Analysis and optimization of dynamic dataflow programs

THÈSE N^{O} 6663 (2015)

PRÉSENTÉE LE 8 JUIN 2015
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

Simone CASALE BRUNET

acceptée sur proposition du jury:

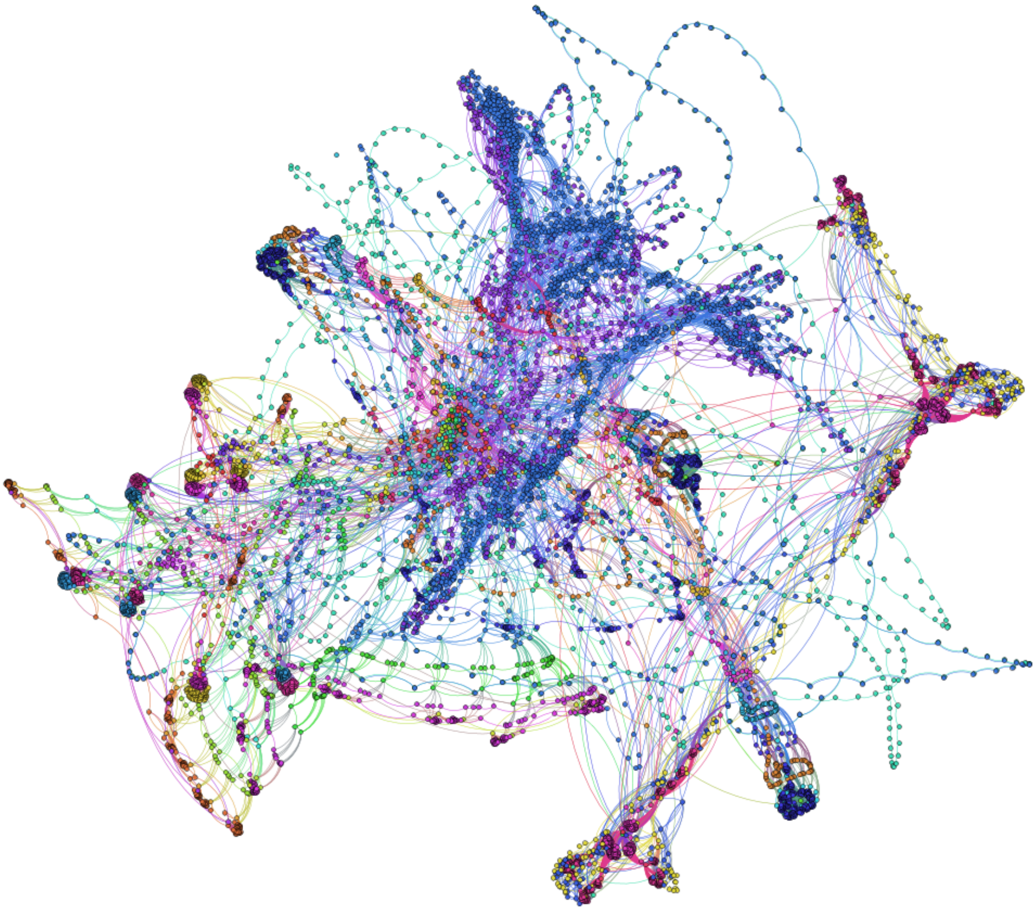
Dr J.-M. Vesin, président du jury
Dr M. Mattavelli, directeur de thèse
Prof. J. Castrillon, rapporteur
Prof. N. Zufferey, rapporteur
Prof. A. P. Burg, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

This thesis is dedicated to the loving memory of my younger brother Edo

Multas per gentes et multa per aequora uectus
Advenio has miseras, frater, ad inferias,
Vt te postremo donarem munere mortis
Et mutam nequiquam adloquerer cinerem,
Quandoquidem fortuna mihi tete abstulit ipsum,
Heu miser indigne frater adempte mihi.
Nunc tamen interea haec, prisco quae more parentum
Tradita sunt tristi munere ad inferias,
Accipe fraterno multum manantia fletu
Atque in perpetuum, frater, aue atque uale.
— Catullus (Carmi, CI. Ad inferias)



Acknowledgements

First of all, I would like to express my deepest sense of gratitude to my supervisor, Dr. Marco Mattavelli, who offered his continuous advice and encouragement throughout the course of this thesis.

I would also like to express my very sincere gratitude to Dr. Jorn W. Janneck, from the Lund University, for his support and systematic guidance to this thesis. Special thanks to Prof. Massimo Canale, from the Politecnico di Torino, who encouraged me to pursue the vision to become a PhD.

The work presented in this thesis was partly supported by the Fonds National Suisse pour la Recherche Scientifique under grant 200021.138214. This support is gratefully acknowledged.

I am thankful to all lab colleagues. A very special thanks to Endri Bezati for his precious friendship and technical assistance to my project. I also take this opportunity to express my gratitude to my friends Christian and Nicoletta, Giacomo, Aurora and Tia, Lucianone, Marco, Martina, Rinaldo and Jenny, Renato and Sandra.

A very special and warm thanks with my profound gratitude to Jessica, who loved and supported me during the writing of my dissertation. She made me feel like everything was possible and incredible. I love you ∞^{∞} Principessinadellafavolapiùbella ♡

I thank my parents, Patrizia and Andrea, who have always given me the strength and wisdom to be sincere in my work, for setting high moral standards and supporting me through their hard work, and for their unselfish love and affection.

