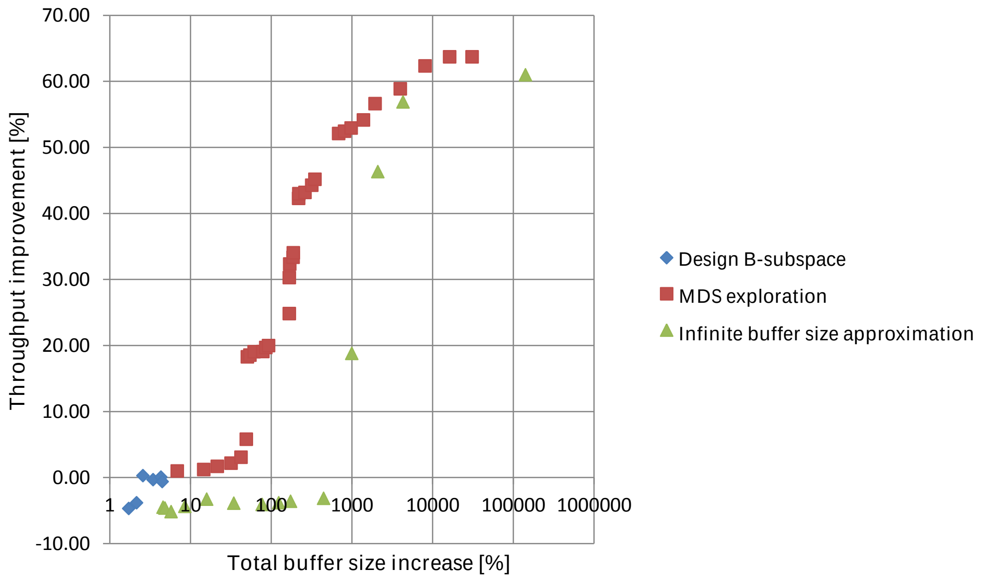


(b) HEVC, 3 cores partitioning.

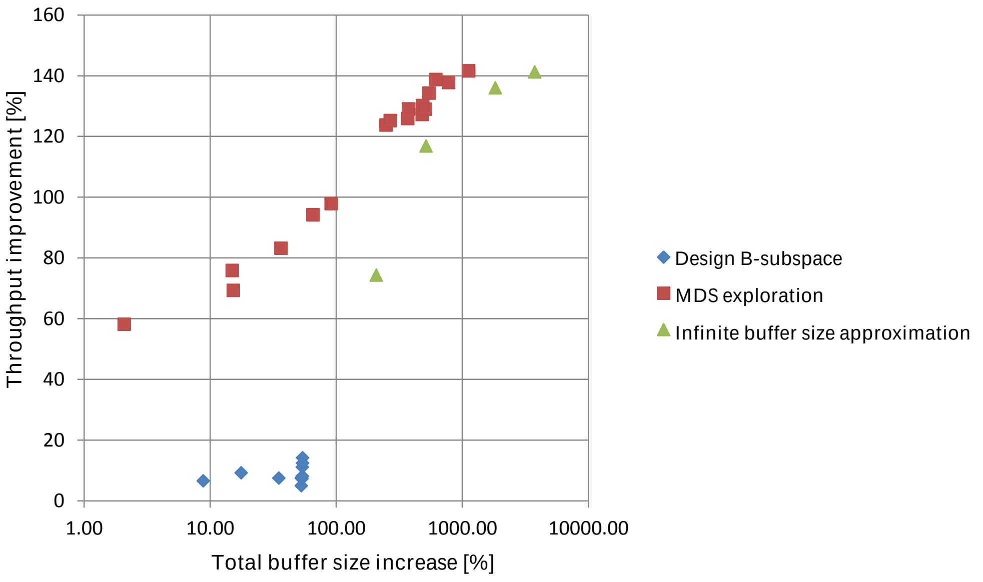


(c) HEVC, 7 cores partitioning.

Figure 10.8 – Top-down buffer dimensioning heuristic.



(a) MPEG4-SP Parallel decoder, 3 processing units.



(b) HEVC decoder, 7 processing units.

Figure 10.9 – *MDS* vs design *B*-subspace vs infinite buffer.

figuration, a solution which is good in terms of performance, gives at least 2 times better throughput than the minimal configuration. Furthermore, for instance, the termination point of the top-down algorithm (the one with the smallest B_{total}) is very close to the minimal one in terms of B_{total}, but still has up to 60% better throughput. The solutions generated with the bottom-up algorithm remain rather closer to the minimal configuration regarding the throughput. Finally, the refinement procedure brings a significant improvement of the throughput with only a slight increase of B_{total} (depending on the configuration). In the end, the top-down algorithm along with the refinement procedure managed to find a configuration with a comparable throughput (or even better, as for the case of the 2 cores partitioning), but a B_{total} at least two times smaller than for the "all = 8192" configuration.

Following the discussion about the exploration in the design B-subspace, one can observe that, in contrast with the design points in D, the design points in C build a quasi-monotonic curve, and are thus of better quality. In fact, the exploration of the design B-subspace terminates very quickly in local optima. More generally, if we disregard some dimensions (namely P and S) of the overall optimization problem, the algorithm working on the other dimensions (namely B) will not be as efficient as a general method with full flexibility ($i.e.$, accounting for P, S and B). This contrasts with some optimization problems for which fixing a dimension is a better approach [234, 235]. This is explained by the fact that in the DSE problem, all the dimensions are strongly correlated.

Another part of the observations related to Figures 10.9a and 10.9b is a comparison between the solutions obtained with the proposed buffer dimensioning heuristics and the solutions obtained without any systematic approach, that is, by increasing the size of *all* buffers or a randomly chosen fraction of buffers, until a saturation of performance is achieved. This curve of solutions is referred to as the *infinite buffer size approximation*. It is observed that the curve of solutions generated by using buffer dimensioning heuristics eventually converges to the curve of approximately infinite buffer size configurations. However, the curves clearly show that for different values of B_{total}, the solutions generated with the buffer dimensioning heuristics outperform the solutions in the other curve in terms of throughput improvement. In some cases, the solutions with comparable throughput are orders of magnitude apart from each other.

As a conclusion it can be stated that the bottom-up approaches might be more useful only for fully parallel, hardware implementations, where resources optimization plays an essential role and the optimization problem as a whole is less affected by various uncertainties. For the case of partitioned, software implementations, the top-down approach remains much more effective. The choice of the heuristic might depend also on the considered optimization scenario (described in Section 7.6). Observing the shape of the curves, the bottom-up (top-down) approach is more appropriate for the optimization scenario (S1) ((S3)), respectively.

Scenario (S2) might require both approaches and eventually, more priority has to be assigned either to U^k or U^b.

10.5 Scheduling

The experimental results presented in this Section considered the exploration of the design space regarding the configurations of S. The setup of P and B configurations is fixed. In all experiments, the two sets of P configurations spanned on up to 8 processors have been compared. The first set contained configurations where the overall workload of each partition is balanced (generated using the WB algorithm), whereas the second one was created as random configurations. Analyzing these two sets verifies whether a certain tendency in the performance for different scheduling policies occurs independently from the quality of partitioning. As for the buffer dimensioning, in order to minimize its influence on the results, a buffer size of 8192 tokens has been fixed as a reasonable approximation of an infinite buffer. This value has been used for profiling, platform execution and performance estimation.

10.5.1 Performance potential

For each partitioning configuration, the performance estimation tool calculated the execution times for 6 different scheduling policies (defined in Section 8.3.2) which have been used to obtain the speed-up versus the mono-core execution. The results for the balanced (random) partitioning configurations are presented in Table 10.22 (10.23), respectively.

No. of units	*NnP*	*RR*	*NnP/P*	*CNnP*	*COW*	*ECO*
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.78	1.99	1.79	1.70	1.89	1.99
3	2.27	2.84	2.36	2.31	2.30	2.79
4	2.72	3.57	2.75	2.68	3.28	3.46
5	3.14	4.20	3.29	3.62	3.85	4.14
6	4.41	4.67	4.43	4.67	4.72	4.68
7	5.04	5.12	4.99	5.14	5.10	5.12
8	5.41	5.46	5.41	5.47	5.49	5.46

Table 10.22 – Estimated speed-ups: balanced partitioning configurations.

It can be clearly observed that some policies tend to perform much better than others for almost *any* set of configurations. For example, *RR* outperforms *NnP* by more than 10% on average, and up to even 25%. The strategies relying entirely on IAP policy (*RR, COW, ECO*) are also in general more efficient than *NnP* and its derivatives which use the $IANnP$ at least partially. Surprisingly, *CNnP* does not perform very well. This may be due to the fact that, as

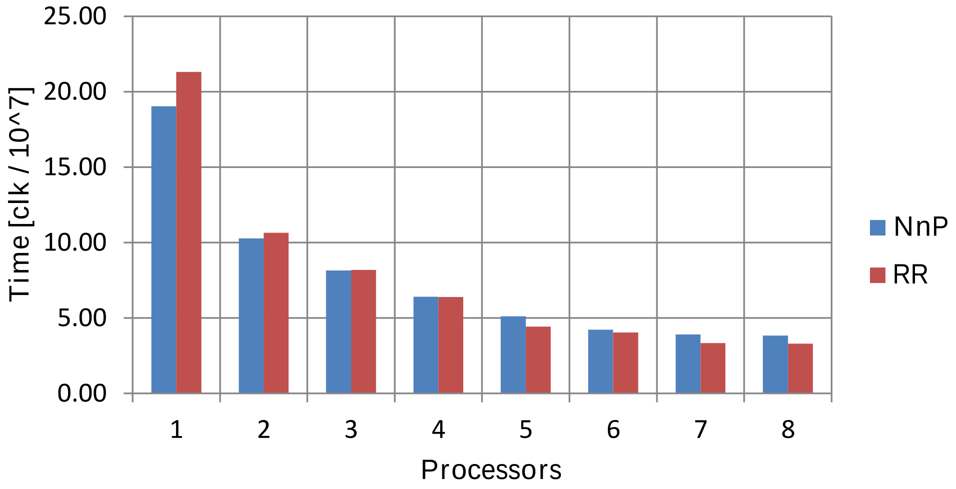
No. of units	*NnP*	*RR*	*NnP/P*	*CNnP*	*COW*	*ECO*
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.61	1.64	1.61	1.54	1.59	1.62
3	2.30	2.48	2.31	2.19	2.47	2.48
4	2.59	2.97	2.68	2.45	2.73	2.97
5	2.84	3.21	2.87	3.03	2.97	3.21
6	2.73	2.86	2.72	2.82	2.67	2.87
7	2.83	2.98	2.83	2.85	2.98	2.98
8	4.47	5.01	4.46	4.70	4.63	5.02

Table 10.23 – Estimated speed-ups: random partitioning configurations.

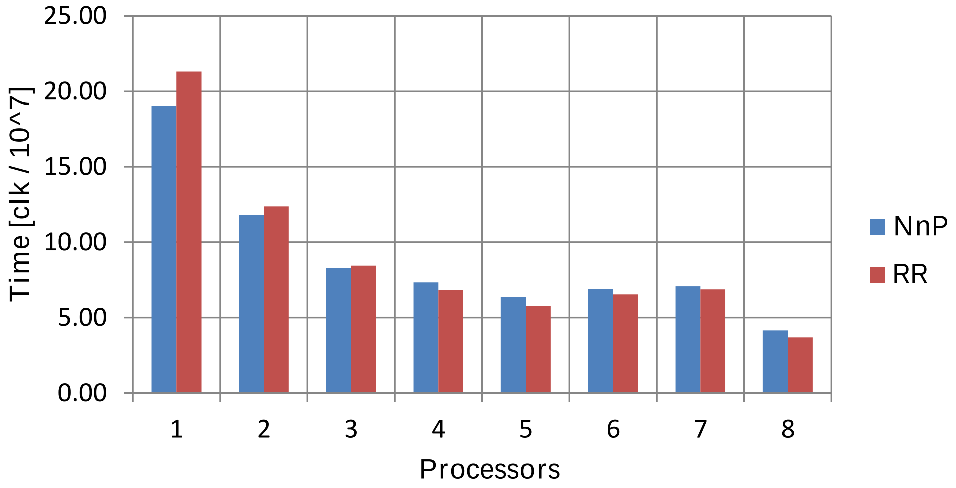
for the scheduling policy, when the critical firings were given a priority to fire, the critical path might have been modified by the concurrent decision of the scheduler. At the higher processor count all policies start to perform very similarly. This is due to the fact that as the average number of actors in one processor decreases, the possible choices of the scheduler become limited and less sensitive to the strategy it is using.

Another observation is that the balanced partitioning configurations resulted in much more diversity in the generated solutions than the random ones. This leads to the conclusion that the partitioning problem should be, in fact, considered dominant over the scheduling problem, as it is responsible for the room for improvement available to the scheduling policies. The same kind of observation was made in order and acceptance scheduling problems [236]. For further experiments, the two relatively extreme strategies (in terms of the overall number of preemptions assumed by the policy, but also in terms of performance differences) *RR* and *NnP* have been chosen. The scheduler inside the *TTA* back-end of *ORCC* has been modified to perform the scheduling on both an *NnP* and *RR* basis, so that a comparison of performances is also possible on the platform. The execution times are presented in Fig. 10.10a and 10.10b for balanced and random configurations, respectively.

The same tendency can be, again, observed in both sets of partitioning configurations. It thus confirms the legitimacy of the partitioning setup applied to the design space for the exploration of scheduling. Since partitioning configurations of different quality behave in the same way for different scheduling policies, using performance estimation in order to tune the scheduling policy for the metaheuristic search of high-quality partitioning configuration seems to be a justified direction. At the beginning, that is, up to 3 units, *NnP* outperforms *RR* on the platform. However, the difference between them gradually decreases. At 4 units and above, *RR* achieves a better performance. In spite of a higher discrepancy and inefficiency for a small number of units, the modified scheduler *RR* brought up to a 14.5% of improvement.



(a) Balanced partitioning configurations.



(b) Random partitioning configurations.

Figure 10.10 – TTA platform execution.

10.5.2 Scheduling cost

In order to find an explanation for the discrepancies reported earlier, the metrics for the numbers of checked and, respectively, failed conditions is used. Tables 10.24 and 10.25 present the normalized numbers of clock-cycles obtained for the two considered scheduling policies (NnP and RR). A positive value of difference (expressed in %) indicates that the real value is larger than the estimated value (underestimation), otherwise the execution time is overestimated. For each scheduling policy, the numbers of checked and failed conditions (as described in Section 8.3.3) have been counted and the normalized values are presented in Tables 10.26 and 10.27. The numbers of checked and failed conditions are calculated per 100 successful firings, per partition.

Proc.	TTA [clk]	PE [clk]	diff [%]
1	19.04	19.01	0.15
2	10.27	10.67	-3.81
3	8.15	8.36	-2.52
4	6.40	7.00	-9.37
5	5.12	6.05	-18.22

Table 10.24 – Execution times *NnP*.

Proc.	TTA [clk]	PE [clk]	diff [%]
1	21.31	19.01	10.81
2	10.64	9.57	10.10
3	8.18	6.69	18.23
4	6.39	5.33	16.65
5	4.42	4.53	-2.45

Table 10.25 – Execution times *RR*.

Proc.	NnP	RR
1	33.051	143.934
2	27.223	90.242
3	25.547	64.115
4	28.198	55.711
5	23.891	46.761

Table 10.26 – Conditions checked.

Proc.	NnP	RR
1	0.017	47.968
2	0.526	24.048
3	0.106	17.099
4	0.690	13.817
5	0.368	10.918

Table 10.27 – Conditions failed.

Logically speaking, it is expected that the cost of intra-partition scheduling is proportional to the number of actors in each partition. In other words, if a small number of processors is considered, the number of actors in one processor is relatively large, so the scheduling cost is also higher. This expectation corresponds well with the statistics provided on the numbers of conditions, especially regarding the fraction of checked conditions. In all cases, the *RR* policy is characterized with bigger values than the *NnP*. In some cases the difference is quite large, for instance, for the mono-core configuration the fraction of failed conditions for *RR* is almost 3000-times larger than for the *NnP*. Generalizing the values, it can be concluded that having a successful firing with the *RR* requires checking approximately 60% more conditions than for the case of *NnP*. Hence, since the performance estimation does not model the intra-partition scheduling cost, it can be concluded that for the *NnP* this cost is rather negligible, whereas for the *RR* it cannot be considered negligible.

10.5.3 Discussion

The observations related to the experiments presented in the last Section lead to interesting consideration of extensions and improvements. Naturally, the first direction might be an investigation of the opportunities for measuring and modeling the scheduling cost. It must be noted that this can be performed only as a matter of approximation, since the real cost might be subject to multiple factors, such as the level of dynamism inside the actors in a certain partition, their complexity (*i.e.*, the number of input/output conditions) or even their order of appearance. Nevertheless, the scheduling cost could be modeled as a function of checked/-failed conditions, where each check/failure is assigned a certain value. Furthermore, since the results confirm that an appropriate choice of the scheduling policy can provide not negligible performance improvements, further studies on the development of more sophisticated scheduling policies seem promising. Minimizing the numbers of conditions checked/failed, as provided by the estimation, could be taken as an indicative optimization criterion.

On the other hand, as confirmed by the experiments, any scheduling policy is related to two aspects: the performance gain coming purely from enforcing a certain order of execution of the firings and the run-time cost of establishing this order. According to the estimation, regarding the first aspect the NnP policy is actually the *worst* choice among the described policies. It is, however, very *economical* in terms of the number of checked conditions. For this reason, using the NnP policy for SW implementations is a popular choice. Instead, it would be worth investigating if the cost of other policies can be reduced by, for instance, identifying some static regions in the dynamic execution.

10.6 Performance estimation

This Section describes the experiments related to validating the performance estimation (PE) tool on different platforms. The PE tool is also referred to as the Trace Processor (TP). The objective of the experiments was the estimation accuracy regarding the correspondence of the estimated number of clock-cycles and the real number obtained on a platform, as illustrated in Figure 10.11.

10.6.1 Estimation accuracy on TTA platform

The initial validation targeted a deterministic, predictable and easily measurable platform, such as TTA. In this case, the test application was the MPEG4-SP Serial decoder. In the first stage, the number of clock-cycles obtained from the TP was compared with the values produced by the cycle-accurate multi-core TTA simulator for different numbers of units. As a starting point, the partitioning configurations generated by the BP algorithm (at one of the

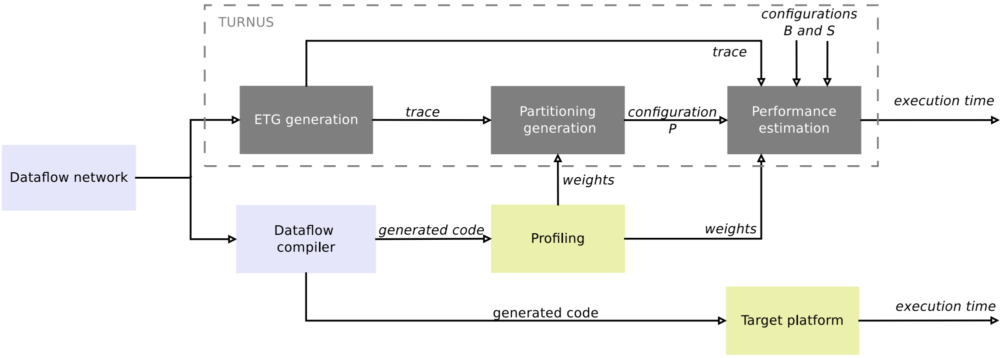


Figure 10.11 – Performance estimation validation - experimental workflow.

development stages) were taken and they can be reasonably considered as configurations of high-quality. Next, it has been verified, whether a simple modification evaluated by the TP as *good* corresponds to a decrease in execution time also on the TTA side. Different types of moves included: (1) a single move, when only one actor is moved to another unit, (2) a swap, when two actors from different units are swapped, and (3) multiple moves, when 3 or more actors are randomly moved to different units. Various moves were performed for 4 and 5 units, as for these numbers the highest estimation discrepancy was observed. Figure 10.12 presents a comparison of execution times expressed in clock-cycles for different numbers of units. Then, Tables 10.28-10.29 focus on the experiments with different types of moves.