Lausanne, 8 May 2015 Simone Casale Brunet

Abstract

All computing platforms, from mobile to supercomputers, are becoming more and more heterogeneous and massively parallel. While they can provide higher power efficiency and computation throughput, effective and confident use of these systems always requires knowledge about low-level programming. The average time necessary to develop and optimize a design on heterogeneous platforms is higher and higher compared to typical homogeneous systems. Dataflow models of computation (MoC) are quickly becoming the common practice in heterogeneous systems development. In domains such as signal processing and multimedia communication, dataflow MoCs have become accepted as standard. However, the shift from a sequential and architecture-specific MoC to a dataflow MoC still uncovers several programming and development challenges. The Cal Actor Language (CAL) is a recently-specified dataflow and actor-based language capable of concisely expressing complex and general purpose parallel applications. However, design tools supporting this language are generally not adequate to fully exploit its features and expressiveness power. In fact, they generally restrict its MoC in order to reduce the design space exploration (DSE) effort. The objective of this thesis is to provide a DSE methodology where all the features of CAL and dynamic dataflow MoCs can be exploited in a more general and effective manner. This dissertation illustrates a novel profiling, analysis and performance estimation methodology for the DSE of dynamic dataflow programs. The main research contributions of this thesis are: the formalization of a graph-based representation of the program execution called an execution trace graph (ETG); the formalization of a systematic methodology for profiling generic dynamic dataflow programs through their code interpretation; the formalization of a complete DSE methodology for dynamic dataflow programs in order to efficiently identify close-to-optimal design points according to various and tailored performance merit functions. In particular, the following design space optimization problems for dynamic dataflow programs are addressed: the analysis of the hotspots and the algorithmic bottlenecks of a parallel program; the bounding and optimization of the buffer size configuration for complex designs; the dynamic power dissipation minimization of programs implemented in multi-clock domain architecture. Furthermore, theoretical concepts like the design space critical path and the potential speedup of a dataflow application have been defined and revisited, respectively. The thesis also presents a DSE framework developed in order to demonstrate the effectiveness of this design methodology.

Key words: dynamic dataflow, design space exploration, heterogeneous computing, CAL

Résumé

De nos jours, des mobiles aux super-ordinateurs, toutes les plates-formes informatiques deviennent de plus en plus hétérogènes et massivement parallèles ce qui les rend très efficaces en termes de puissance et de calcul. Pour obtenir une très bonne utilisation de ces systèmes, il est nécessaire d'avoir toujours plus de connaissances de programmation bas niveau. De plus, le temps moyen nécessaire pour développer et optimiser ce type de système est de plus en plus élevé par rapport aux systèmes typiquement séquentiels. Les modèles de calcul flux de données deviennent rapidement la pratique la plus courante dans le développement des systèmes hétérogènes. Dans des domaines, tels que le traitement du signal et le multimédia, ces modèles de calcul flux de données sont devenus un standard largement accepté. Cependant, le passage d'une méthode séquentielle et spécifique à l'architecture à une méthode flux de données, montre que plusieurs défis de programmation et de développement sont encore à découvrir. Pour répondre à ce passage, un langage de programmation flux de données, récemment spécifié, a été développé. Ce langage, appelé Cal Actor Language (CAL), est capable d'exprimer de manière concise des applications parallèles complexes avec un formalisme simple et générique. Malgré cela, les outils de conception basés sur ce langage ne sont généralement pas suffisants pour exploiter entièrement toutes ses caractéristiques, surtout sa puissance d'expression. En général, les outils actuellement disponibles limitent énormément son modèle de calcul afin de réduire l'effort de l'exploration de l'espace de conception. L'objectif de cette thèse est donc de fournir une méthodologie d'exploration de l'espace de conception où toutes les fonctionnalités du CAL et de son modèle de calcul peuvent être exploitées d'une manière plus générale et plus efficace. Elle démontre aussi une nouvelle méthodologie d'estimation et d'analyse des performances pour les applications flux de données dynamiques. Les principales contributions à la recherche de cette thèse sont: la formalisation d'une représentation de l'exécution du programme basée sur la théorie des graphes et appelée "graphe de trace d'exécution"; la formalisation d'une méthodologie systématique pour le profilage des programmes flux de données dynamiques génériques à travers l'interprétation de haut niveau de leur code source; la formalisation d'une méthodologie complète de l'exploration de l'espace de conception pour des programmes flux de données dynamiques. En outre, les problèmes d'optimisation de l'espace de conception du design pour les programmes flux de données dynamiques abordés sont: l'analyse des goulets d'étranglement algorithmiques d'un programme; la sélection et l'optimisation de la configuration de la taille de mémoire pour des applications complexes; la minimisation de la dissipation de puissance dynamique des programmes mis en œuvre dans une architecture multi-horloges. De plus, les concepts théoriques comme l'espace du chemin critique et l'accélération potentielle d'une application flux de données ont été respectivement définis et revisités. La thèse présente, également, un logiciel d'exploration de l'espace de conception développé afin de démontrer l'efficacité de cette méthode.

Mots clefs: flux de données, exploration de l'espace du design, computation parallèle, plates-formes hétérogènes, CAL

Contents

Acknowledgements i

Abstract iii

Résumé v

List of symbols xiii

List of figures xix

List of tables xxiii

List of listings xxv

List of algorithms xxvii

1 Introduction 1
- 1.1 Heterogeneous systems development . . . 1
  - 1.1.1 Requirements for effective design development . . . 3
  - 1.1.2 Models of computation . . . 3
  - 1.1.3 Design space exploration . . . 4
- 1.2 Motivation of this thesis . . . 5
- 1.3 System development design flow . . . 6
- 1.4 Research contributions of this thesis . . . 8
- 1.5 Thesis organization . . . 9

2 Dataflow programming 11
- 2.1 Dataflow programs . . . 11
  - 2.1.1 Kahn process networks . . . 12
  - 2.1.2 Dataflow process networks . . . 13
  - 2.1.3 Actor transition systems . . . 14
- 2.2 Dataflow paradigm . . . 16
  - 2.2.1 Modular programming . . . 16
  - 2.2.2 Parallelism flavors . . . 16
- 2.3 Dataflow classes . . . 18
  - 2.3.1 Static dataflow programs . . . 18

2.3.2 Cyclo-static dataflow programs 21
2.3.3 Dynamic dataflow programs 22
2.4 Code interpretation and generation 22
2.4.1 Abstract syntax tree 23
2.4.2 Intermediate representation 23
2.4.3 Control flow graph 23
2.5 The Cal Actor Language 24
2.5.1 CAL program 25
2.5.2 Execution model 25
2.5.3 CAL syntax and semantics 26
2.5.4 An example of a CAL program 30
2.5.5 RVC-CAL 32
2.5.6 Compiler infrastructure 32
2.6 Conclusions 33

3 Profiling CAL programs 35
3.1 Actor classification 35
3.2 Static analysis 36
3.2.1 Source lines of code 36
3.2.2 Operators count 37
3.2.3 Cyclomatic complexity 37
3.2.4 Halstead metrics 39
3.3 Data-dependent analysis 40
3.3.1 Computational load 41
3.3.2 Data-transfers and storage load 41
3.4 Conclusions 42

4 Exploring the design space of dataflow programs 43
4.1 Orthogonalization of concerns 43
4.1.1 Model of computation 44
4.1.2 Model of architecture 44
4.1.3 Mapping 45
4.2 The design space of a dataflow program 45
4.2.1 Design space and design points 46
4.2.2 Exploration methods 48
4.2.3 Performance estimation 49
4.3 Related work 49
4.4 Advances in design space exploration of CAL programs 54
4.4.1 Space for improvement 54
4.4.2 New requirements 54
4.5 Conclusions 55

5 Execution trace graph 57
5.1 Geometry of execution . . . 57
5.1.1 Partially-ordered space . . . 57
5.1.2 Execution trace . . . 58
5.1.3 Execution trace space . . . 58
5.2 Execution trace graph . . . 59
5.2.1 Firings . . . 59
5.2.2 Dependencies . . . 60
5.2.3 Example of an execution trace graph . . . 63
5.3 Properties . . . 66
5.3.1 Topological order . . . 66
5.3.2 Mapping independence . . . 66
5.3.3 Untimed . . . 68
5.3.4 Maximum parallelism . . . 68
5.3.5 Data dependent . . . 70
5.3.6 Modeling a dynamic program execution . . . 70
5.4 Timed execution trace graph . . . 74
5.4.1 Firing weight . . . 75
5.4.2 Dependency weight . . . 76
5.5 Transformations . . . 77
5.5.1 Firing expansion . . . 77
5.5.2 Dependency amalgamation . . . 78
5.5.3 Event-driven system representation . . . 78
5.6 Conclusions . . . 83

6 TURNUS: a design space exploration environment for CAL programs 85
6.1 Design flow features and capabilities . . . 85
6.1.1 Profiler . . . 85
6.1.2 Execution trace graph post-mortem scheduling and analysis . . . 86
6.2 High-level models . . . 88
6.2.1 CAL dataflow program . . . 88
6.2.2 Architecture and constraints . . . 95
6.2.3 Execution trace graph . . . 99
6.2.4 Profiling information . . . 101
6.3 Integration with third-party CAL dataflow environments . . . 102
6.4 Conclusions . . . 103

7 Profiling CAL programs with TURNUS 105
7.1 Advances in profiling CAL programs . . . 105
7.2 Data collection . . . 108
7.2.1 Firing data . . . 108
7.2.2 Action data . . . 109
7.2.3 Actor data . . . 110

7.2.4 Buffer data 112
7.2.5 Statistical data 114
7.2.6 Profiled token 114
7.3 Building of the execution trace graph 115
7.4 Application programming interface 116
7.5 Conclusions 117

8 Design space exploration and optimization with TURNUS 119
8.1 Performance estimation 119
8.1.1 Post-mortem scheduler models 120
8.1.2 Execution trace graph post-mortem scheduling 123
8.1.3 Execution statistics 126
8.1.4 Analysis of a collection of execution trace graphs 126
8.2 Design space critical path 126
8.2.1 Critical path length 127
8.2.2 Algorithmic critical path 131
8.2.3 Throughput and design space critical path 132
8.2.4 Potential speedup 132
8.3 Hotspot analysis 135
8.3.1 Critical actions ranking 135
8.3.2 Impact analysis 136
8.4 Buffer size dimensioning 138
8.4.1 Related work 138
8.4.2 Deadlock and feasible regions 139
8.4.3 Minimization by the use of a model predictive control approach 139
8.4.4 Optimization by the exploration of the design space critical path 142
8.5 Dynamic power dissipation minimization 145
8.5.1 Related work 147
8.5.2 Multi-clock domain partitioning 148
8.5.3 Linear programming formulation 148
8.5.4 Heuristic approach 149
8.6 Conclusions 150