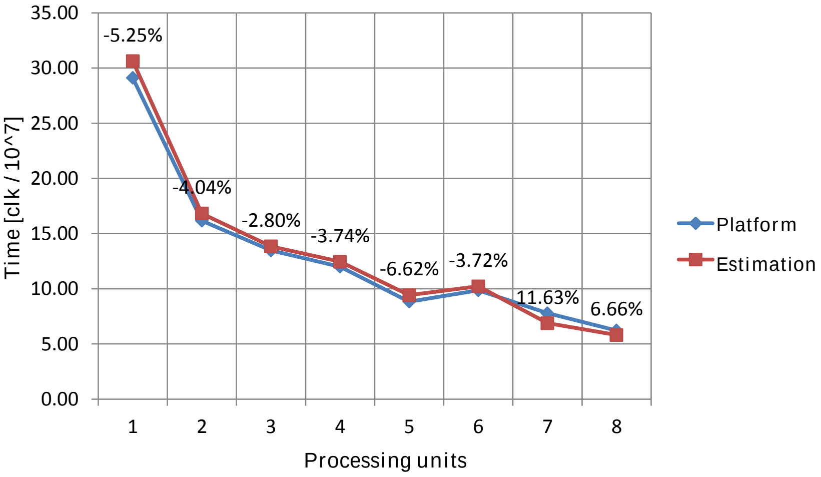


Figure 10.12 – TTA clock-cycles comparison.

Configuration	TTA [%]	TP [%]
single move 1	1	-1
single move 2	8	7
single move 3	-1	-1
swap 1	-4	-3
swap 2	26	31
swap 3	-1	-3
multiple move 1	8	9
multiple move 2	-6	-6
multiple move 3	-9	-12

Table 10.28 – Execution time change vs the initial configuration (4 units).

Configuration	TTA [%]	TP [%]
single move 1	0	0
single move 2	6	1
single move 3	2	1
swap 1	6	2
swap 2	7	1
swap 3	1	-1
multiple move 1	25	9
multiple move 2	74	74
multiple move 3	25	15

Table 10.29 – Execution time change vs the initial configuration (5 units).

Taking into account all analyzed cases, it is observed that the TP corresponds very well to the TTA simulator. The average difference between the number of cycles obtained for both is only 4.12% and tends to grow slightly for a larger number of units. Nevertheless, both single and complex moves are evaluated properly by the TP. This phenomenon legitimizes to the use of performance estimation in the entire process of analyzing dataflow applications, including the DSE heuristics and bottleneck analysis. Regarding the profiling methodology for TTA, it must be taken into account that although the used time-stamp operation is minimally intrusive, it results in the generated code with profiling being constantly ca. 1.5% slower than the generated code without profiling.

10.6.2 Estimation accuracy on Intel 86x64 platforms

The other set of experiments included benchmarking the applications (JPEG, MPEG4-SP Serial/Parallel decoder) on Intel platforms. Due to the challenges described in Section 5.3, the initial step was to validate the TP with a very simple design such as the JPEG decoder (Fig. 10.13). Secondly, both the MPEG4-SP designs have been tested on two Intel 86x64 machines (M1 and M2, as defined in Section 10.1.4) for partitioning configurations on 1, 2 and 3 cores. For each multi-core execution, 3 partitioning configurations of different quality were analyzed. The estimated and real execution times along with the estimation error are presented in Figure 10.14.

In terms of the average absolute error value, the precision decreases along with increasing the number of cores (4.94% in mono-core, 13.56% for 3 cores). This proportion can be explained by the neglected communication cost which grows in importance when more cores are used. Hence, another experiment was to incorporate the communication cost using the modeling approach and profiling methodology described in Section 5.3.3. The partitioning configurations with the largest discrepancy (MPEG4-SP Serial, configurations 5-7) have been profiled on both

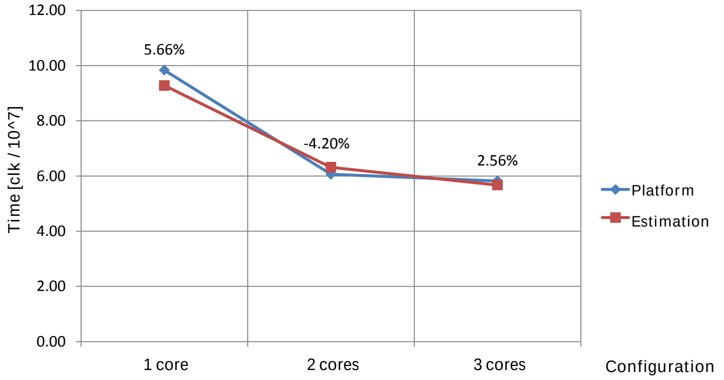
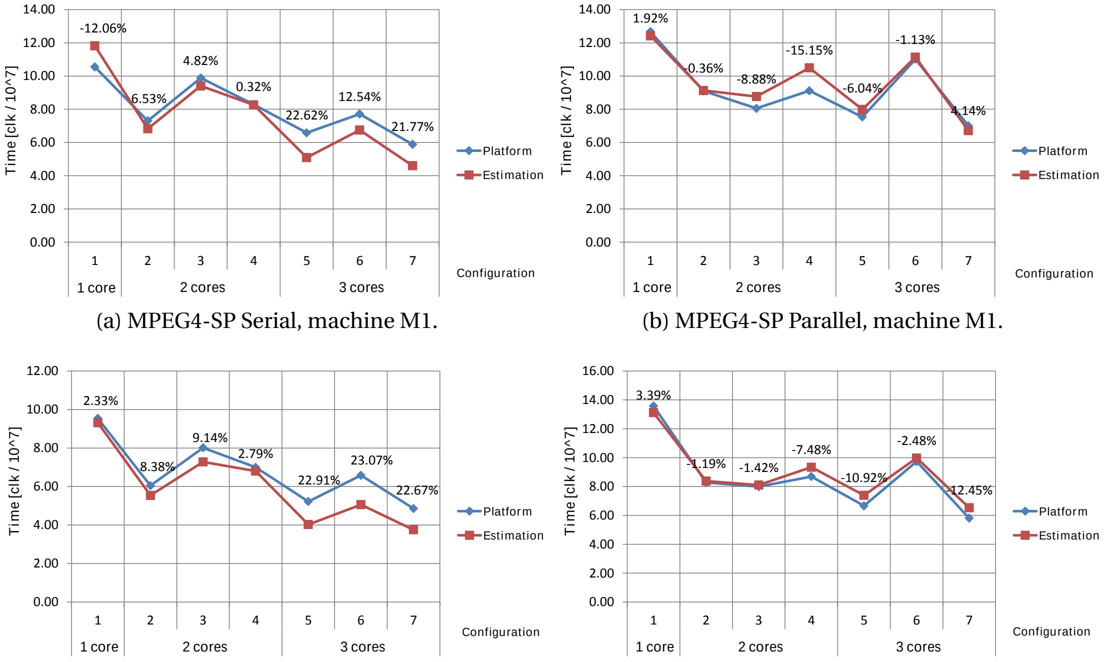


Figure 10.13 – JPEG, machine M2 (discrepancy as % value).



(a) MPEG4-SP Serial, machine M1.

(b) MPEG4-SP Parallel, machine M1.

(c) MPEG4-SP Serial, machine M2.

(d) MPEG4-SP Parallel, machine M2.

Figure 10.14 – MPEG-4 SP decoder estimation results (discrepancy as % value).

machines in order to provide the TP with the communication weights. Table 10.30 presents the improvement in estimation accuracy after applying the communication measures.

Configuration	[%] M1		[%] M2	
	without	with	without	with
5	22.62	11.38	22.91	7.74
6	12.54	-1.94	23.07	10.48
7	21.77	8.62	22.67	3.34

Table 10.30 – MPEG-4 SP Serial: estimation with the communication cost, discrepancy.

The most important outcome of the experiments is the observation that the estimation accuracy for the Intel platforms (under all the circumstances described in Section 5.3) is only slightly worse than for the case of TTA. An obvious complication to the model is the communication cost, which does not remain negligible for the estimation accuracy. However, as demonstrated, the discrepancy resulting from this cost can be remarkably reduced. In fact, considering it in the estimation demonstrates a large improvement of the accuracy, since the average absolute error for 3 cores is reduced to 6.72% (vs the previous 13.56%). From the perspective of design space exploration, the level of estimation accuracy is acceptable if it allows correct evaluation and comparison of different design points and to perform the moves in the space. This task has been successfully accomplished for both types of platform.

10.6.3 Estimation accuracy across different platforms

Fig. 10.15 summarizes the estimation accuracy obtained for different platforms. Compared to the results discussed earlier, apart from the TTA, multiple Intel platforms (in the chart grouped together as software executions *(SW)*), an execution on the Xilinx ZC702 platform (indicated as *HW*) is also included. The set of tested applications is extended by the HEVC decoder. For each test case (platform, application), the estimation discrepancy for the mono-core configuration (or, respectively, *HW*) expressed in % is indicated. The partitioning configurations for *SW* consist of different numbers of machines (cores) and have been established in order to demonstrate the scalability of each application. The execution times are expressed in clock-cycles. For HEVC, the input sequence was a full HD *BQTerrace* test sequence, whereas for the other applications a QCIF *Foreman* test sequence was used.

An important improvement compared to the results reported earlier is an implementation of the outliers filtering (as described in Section 5.3). Thanks to the filtering the results became more stable and different runs of profiling provided a very close set of weights. Summarizing the discrepancies, they range from only 0.18% for HW, through 3.5% for TTA, up to 6.2% for SW. The biggest single discrepancies occur, comprehensibly, for the HEVC. Since it is a complex design with a very high notion of dynamism in the implementation, the profiling approach

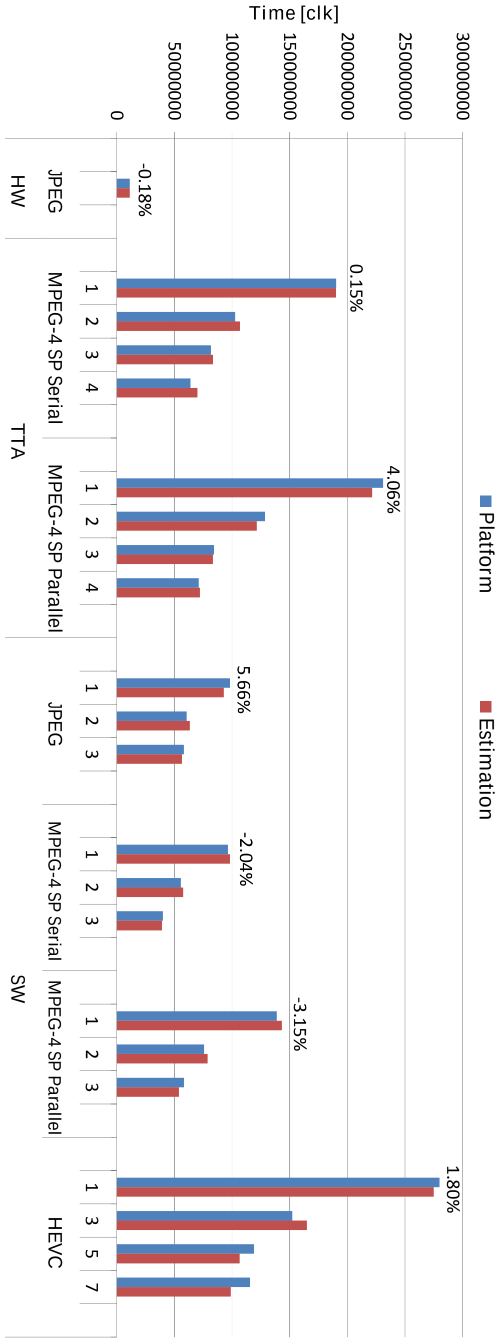


Figure 10.15 – Estimation accuracy: different platforms.

based on generating an averaged weight for each action might be burdened with a higher error than for the other cases, since different executions of the same action may be related to different parts of the code and hence result in different execution times. Alternatively, it is possible to consider independent weights for each firing, however, due to the size of the generated *ETG* it would remarkably increase the estimation time and memory requirements. Since the approach based on averaging provides enough accuracy to appropriately compare different design points, it remains preferable.

10.6.4 Discussion

Apart from the sources of discrepancy mentioned earlier, another possible reason for the sporadic occurrence of *peaks* in the overall discrepancy might be related to the modeling of the scheduling. First, for the case of TTA, the cost of intra-actor scheduling is modeled in a simplified way (as described in Section 5.2). A more accurate model should rather take into account the actor FSM structure, complexity of each guard, number of input/output ports per action and priorities. Second, the profiling methodology for none of the platforms contains the profiling of intra-partition scheduling overhead, which is related to the choices of the scheduler [237]. This cost is difficult to track, because it may depend on multiple factors, such as: the number of actors in one partition, the properties of a scheduling policy, the number of conditions to be checked before an actor is executed, or even the order of appearance of actors on the list representing each partition. As demonstrated and discussed previously with the experiments related to scheduling (Section 10.5), using the *NnP* scheduling policy keeps this cost at a, generally, negligible level.

Apart from estimation accuracy, an important aspect of a PE methodology is its efficiency and requirements. Due to the small memory requirements (even for complex applications) all reported results have been obtained on a standard PC. The estimation forthe MPEG4-SP designs operating on an input stimulus of 30 frames succeeded in a reasonable time (a couple of minutes). For JPEG, the estimation time in measured in seconds and for the HEVC operating on an input stimulus of 10 frames in HD resolution the estimation time is around 15-20 minutes. The TP seems to be promising in terms of future estimation of complex platforms ($i.e.$, heterogeneous). Increasing the number of elements in the model ($i.e.$, processing units) does not remarkably increase the estimation time. For instance, the difference in the time required for estimation of a mono-core platform and a platform with 30 processing units intended for a software execution is ca. 20%.

10.7 Variable Space Search

The experimental results discussed in this Section aimed at the validation of the concept of VSS introduced and discussed in Chapter 7. They have been performed using the recent dataflow implementation of the HEVC decoder, which is constantly under development, also in terms of performance improvement. In this context, the analysis of the decoder with regards to its bottlenecks and identification of the most promising directions of improvement is an important step providing some necessary information to the developers. The experiments were performed on machine M3, which contains enough cores (2 x 10) not limiting the exploration. Among the proposed optimization scenarios, the one aimed at throughput optimization (S1) has been chosen.

10.7.1 Design spaces

The experiments based on the proposed VSS algorithm involved 13 different design spaces summarized in Table 10.31. The initial space was created according to the basic dataflow design of the HEVC decoder, as depicted in Figure 10.3b. The transition from one space to another (according to the order given in the Table) was based on the results of bottlenecks and impact analysis performed with the performance estimation tool. According to these results, in each iteration an actor (or a set of actors) was chosen for modifications. The modifications were possible in two directions. First, the parts of the target actors ($i.e.$, the most critical actions and/or the ones with the highest impact analysis) were considered for *algorithmic* optimizations implying rewriting some parts of the code to make it more efficient and/or concise. These modifications can potentially improve the performance by reducing the execution time of a program, but they do not increase the potential parallelism (according to the definition in Section 4.2.4). Theoretically, as discussed earlier in Section 7.1.3, these modifications might lead to some new feasible design points and, consecutively, to a new design space. In practice, they have been verified to bring a negligible improvement of performance or, in some cases, they were applied only in order to improve the readability of the code, hence they are not listed as separate spaces.

After applying the modifications of the algorithmic parts of the code, a given actor (or a set of actors) was considered for a *parallelization*. This kind of modification implies replacing an actor with a set of actors so that a given processing part can be portioned among them. As a result, it is expected that the potential parallelism of a program will increase, which means that the results of better quality will become achievable. In most cases, such a modification relied on introducing a *component-based* implementation, where the processing of *Luma (Y)* and *Chroma (U, V)* components is performed, possibly in parallel, by separate actor instances. Another option is a *pipelined* implementation, where different instances of the same actor exchange some control indexes responsible for distributing different portions of the data among the instances. A pipelined implementation can consist of two or more (multi-stage implementation) instances of the same actor. A similar concept of portioning the data is present in the transition from a basic implementation with one *Parser* to a multi-parser implementation. Finally, since the considered implementation of HEVC uses the concept of *shared memory*, a modification can also rely on implementing certain communication procedures between the actors as reading/writing of shared memory variables. These modifications lead to a simplification of communication and a reduction of the number of tokens exchanged between the actors.

D_i	description	actors
0	basic design	13
1	D_0 with component-based implementation of *Inter Prediction*	15
2	D_0 with component-based pipelined implementation of *Inter Prediction*	16
3	D_0 with component-based multi-stage pipelined implementation of *Inter Prediction*	19
4	D_3 with multi-parser implementation (2 *Parsers*)	21
5	D_4 with separate processing of U and V component in *Inter Prediction Chroma*	22
6	D_4 with pipelined implementation of *Deblocking Filter* (horizontal/vertical processing)	22
7	D_4 with component-based implementation of *Deblocking Filter*	23
8	D_4 with component-based implementation of *Sample Adaptive Offset Filter (SAO)*	23
9	D_4 with component-based implementation of *Intra Prediction*	23
10	D_4 with component-based implementation of *Reconstruct Coding Unit (Reconstruct CU)*	25
11	D_4 with shared memory implementation for exchanging the information about neighboring pixels	21
12	combined changes applied in D_9, D_{10}, D_{11}	25

Table 10.31 – HEVC: summary of the considered design spaces.

After performing a transition from one space to another, several performance tests have been conducted on the target platform. A set of input sequences included: *BQTerrace*, *Kimono* (both in HD resolution), *Traffic* (crop 4K) and *Jockey* (full 4K). For each sequence, 4 different values of the quantization parameter (QP) have been tested: 22, 27, 32, 37. Small values of QP correspond to a higher image quality at a cost of lower performance. Also, the smaller is the QP, the more workload during the processing goes through the *Parser* and less through the other parts of the decoder. For the performance tests on the platform, the full length sequences were used (300 frames), whereas for the bottleneck and impact analysis short, 10 frames sequences were used instead. Table 10.32 summarizes the results obtained for each design space. It contains the following information:

- The potential parallelism (Pot. parall.) obtained for *BQTerrace, QP 37*;
- The parallelism (Parall.) of the best design points $X^*(D_i)$ obtained for *BQTerrace, QP 37*;
- The potential parallelism (Pot. parall) obtained for *BQTerrace, QP 37*;
- The referenced design space (Ref. D_i) the performance comparison is done with;
- The maximum improvement of the performance (Max. impr.) obtained for an individual sequence, compared to the referenced design space;
- The fraction of sequences (Impr.) for which the throughput has been improved, compared to the referenced design space;
- The fraction of sequences (Sim.) for which the throughput was similar (+/- 1%), compared to the referenced design space;
- The fraction of sequences (Decr.) for which the throughput has decreased, compared to the referenced design space.

D_i	Pot. parall.	Parall. ($X^*(D_i)$)	Ref. D_i	Max. impr. [%]	Impr. [%]	Sim. [%]	Decr. [%]
D_0	2.45	1.64	-	-	-	-	-
D_1	3.13	1.66	D_0	36.37	68.8	12.5	18.75
D_2	3.69	1.76	D_0	34.71	68.8	12.5	18.75
D_3	3.85	2.71	D_0	43.35	81.3	6.3	12.5
D_4	4.05	2.89	D_3	121.88	87.5	6.3	6.3
D_5	4.19	2.91	D_4	3.04	6.3	12.5	81.3
D_6	4.28	3.35	D_4	4.3	25.0	56.3	18.8
D_7	4.27	3.19	D_4	13.12	18.8	31.3	50.0
D_8	4.21	2.76	D_4	1.91	6.3	6.3	87.5
D_9	4.42	2.73	D_4	-	0.0	0.0	100.0
D_{10}	4.55	2.92	D_4	-	0.0	0.0	100.0
D_{11}	4.21	2.85	D_4	3.41	25.0	62.5	12.5
D_{12}	5.34	3.32	D_4	17.53	43.75	0.00	56.25

Table 10.32 – HEVC: summary of the improvement achieved in various design spaces.