9 Experimental results 153
9.1 Design cases 153
9.1.1 JPEG decoder 153
9.1.2 MPEG4-SP decoder 154
9.1.3 MPEG-HEVC decoder 154
9.2 CAL source code static and dynamic profiling 154
9.2.1 Source code static analysis 155
9.2.2 Memory requirements and utilization 156
9.2.3 Execution trace graph 157
9.2.4 Initial design-refactoring directions 157

9.3 Design refactoring 161
9.3.1 Critical action ranking 162
9.3.2 Impact analysis 163
9.4 Bounded buffer size configuration 164
9.5 Buffer size optimization 167
9.6 Dynamic power dissipation minimization 169
9.7 Conclusions 171

10 Conclusions 173
10.1 Future work 176

A Discrete event system and simulation 179
A.1 Petri nets 179
A.1.1 State 180
A.1.2 Transition firing 180
A.2 Discrete event system specification 181
A.2.1 The atomic model 181

B Model predictive control 183

C A CAL esoteric example 185
C.1 A Chef chocolate cake 185
C.2 From a sequential to a dataflow program specification 187
C.3 The first CAL chocolate cake 188
C.4 A dynamic refrigerator 191
C.5 Design space exploration of a kitchen 193

Bibliography 211

Curriculum Vitae

List of Symbols

Acronyms

AAM	Algorithm architecture adequation matching
ACP	Algorithmic critical path
API	Application programming interface
ASIC	Application-specific integrated circuit
ATM	Actor transition system
CAL	Cal Actor Language
CIF	Common interchange format
CP	Critical path
CPL	Critical path length
CSDF	Cyclo-static dataflow program
DAG	Directed acyclic graph
DCT	Discrete cosine transform
DDF	Dynamic dataflow program
DPN	Dataflow process network
DSCP	Design space critical path
DSE	Design space exploration
ETG	Execution trace graph
FID	Firing identifier
FIFO	First in, first out
FNL	Functional unit network language

FPGA	Field-programmable gate array
GALS	Globally asynchronous locally synchronous
HEVC	High efficiency video coding
HW	Hardware
IDCT	Inverse DCT
ILP	Integer linear programming
IQ	Inverse quantization
JPEG	Joint photographic experts group
KPN	Kahn process network
LP	Linear programming
LTS	Labeled transition systems
LUB	Least upper bound
MCD	Multiple-clock domain
MoA	Model of architecture
MoC	Model of computation
MPC	Model predictive control
MPEG	Moving picture experts group
MPSoC	Multiprocessor system on chip
NPM	Native programming model
Orcc	Open RVC-CAL compiler infrastructure
PAPI	Performance application programming interface
PAPS	Periodic admissible parallel schedule
PASS	Periodic admissible sequential schedule
PiMM	Parameterized and interfaced dataflow meta-model
PN	Petri net
QCIF	Quarter CIF
RMC	Reconfigurable media coding

RTL	Register-transfer level
RVC	Reconfigurable video coding
S-LAM	System-level architecture model
SDF	Static or synchronous dataflow program
SoC	System on chip
SP	Simple profile
SW	Software
TETG	Timed execution trace graph
UML	Unified modeling language
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit
VLSI	Very large scale integration
XDF	XML dataflow format
XML	Extensible markup language

Operators

$[.]'$	The matrix transpose operator
$\mathrm{argmax}(.)$	The argument of the maximum operator
$\mathrm{argmin}(.)$	The argument of the minimum operator
$\bullet$	The amalgamation operator
$\lvert . \rvert$	The cardinality operator
$\lVert . \rVert_n$	The n-norm operator
$\lvert \underset{.}{\rightarrow} \rvert$	The directed path length operator
$\max(.)$	The maximum value operator
$\min(.)$	The minimum value operator
$\oplus$	The concatenation operator
$\underset{.}{\rightarrow}$	The directed path operator
$E[.]$	The expected value operator

$Var(.)$	The variance operator

Variables

$(X, \leq)$	A partially-ordered space, also called po-space
(X, dX)	A directed topological space, also called d-space
$\beta \in C_\beta$	A buffer size configuration
η	Action execution index
$\kappa \in K$	The CAL actor-classes
$\lambda \in \Lambda$	The CAL action
Λ	The set of CAL actions
$\Lambda_{CP} \subseteq \Lambda$	The set of actions that have at least one action firing along the critical path
$\lvert\overrightarrow{CP}\rvert$	The critical path length
$\widehat{\mathrm{T}}$	The estimated design throughput
$\mathrm{S}(n)$	The speedup of a program when executed in n processing elements
T	The design throughput
μ	Dependency kind
$\overrightarrow{1} = [0, 1]$	The closed and directed unit interval
$\overrightarrow{CP}$	The critical path
$\overrightarrow{CP}_{algo}$	The algorithmic critical path
$\overrightarrow{p}$	A directed path, also called d-path
$\perp$	The empty sequence
$\rho \in C_\rho$	A partitioning configuration
$\sigma \in C_\sigma$	A scheduling configuration
Θ	The set clock-accurate profiling information
$a \in A$	The CAL actor
A	The set of CAL actors
$A_{CP} \subseteq A$	The set of actors that have at least one action firing along the critical path
B	The set of buffers of a dataflow program