10.7.2 Critical path and impact analysis

As mentioned earlier, the basis for making a transition from one space to another was the bottleneck and impact analysis performed for the best design point in a given space, obtained by means of exploration, as well as for the full parallel configuration. The first analysis is referred to as the *scheduled* bottleneck (or impact) and the second as the *algorithmic* bottleneck (or impact) analysis. An important choice was to decide on which input sequence the analysis should be based on. The first demand was to use a *real* resolution, such as HD or 4K. Furthermore, as mentioned earlier, low values of QP tend to locate most of the workload in the process of parsing the data. Hence, no matter which type of analysis is performed (algorithmic/scheduled) and which P, S, B configurations are used, the bottleneck analysis points to the *Parser*, hence the actors composing the decoder are not adequately represented in the analysis. Using low values of QP has also a side effect, because large amounts of high-quality data must be processed and a single run of performance estimation is very long (few hours). Furthermore, since they require using generally bigger buffer sizes, the memory requirements for the bottleneck and impact analysis are very high making it difficult to run multiple analyses at the same time. For these reasons it is preferable to base the analysis on high values of QPs, and use the low QPs only occasionally. Finally, as verified experimentally, the high-quality design points established for the high QPs remain, in general, high-quality also for smaller QPs, whereas the other way around this property is not preserved.

The other observation is that for the same values of QP, different sequences behave differently. For instance, finding an appropriate partitioning configuration and/or moving from one space to another so that the performance is improved is an easy task for some sequences, while for some others the DSE takes a lot of time and only a slight performance improvement can be observed. It was decided to focus on the *worst case* and hence, the *BQ Terrace* sequence was chosen as the reference for the analysis.

Tables 10.33-10.45 contain the results of the algorithmic and scheduled bottleneck analysis. For each design space, the 10 most critical actions are listed, along with the percentage of their executions which are contained in the CP of the design and the estimated execution time (EET) (normalized) corresponding to the overall length of the CP for a given design point.

Figures 10.16-10.22 illustrate the chosen results (D_0-D_4 and D_{11}-D_{12}) of the impact analysis (both, algorithmic and scheduled) carried out for the considered design spaces. The results for the design spaces D_5-D_{10} have been skipped in this summary due to their similarity to the results obtained for the other design spaces. In each case, the 3 most critical actions (according to the bottleneck analysis) have been analyzed. The analysis used three points, corresponding to the reduction of the weight of an action by 33, 66 and 100%, respectively. This information is complementary to that obtained during the bottleneck analysis, because it demonstrates the actual improvement potential coming from a reduction of the complexity, and hence, the

Algorithmic: EET 1.60		
Actor	Action	CP%
InterPrediction	interpolateSamples	38.00
InterPrediction	applyWeights	11.35
IntraPrediction	computeIntraPu_is4x4	6.98
ReconstructCU	getTuIntra_is4x4	6.31
IntraPrediction	computeIntraPu_is8x8	6.28
ReconstructCU	getTuIntra_isNot4x4	5.04
ReconstructCU	sendNeighb_y_is4x4	3.17
IntraPrediction	computeIntraPu_is16x16	2.72
ReconstructCU	sendNeighb_chr_is4x4	2.31
IntraPrediction	computeIntraPu_is32x32	1.93

(a) Algorithmic bottlenecks.

Scheduled: EET 2.45		
Actor	Action	CP%
InterPrediction	interpolateSamples	23.40
InterPrediction	applyWeights	6.99
IntraPrediction	computeIntraPu_is4x4	4.57
ReconstructCU	getTuIntra_is4x4	4.13
IntraPrediction	computeIntraPu_is8x8	4.11
ReconstructCU	getTuIntra_isNot4x4	3.30
Parser	read_ResidualCoding…	3.24
Parser	read_ResidualCoding…	2.78
Parser	read_ResidualCoding…	2.73
Parser	read_CodingUnit…	2.65

(b) Scheduled bottlenecks.

Table 10.33 – Design space D_0.

Algorithmic: EET 1.31		
Actor	Action	CP%
InterPrediction_InterpLuma	interpolateSamples	31.64
IntraPrediction	computeIntraPu_is4x4	8.86
IntraPrediction	computeIntraPu_is8x8	7.98
ReconstructCU	getTuIntra_is4x4	7.23
InterPrediction_InterpLuma	applyWeights	6.61
ReconstructCU	getTuIntra_isNot4x4	5.90
ReconstructCU	sendNeighb_y_is4x4	3.88
IntraPrediction	computeIntraPu_is16x16	3.76
ReconstructCU	sendNeighb_chr_is4x4	2.88
IntraPrediction	computeIntraPu_is32x32	2.40

(a) Algorithmic bottlenecks.

Scheduled: EET 2.76		
Actor	Action	CP%
InterPrediction_InterpLuma	interpolateSamples	17.11
DBFilter_DeblockFilter	filterEdges	5.62
IntraPrediction	computeIntraPu_is4x4	3.96
IntraPrediction	computeIntraPu_is8x8	3.49
InterPrediction_InterpLuma	applyWeights	3.36
SAO	getSaoMerge_merge	3.33
Parser	read_ResidualCoding…	3.07
ReconstructCU	getTuIntra_is4x4	3.00
ReconstructCU	getTuIntra_isNot4x4	2.90
Parser	read_ResidualCoding…	2.53

(b) Scheduled bottlenecks.

Table 10.34 – Design space D_1.

Algorithmic: EET 1.13		
Actor	Action	CP%
IntraPrediction	computeIntraPu_is4x4	10.64
ReconstructCU	getTuIntra_is4x4	9.51
IntraPrediction	computeIntraPu_is8x8	9.46
InterPrediction_InterpChroma	interpolateSamples	9.44
ReconstructCU	getTuIntra_isNot4x4	7.03
SAO	getSaoMerge_merge	6.41
DBFilter_DeblockFilter	filterEdges	4.82
ReconstructCU	sendNeighb_y_is4x4	4.52
IntraPrediction	computeIntraPu_is16x16	4.13
InterPrediction_InterpChroma	applyWeights	3.83

(a) Algorithmic bottlenecks.

Scheduled: EET 2.82		
Actor	Action	CP%
InterPrediction_InterpLuma_1_0_0	interpolateSamples	10.02
DBFilter_DeblockFilter	filterEdges	6.58
SAO	getSaoMerge_merge	4.42
IntraPrediction	computeIntraPu_is4x4	4.13
IntraPrediction	computeIntraPu_is8x8	3.58
Parser	read_ResidualCoding…	3.36
ReconstructCU	getTuIntra_is4x4	3.16
Parser	read_ResidualCoding…	3.12
ReconstructCU	getTuIntra_isNot4x4	3.08
Parser	read_CodingUnit…	2.64

(b) Scheduled bottlenecks.

Table 10.35 – Design space D_2.

Algorithmic: EET 1.12		
Actor	Action	CP%
IntraPrediction	computeIntraPu_is4x4	10.41
IntraPrediction	computeIntraPu_is8x8	9.66
ReconstructCU	getTuIntra_is4x4	9.23
InterPrediction_InterpChroma	interpolateSamples	9.09
ReconstructCU	getTuIntra_isNot4x4	7.09
SAO	getSaoMerge_merge	6.32
DBFilter_DeblockFilter	filterEdges	4.79
ReconstructCU	sendNeighb_y_is4x4	4.71
IntraPrediction	computeIntraPu_is16x16	3.95
InterPrediction_InterpChroma	applyWeights	3.76

(a) Algorithmic bottlenecks.

Scheduled: EET 1.89		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	11.85
DBFilter_DeblockFilter	filterEdges	8.37
IntraPrediction	computeIntraPu_is4x4	6.14
SAO	getSaoMerge_merge	6.01
ReconstructCU	getTuIntra_is4x4	5.60
IntraPrediction	computeIntraPu_is8x8	5.34
ReconstructCU	getTuIntra_isNot4x4	4.73
InterPrediction_InterpChroma	applyWeights	4.61
DecodingPictureBuffer	getMvInfo_launch	3.44
DBFilter_DeblockFilter	getCuPix_launch…	3.04

(b) Scheduled bottlenecks.

Table 10.36 – Design space D_3.

Algorithmic: EET 1.16		
Actor	Action	CP%
InterPrediction_Interp_1_0_0	interpolateSamples	12.65
IntraPrediction	computeIntraPu_is4x4	11.31
ReconstructCU	getTuIntra_is4x4	9.60
IntraPrediction	computeIntraPu_is8x8	9.28
ReconstructCU	getTuIntra_isNot4x4	6.99
SAO	getSaoMerge_merge	6.26
DBFilter_DeblockFilter	filterEdges	4.69
ReconstructCU	sendNeighb_y_is4x4	4.66
IntraPrediction	computeIntraPu_is16x16	4.23
ReconstructCU	sendNeighb_chr_is4x4	3.2

(a) Algorithmic bottlenecks.

Scheduled: EET 1.84		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	12.02
DBFilter_DeblockFilter	filterEdges	8.53
IntraPrediction	computeIntraPu_is4x4	6.53
SAO	getSaoMerge_merge	6.52
IntraPrediction	computeIntraPu_is8x8	5.56
ReconstructCU	getTuIntra_is4x4	5.37
ReconstructCU	getTuIntra_isNot4x4	4.81
InterPrediction_InterpChroma	applyWeights	4.81
DecodingPictureBuffer	getMvInfo_launch	3.03
ReconstructCU	sendNeighb_y_is4x4	3.02

(b) Scheduled bottlenecks.

Table 10.37 – Design space D_4.

Algorithmic: EET 1.13		
Actor	Action	CP%
InterPrediction_Interp_1_0_0	interpolateSamples	13.04
IntraPrediction	computeIntraPu_is4x4	10.60
IntraPrediction	computeIntraPu_is8x8	9.74
ReconstructCU	getTuIntra_is4x4	8.63
ReconstructCU	getTuIntra_isNot4x4	7.05
SAO	getSaoMerge_merge	6.41
ReconstructCU	sendNeighb_y_is4x4	4.79
DBFilter_DeblockFilter	filterEdges	4.70
IntraPrediction	computeIntraPu_is16x16	4.27
ReconstructCU	sendNeighb_chr_is4x4	3.29

(a) Algorithmic bottlenecks.

Scheduled: EET 1.84		
Actor	Action	CP%
InterPrediction_Interp_1_0_0	interpolateSamples	8.22
DBFilter_DeblockFilter	filterEdges	6.41
IntraPrediction	computeIntraPu_is4x4	5.82
IntraPrediction	computeIntraPu_is8x8	5.22
SAO	getSaoMerge_merge	4.60
ReconstructCU	getTuIntra_is4x4	4.37
ReconstructCU	getTuIntra_isNot4x4	4.08
ReconstructCU	sendNeighb_y_is4x4	3.11
InterPrediction_InterpChromaU	interpolateSamples	3.00
ReconstructCU	sendNeighb_chr_is4x4	2.47

(b) Scheduled bottlenecks.

Table 10.38 – Design space D_5.

Algorithmic: EET 1.11		
Actor	Action	CP%
InterPrediction_Interp_1_0_0	interpolateSamples	19.60
IntraPrediction	computeIntraPu_is4x4	11.00
IntraPrediction	computeIntraPu_is8x8	9.55
ReconstructCU	getTuIntra_is4x4	8.93
ReconstructCU	getTuIntra_isNot4x4	7.26
SAO	getSaoMerge_merge	6.45
ReconstructCU	sendNeighb_y_is4x4	4.91
IntraPrediction	computeIntraPu_is16x16	3.87
ReconstructCU	sendNeighb_chr_is4x4	3.26
IntraPrediction	computeIntraPu_is32x32	2.62

(a) Algorithmic bottlenecks.

Scheduled: EET 1.63		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	13.70
IntraPrediction	computeIntraPu_is4x4	7.06
IntraPrediction	computeIntraPu_is8x8	6.36
ReconstructCU	getTuIntra_is4x4	5.65
InterPrediction_InterpChroma	applyWeights	5.46
ReconstructCU	getTuIntra_isNot4x4	5.4
SAO	getSaoMerge_merge	5.13
ReconstructCU	sendNeighb_y_is4x4	3.44
DBFilter_DeblockFilterVert	filterEdges	3.06
ReconstructCU	sendNeighb_chr_is4x4	2.76

(b) Scheduled bottlenecks.

Table 10.39 – Design space D_6.

Algorithmic: EET 1.16		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	11.50
IntraPrediction	computeIntraPu_is4x4	11.39
IntraPrediction	computeIntraPu_is8x8	9.85
ReconstructCU	getTuIntra_is4x4	8.67
ReconstructCU	getTuIntra_isNot4x4	7.02
SAO	getSaoMerge_merge	6.25
ReconstructCU	sendNeighb_y_is4x4	4.90
InterPrediction_InterpChroma	applyWeights	4.63
IntraPrediction	computeIntraPu_is16x16	4.29
ReconstructCU	sendNeighb_chr_is4x4	3.45

(a) Algorithmic bottlenecks.

Scheduled: EET 1.78		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	12.60
IntraPrediction	computeIntraPu_is4x4	6.36
IntraPrediction	computeIntraPu_is8x8	5.83
DBFilter_DeblockFilterChroma	filterEdges	5.71
ReconstructCU	getTuIntra_isNot4x4	5.08
ReconstructCU	getTuIntra_is4x4	5.07
InterPrediction_InterpChroma	applyWeights	5.03
SAO	getSaoMerge_merge	4.98
DecodingPictureBuffer	getMvInfo_launch	3.15
ReconstructCU	sendNeighb_y_is4x4	3.13

(b) Scheduled bottlenecks.

Table 10.40 – Design space D_7.

Algorithmic: EET 1.15		
Actor	Action	CP%
IntraPrediction	computeIntraPu_is4x4	11.19
IntraPrediction	computeIntraPu_is8x8	9.45
ReconstructCU	getTuIntra_is4x4	9.25
InterPrediction_InterpChroma	interpolateSamples	9.18
ReconstructCU	getTuIntra_isNot4x4	6.75
SAO	getSaoMerge_merge	6.17
ReconstructCU	sendNeighb_y_is4x4	4.75
DBFilter_DeblockFilter	filterEdges	4.58
IntraPrediction	computeIntraPu_is16x16	4.16
InterPrediction_InterpChroma	applyWeights	3.65

(a) Algorithmic bottlenecks.

Scheduled: EET 1.97		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	11.49
IntraPrediction	computeIntraPu_is4x4	5.73
IntraPrediction	computeIntraPu_is8x8	4.83
DBFilter_DeblockFilter	filterEdges	4.52
InterPrediction_InterpChroma	applyWeights	4.43
SAO_Luma	getSaoMerge_merge	4.26
ReconstructCU	getTuIntra_is4x4	4.24
ReconstructCU	getTuIntra_isNot4x4	3.68
InterPrediction_Interp_1_0_0	interpolateSamples	3.39
ReconstructCU	sendNeighb_y_is4x4	2.80

(b) Scheduled bottlenecks.

Table 10.41 – Design space D_8.

Algorithmic: EET 1.12		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	10.97
IntraPrediction_IntraPredLuma	computeIntraPu_is4x4	10.57
ReconstructCU	getTuIntra_is4x4	10.18
ReconstructCU	getTuIntra_isNot4x4	7.49
IntraPrediction_IntraPredChroma	computeIntraPu_is8x8	7.04
DBFilter_DeblockFilter	filterEdges	6.95
ReconstructCU	sendNeighb_y_is4x4	5.62
InterPrediction_InterpChroma	applyWeights	4.54
ReconstructCU	sendNeighb_chr_is4x4	3.31
ReconstructCU	sendNeighb_y_is8x8	2.82

(a) Algorithmic bottlenecks.

Scheduled: EET 2.09		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	10.64
DBFilter_DeblockFilter	filterEdges	7.66
DBFilter_GenerateBs	getSplitTrafo...	5.3
ReconstructCU	getTuIntra_is4x4	5.14
ReconstructCU	getTuIntra_isNot4x4	4.79
SAO	getSaoMerge_merge	4.48
InterPrediction_InterpChroma	applyWeights	4.41
IntraPrediction_IntraPredLuma	computeIntraPu_is4x4	3.51
ReconstructCU	sendNeighb_y_is4x4	2.75
IntraPrediction_MergeProcessDone	untagged_0	2.65

(b) Scheduled bottlenecks.

Table 10.42 – Design space D_9.

Algorithmic: EET 1.11		
Actor	Action	CP%
IntraPrediction	computeIntraPu_is4x4	12.20
IntraPrediction	computeIntraPu_is8x8	10.67
InterPrediction_InterpChroma	interpolateSamples	9.33
ReconstructCU_Luma	getTuIntra_is4x4	8.84
ReconstructCU_Luma	getTuIntra_isNot4x4	7.10
SAO	getSaoMerge_merge	6.55
ReconstructCU_Luma	sendNeighb_y_is4x4	6.12
DBFilter_DeblockFilter	filterEdges	4.73
IntraPrediction	computeIntraPu_is16x16	4.45
InterPrediction_InterpChroma	applyWeights	3.76

(a) Algorithmic bottlenecks.

Scheduled: EET 2.13		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	6.44
DBFilter_GenerateBs	getSplitTrafo...	5.46
DBFilter_DeblockFilter	filterEdges	5.41
IntraPrediction	computeIntraPu_is4x4	5.23
IntraPrediction	computeIntraPu_is8x8	4.86
SAO	getSaoMerge_merge	4.67
InterPrediction_Interp_1_0_0	interpolateSamples	4.55
ReconstructCU_Luma	getTuIntra_is4x4	4.42
ReconstructCU_Luma	getTuIntra_isNot4x4	4.35
ReconstructCU_Luma	sendNeighb_chr_is4x4	3.21

(b) Scheduled bottlenecks.

Table 10.43 – Design space D_{10}.

Algorithmic: EET 1.11		
Actor	Action	CP%
IntraPrediction	computeIntraPu	27.39
DBFilter_DeblockFilter	filterEdges	11.94
ReconstructCU	getTuIntra_is4x4	9.93
InterPrediction_InterpChroma	interpolateSamples	9.44
ReconstructCU	sendNeighb	8.45
ReconstructCU	getTuIntra_isNot4x4	7.49
InterPrediction_InterpChroma	applyWeights	4.02
DBFilter_DeblockFilter	getCuPix...	3.13
Source	sendData_launch	2.06
ReconstructCU	getSplitTrafo...	1.45

(a) Algorithmic bottlenecks.

Scheduled: EET 1.87		
Actor	Action	CP%
IntraPrediction	computeIntraPu	14.47
DBFilter_DeblockFilter	filterEdges	11.98
InterPrediction_InterpChroma	interpolateSamples	8.94
ReconstructCU	sendNeighb	4.89
ReconstructCU	getTuIntra_is4x4	4.22
InterPrediction_InterpChroma	applyWeights	3.89
ReconstructCU	getTuIntra_isNot4x4	3.68
InterPrediction_Interp_1_0_0	interpolateSamples	3.68
Source	sendData_launch	3.09
DBFilter_DeblockFilter	getCuPix...	2.41

(b) Scheduled bottlenecks.

Table 10.44 – Design space D_{11}.

Algorithmic: EET 0.98		
Actor	Action	CP%
IntraPrediction_IntraPredLuma	computeIntraPu	23.93
InterPrediction_InterpChroma	interpolateSamples	11.12
ReconstructCU_Luma	getTuIntra_is4x4	11.08
ReconstructCU_Luma	getTuIntra_isNot4x4	8.48
SAO	getSaoMerge_merge	7.42
ReconstructCU_Luma	sendNeighb	6.22
DBFilter_DeblockFilter	filterEdges	5.33
InterPrediction_InterpChroma	applyWeights	4.89
Source	sendData_launch	2.51
DBFilter_DeblockFilter	getCuPix...	1.93

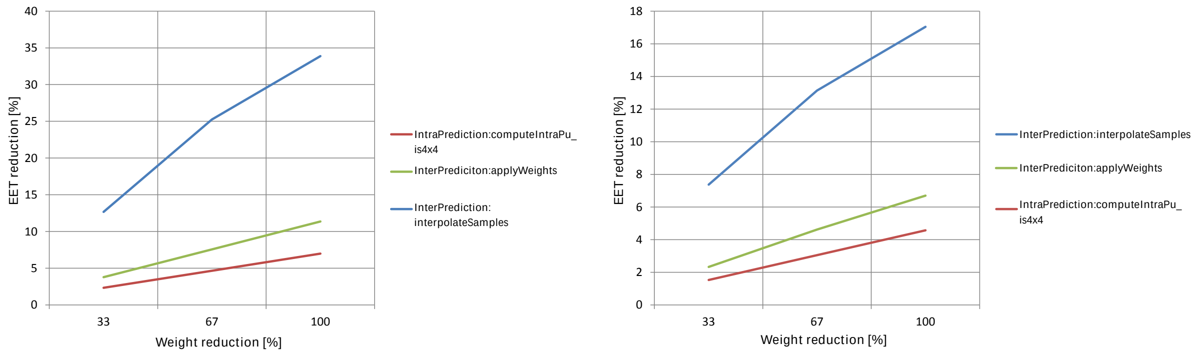
(a) Algorithmic bottlenecks.

Scheduled: EET 1.82		
Actor	Action	CP%
InterPrediction_InterpChroma	interpolateSamples	11.43
IntraPrediction_IntraPredLuma	computeIntraPu	10.49
ReconstructCU_Luma	getTuIntra_isNot4x4	5.83
ReconstructCU_Luma	getTuIntra_is4x4	5.38
InterPrediction_InterpChroma	applyWeights	4.86
SAO	getSaoMerge_merge	4.85
DBFilter_DeblockFilter	filterEdges	3.90
ReconstructCU_Luma	sendNeighb	3.37
TwoParsers_P0	read_ResidualCoding...	2.41
TwoParsers_P0	read_nal_launch	2.18

(b) Scheduled bottlenecks.

Table 10.45 – Design space D_{12}.

weight of an action. In some cases, the action which appears to be the most critical according to the bottlenecks, is not the one that offers the biggest improvement potential.



(a) Algorithmic impact analysis.

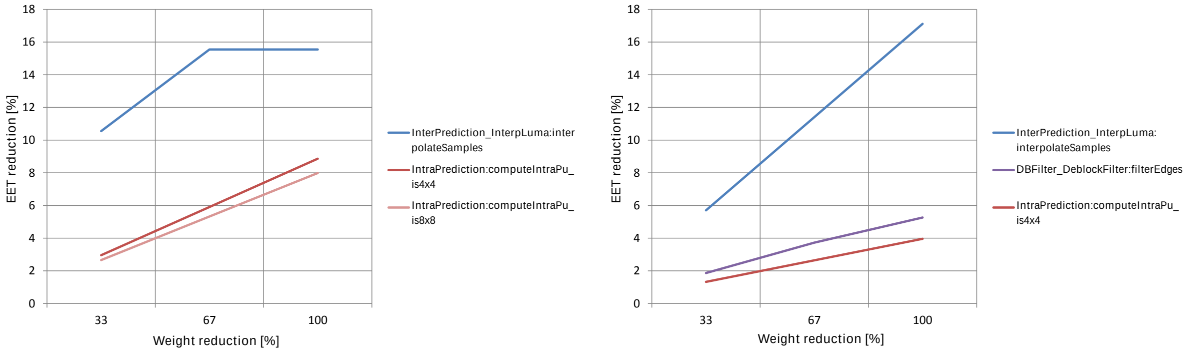
(b) Scheduled impact analysis.

Figure 10.16 – Design space D_0.

10.7.3 Solutions in multiple spaces

The results presented in this Section aim at summarizing the overall improvement achieved during the VSS. The first set of charts (Figures 10.23a-10.24b) demonstrates the throughput for different design points established during the exploration in each space. Each chart targets a different quality point, dependending on the QP.

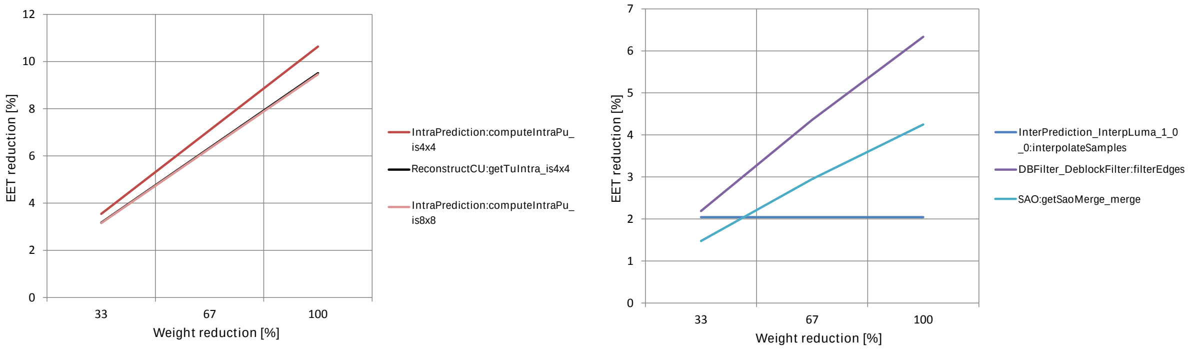
The second set of Figures (Fig. 10.25a-10.25b) locates the best design point from each space in a 3-dimensional space consisting of the throughput (expressed in FPS), the total buffer size B_{total} (expressed in tokens) and the number of machines m^X. These dimensions correspond to the criteria used by the optimization scenarios described in Section 7.6. The Figures have been generated for a representative fraction among the considered input sequences. The



(a) Algorithmic impact analysis.

(b) Scheduled impact analysis.

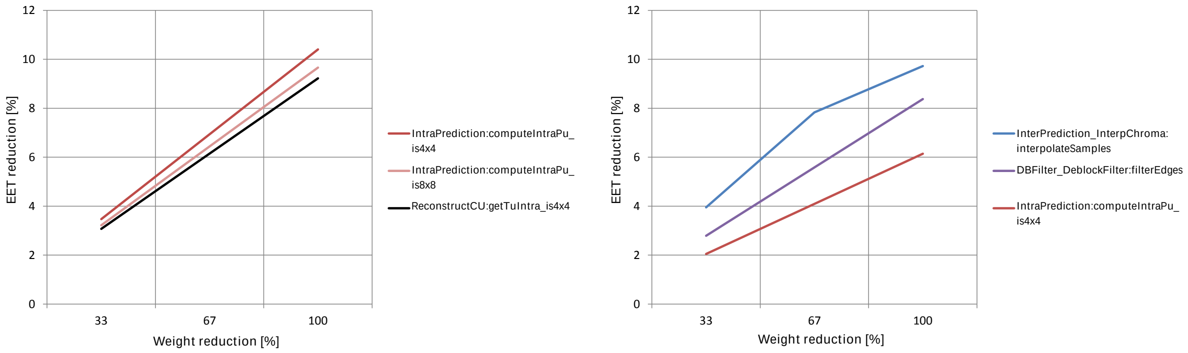
Figure 10.17 – Design space D_1.



(a) Algorithmic impact analysis.

(b) Scheduled impact analysis.

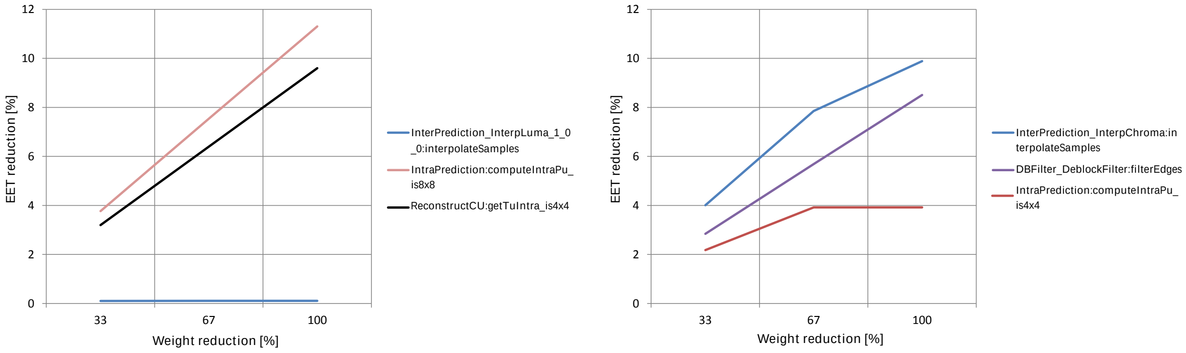
Figure 10.18 – Design space D_2.



(a) Algorithmic impact analysis.

(b) Scheduled impact analysis.

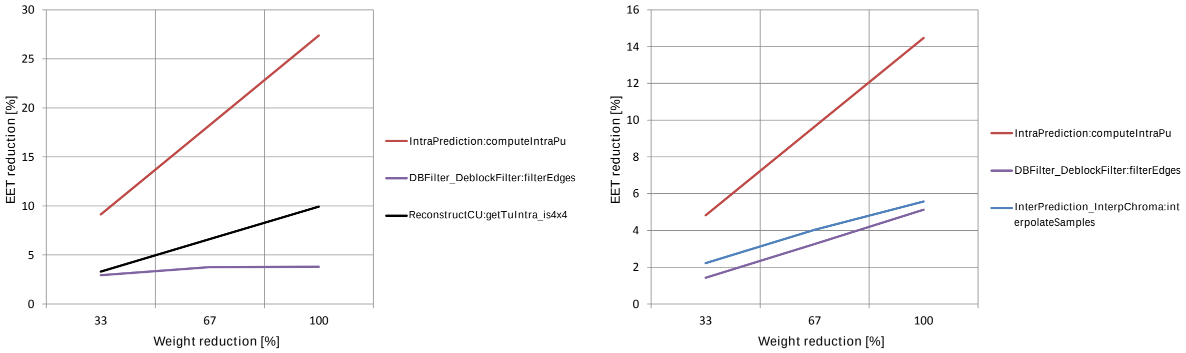
Figure 10.19 – Design space D_3.



(a) Algorithmic impact analysis.

(b) Scheduled impact analysis.

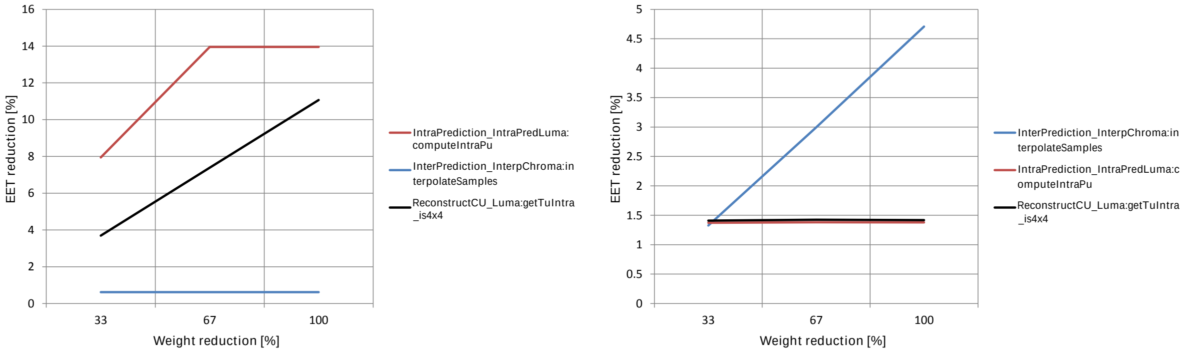
Figure 10.20 – Design space D_4.



(a) Algorithmic impact analysis.

(b) Scheduled impact analysis.

Figure 10.21 – Design space D_{11}.



(a) Algorithmic impact analysis.

(b) Scheduled impact analysis.

Figure 10.22 – Design space D_{12}.

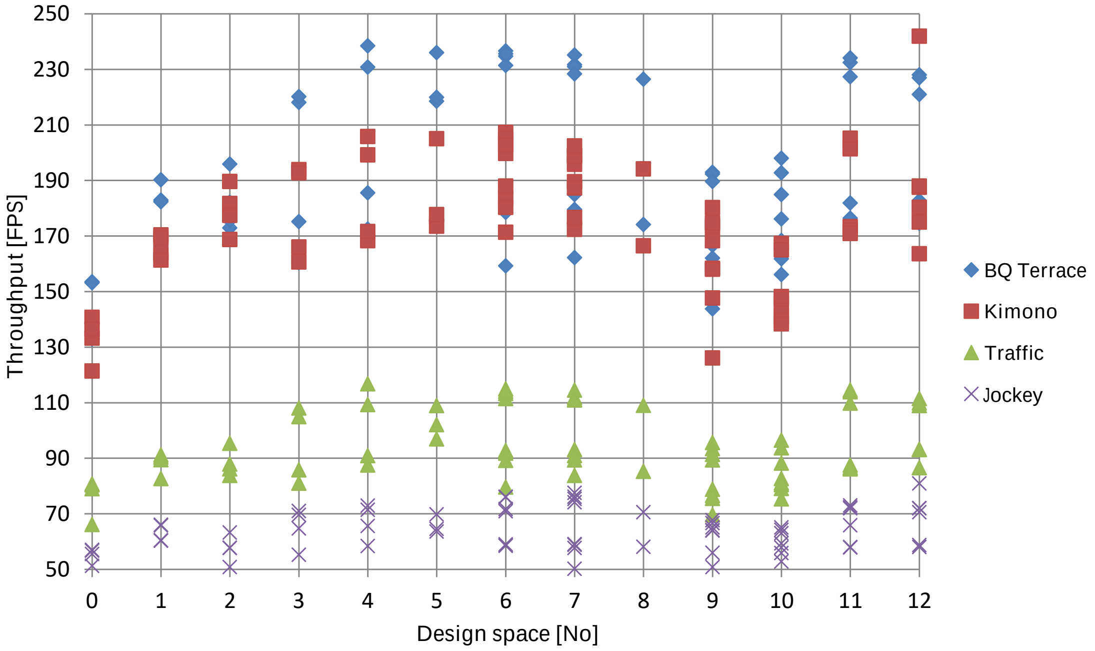
experimental cases (*i.e.*, the sequence-QP combinations) have been chosen according to the location of the design points allowing a visual evaluation in a 3-dimensional space.

10.7.4 Discussion

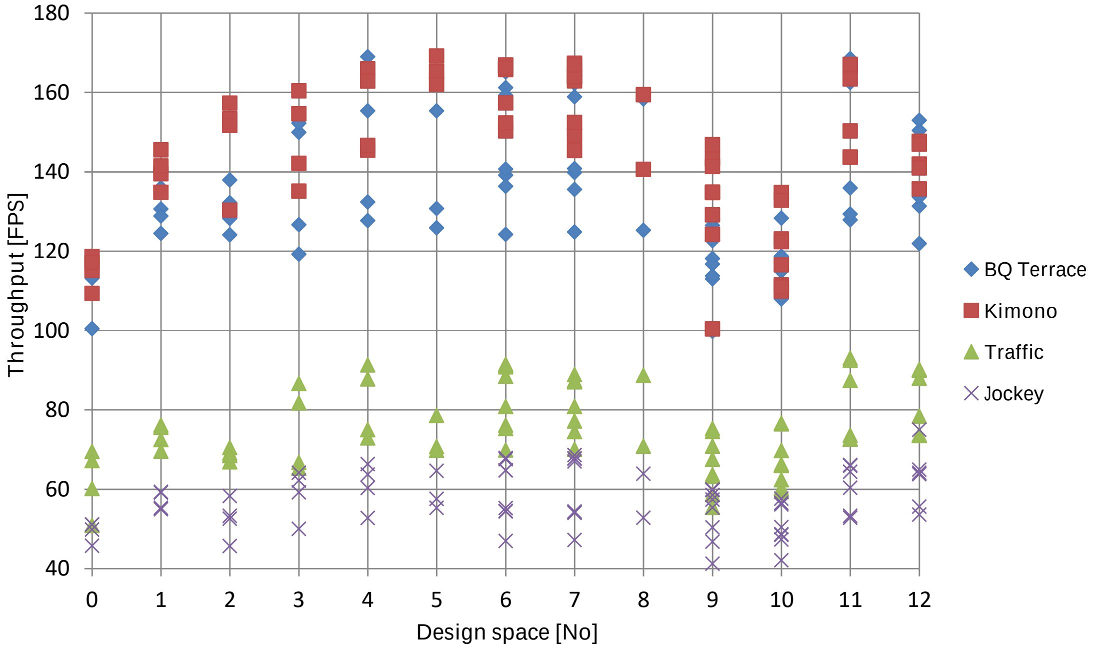
The analysis of the initial design space D_0 indicates clearly the most critical part of the decoder. The *interpolate Samples* action inside the *Inter Prediction* clearly standouts from the other actions in terms of the percentage of the CP it takes. Moreover, the second most critical action *apply Weights* also belongs to the same actor. Hence, a parallelization of this actor is responsible for a transition to the next design space D_1. Then, it can be observed that the EET for the algorithmic bottlenecks is reduced (potential parallelism has increased, as indicated earlier in Table 10.32), however, the same actions responsible for the processing of the *Luma* part still appear quite high on the list of most critical parts, for both, algorithmic and bottleneck analysis. Hence, further parallelization is applied, leading to design spaces D_2 and D_3. It can be observed that during this process the criticality of the parts related to the *Luma* processing within *Inter Prediction* decreases and, eventually, it is the *Chroma* part that starts to appear among the most critical parts instead. The potential parallelism, the parallelism of the $X^*(D_i)$ and the values of EET for both types of analysis generally keep improving until this point.

As mentioned earlier, for the case of low QPs, the bottleneck analysis points only to the *Parser*. Considering this property, as well as the fact that in some design spaces even for a high QP the *Parser* appears among the most critical parts, a transition from D_3 to D_4 has been obtained by applying a multi-parser configuration, *i.e.*, with two instances of the *Parser* capable of processing the data in parallel. This transition brings some improvement for high QPs and a remarkable improvement (up to 121.88%) for low QPs. Considering the algorithmic bottleneck analysis performed in these 2 cases for *BQ Terrace* and QP 22, the potential parallelism grows from 2.41 to 4.71, which fully corresponds to the results obtained on the platform.

Analyzing the design space D_4, the algorithmic and scheduled bottleneck analysis point to slightly different parts of the decoder. It is preferable to rely more on the scheduled analysis, since it corresponds to the configuration leading to the best throughput really achieved on the platform. Hence the next candidates for parallelization are: *Inter Prediction Chroma*, *Deblocking Filter* and *SAO*. The parallelization of these actors (design spaces D_5-D_8) is characterized by three properties: (1) it does not completely eliminate the actor/action from the list of most critical parts, because a component (*i.e.*, *Luma*) always remains one of the most critical parts, (2) it does not provide any remarkable reduction of the EET, (3) in terms of a real execution platform, it brings improvement only for some input sequences, being rather a minority among the tested fraction.