$b_i \in B$	The i-th buffer of a dataflow program
C_β	The set of buffer size configurations
C_ρ	The set of partitioning configurations
C_σ	The set of scheduling configurations
D	The ETG dependencies set
d	Dependency direction
$D^\bullet$	The set of amalgamated dependencies
$D_c \subseteq D$	The critical dependencies set
$D_f \subseteq D$	The finite state machine dependencies set
$D_g \subseteq D$	The guard dependencies set
$D_p \subseteq D$	The port dependencies set
$D_t \subseteq D$	The tokens dependencies set
$D_v \subseteq D$	The internal variables dependencies set
$D_{CP} \subseteq D_c$	The dependencies set along the critical path
$E(X, dX)$	The execution trace space
$e_n^\bullet \in D^\bullet$	The n-th amalgamated dependency
$e_n = (s_i, s_j) \in D$	The n-th dependency of the ETG, with s_i and s_j source and target firings
$G(PU, ME, L)$	The platform architecture model
$G(V, E)$	A generic graph with V and E the sets of vertexes and edges
K	The set of CAL actor-classes
$K_{CP} \subseteq K$	The set of actor-classes that have at least one action firing along the critical path
L	The set of links available on the architecture
$l_i \in L$	The i-th link (i.e. interconnection between a processing element and a medium)
M	The set of mapping point
ME	The set of media of an architecture
$me_i \in ME$	The i-th medium
P	The set of Petri net places

$p_i \in P$	The i-th of Petri place
$P_{\emptyset^-}$	The set of fictive Petri net places
PU	The set of processing elements of an architecture
$pu_i \in PU$	The i-th processing element
S	The ETG action firings set
$S_c \subseteq S$	The critical action firings set
$s_i \in S$	The i-th firing of the ETG
$S_{CP} \subseteq S_c$	The action firings set along the critical path
T	The set of Petri net transitions
$t_i \in T$	The i-th of Petri transition
$w(s_i)$	Execution time (or time weight) required by action firing s_i
$w(s_i, s_j)$	Execution time (or time weight) required by dependency (s_i, s_j)

List of Figures

1.1 Simplified typical design flow of a heterogeneous hardware and software system. 2

1.2 Heterogeneous system development design flow for CAL dataflow programs. . 7

2.1 Pipeline parallelism. 17

2.2 Task parallelism. 17

2.3 Data parallelism. 18

2.4 Dataflow MoCs classes. 18

2.5 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

2.6 Dataflow graph example. 20

2.7 Code compiler and interpreter flowcharts. 23

2.8 CAL network and actors structure. 24

2.9 Action execution model according to Equation 5.10. 26

2.10 Basic dataflow program. 30

2.11 The RVC-CAL compiler and Xronos infrastructure integrated in the design flow presented in Figure 1.2. 33

4.1 Mapping from an application to an architecture. Constraints represent the feasible regions of the design space. 45

4.2 The design space $M = C_\rho \times C_\sigma \times C_\beta = \{m_1, m_2, \ldots, m_{n_M}\}$ and the corresponding performance $\mathrm{T}(m)$ and estimated performance $\widehat{\mathrm{T}}(m)$. 46

4.3 Platform independent simulation of the CAL network depicted in Fig. 2.10 with the mapping configurations described in Table 4.1. The execution of each action is supposed to take at least one (abstract) clock cycle (when there are no blocking output buffers), the overhead introduced by the action selection and buffer access overheads are both neglected. In gray the actor execution with the corresponding action firing. In striped-gray the actor execution is postponed due to the unavailability of a token (i.e. blocking reading). 47

5.1 Execution space in $\mathbb{R}^2$ of two actors `A` and `B` mapped on two processing units pu_1 and pu_2, respectively. The dashed arrow represents a possible execution path of the program. 58

5.2 Execution trace graph obtained after the execution of the CAL program described in Section 2.5.4. The firing set S is summarized in Table 2.2, and the dependencies set D is summarized in Table 5.2. 64
5.3 Execution Trace Graphs of the CAL network depicted in Fig. 2.10. Dashed lines represent additional edges that model a particular scheduling configuration defined within the mapping configurations described in Table 4.1. 69
5.4 Two possible execution paths of the `GuardedInverter` actor illustrated in Listing 5.1. The corresponding execution trace graphs do not take into account the guard enable and disable dependencies. 72
5.5 Guard enable and disable dependencies couples that model the guard enable windows $n = 1$ and $n = 2$ depicted in Figure 5.4. The firing s_b represents a generic firing of the action `B`. 74
5.6 The ETGs related to the execution paths depicted in Figure 5.4a and Figure 5.4b where for each firing of `B` a couple of guard enable and disable has been considered in order to model the guard enabled windows $n = 1$ and $n = 2$. . . . 75
5.7 Firings expansion of an execution trace graph. 78
5.8 Amalgamation of the execution trace graph illustrated in Figure 5.2. 79
5.9 Petri net obtained from the execution trace graph depicted in Figure 5.2. 82

6.1 TURNUS design flow. 87
6.2 The `Network` object. 89
6.3 The `ActorClass` object. 90
6.4 The `Actor` object. 90
6.5 The `Action` object. 91
6.6 The `Quid` object. 91
6.7 The `Procedure` object. 92
6.8 The `Variable` object. 92
6.9 The `Guard` object. 93
6.10 The `Port` object. 93
6.11 The `Buffer` object. 94
6.12 The `Type` object. 94
6.13 The `Version` object. 95
6.14 Xilinx Zynq-7 ZC702 evaluation-board architecture model. 96
6.15 The `Platform` object. 96
6.16 The `ProcessingElement` object. 97
6.17 The `Medium` object. 98
6.18 The `Link` object. 98
6.19 The `Scheduler` object. 99
6.20 The `Trace` object. 99
6.21 The `Firing` object. 100
6.22 The `Dependency` object. 101
6.23 The `NetworkProfilingData` and `ActionProfilingData` objects. . . 102

6.24 The open RVC-CAL compiler (Orcc) and Xronos infrastructure integrated in the TURNUS design flow. 104

7.1 CAL profilers design flow. 106
7.2 The `FiringData` object. 109
7.3 The `ActionData` object. 110
7.4 The `ActorData` object. 112
7.5 The `ActorTracingData` object. 113
7.6 The `BufferData` object. 113
7.7 The `Statistics` object. 114
7.8 The `Token` object. 114

8.1 Execution trace graph post-mortem scheduler: simulation models. 121
8.2 Sequence diagram for the DEVS atomic implementation of an actor. 124
8.3 Design space critical path. 133
8.4 Critical path length linear model $|CP|(\rho_n)$. 134
8.5 Theoretical speedup $S(n)$ defined in Equation (8.27) for different values of $h = |CP|_{algo}/w \in [0,1]$ when $n_A = 10$. 135
8.6 Example of impact analysis for three actions λ_1, λ_2 and λ_3. 138
8.7 Critical path design space given different buffer size configurations. 139
8.8 Bounded buffer scheduling with deadlock avoidance approach. 143
8.9 Bounded buffer scheduling with deadlock recovery approach. 144

9.1 JPEG decoder. 154
9.2 MPEG4-SP decoder. 154
9.3 HEVC decoder. 155
9.4 The rendering of a small portion (i.e. approximatively 80000 action firings and 350000 dependencies) of the execution trace graph described in Table 9.4. Action firings are colored according to the corresponding actor. 159
9.5 Refactoring strategies for the `Inter-Prediction` actor. 161
9.6 Impact analysis for the initial version of the Shared-Memory MPEG-HEVC decoder. 164
9.7 Buffer size optimization of the MPEG4-SP decoder implemented on an ST Microelectronics STHorm platform. 168

C.1 A dataflow representation of the `Hello World Cake with Chocolate sauce` Chef program illustrated in Listing C.1. 187
C.2 The `Liquify` CAL actor defined in Listing C.4. 191
C.3 The modified version of the `ChocolateSauce` CAL actor. 191

List of Tables

2.1 CAL lexical tokens. 27

2.2 Firing of the CAL program described in Section 2.5.4. 31

3.1 Profiled executed operators and statements. 38

4.1 Mapping configurations for the dataflow network illustrated in Figure 2.10. For brevity, the actors Producer, Filter and Consumer are denoted with P, F, C, respectively. The partitioning of the buffers is not considered. 48

5.1 Dependencies kinds, directions, parameters and additional attributes. 63

5.2 Dependencies set S of the execution trace graph depicted in Figure 5.2. 65

5.3 Firings sequence of the CAL actor `Split` defined in Listing 2.2 when two input sequences are available in its input port `I`: $I_1 = \{0, 1, -10, -5\}$ and $I_2 = \{-1, -1, 0, -1\}$, respectively. 70

5.4 Firings with the corresponding internal variable and guard values for the execution trajectories and graphs depicted in Figure 5.4. 73

5.5 Firing weight parameters for the linear model of Equation 5.10. 76

7.1 CAL profilers features. 107

9.1 Static code complexity of the MPEG-HEVC decoder. 156

9.2 Actor memory requirements for the initial HEVC design. 156

9.3 Buffer utilization profiling data of the MPEG-HEVC decoder. 157

9.4 Execution trace graph configuration of the Ref-Standard MPEG-HEVC decoder. 158

9.5 Memory requirement for the actor internal variables of the Shared-Memory MPEG-HEVC decoder. 160

9.6 Execution trace graph configuration of the Shared-Memory MPEG-HEVC decoder. 160

9.7 Critical action ranking analysis of the MPEG-HEVC decoder. Results are for the initial and the full-parallel version, summarized for 5 different actions in Table 9.7a and 9.7b respectively. 162

9.8 Description of different configurations of the HEVC decoder design and corresponding speedup, computational complexity and critical path length values. . 163

9.9 Design sizes: numbers of actors, buffers and action firings. 165

9.10 Bounded buffer size configurations of the JPEG decoder using the MPC approach. Results are compared to state of the art approaches. 165
9.11 Bounded buffer size configurations of the MPEG-HEVC decoder using the MPC approach. Results are compared to state of the art approaches. 166
9.12 2-Clock domains dynamic power minimization results of the MPEG-4 SP decoder implemented in a Xilinx Virtex-5 FPGA. Nominal: all the domains use the maximum available frequency; Optimized: with the clock frequencies illustrated in Table 9.12a. Δ% defines the percentage reduction between the nominal and optimized case of each contribution. 169
9.13 4-Clock domains dynamic power minimization results of the MPEG-4 SP decoder implemented in a Xilinx Virtex-5 FPGA. Nominal: all the domains use the maximum available frequency; Optimized: with the clock frequencies illustrated in Table 9.13a. Δ% defines the percentage reduction between the nominal and optimized case of each contribution. 170