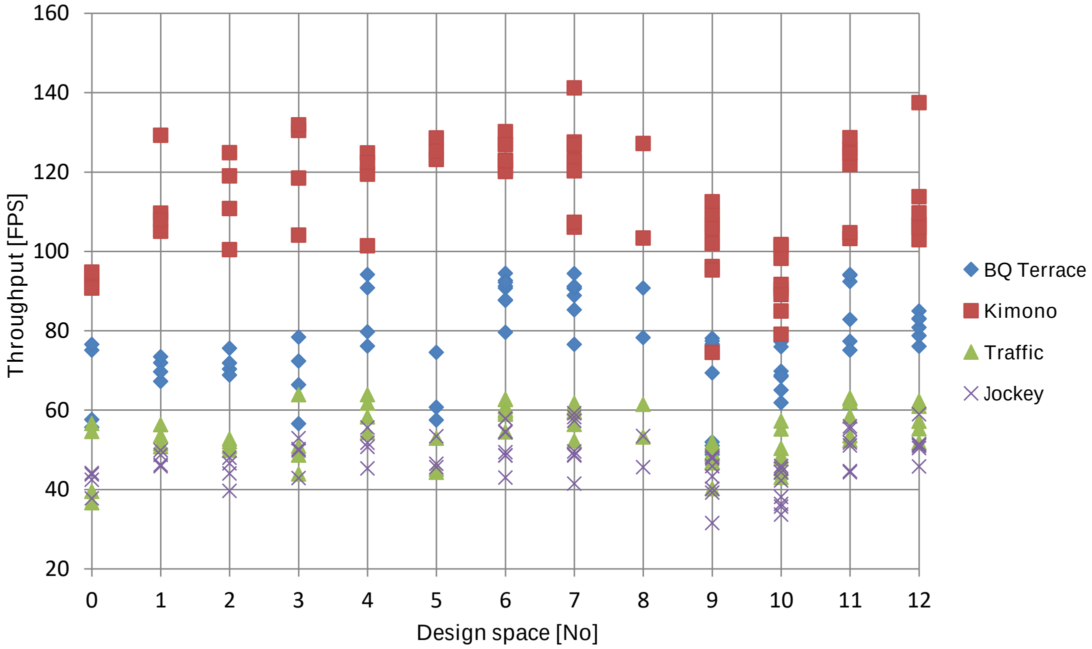


(a) QP37.

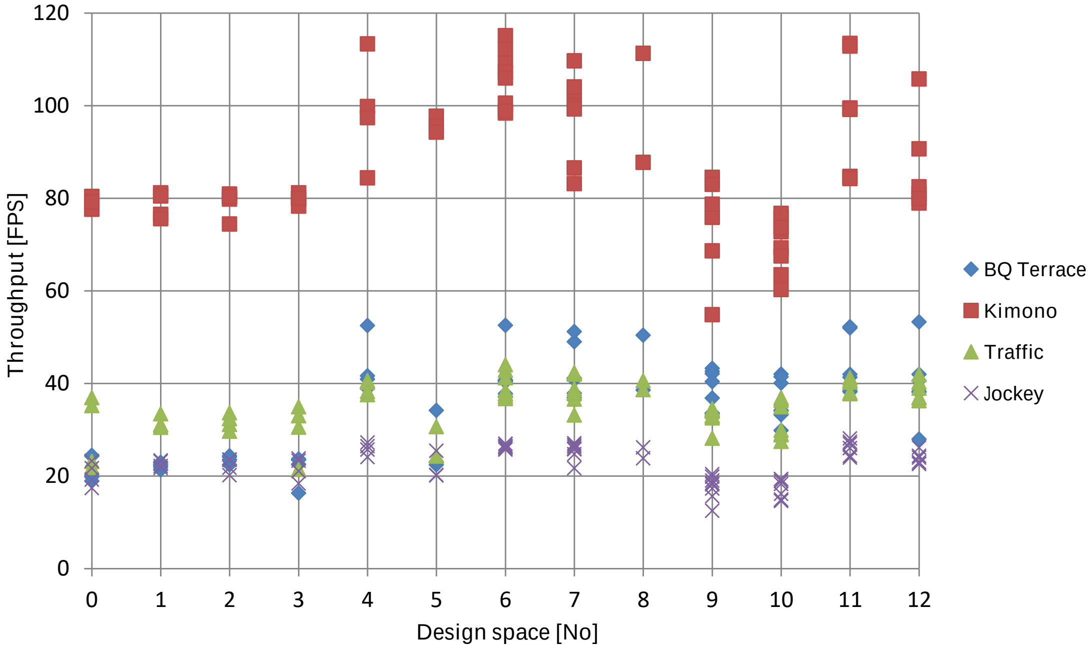


(b) QP32.

Figure 10.23 – Throughput improvement summary (1).

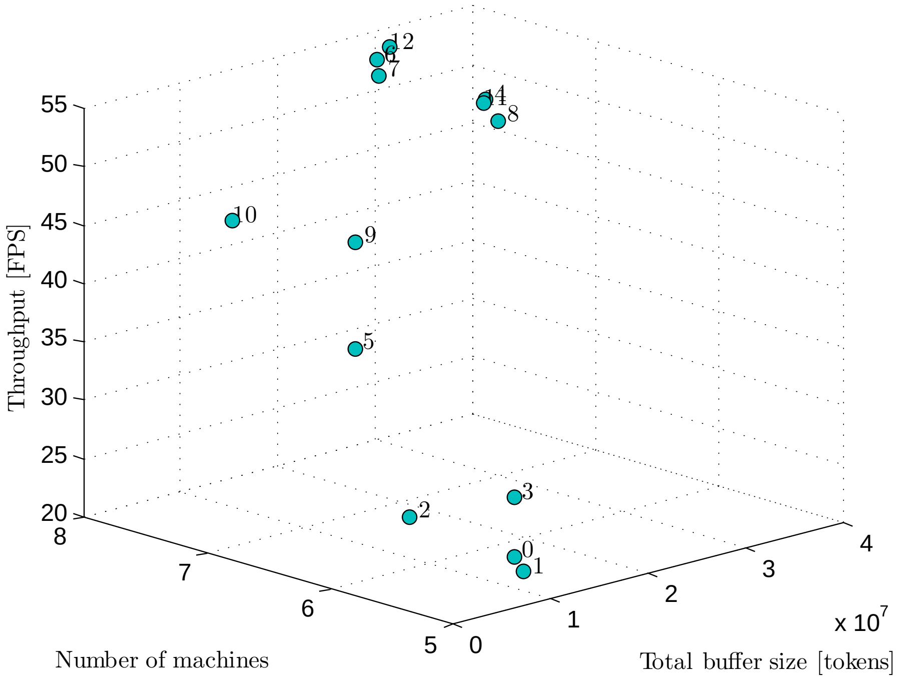


(a) QP27.

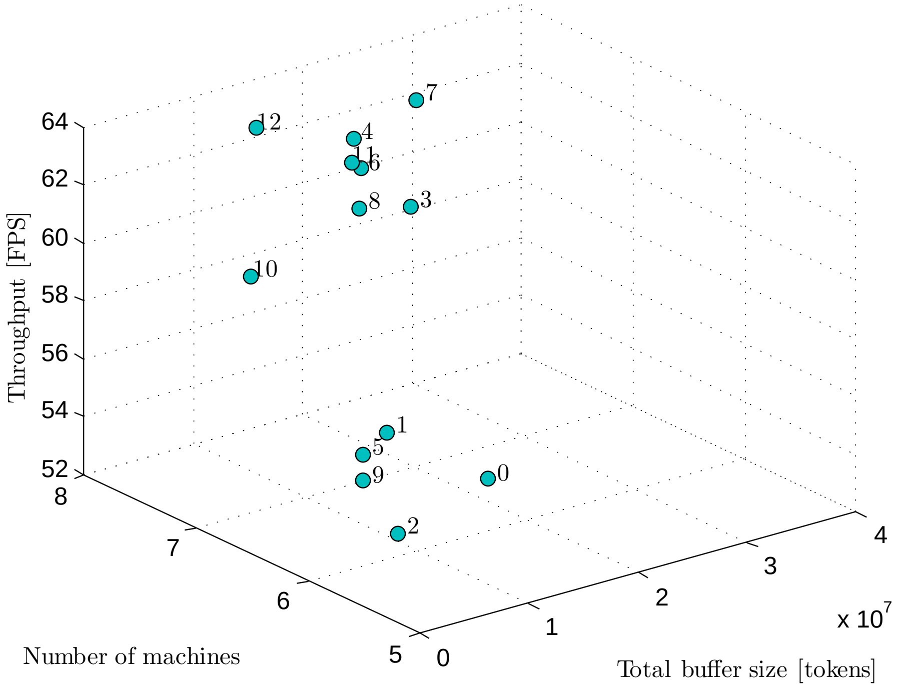


(b) QP22.

Figure 10.24 – Throughput improvement summary (2).

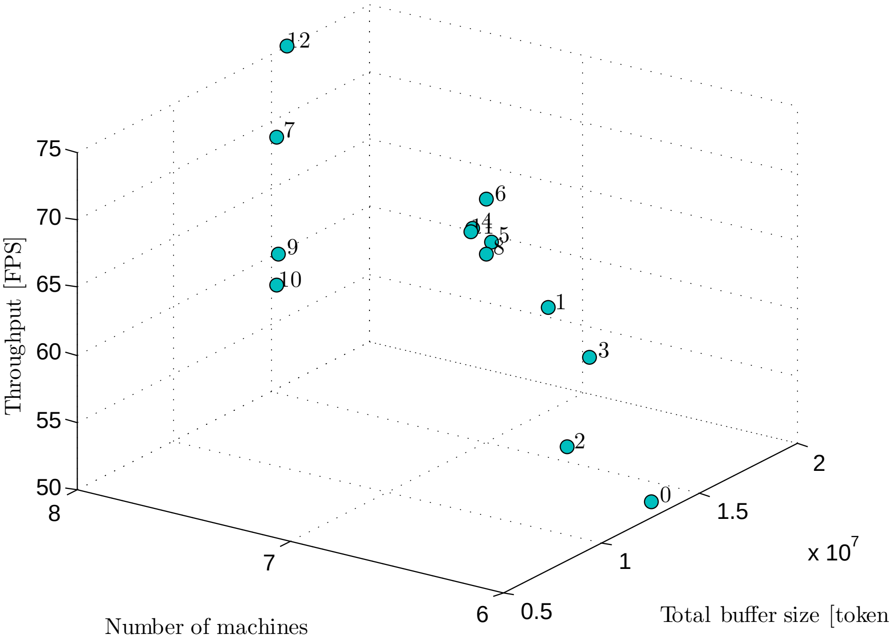


(a) BQTerrace QP22.

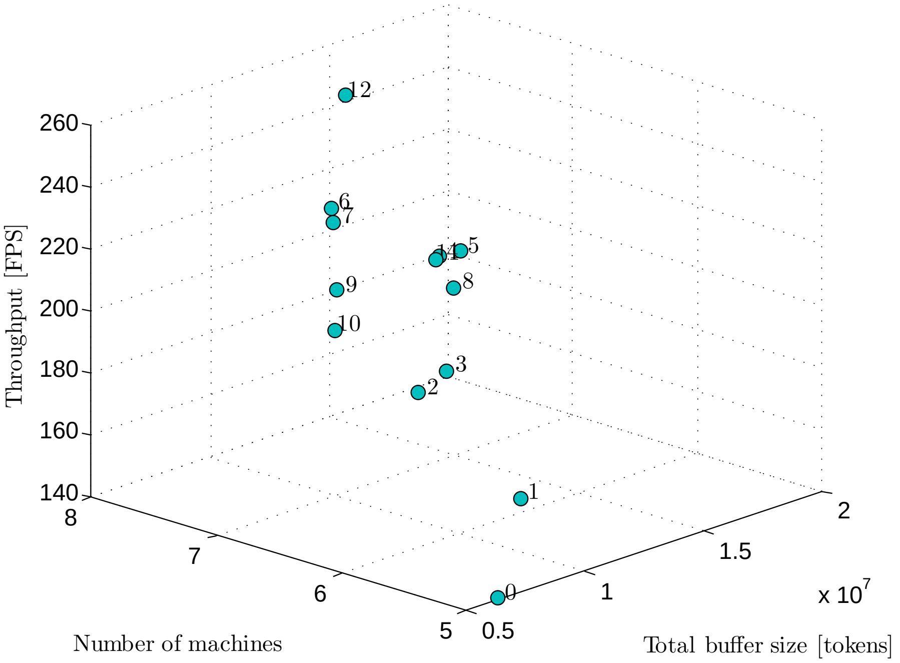


(b) Traffic QP27.

Figure 10.25 – Throughput and resources in different design spaces: summary (1).



(a) Jockey QP32.



(b) Kimono QP37.

Figure 10.26 – Throughput and resources in different design spaces: summary (2).

Summarizing the results for design spaces D_5-D_8, the next candidates are the *Intra Prediction* and *Reconstruct CU*, leading to design spaces D_9 and D_{10}. Unlike for the previous case, in this design space a remarkable *decrease* of the throughput occurs for *all* considered sequences. This is quite surprising, since the actions belonging to these actors appear quite high on the list of most critical actions starting already from the initial design space. A deep analysis of the structure of these two actors supported by profiling (*i.e.*, using the *numap* library) leads to an explanation of this behavior. These two actors and, in particular, the involved actions exchange many tokens with each other. Hence, as long as these two actors are partitioned together in one processing unit, the communication cost is kept much lower than for the case when they are separated. Since parallelization always implies partitioning the parallelized components on different processing units, it leads to a remarkable increase of the communication cost which apparently outstrips any potential gain coming from the parallelization itself.

This analysis leads to yet another design space D_{11}, where the parts identified to be responsible for the high communication cost are implemented using the concept of shared-memory. In this case, the time required for making a copy of the data in order to transfer it as tokens is eliminated, because it is immediately available as shared-memory variables. This design space seems to be comparable to its original predecessor (D_4), with some slight improvement for many sequences. Finally, the last design space D_{12} comprising the modifications attempted in the design spaces D_9, D_{10} and D_{11} remarkably improves the value of potential parallelism to the value of 5.34 (compared to the initial 2.45 and 4.05 for D_4). Although the parts responsible for the processing of the *Luma* component remain among the most critical actions, they lead to the workload in CP which is much better distributed among different parts of the decoder.

Studying the impact data for the initial design spaces (*i.e.*, D_0, D_1, ...), it can be stated that the results of the impact analysis fully correspond to those observed in the bottleneck analysis. The parallelization of *Inter Prediction* offers the greatest improvement potential and this is verified by the execution in the platform. However, later on, the improvement coming from further parallelization of *Inter Prediction* is very small. In fact, as it can be observed first in Fig. 10.18b and then in Fig. 10.20a, a *saturation* of the improvement occurs, which diverts the next transitions to the new spaces to other parts of the decoder. Along the design spaces, the modifications of *compute Intra Pu* in the *Intra Prediction* remain the most promising direction for optimizations. Nevertheless, for the design space with the best potential parallelism (D_{12}), the scheduled impact analysis also points to a quick *saturation* of the improvement for this particular action.

As discussed earlier in Section 7.4, the main indication about the quality of the design space is the quality of its best design point. This criterion is considered here during the evaluation of the different spaces. In general, the transitions between D_0 and D_4 lead to visible improvements of the throughput for all or almost all sequences. Between D_5 and D_8 some small improvements appear occasionally, however in most cases the performance is slightly worsened. In design spaces D_9 and D_{10} the throughput is remarkably decreased in all the cases. Design space D_{11} provides similar, or slightly better results. Finally, for the case of design space D_{12} the evaluation fully depends on the considered sequence and the value of QP, and may result in a decrease of performance, a similar performance, as well as a remarkable improvement of performance in some cases. Comparing different charts it is observed that whereas QPs of 37, 32 and 27 represent a generally similar shape, the chart for QP 22 is completely different. This is consistent with the earlier observation that high-quality design points obtained for this QP do not result in high-quality points for the other values, unlike the other way around.

Analyzing the location of different $X^*(D_i)'s$ regarding the possible optimization criteria, notice that, in general, obtaining higher throughputs implies also increasing the resources in terms of the number of machines and the B_{total}. In fact, approximating the "path" arising between the points coming from different design spaces a quasi-monotonic curve is obtained in each case. Exceptions (if any) can be interpreted as a low-quality design space, where the increased resources do not correspond to an improvement of the throughput. This relationship between the improvement of the throughput and the increase of the resources can be intuitively explained. Nevertheless, it must be emphasized that finding the aforementioned "path" between different $X^*(D_i)'s$ is a difficult task, infeasible without the support of tools, such as DSE heuristics, performance estimation and bottleneck analysis. Moreover, attempting to perform the same without such support, can easily lead to design spaces, where the $X^*(D_i)$ makes very high resource demands that do not necessarily translate into a performance improvement.

10.8 Conclusions

The results described in this Chapter provided a solid verification of the methodologies introduced in this Thesis. The applications chosen for the experiments ranged from a simple design, where the exploration is manageable manually (JPEG decoder), up to a complex design resembling state-of-the art of video decoding (HEVC decoder). The latter is characterized by a high level of dynamism and strong dependence on the used input sequence, hence, it shows the properties typical for DDF, which are the target $MoCs$ for this work. Different platforms used in the experiments encompass architectures that can be translated into a simple highly-accurate model or a more complex and less accurate model.

The first part of the experiments considered a thorough verification of the proposed partitioning heuristics. For different algorithms it has been observed how they approach the potential parallelism of the considered design or outperform state-of-the-art dataflow partitioning methodologies. The more complex heuristics (*i.e.*, based on local search or tabu search) lead to better solutions than the simpler greedy heuristics, but quite naturally, require more operating time. The best quality solutions were obtained with the most advanced heuristic combining different variants of tabu search.

In the second part, the buffer dimensioning heuristics were verified to successfully establish a trade-off between the program throughput and the total buffer size corresponding to the used resources. The heuristics provided an entire curve of solutions that can be chosen according to the used optimization scenario. An important part of the experiments related to the buffer dimensioning was to compare the results obtained with the proposed heuristics with an approximation of an infinite buffer size. The solutions eventually converge to the ones obtained for an infinite buffer, but with the buffer sizes smaller by orders of magnitude. Finally, the experiments in this part have put in evidence the importance of a multidimensional exploration, since they demonstrated that narrowing the other dimensions prevents the discovery of high-quality solutions.

Next, experiments with different scheduling policies illustrated that different orders of execution of the firings lead to very different execution times. The estimated differences translate also to an improvement on a platform, but identify another property of the scheduling problem, which is the run-time cost of establishing the schedule.

All of the heuristics relied on the rich performance metrics carried by an execution trace. These metrics were tracked and extracted by means of performance estimation. The performance estimation *SW* tool was experimentally verified to provide a very high accuracy of the estimation that allowed correct evaluation of the moves in the space for the considered platforms, independently from their complexity. Furthermore, the performance estimation illustrated that a single model of execution, when provided with appropriate timing information can be successfully used on different types of platforms.

The experiments related to the *VSS* methodology have comprised all partial results discussed in the preceding Sections, because different stages of the methodology consisted of *DSE*, performance estimation and bottleneck analysis implemented on top of it. Using a complex design case, the experiments illustrated how the methodology leads a dataflow designer through different stages of the design flow and provides refactoring directions enabling discovery of the most promising design space, corresponding to a specific implementation variant of the design. It must be emphasized, that the programming effort required in the methodology is reduced to a minimum, since refactoring of an application in terms of code modifications is required only after exploring the available configurations. Furthermore, it is clearly indicated which parts of the code should be considered for modification.

11 Conclusions

This Thesis provides a systematic methodology for design space exploration of dynamic dataflow programs. It introduces a novel formulation of the problem, not presented in the literature so far, which considers a fully dynamic execution of a program and allows an exploration of different design alternatives. Furthermore, the formulation can be easily referred to different types of architecture with regard to the architecture-specific constraints and properties. A detailed execution model required to explore the design space according to the formulation is provided as an ETG supplied with accurate timing information. The formulation and the execution model are a base for the set of tools corresponding to different stages of the system development design flow. Unlike many state-of-the-art methodologies, the flow avoids designing application-specific architectures at a detailed level and hence offers much wider exploration opportunities of different design points. This approach favors defining and finding trade-offs between the performance, resource utilization and programmability, increasing the efficiency of a program and fully exploiting its portability.