List of Listings

2.1 Inverter.cal 28
2.2 Split.cal 28
2.3 ParametrizedProducer.cal 29
2.4 PingPongMerge.cal 29
2.5 BiasedMerge.cal 30
2.6 TokenConsumer.cal 30
2.7 BasicNetwork.nl 31
2.8 BasicNetwork.xdf 31
5.1 GuardedInverter.cal 71
C.1 Cake.chef 186
C.2 Cake.cal 189
C.3 ChocolateSauce.cal 190
C.4 Liquify.cal 191
C.5 ModifiedChocolateSauce.cal 192
C.6 Refrigerate.cal 192
C.7 Refrigerate.cal 193

List of Algorithms

1 Compute the set of parameters $ES(s_i), EF(s_i), LS(s_i), LF(s_i)$ for each $s_i \in S$. 129
2 Compute the slack value $SL(s_i)$ for each $s_i \in S$ and $SL(s_i, s_j)$, and the set of critical firings set S_c and critical dependencies set D_c. 130
3 Critical path extraction. 130
4 Impact analysis for the set of critical actions Λ_{CP}. 137
5 Critical path length reduction by increasing the size of critical buffers. 146
6 Heuristic algorithm for solving the problem of the multi-clock domain partitioning defined in Equation (8.45). 150

1 Introduction

This thesis addresses the problem of analyzing complex design applications modeled with emerging dataflow programming languages. In the last decades, there has been a great deal of activity and advancement in the field of dataflow programming languages. The motivations of such interest are related to the fact that the increasing demand of computing power can be coped difficultly only by the improvement of device technology. Nowadays, the availability of heterogeneous parallel platforms, that combine the processing features of FPGAs with multi-core CPUs, offer in a single silicon die a potential amount of computing power that exceeds by far what was available in the past years. However, the programming experience of these platforms becomes more and more complex. Consequently, designers have to implement increasingly complex applications for increasingly complex and networked platforms. The potential power of those platforms can only be exploited if existing design flows are able to support the new heterogeneous architectures. Designs capable of efficiently exploiting the architecture characteristics must encompass both hardware and software design concepts, which are currently expressed by using completely different abstractions. This thesis defines a complete design flow, supported by a software tool environment, such that the designer can be efficiently and easily guided during the entire application development process.

1.1 Heterogeneous systems development

All computing platforms, from mobile to supercomputers, are becoming more and more heterogeneous and massively parallel. In a time when new hardware meant higher clock frequencies, old programs almost always ran faster on more modern equipment. However, this is not the case anymore when programs written for single-core systems will have to execute, as an example, on multi-core platforms at possibly lower clock speeds on low-power platforms. While these heterogeneous and massively parallel platforms can provide higher power efficiency and computational throughput, their effective and confident use always requires knowledge about low-level programming. Hence, the average time necessary to develop and optimize a design on heterogeneous platforms is higher and higher compared to typical homogeneous systems. A common practice is to choose "a-priori" partition of the design. Each part of the design is specifically developed for the assigned computing element. This typical design flow is depicted in Figure 1.1 which starts with a behavioral description of the application. This description is generally made using a plethora of different programming language. Application parts that are implemented in hardware (HW) are generally specified using parallel languages (e.g. VHDL, Verilog), and application parts that are implemented in software (SW) are generally specified using sequential languages (e.g. C/C++, CUDA, OpenCL) that sometimes make use of parallel pragmas (e.g. OpenMP) specified by the designer. This initial choice can affect all the development stages. In fact, if the design requirements and constraints are not satisfied, then the design should be optimized. If the modification requires that the partitioning configuration should be changed, then part of the (or the entire) design should be rebuilt from scratch. This can be frustrating for the designer, but it also increases the time-to-market of the application. As a consequence, this typical design flow cannot be considered as an adequate and productive methodology for the design development on heterogeneous platforms.