The tools form a complete methodology supporting the designer in the process of application development. Hence, an important aspect are the DSE heuristics that allow finding high-quality configurations of the program without requiring any piece of code to be modified. If the current design does not allow finding the points satisfying the design constraints ($i.e.$, in terms of performance), a set of analyses clearly identifies the parts of the code that should be subject to some programming effort in order to improve the value(s) of the assumed objective function(s). Both tasks, that is, DSE and bottleneck identification are accomplished by a highly-accurate performance estimation that makes it possible to evaluate different design variants and configurations without having to execute them on a physical platform. The methodology leads to results which, due to the complexity and the level of dynamism inside the applications, are not possible to obtain manually without appropriate support.

11.1 Achievements of the Thesis

The contributions of this Thesis start from the aforementioned formulation of the design space exploration problem. This formulation acts as a base for all steps. They include: modeling of a dynamic execution as an execution trace graph, timing it according to the considered platform, design space exploration in terms of partitioning, buffer dimensioning and scheduling, performance estimation and a formalization of the Variable Space Search methodology comprising all previous steps in order to efficiently and systematically analyze and improve dynamic dataflow programs.

Rigorous design space exploration problem formulation

A novel formulation of the design space exploration problem in terms of partitioning, scheduling and buffer dimensioning has been provided. The formulation operates at the level of action firings (considered as jobs), which is appropriate to describe fully dynamic applications. It handles the dependence of different subproblems on each other and does not impose any specific order in which the solutions to these problems should be provided. Hence, it does not limit the exploration procedure. The formulation considers the dynamic execution of a program without narrowing to static or quasi-static dataflow $MoCs$, as often happens in the literature.

Furthermore, the formulation has been also referred to two different types of platforms: homogeneous and heterogeneous. Each type implies a more precise specification of some of the constraints ($i.e.$, in terms of the cost related to the communication) and/or introduces additional constraints impacting the problem to be solved. Apart from providing a formulation in terms of decision variables, objective functions and constraints, a set of examples has demonstrated the size and the complexity of the problem which make it manually unmanageable.

Definition of an accurate dynamic dataflow program execution model for DSE

An execution trace graph (ETG) has been used as a basic abstract model of a dynamic execution. It is detailed enough to provide the necessary information for the DSE problem as it has been formulated. Translating an abstract execution model into a real execution on a physical platform requires providing it with appropriate timing information obtained by means of profiling. It has been demonstrated and experimentally verified that using the profiling tools available for different platforms it is possible to extract such timing information which is accurate enough to provide rich performance metrics for the purpose of DSE. Such an approach keeps the application and architecture models separated because a single ETG can be used for analysis purposes on different platforms, for which models of different accuracy can be constructed.

Efficient DSE heuristics

In order to make the exploration process effective, efficient and automated, several heuristics have been provided. Each of them targets one of the subproblems of the *DSE*, that is partitioning, buffer dimensioning or scheduling.

- For the case of **partitioning**, the heuristics can be ordered according to the complexity as: greedy constructive heuristics, descent local search, tabu search and advanced tabu search extended with a probabilistic approach and/or an iterative re-profiling procedure responding to the demands of *NUMA* platforms. Experimental results have verified that the more complex is the heuristic, the better results it provides, but in a longer time. The performances of the applications under analysis have been improved in terms of the approach to their potential parallelism in a fully automated way and without making any modifications to the algorithmic part;

- The **buffer dimensioning** heuristics are, to the best of the author's knowledge, the first heuristics targeting the problem of buffer dimensioning for the purpose of throughput improvement with regards to the applied partitioning and scheduling configurations. The two general approaches, bottom-up and top-down have been introduced. They can be used in different optimization scenarios enabling finding a trade-off between the performance of the program and the utilization of memory resources. The experiments with this particular dimension of the design space have also verified the legitimacy of the novel multidimensional formulation of the problem, since high-quality solutions can be found during the exploration only by properly expanding the dimensions;

- The difference between the **scheduling** and the other subproblems of the *DSE* in terms of the necessity to perform decisions dynamically lead to the development of different dynamic scheduling policies. It has been defined and experimentally verified that the efficiency of a policy: (1) depends on the other configurations, for instance, an unfavorable partitioning configuration limits the opportunities of applying an efficient scheduling, (2) is subject to two factors, including the performance potential coming from different orders of execution and the cost of establishing this order. A figure of merit has been introduced to express the cost of a policy and an approach for establishing this cost by means of performance estimation has been presented.

Highly-accurate performance estimation

Based on the *ETG* complemented with the timing information, a performance estimation *SW* tool performing a post-processing of the trace has been developed. The tool allows estimating the execution time of different design points without having to execute them on a physical platform. The tool is an essential component of the design flow, because it provides rich performance metrics extracted from the trace. Different properties tracked during the execution of a program are then supplied to the *DSE* heuristics. Experimental results have verified this tool to be highly accurate and to allow a precise evaluation of different design points, even if the structural difference between them is very small. Furthermore, the experiments with the tool confirmed the portability of the *ETG*, since the same abstract execution has been used for estimating the performances of a given dataflow program on different platforms.

Another important aspect of the performance estimation tool is the bottleneck and impact analysis implemented on top of it. The algorithms for these analyses were originally proposed in Chapter 8 of [121], but allowed only the analysis for a fully parallel execution without the notion of limited buffer sizes. Thanks to the novel problem formulation and the performance estimation tool, these algorithms can be also applied to a given design point giving a more realistic indication about the bottlenecks.

VSS methodology formalization

The stages of the system development design flow related to the profiling, analysis and providing refactoring directions have been summarized and formalized as a consistent and complete analysis and improvement methodology. This methodology relies on the concept of Variable Space Search introduced originally for the graph coloring problem, where the search for close-to-optimal points is performed in differently defined spaces. Following the proposed *DSE* problem formulation, the concepts of design points, design spaces and the transitions between them have been introduced and expressed with appropriate notation capturing the multidimensionality of the problem. Different optimization scenarios, in terms of objective functions and constraints, have been introduced in order to be used in the process of optimization of a dynamic dataflow program, without narrowing the possible trade-off options.

This methodology minimizes the programming effort, because is relies on an automated exploration and requires an intervention into the code only if the constraints or the values of the objective functions are not satisfied. Furthermore, the directions of the optimizations are clearly given. This approach differs significantly from the exploration methodologies defined in the literature, where dissatisfaction of the design constraints usually leads to considerations about the possible modifications of the target platform. Instead, the focus of the *VSS* methodology is to improve the efficiency of the design itself, where different target platforms can be applied.

This methodology, comprising all of the previously described stages, has been thoroughly verified using the HEVC decoder, which is the current state-of-the art in video decoding. The results obtained illustrate a consequent improvement of the performance along the transitions between different design spaces, followed by an increase of the resource requirements. The complexity of the HEVC decoder prevents such results from being obtained manually, without systematic support.

11.2 Work extensions

The results reported here constitute a good basis for further investigations aiming at maximizing the efficiency and usability of the system development design flow and its underlying *DSE* methodology. The set of tasks belonging to future work can be separated into two groups. The first one considers possible improvements and extensions to the provided tools, whereas the second one identifies some open problems to be investigated.

11.2.1 Improvements

Profiling of heterogeneous platforms

It has been verified that an accurate execution model on different platforms can be built using the *ETG* representation. In order to further extend the *DSE* and *VSS* methodologies to various types of platforms, the set of supported and verified target architectures should be constantly extended. For each new platform, it is essential to analyze the opportunities for obtaining accurate profiling information. In this context, it is especially interesting to focus on heterogeneous platforms and dedicate the profiling efforts to the boarder between the *SW* and *HW* components and the communication taking place between them. Such an investigation can extend and possibly make more precise the constraints identified and discussed in Section 6.3.2.

Profiling of intra-partition scheduling cost

The biggest source of discrepancy of the performance estimation has been identified as the missing cost of intra-partition scheduling. Incorporating this cost into the estimation model is expected to strongly reduce the discrepancy and make it dependent only on the accuracy of the profiling, not on the accuracy of the model itself. This cost might be subject to multiple factors, such as: the number and order of appearance of the actors inside each partition, the number of conditions required to be checked for each firing, the level of dynamism within the actors *etc*. Hence, an investigation on the impact of these factors on the accuracy of the estimation results and the profiling opportunities would be beneficial.

Modeling of caches

As stated in Chapter 5, the profiling results are subject to the availability of data in the caches. Furthermore, the profiling of the communication cost is related to the levels of caches serving a given read/write request. Currently, the communication cost assigned to a buffer is estimated according to the latencies profiled for different memory levels. In order to eliminate this approximation and hence improve the estimation accuracy, it can be considered to model the caches and include them in the components composing the *DEVS* system used in the performance estimation module. Such a modeling can eliminate, or at least reduce the impact of a partitioning configuration on the profiling results and enable modeling the execution times of specific firings more precisely.

Joint multidimensional exploration

The proposed *DSE* heuristics focus on finding a high-quality solution to only one of the subproblems. Hence, it must be chosen in which order the heuristics should be applied. The current approach is repetitive, for instance, after the initial tuning of buffers, a high-quality partitioning configuration is established and the buffer size is eventually finally tuned. An alternative to this approach could be to use a heuristic, such as tabu search, which considers *all* dimensions when defining the set of possible moves and neighborhoods. Such an implementation can increase the level of automation of the *DSE* stage of the design flow.

Acceleration of the performance estimation

The time required for a single run of the performance estimation depends on the size of the *ETG*, which rapidly grows as the number of action firings increases. Considering complex designs in conjunction with long input stimuli (*e.g.*, high resolutions for the case of video decoders), the resulting *ETG* can quickly grow up to billions of nodes and dependencies. In order to make a single estimation run faster, the opportunities for using efficient graph databases to process the *ETG* can be investigated. Accelerating a single estimation run can enable evaluating a higher number of moves in different dimensions within the same time frame and, in consequence, improve the quality of the final solution. Besides that, a faster exploration and estimation can make the usage of the design flow and, in particular, the *DSE* stage more robust.

Mixed evaluation of the moves

Once of the concepts in the field of performance estimation is to combine multiple models with different levels of detail and accuracy (Section 9.1). In general, the less accurate, but faster model is used more often, and the more accurate, but time consuming one is used only occasionally. In this particular case, a faster model can be replaced with an execution on the target platform. In this way, a move can be evaluated quickly, but without extracting the execution properties, as for the case of performance estimation. An interesting aspect of this research would be to establish the appropriate *usage ratio* for the platform execution and performance estimation, so that the exploration process is maximally efficient. On the other hand, studying the correlation between performance estimation and platform execution can lead to developing some learning features, similarly to the machine learning-based estimation methods. As in the previous point, a faster evaluation of the moves can increase the number of moves evaluated during the exploration and, in consequence, improve the quality of the final solution.

11.2.2 Open problems

Identification of static regions in a dynamic execution

The experiments with different scheduling policies demonstrated two aspects of the scheduling problem: the performance gain coming from different execution orders and the run-time cost required to establish this order. An interesting problem is to investigate the existence of some static regions in a dynamic execution, as already attempted in the literature. Identifying such regions and eliminating the run-time checking of the firing conditions for them can lead to a performance improvement. The success of this approach is subject to an integration of such pattern-extracting solutions to the toolchain under two conditions. First, an identification of static regions must be performed in conjunction with the bottleneck analysis. For instance, even if a static region is identified, but it does not belong to the CP, eliminating the scheduler run-time cost will not lead to a performance improvement, since the length of the CP is not reduced. Second, the proportions between the sizes of the static and dynamic regions in the overall execution must be studied. A performance improvement is possible only if the size of the static regions is remarkably bigger than the dynamic ones. Hence, this approach may be successful only for a limited set of applications.

Reduction of the intra-partition scheduler working time

Another scheduling-related problem applies to a situation when none of the actors in a partition is eligible to execute. In this particular case, the core the actors are partitioned to is idle and the scheduler keeps iterating over the actors and checking the firing conditions in vain. Identifying the time slots when such a situation occurs and establishing the executions in other partitions triggering an execution in the idle partition, can lead to a decision about switching off the core during the unused slot and hence reduce power consumption. An alternative path towards run-time partitioning is to consider assigning another actor to that core during the identified time slots. The identification of the idle time slots and the operating time slots for the actors considered for a run-time partitioning, could be performed by means of performance estimation.

Identification of alternative improvement paths

If the CP workload is quasi-equally distributed among different components of the design and further parallelization or algorithmic optimization effort does not bring much improvement, the rich information obtained in the VSS process can lead to some considerations of new architectural solutions for the analyzed designs. For the case of the HEVC decoder, one of the possibilities is to modify the currently used concept of a multi-parser (described in Section 10.1.3). In its current implementation, it assumes merging different portions of the parsed data before it is transferred to the rest of the decoder. An alternative to this approach is to create multiple instances of the decoder operating directly on the portioned and parsed data. In this way, the merging process could be eliminated and the potential parallelism of the decoder should grow drastically. The main drawback of this approach is the explosion of the complexity of the decoder implying also the complexity of the design space and the much increased resource requirements. Hence, an investigation is required to determine if this cost is worth the possible performance gain.

Bibliography

[1] S. Borkar and A. A. Chien, “ The future of microprocessors,” in *Commun. ACM*, vol. 54, pp. 67–77, 2011.

[2] “Epiphany architecture reference.” http://www.adapteva.com/docs/epiphany_arch_ref.pdf. Accessed: November 2016.

[3] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Leger, B. Orgogozo, J. Reybert, and T. Strudel, “A distributed run-time environment for the kalray mppa-2556 integrated manycore processor,” in *Procedia Computer Science*, 2013.

[4] J. Diaz, C. Munoz-Caro, and A. Nino, “A survey of parallel programming models and tools in the multi and many-core era,” in *IEEE Transactions on parallel and distributed systems*, vol. 23, pp. 1369–1386, 2012.

[5] A. Geist, A. Beguelin, J. D. adn W. Jiang, B. Manchek, and V. Sunderam, “PVM: Parallel Virtual Machine - A Users Guide and Tutotial for Netowrk Parallel Computing,” tech. rep., MIT Press, Cambridge, MA, 1994.

[6] M. P. Forum, “Mpi: A message-passing interface standard,” tech. rep., Knoxville, TN, USA, 1994.

[7] “OpenMP 4.0 API C/C++ Syntax Quick Reference Card.” http://openmp.org/mp-documents/OpenMP-4.0-C.pdf. Accessed: November 2016.

[8] E. Lee, “The problem with threads,” *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[9] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, “A catalog of stream processing optimizations,” *ACM Computing Surveys*, vol. 46, no. 4, 2014.

[10] M. Michalska, N. Zufferey, E. Bezati, and M. Mattavelli, “Design space exploration problem formulation for dynamic dataflow programs on heterogeneous architectures,” *10th International Symposium on Embedded Multicore/Many-core Systems on Chip, Lyon, France, September 21-23*, 2016.

[11] M. Michalska, J. Boutellier, and M. Mattavelli, “A methodology for profiling and partitioning stream programs on many-core architectures,” in *Procedia Computer Science Ed. International Conference on Computational Science (ICCS), Reykjavik, Iceland, June 1-3*, vol. 51, p. 2962–2966, 2015.

[12] S. Casale-Brunet, M. Michalska, J. J. Ahmad, E. Bezati, and M. Mattavelli, “High-accuracy performance estimation of dynamic dataflow programs on multi-core platforms,” *Integration, the VLSI Journal (to appear)*, 2017.

[13] M. Michalska, S. Casale-Brunet, E. Bezati, M. Mattavelli, and J. Janneck, “Trace-based manycore partitioning of stream-processing applications,” *50th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, USA, November 6-9*, 2016.

[14] S. Casale-Brunet, M. Michalska, J. Ahmad, M. Mattavelli, M. Selva, K. Marquet, and L. Morel, “Memory profiling of dynamic dataflow programs,” in *Colloque SoC-SIP, Nantes, France, June 8-10*, 2016.

[15] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli, “Execution trace graph based multi-criteria partitioning of stream programs,” in *Procedia Computer Science Ed. International Conference on Computational Science (ICCS), Reykjavik, Iceland, June 1-3*, vol. 51, pp. 1443–1452, 2015.

[16] S. Casale-Brunet, M. Wiszniewska, E. Bezati, M. Mattavelli, J. Janneck, and M. Canale, “TURNUS: an open-source design space exploration framework for dynamic stream programs,” *2014 Conference on Design and Architectures for Signal and Image Processing (DASIP), Madrid, Spain, October 8-10*, 2014.

[17] M. Michalska, N. Zufferey, and M. Mattavelli, “Tabu search for partitioning dynamic dataflow programs,” in *Procedia Computer Science Ed. International Conference on Computational Science (ICCS), San Diego, California, USA, June 6-8*, vol. 80, pp. 1577–1588, 2016.

[18] M. Michalska, N. Zufferey, and M. Mattavelli, “Performance estimation based multi-criteria partitioning approach for dynamic dataflow programs,” in *Journal of Electrical and Computer Engineering, vol. 2016, Article ID 8536432, 15 pages*, 2016.

[19] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli, “High-accuracy performance estimation for design space exploration of dynamic dataflow programs,” *IEEE Transactions on Multi-Scale Computing Systems: Special Issue on Emerging Technologies and Architectures for Manycore (to appear)*, 2017.

[20] M. Michalska, N. Zufferey, J. Boutellier, E. Bezati, and M. Mattavelli, “Efficient scheduling policies for dynamic dataflow programs executed on multi-core,” *9th International*

Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG), Prague, Czech Republic, January 18, 2016.

[21] M. Michalska, E. Bezati, S. Casale-Brunet, and M. Mattavelli, "A partition scheduler model for dynamic dataflow programs," in *Procedia Computer Science Ed. International Conference on Computational Science (ICCS), San Diego, California, USA, June 6-8*, vol. 80, pp. 2287–2291, 2016.

[22] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli, "High-precision performance estimation of dynamic dataflow programs," *10th International Symposium on Embedded Multicore/Many-core Systems on Chip, Lyon, France, September 21-23*, 2016.

[23] M. Michalska, J. J. Ahmad, S. Casale-Brunet, E. Bezati, and M. Mattavelli, "Performance estimation of program partitions on multi-core platforms," *International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS), Bremen, Germany, September 21-23*, 2016.

[24] E. Lee and A. Sangiovanni-Vincentelli, "Comparing models of computation," *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pp. 234 – 241, 1997.

[25] W. M. Johnston, J. R. P. Hanna, and R. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36 (1), pp. 1 – 34, 2004.

[26] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing (J. L. Rosenfeld, ed.)*, pp. 471 – 475, 1974.

[27] E.Lee and T.Parks, "Dataflow process networks," *Proceedings of the IEEE*, pp. 773 – 799, 1995.

[28] J. Janneck, "Actor machines: A machine model for dataflow actors and its applications," *Technical Memo LTH Report 96, 2011 (corrections 2013-03-01), Lund University, Computer Science Department*, 2013.

[29] A. Grabowski, "Scott-continuous functions," *Journal of Formalized Mathematics*, vol. 10, 1998.

[30] D. McAllester, P. Panangaden, and V.Shanbhogue, "Nonexpressibility of fairness and signaling," *J. Comput. Syst. Sci*, vol. 47, pp. 287 – 321, 1993.

[31] J. Dennis, "First versionof a dataflow procedure language," *Proceedings Colloque Sur La Programmation*, pp. 362 – 376, 1974.

[32] E. Lee and E. Matsikoudis, "A denotational semantics for dataflow with firing," *Memorandum UCB/ERL M97/3, ElectronicsResearch*, 1997.

[33] J. Johnston, W.and Hanna and R. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[34] C. Lucarz, G. Roquier, and M. Mattavelli, "High level design space exploration of RVC codec specifications for multi-core heterogeneous platforms," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pp. 191–198, Oct. 2010.

[35] S. Bhattacharyya, J. Eker, J. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *Journal of Signal Processing Systems*, vol. 63, pp. 251 – 263, 2011.

[36] C. Lucarz, M. Mattavelli, and J. Janneck, "Optimization of portable parallel signal processing applications by design space exploration of dataflow programs," in *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pp. 43 –48, Oct. 2011.

[37] C. Lucarz, *Dataflow Programming for Systems Design Space Exploration for Multicore Platforms.* PhD thesis, EPFL - STI - EDIC, Lausanne, 2011.

[38] J. Castrillon, A. Tretter, R. Leupers, and G. Ascheid, "Communication-aware Mapping of KPN Applications Onto Heterogeneous MPSoCs," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), pp. 1266–1271, ACM, 2012.

[39] S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, eds., *Handbook of Signal Processing Systems.* Springer, 2013.

[40] W. Najjar, E. Lee, and G. Gao, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, no. 13, pp. 1907–1929, 1999.

[41] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, pp. 24–35, 1987.

[42] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Comput. Surv.*, vol. 31, pp. 406–471, Dec. 1999.

[43] Z. Gu, M. Yuan, N. Guan, M. Lv, X. He, Q. Deng, and G. Yu, "Static Scheduling and Software Synthesis for Dataflow Graphs with Symbolic Model-Checking," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, (Washington, DC, USA), pp. 353–364, IEEE Computer Society, 2007.

[44] T. Parks, J. Pino, and E. A. Lee, "A comparison of synchronous and cyclo-static dataflow," *Asilomar Conference on Signals, Systems and Computers*, 1995.

[45] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, “Cycle-static dataflow,” *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397–408, 1996.

[46] J. Buck and E. Lee, “Scheduling dynamic dataflow graphs with bounded memory using the token flow model,” in *IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP-93*, vol. 1, pp. 429–432, 1993.

[47] J. Buck, “Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams,” in *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, vol. 1, pp. 508–513, 1994.

[48] B. Bhattacharya and S. Bhattacharyya, “Parameterized dataflow modeling for dsp systems,” in *IEEE Transactions on Signal Processing*, 2011.

[49] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk, “A scenario-aware dataflow 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*, pp. 185–194, 2006.

[50] P. Fradet, A. Girault, and P. Poplavko, “Spdf: A schedulable parametric data-flow moc,” in *DATE*, pp. 769–774, 2012.

[51] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueu, “Bpdf: A statically analyzable dataflow model with integer and boolean parameters,” in *EMSOFT*, pp. 3–10, 2013.

[52] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, “Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator,” in *DATE*, pp. 983–987, 2012.

[53] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueu, “Bpdf: Boolean parametric data flow,” in *Research Report RR-8333, INRIA*, 2013.

[54] X. K. Do, *A model of programming languages for dynamic real-time streaming applications.* PhD thesis, Universite Pierre et Marie Cuire, France, 2016.

[55] S. Bhattacharyya, E. Deprettere, and B. Theelen, “Dynamic dataflow graphs,” in *Handbook of Signal Processing Systems*, pp. 905–944, Springer, 2013.

[56] M. Geilen and T. Basten, “Kahn process networks and a reactive extension,” in *Handbook of Signal Processing Systems*, pp. 1041–1081, Springer, 2013.

[57] S. Casale-Brunet, A. Elguindy, E. Bezati, R. Thavot, G. Roquier, M. Mattavelli, and J. W. Janneck, “Methods to Explore Design Space for MPEG RMC Codec Specifications,” *Image Commun.*, vol. 28, pp. 1278–1294, Nov. 2013.

[58] S. Casale-Brunet, E. Bezati, M. Mattavelli, M. Canale, and J. Janneck, "Execution trace graph analysis of dataflow programs: bounded buffer scheduling and deadlock recovery using model predictive control," in *Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2014.

[59] M. Canale, S. Casale-Brunet, E. Bezati, M. Mattavelli, and J. Janneck, "Dataflow programs analysis and optimization using model predictive control techniques: An example of bounded buffer scheduling," in *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1–6, Oct. 2014.

[60] J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli, "Scheduling of Dynamic Dataflow Programs Based on State Space Analysis," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 1661–1664, IEEE, 2012.

[61] H. Yviquel, J. Boutellier, M. Raulet, and E. Casseau, "Automated design of networks of transport-triggered architecture processors using dynamic dataflow programs," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1295 – 1302, 2013.

[62] J. Ersfolk, *Scheduling Dynamic Dataflow Graphs with Model Checking*. PhD thesis, TUCS, 2014.

[63] H. Yviquel, A. Sanchez, P. Jaaskelainen, J. Takala, M. Raulet, and E. Casseau, "Embedded Multi-Core Systems Dedicated to Dynamic Dataflow Programs," *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 121–136, 2015.

[64] J. Eker and J. Janneck, "CAL Language Report: Specification of the CAL Actor Language," Technical Memo UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley, Dec. 2003.

[65] F. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM*, vol. 19, pp. 137–147, Mar. 1976.

[66] I. 23001-4:2011, "Information technology - MPEG systems technologies - Part 4: Codec configuration representation," 2011.

[67] M. Mattavelli, J. Janneck, and M. Raulet, "MPEG Reconfigurable Video Coding," in *Handbook of Signal Processing Systems* (S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, eds.), pp. 43–67, Springer US, 2010.

[68] M. Mattavelli, "MPEG reconfigurable video representation," in *The MPEG Representation of Digital Media* (L. Chiariglione, ed.), pp. 231–247, Springer New York, 2012.

[69] E. Jang, M. Mattavelli, M. Preda, M. Raulet, and H. Sun, "Reconfigurable Media Coding: An overview ," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1215–1223, 2013.

[70] “The Open RVC-CAL Compiler, Orcc.” http://github.com/orcc. Accessed: November 2016.

[71] M. Wipliez, *Compilation infrastructure for dataflow programs.* Theses, INSA de Rennes, Dec. 2010.

[72] H. Yviquel, A. Lorence, K. Jerbi, and G. Cocherel, “Orcc: Multimedia development made easy,” *Proceedings of the 21st ACM International Conference on Multimedia*, pp. 863–866, 2013.

[73] “Eclipse IDEs.” http://eclipse.org/ide. Accessed: November 2016.

[74] “Eclipse modeling framework.” http://eclipse.org/modeling/emf. Accessed: November 2016.

[75] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd ed., 2009.

[76] “Xtext: Language development made easy!.” http://eclipse.org/Xtext. Accessed: November 2016.

[77] “Xtend: Modernized java.” http://eclipse.org/xtend. Accessed: November 2016.

[78] “Daedalus: System-Level Design For Multi-Processor System-on-Chip.” http://daedalus.liacs.nl. Accessed: November 2016.

[79] M. Thompson, H. Nikolov, T. Stefanov, A. Pimentel, C. Erbas, S. Polstra, and E. Deprettere, “A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs,” in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS ’07, (New York, NY, USA), pp. 9–14, ACM, 2007.

[80] H. Nikolov, T. Stefanov, and E. Deprettere, “Systematic and Automated Multiprocessor System Design, Programming, and Implementation,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, pp. 542–555, Mar. 2008.

[81] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, “MAPS: an integrated framework for MPSoC application parallelization,” in *Proceedings of the 45th annual Design Automation Conference*, pp. 754–759, ACM, 2008.

[82] J. Castrillon, R. Velasquez, A. Stulova, W. Sheng, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr, “Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms,” *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2010.

[83] R. Leupers and J. Castrillon, “MPSoC programming using the MAPS compiler,” in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pp. 897–902, Jan. 2010.

[84] J. Castrillon, R. Leupers, and G. Ascheid, “MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs,” *IEEE Transactions on Industrial Informatics*, pp. 527 – 545, 2013.

[85] R. Leupers, M. Aguilar, J. Eusse, and W. Sheng, “MAPS: A software development environment for embedded multicore applications,” *Handbook of Hardware/Software Codesign*, 2017.

[86] A. Mihal, C. Kulkarni, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, C. Sauer, K. Vissers, and S. Malik, “Developing Architectural Platforms: A Disciplined Approach,” *IEEE Des. Test*, vol. 19, pp. 6–16, Nov. 2002.

[87] M. Gries and K. Keutzer, *Building ASIPs: The Mescal Methodology.* Springer Publishing Company, Incorporated, 1st ed., 2010.

[88] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, “Metropolis: an integrated electronic system design environment,” *Computer*, vol. 36, pp. 45–52, 2003.

[89] “Metropolis: Design Environment for Heterogeneous Systems.” https://embedded.eecs.berkeley.edu/metropolis. Accessed: November 2016.

[90] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. Joo, “PeaCE: A Hardware-software Codesign Environment for Multimedia Embedded Systems,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 1–25, May 2008.

[91] “PeaCE : Codesign Environment.” http://peace.snu.ac.kr/research/peace/index.php. Accessed: November 2016.

[92] “PREESM: the parallel and real-time embedded executives scheduling method.” http://sourceforge.net/projects/preesm/. Accessed: November 2016.

[93] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.-F. Nezan, “An open framework for rapid prototyping of signal processing applications,” *EURASIP Journal on Embedded Systems*, 2009.

[94] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. Nezan, and S. Aridhi, “Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming,” in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pp. 36–40, IEEE, 2014.

[95] "Ptolemy project: heterogeneous modeling and design." http://ptolemy.eecs.berkeley.edu. Accessed: November 2016.

[96] E. Lee, "Overview of The Ptolemy Project," Technical Memo UCB/ERL M98/71, Electronics Research Laboratory, University of California at Berkeley, Nov. 1998.

[97] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF for free," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pp. 276–278, Jun. 2006.

[98] "SDF3." http://www.es.ele.tue.nl/sdf3/. Accessed: November 2016.

[99] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *Computers, IEEE Transactions on*, vol. 55, pp. 99–112, Feb. 2006.

[100] "RDF4J." http://rdf4j.org/. Accessed: November 2016.

[101] "Space Codesign Systems." http://www.spacecodesign.com. Accessed: November 2016.

[102] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois, and E. Aboulhamid, "A SystemC refinement methodology for embedded software," *Design Test of Computers, IEEE*, vol. 23, pp. 148–158, Mar. 2006.

[103] D. Turaga, H. Andrade, B. Gedik, C. Venkatramani, O. Verscheure, J. Harris, J. Cox, W. Szewczyk, and P. Jones, "Design Principles for Developing Stream Processing Applications," *Softw. Pract. Exper.*, vol. 40, pp. 1073–1104, Nov. 2010.

[104] B. Gedik, H. Andrade, K. Wu, P. Yu, and M. Doo, "SPADE: The System S Declarative Stream Processing Engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 1123–1134, ACM, 2008.

[105] W. De Pauw, M. Leţia, B. Gedik, H. Andrade, A. Frenkiel, M. Pfeifer, and D. Sow, "Visual Debugging for Stream Processing Applications," in *Runtime Verification* (H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, eds.), vol. 6418 of *Lecture Notes in Computer Science*, pp. 18–35, Springer Berlin Heidelberg, 2010.

[106] "SynDEx." http://www.syndex.org. Accessed: November 2016.

[107] "SystemCoDesigner." http://www.mycodesign.com/research/scd. Accessed: November 2016.

[108] C. Haubelt, M. Meredith, T. Schlichter, and J. Keinert, "SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioral Models," in *Proceedings*

of the 45th Design Automation Conference (DACÓ8), (Anaheim, CA, USA.), pp. 580–585, Jun. 2008.

[109] J. Keinert, M. Streubuhr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, “SystemCoDesigner: an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, pp. 1–23, Jan. 2009.

[110] S. Verdoolaege, H. Nikolov, and T. Stefanov, “Pn: A Tool for Improved Derivation of Process Networks,” *EURASIP J. Embedded Syst.*, vol. 2007, pp. 19–19, Jan. 2007.

[111] J. Castrillon, *Programming Heterogeneous MPSoCs.* Springer, 2013.

[112] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, and H. Meyr, “A high-level virtual platform for early MPSoC software development,” in *CODES+ISSS ’09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 11–20, ACM, 2009.

[113] K. Desnos, M. Pelcat, J. Nezan, S. Bhattacharyya, and S. Aridhi, “PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pp. 41–48, Jul. 2013.

[114] 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 *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE ’03, (Washington, DC, USA), pp. 123–133, IEEE Computer Society, 2003.

[115] “Forte Synthesizer.” http://www.cadence.com/products/sd/cynthesizer/. Accessed: November 2016.

[116] M. Lukasiewycz, M. Glass, C. Haubelt, and J. Teich, “ Efficient symbolic multi-objective design space exploration,” in *Proceedings of the 13th Asia and South Pacific Design Automation Conference (ASP-DAC 2008)*, pp. 691–696, 2008.

[117] “CAL design suite.” http://sourceforge.net/projects/caldesignsuite/. Accessed: November 2016.

[118] C. Lucarz, G. Roquier, and M. Mattavelli, “ High level design space exploration of RVC codec specifications for multi-core heterogeneous platforms,” in *Conference on Design and Architectures for Signal and Image Processing, DASIP*, 2010.

[119] “COMPA Project.” http://www.compa-project.org. Accessed: May 2015.

[120] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” in *Science 220*, pp. 671–680, 1983.

[121] S. Casale-Brunet, *Analysis and optimization of dynamic dataflow programs.* PhD thesis, EPFL, Switzerland, 2015.

[122] E. Bezati, *High-level synthesis of dataflow programs for heterogeneous platforms: design flow tools and design space exploration.* PhD thesis, EPFL, Switzerland, 2015.

[123] J. Janneck, I. Miller, and D. Parlour, “Profiling dataflow programs,” in *Multimedia and Expo, 2008 IEEE International Conference on*, pp. 1065–1068, Jun. 2008.

[124] C. Zebelein, C. Haubelt, J. Falk, T. Schwarzer, and J. Teich, “Representing mapping and scheduling decisions within dataflow graphs,” in *Specification Design Languages (FDL), 2013 Forum on*, pp. 1–8, Sept 2013.

[125] G. M. Amdahl, “Validity of the single-processor approach to achieving large scale computing capabilities,” in *AFIPS Conference Proceedings*, vol. 30, pp. 483–485, 1967.

[126] P. Mucci, S. Browne, C. Deane, and G. Ho, “PAPI: A Portable Interface to Hardware Performance Counters,” in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10, 1999.

[127] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, “Codesign toolset for application-specific instruction-set processors,” *Multimedia on Mobile Devices, Proc. SPIE*, vol. 6507, 2007.

[128] O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez, “Customized exposed datapath soft-core design flow with compiler support,” in *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL ’10, (Washington, DC, USA), pp. 217–222, IEEE Computer Society, 2010.

[129] J. Helkala, T. Viitanen, H. Kultala, P. Jääskeläinen, J. Takala, T. Zetterman, and B. Heikki, “Variable length instruction compression on transport triggered architectures,” *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XIV)*, 2014.

[130] P. Jääskeläinen, H. Kultala, T. Viitanen, and J. Takala, “Code density and energy efficiency of exposed datapath architectures,” *Journal of Signal Processing Systems*, vol. 80, 2015.

[131] H. Yviquel, *From dataflow-based video coding tools to dedicated embedded multi-core platforms.* PhD thesis, Université Rennes, 2013.

[132] M. Selva, *Performance Monitoring of Throughput Constrained Dataflow Programs Executed On Shared-Memory Multi-core Architectures.* PhD thesis, INSA Lyon, France, 2015.

[133] H. Yviquel, A. Sanchez, P. Jaaskelainen, J. Takala, M. Raulet, and E. Casseau, "Embedded multi-core systems dedicated to dynamic dataflow programs.," *Journal of Signal Processing Systems*, pp. 1–16, 2014.

[134] G. Paoloni, "How to Benchmark Code Execution Times on Intel ®IA-32 and IA-64 Instruction Set Architectures," tech. rep., Intel Corporation, 09 2010.

[135] J. Ahmad, S. Li, R. Thavot, and M. Mattavelli, "Secure Computing with the MPEG RVC Framework," *Signal Processing: Image Communication, special issue on Recent Advances on MPEG Codec Configuration Framework*, vol. 28, pp. 1315–1334, November 2013.

[136] M. Selva, L. Morel, and K. Marquet, "numap: A portable library for low level memory profiling," *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XVI)*, 2016.

[137] V. Weaver, D.Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," *IEEE International Symposium on Performance Analysis of Systems and Software, Austin*, 2013.

[138] O. Sinnen, *Task scheduling for parallel systems.* Wiley Series on Parallel and Distributed Computing, 2007.

[139] M. Tanaka and O. Tatebe, "Workflow scheduling to minimize data movement using multi-constraint graph partitioning," in *Proceedings of the 2012 12th International Symposium on Cluster, Cloud and Grid Computing*, pp. 65–72, IEEE, 2012.

[140] I. Ahmad and Y. Kwok, "On parallelizing the multiprocessor scheduling problem," in *Transactions on Parallel and Distributed Systems*, pp. 424–432, IEEE, 1999.

[141] R. Diekman and R. Preis, "Load balancing strategies for distributed memory machines," in *Parallel and distributed processing for computational mechanics*, pp. 124–157, 1999.

[142] G. N. Khan and M. Jin, "A new graph structure for hardware-software partitioning of heterogeneous systems," in *Canadian Conference on Electrical and Computer Engineering*, pp. 229–232, IEEE, 2004.

[143] J. Wang, E. Sha, and N. L. Passos, "Minimization of memory access overhead for multidimensional DSP applications via multilevel partitioning and scheduling," in *Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, pp. 741–753, IEEE, 1997.

[144] W. Jigang, T. Srikanthan, and G. Chen, "Algorithmic aspects of hardware/software partitioning: 1D search algorithms," in *Transactions on Computers*, pp. 532–544, IEEE, 2010.

[145] L. Wang, J. Liu, J. Hu, Q. Zhuge, and E. Sha, "Optimal assignment for tree-structure task graph on heterogeneous multicore systems considering time constraint," in *International Symposium on Embedded Multicore Socs (MCSoC)*, pp. 121–127, IEEE, 2012.

[146] B. Tafesse, A. Raina, J. Suseela, and V. Muthukumar, "Efficient scheduling algorithms for MpSoC systems," in *Eighth International Conference on Information Technology: New Generations (ITNG)*, pp. 683–688, IEEE, 2011.

[147] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet, "Efficient multicore scheduling of dataflow process networks," *IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 198 – 203, 2011.

[148] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, 2008.

[149] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," *Queue - Development*, vol. 11, 2013.

[150] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," Tech. Rep. MSR-TR-2013-102, Microsoft Research, February 2013.

[151] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study," in *Workshop on Signal Processing Systems*, pp. 281–286, IEEE, 2008.

[152] S. Padmanabhan, Y. Chen, and R. D. Chamberlain, "Convexity in non-convex optimizations of streaming applications," in *IEEE 18th International Conference on Parallel and Distributed Systems*, pp. 668–675, 2012.

[153] T. Saxena and G. Karsai, "Novel heuristic mapping algorithms for design space exploration of multiprocessor embedded architectures," in *2011 18th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pp. 71–80, 2011.

[154] S. Padmanabhan, Y. Chen, and R. D. Chamberlain, "Optimal design-space exploration of streaming applications," in *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 227–230, 2011.

[155] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient software performance estimation methods for hardware/software codesign," in *33rd Design Automation Conference Proceedings 1996*, pp. 605–610, 1996.

[156] S. Tamaskar, K. Neema, T. Kotegawa, and D. DeLaurentis, "Complexity enabled design space exploration," in *2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 1250–1255, 2011.

[157] S. Sinaei and O. Fatemi, "A meta-framework for design space exploration," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 801–804, 2016.

[158] 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*, vol. 10, pp. 358–374, 2006.

[159] T. Givargis, F. Vahid, and J. Henkel, "System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 25–30, 2001.

[160] J. Teich, T. Blickle, and L. Thiele, "An evolutionary approach to system-level synthesis," in *In Proc. of Codes/CASHE 97 - the 5th Int. Workshop on Hardware/Software Codesign*, (Braunschweig, Germany), pp. 167–171, Mar 1997.

[161] T. Schlichter, M. Lukasiewycz, C. Haubelt, and J. Teich, "Improving system level design space exploration by incorporating sat-solvers into multi-objective evolutionary algorithms," in *2006 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2006), 2-3 March 2006, Karlsruhe, Germany*, pp. 309–316, 2006.

[162] T. Saxena and G. Karsai, "Towards a generic design space exploration framework," in *2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)*, pp. 1940–1947, 2010.

[163] A. Hertz, D. Schindl, and N. Zufferey, "Lower bounding and tabu search procedures for the frequency assignment problem with polarization constraints," *4OR*, vol. 3 (2), pp. 139 – 161, 2005.

[164] P. Garg, A. Gupta, and J. W. Rozenblit, "Performance analysis of embedded systems in the virtual component co-design environment," in *11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pp. 61–68, 2004.

[165] E. Silver and N. Zufferey, "Inventory control of an item with a probabilistic replenishment lead time and a known supplier shutdown period," *International Journal of Production Research*, vol. 49, pp. 923–947, 2011.

[166] T. Taghavi, A. D. Pimentel, and M. Thompson, "System-level mp-soc design space exploration using tree visualization," in *2009 IEEE/ACM/IFIP 7th Workshop on Embedded Systems for Real-Time Multimedia*, pp. 80–88, 2009.

[167] L. Wang, Y. Tang, Y. Deng, F. Qin, Q. Dou, G. Zhang, and F. Zhang, "A scalable and fast microprocessor design space exploration methodology," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 33–40, 2015.

[168] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporal, "Automated bottleneck-driven design-space exploration of media processing systems," in *2010 Design, Automation and Test in Europe Conference and Exhibition (DATE 2010)*, pp. 1041–1046, 2010.

[169] A. Hertz, M. Plumettaz, and N. Zufferey, "Variable space search for graph coloring," *Discrete Applied Mathematics*, vol. 156, pp. 2551–2560, 2008.

[170] Q. Wu and M. S. Hsiao, "A new simulation-based property checking algorithm based on partitioned alternative search space traversal," in *Tenth IEEE International High-Level Design Validation and Test Workshop*, pp. 121–126, 2006.

[171] M. A. Beck and I. C. Parmee, "Design exploration: extending the bounds of the search space," *Proceedings of the 1999 Congress on Evolutionary Computation*, vol. 1, 1999.

[172] J. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, pp. 384 – 393, 1975.

[173] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Seventh International Conference on Application of Concurrency to System Desig*, pp. 29–40, IEEE, 2007.

[174] M. Garey and D. Johnson, *A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[175] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, pp. 268 – 308, 2003.

[176] I. Osman and G. Laporte, "Metaheuristics: A bibliography," *Annals of Operations Research*, pp. 513–623, 1996.

[177] M. Gendreau and J.-Y. Potvin, *Handbook of Metaheuristics.* Springer, 2010.

[178] N. Zufferey, "Metaheuristics: Some principles for an efficient design," *Computer Technology and Application*, vol. 3, pp. 446–462, 2012.

[179] M. A. Arslan, J. W. Janneck, and K. Kuchcinski, "Partitioning and mapping dynamic dataflow programs," *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pp. 1452–1456, 2012.

[180] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, pp. 99–112, 2006.

[181] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," *Algorithm Engineering: Selected Results and Surveys, LNCS 9220*, 2015.

[182] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359 – 392, 1998.

[183] F. Pellegrini and J. Roman, "A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," *Proceedings of HPCN'96, Brussels, Belgium*, pp. 493 – 498, 1996.

[184] H. Yviquel, E. Casseau, M. Raulet, P. Jaaskelainen, and J. Takala, "Towards run-time actor mapping of dybamic dataflow programs onto multi-core platforms," *8th International SYmposium on Image and Signal Processing and Analysis*, pp. 732 – 737, 2013.

[185] T. Tabirca, S. Tabirca, L. Freeman, and L. T. Yang, "A static workload balance scheduling algorithm," *International Conference on Parallel Processing Workshops*, 2002.

[186] S. Thevenin, N. Zufferey, and M. Widmer, "Metaheuristics for a scheduling problem with rejection and tardiness penalties," *Journal of Scheduling*, vol. 18, pp. 89–105, 2015.

[187] F. Glover, "Tabu search - part I," *ORSA Journal on Computing*, vol. 1, pp. 190–205, 1989.

[188] G. Kahn, "The semantics of simple language for parallel programming," *IFIP Congress*, 1974.

[189] T. Parks, J. Pino, and E. Lee, "A comparison of synchronous and cycle-static dataflow," in *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, vol. 1, pp. 204–210, IEEE, 1995.

[190] W. Liu, Z. Gu, J. Xu, Y. Wang, and M. Yuan, "An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking," in *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, (New York, NY, USA), pp. 61–70, ACM, 2009.

[191] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, (New York, NY, USA), pp. 819–824, ACM, 2005.

[192] S. Stuijk, M. Geilen, and T. Baste, “Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs,” in *Proceedings of the 43rd annual conference on Design automation*, pp. 899–904, ACM, 2006.

[193] S.-I. Han, X. Guerin, S.-I. Chae, and A. A. Jerraya, “Buffer memory optimization for video codec application modeled in simulink,” in *Proceedings of the 43rd annual conference on Design automation*, pp. 689–694, ACM, 2006.

[194] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-vincentelli, “Scheduling for embedded real-time systems,” in *Des. Test*, pp. 71–82, IEEE, 1998.

[195] H. Salunkhe, A. Lele, O. Moreira, and K. v. Berkel, “Buffer allocation for dynamic real-time streaming applications running on a multi-processor without back-pressure,” *Euromicro Conference on Digital System Design*, pp. 250–254, 2015.

[196] E. Cheung, H. Hsieh, and F. Balarin, “Automatic buffer sizing for rate-constrained kpn applications on multiprocessor system-on-chip,” *IEEE International High Level Design Validation and Test Workshop*, pp. 37–44, 2007.

[197] S. Casale-Brunet, M. Mattavelli, and J. W. Janneck, “Buffer optimization based on critical path analysis of a dataflow program design,” in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pp. 1384–1387, IEEE, 2013.

[198] T. Parks, *Bounded scheduling of process networks.* PhD thesis, University of California at Berkeley, USA, 1995.

[199] D. J. D. S. Jorre, D. Renzi, S. C. Brunet, M. Wiszniewska, E. Bezati, and M. Mattavelli, “MPEG high efficient video coding stream programming and many-cores scalability,” *2014 Conference on Design and Architectures for Signal and Image Processing (DASIP), Madrid, Spain, October 8-10*, 2014.

[200] L. Benini, M. Lombardi, M. Milano, and M. Ruggiero, “Optimal resource allocation and scheduling for the CELL BE platform,” *Annals of Operations Research*, pp. 51–77, 2011.

[201] M. Eisenring, J. Teich, and L. Thiele, “Rapid prototyping of dataflow programs on hardware/software architectures,” in *Proc. of HICSS-31, Proc. of the Hawai Int. Conf. on System Sciences*, pp. 187–196, 1998.

[202] S. Casale-Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. W. Janneck, “Partitioning and optimization of high level stream applications for multi clock domain architectures,” *IEEE Workshop on Signal Processing*, pp. 177–182, 2013.

[203] K. R. Baker and D. Trietsch, *Principles of Sequencing and Scheduling.* Wiley, 2009.

[204] J. Boutellier, V. Sadhanala, C. Lucarz, P. Brisk, and M. M. Mattavelli, "Scheduling of dataflow models within the reconfigurable video coding framework," *IEEE Workshop on Signal Processing Systems*, pp. 182–187, 2008.

[205] J. W. Janneck, "Actors and their composition," *Formal Aspects Comput.*, pp. 349–369, 2003.

[206] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silven, "Actor merging for dataflow process networks," *IEEE Transactions on Signal Processing*, pp. 2496–2508, 2015.

[207] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts.* Wiley, 2005.

[208] S. Singh, "Round-robin with credits: an improved scheduling strategy for rate-allocation in high-speed packet-switching," *Global Telecommunications Conference*, 1994.

[209] M. Obaidat and G. Papadimitriou, *Applied System Simulation: Methodologies and Applications.* Springer Publishing Company, 2013.

[210] S. Pllana, I. Brandic, and S. Benkner, "Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art," *First International Conference on Complex, Intelligent and Software Intensive Systems*, pp. 279–284, 2007.

[211] P. Bose and T. Conte, "Performance analysis and its impact on design," *Computer*, pp. 41–49, 1998.

[212] H. J. Sri Parameswaran, Alex Ignatovic, "Performance estimation of pipelined multiprocessor system-on-chips (mpsocs)," in *IEEE Transactions on Parallel & Distributed Systems*, 2013.

[213] S. L. Shee and S. Parameswaran, "Design methodology for pipelined heterogeneous multiprocessor systems," in *Proc. 44th ACM/IEEE Annu. DAC*, pp. 811–816, 2007.

[214] Q. Hu, J. Shu, J. Fan, and Y. Lu, "Run-time performance estimation and fairness-oriented scheduling policy for concurrent gpgpu applications," in *45th International Conference on Parallel Processing (ICPP)*, 2016.

[215] N. Frid, D. Ivosevic, and V. Sruk, "Performance estimation in heterogeneous mpsoc based on elementary operation cost," in *39th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2016, Opatija, Croatia, May 30 - June 3, 2016*, pp. 1202–1205, 2016.

[216] D. Ivosevic, N. Frid, and V. Sruk, "Function-level performance estimation for heterogeneous mpsoc platforms," in *Zooming Innovation in Consumer Electronics International Conference (ZINC)*, 2016.

[217] C. Lin, X. Du, X. Jiang, and D. Wang, “An efficient and effective performance estimation method for dse,” in *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2016.

[218] A. Gamatie, R. Ursu, M. Selva, and G. Sassatelli, “Performance prediction of application mapping in manycore systems with artificial neural networks,” *10th International Symposium on Embedded Multicore/Many-core Systems on Chip*, 2016.

[219] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee, “Machine learning based online performance prediction for runtime parallelization and task scheduling,” in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2009.

[220] X. Zheng, L. K. John, and A. Gerstlauer, “Accurate phase-level cross-platform power and performance estimation,” in *Proceedings of the 53rd Annual Design Automation Conference*, DAC ’16, (New York, NY, USA), pp. 4:1–4:6, ACM, 2016.

[221] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, “Learning-based analytical cross-platform performance prediction,” in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XV)*, 2015.

[222] J. Nutaro, *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Wiley Publishing, 2010.

[223] J. Chen and R. M. Clapp, “Critical-path candidates: scalable performance modeling for mpi workloads,” in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pp. 1–10, March 2015.

[224] D. Böhme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer, “Scalable critical-path based performance analysis,” in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 1330–1340, May 2012.

[225] C. Q. Yang and B. P. Miller, “Critical path analysis for the execution of parallel and distributed programs,” in *Distributed Computing Systems, 1988., 8th International Conference on*, pp. 366–373, Jun 1988.

[226] A. Schrijver, *Combinatorial optimization: polyhedra and efficiency*, vol. 24. Springer Science & Business Media, 2003.

[227] E. Bezati, H. Yviquel, M. Raulet, and M. Mattavelli, “A unified hardware/software co-synthesis solution for signal processing systems,” in *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pp. 1–6, Nov. 2011.

[228] document ITU-T Rec. H.265, I.-T. ISO/IEC 23008-2 (HEVC), and ISO/IEC, “High Efficiency Video Coding,” 2013.

[229] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, “ Overview of the H.264/AVC video coding standard,” in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, pp. 560–576, 2003.

[230] J. F. Bossen, B. Bross, K. Suhring, and D. Flynn, “ HEVC complexity and implementation analysis,” in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, pp. 1685–1696, 2012.

[231] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl, “Parallel scalability and efficiency of hevc parallelization approaches,” in *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 1827–1838, Dec. 2012.

[232] S. Casale-Brunet, C. Alberti, M. Mattavelli, and J. Janneck, “Turnus: a unified dataflow design space exploration framework for heterogeneous parallel systems,” *2013 Conference on Design and Architectures for Signal and Image Processing (DASIP), Cagliari, Italy*, October 2013.

[233] “TURNUS.” http://github.com/turnus. Accessed: November 2016.

[234] F.-X. Meuwly, B. Ries, and N. Zufferey, “Solution methods for a scheduling problem with incompatibility and precedence constraints,” *Algorithmic Operations Research*, vol. 5 (2), pp. 75 – 85, 2010.

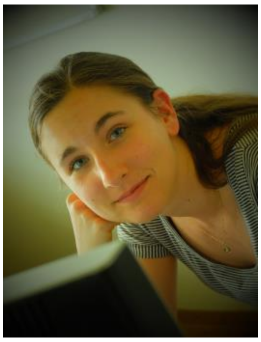
[235] D. Schindl and N. Zufferey, “A Learning Tabu Search for a Truck Allocation Problem with Linear and Nonlinear Cost Components,” *Naval Research Logistics*, vol. 62 (1), pp. 32 – 45, 2015.

[236] S. Thevenin, N. Zufferey, and M. Widmer, “Metaheuristics for a scheduling problem with rejection and tardiness penalties,” *Journal of Scheduling*, vol. 18, pp. 89–105, 2015.

[237] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Computing Surveys*, vol. 43, pp. 1 – 44, 2011.

Małgorzata Maria Michalska

malgorzata.michalska@epfl.ch



(last update: February 2017)

Professional experience

July 2013 - now

Doctoral assistant at **Swiss Federal Institute of Technology (EPFL)**, SCI-STI-MM group, Lausanne, Switzerland
FNS project: Extension and analysis of software specifications of high-level software partitioning and architectural solution explorations

July 2012 - Sep. 2012

Intern at **UBS AG**, London, United Kingdom
IT Business Analysis, Project Management

May 2011 - Sep. 2011

Intern at **PayCo Services GmbH**, Berlin, Germany
Java software development, Technologies: Java Spring, Hibernate, Quartz, MongoDB, JSP

Education

July 2013 - now

Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland
Ph. D: Doctoral School of Electrical Engineering;
Thesis: Systematic design space exploration of dynamic dataflow programs for multi-core platforms

Feb. 2012 - June 2013

Gdansk University of Technology, Gdansk, Poland
M. Sc: Computer Science with specialization in Intelligent Interactive Systems, individual study program;
Thesis: Computer application automatically correcting detuned singing
(Summa cum Laude)

2010 - 2011

Beuth University of Applied Sciences, Berlin, Germany
Erasmus Student Network, field of study: Media Computer Science

2008 - 2012

Gdansk University of Technology, Gdansk, Poland
B. Eng: Electronics and Telecommunications with specialization in Multimedia Systems, individual study program since February 2010;
Thesis: Implementation of an algorithm for pitch shifting of a digital sound
(Summa cum Laude)

2005 - 2008

3rd Bilingual High School, Gdynia, Poland

Scholarships and Awards

September 2016

Best Paper Award at **IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip**, Lyon, France

September 2014

Best Paper Award at **ICT Young Conference**, Gdansk, Poland

June 2012

The **Google Anita Borg Memorial** Scholarship, EMEA *(scholar)*

2009 - 2011:

Merit-based scholarship granted by Gdansk University of Technology, Poland

Computer skills

Programming languages:	Java, C/C++, CAL
Technologies:	HTML, PHP, JSP, .NET
Databases:	SQL, MongoDB, MS Sharepoint
Operating Systems:	Windows, Linux
IDE:	Visual Studio, Qt, Eclipse

Open source projects

TURNUS co-design framework for analysis of static and dynamic dataflow applications, https://github.com/turnus/

Publications

Journals

M. Michalska, S. Casale-Brunet, E. Bezati M. Mattavelli. High-accuracy performance estimation for design space exploration of dynamic dataflow programs. IEEE Transactions on Multi-Scale Computing Systems: Special Issue on Emerging Technologies and Architectures for Manycore, 2017 *(to appear)*.

S. Casale-Brunet, **M. Michalska**, E. Bezati, J. Ahmad, M. Mattavelli. High-accuracy performance estimation of dynamic dataflow programs on multi-core platforms. Integration, the VLSI Journal, 2017 *(to appear)*.

M. Michalska, N. Zufferey, M. Mattavelli. Performance estimation based multi-criteria partitioning approach for dynamic dataflow programs. Journal of Electrical and Computer Engineering, vol. 2016, Article ID 8536432, 15 pages, 2016.

M. Michalska. Application for the Automatic Pitch Detection and Correction of Detuned Singing. Pomiary Automatyka Robotyka (PAR), Volume 20, Pages 25-30, 1/2016.

Conferences

M. Michalska, S. Casale Brunet, E. Bezati, M. Mattavelli, J. Janneck. Trace-Based Manycore Partitioning of Stream-Processing Applications. 50th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, USA, November 6-9 2016.

M. Michalska, S. Casale Brunet, E. Bezati, M. Mattavelli. High-precision performance estimation of dynamic dataflow programs. IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, Lyon, France, September 21-23 2016.

M. Michalska, N. Zufferey, E. Bezati, M. Mattavelli. Design space exploration problem formulation for dataflow programs on heterogeneous architectures. IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, Lyon, France, September 21-23 2016.

M. Michalska, J.J. Ahmad, E. Bezati, S. Casale-Brunet, M. Mattavelli. Performance Estimation of Program Partitions on Multi-core Platforms. International Workshop on Power and Timing Modeling, Optimization and Simulation, Bremen, Germany, September 21-23 2016.

S. Casale-Brunet, **M. Michalska**, J. Ahmad, M. Mattavelli, M. Selva, K. Marquet, and L. Morel. Memory profiling of dynamic dataflow programs. Colloque SoC-SIP, Nantes, France, June 8-10 2016.

M. Michalska, N. Zufferey, M. Mattavelli. Tabu Search for Partitioning Dynamic Dataflow Programs. Procedia Computer Science, Volume 80, Pages 1577-1588, International Conference on Computational Science (ICCS), San Diego, California, USA, June 6-8 2016.

M. Michalska, E. Bezati, S. Casale-Brunet, M. Mattavelli. A Partition Scheduler Model for Dynamic Dataflow Programs. Procedia Computer Science, Volume 80, Pages 2287–2291, International Conference on Computational Science (ICCS), San Diego, California, USA, June 6-8 2016.

M. Michalska, N. Zufferey, J. Boutellier, E. Bezati and M. Mattavelli. Efficient scheduling policies for dataflow programs executed on multi-core. Proceedings of the 9th International Workshop on Programmability and Architectures for Heterogeneous Multicore (MULTIPROG), Prague, Czech Republic, January 18 2016.

M. Michalska, S. Casale-Brunet, E. Bezati and M. Mattavelli. Execution Trace Graph Based Multi-criteria Partitioning of Stream Programs. Procedia Computer Science, Volume 51, Pages 1443 - 1452, International Conference on Computational Science (ICCS), Reykjavik, Iceland, June 1-3 2015.

M. Michalska, J. Boutellier and M. Mattavelli. A Methodology for Profiling and Partitioning Stream Programs on Many-core Architectures. Procedia Computer Science, Volume 51, Pages 2962–2966, International Conference on Computational Science (ICCS), Reykjavik, Iceland, June 1-3 2015.

S. Casale-Brunet, **M. Wiszniewska**, E. Bezati, M. Mattavelli, J. Janneck and M. Canale. TURNUS: an open-source design space exploration framework for dynamic stream programs. In Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP), Madrid, Spain, October 8-10 2014.

D. J. De Saint Jorre, D. Renzi, S. Casale-Brunet, **M. Wiszniewska**, E. Bezati and M. Mattavelli. MPEG high efficient video coding stream programming and many-cores scalability. In Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP), Madrid, Spain, October 8-10 2014.

Scientific interests

Optimization problems in parallel systems
Digital signal processing and multimedia analysis
Music informatics

Additional

Languages

Polish	(mother-tongue)
English	(fluent in spoken and written)
German	(fluent in spoken and written)
French	(intermediate user: level B1/B2)

Hobbies

Music & Choir singing: past member of *Gdansk University of Technology Choir,*
Berliner Cappella, Chœur Universitaire de Lausanne

Historical literature, cooking

References

Dr. Marco Mattavelli, SCI-STI-MM Group, EPFL
marco.mattavelli@epfl.ch, (+41) 21 69 36984

Prof. Bozena Kostek, Audio Acoustics Laboratory, Gdansk University of Technology, bokostek@audioakustyka.org, (+48) 58 347 27 17