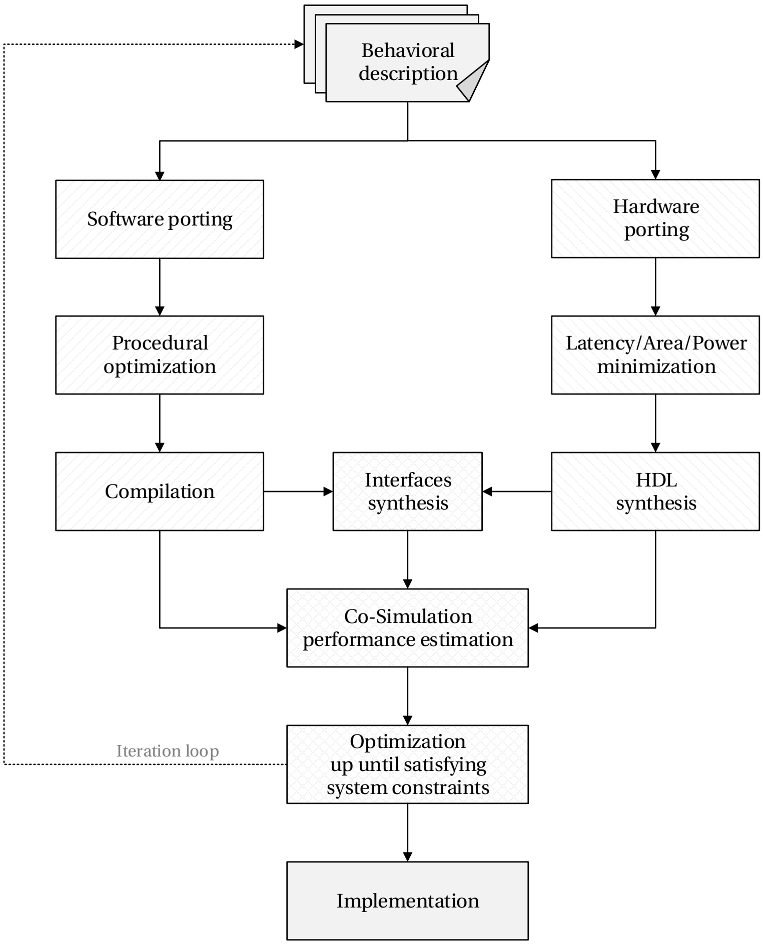


Figure 1.1: Simplified typical design flow of a heterogeneous hardware and software system.

1.1.1 Requirements for effective design development

The main requirements for effective design development on heterogeneous platforms can be summarized in terms of:

- **Design abstraction**: one of the most important questions that a designer faces during the early stage of the development is which level of abstraction should be used. The response is not always trivial by the diverse nature of platforms. Different degrees of abstraction may be employed depending on the amount of details needed to describe the requirements.
- **Modularity**: design abstraction should make opportunities for a more flexible and modular implementation. The functionalities of a program should be separated into independent and interchangeable modules, where each module contains everything necessary to execute only a specific functionality.
- **Composability**: design abstraction should make opportunities to provide recombinant components implementation. These components should be selected and assembled in various combinations in order to satisfy specific design requirements.
- **Reuse**: design abstraction and the modularity of an application should make opportunities for the reuse of program components. As an example, several audio codecs share part of the same functional units, which makes modularity a necessity.

These requirements are essential for an unified computation abstraction for HW and SW that requires the application programming with a model of computation that is modular and, at the same time, abstracts out platform specific details.

1.1.2 Models of computation

One of the main obstacles that may prevent the widespread usage of heterogeneous parallel platforms is the fact that serial models of computations (MoC) and programming methods are still adopted. The vast majority of existing software is written in sequential form. However, efficient parallel implementations are challenging and arduous to achieve using sequential MoCs. In fact, sequential languages are notoriously difficult to parallelize in general, so efficient parallel implementations will usually require significant guidance from the user. Consequently, serious problems are arising for porting existing technologies and applications on the new performing heterogeneous and massively parallel platforms. The understandability, predictability, and determinism properties of purely-sequential MoCs remain the crucial requirement for parallel MoCs. Hence, a shift to a new programming paradigm that exploits the parallelism and diversification interested in heterogeneous systems development is clearly becoming a necessity. In application areas characterized by the use of highly parallel computing platforms, the use of a **dataflow** MoC to describe the algorithms creates opportunities for more flexible implementation and also for more extensive analysis. One of the reasons for this is that a dataflow program describes an algorithm as a, possibly hierarchical, network of communicating computational kernels, also called **actors**. Actors are connected by directed, lossless, order-preserving point-to-point channels. This makes the flow of data explicit between actors, which are not permitted to share data in any other way than by sending each other messages, called **tokens**. Furthermore, this MoC also exposes the application-internal parallelism between actors. As actors are forbidden to share state, implementation tools have a great range of freedom in mapping dataflow programs to hardware and software implementations, or mixtures thereof. Dataflow programs are being analyzed in different ways for different purposes. The subclass of statically schedulable programs, also called static dataflow programs, is amenable to pure compile-time analysis that yields not only a static schedule, but also things such as exact minimal bounds for buffer sizes, exact predictions for throughput and latency, and a guarantee of the absence of deadlocks and so forth. However, for many complex applications (e.g. signal processing), it is not possible to represent all of the functionality in terms of a purely statical schedulable program. Functionality that involves conditional execution or dynamically varying token production and consumption rates can only be directly expressed through a **dynamic dataflow** representation. Intuitively, in dynamic dataflow programs the production and consumption rates of actors can vary in different ways that are not entirely predictable at compile time. As a consequence, compile-time analysis may provide inconclusive results.

1.1.3 Design space exploration

Design space exploration (DSE) refers to the activity of evaluating and exploring the different design alternatives during the system development of an application. For large and complex designs, implemented in heterogeneous and massively parallel platforms, the number of design alternatives easily becomes too big and error-prone for a manual and efficient exploration. For this reason, several DSE methodologies have been investigated in the last decades. Each DSE methodology generally makes use of the following **functionalities**:

- **Rapid prototyping**: DSE is used to generate a set of prototypes prior to implementation. Validating and testing the design before its final implementation may reduce the cost and the time required for solving problems that can arise in the late production cycle of an application. Furthermore, it can increase understanding of the impact of design decisions during the implementation process.
- **Optimization**: even though validation is an important part of the design process, it is possible that the application does not satisfy the requirements. Consequently, feasible design configuration should be explored in order to meet the requirements. If any of those exist, the design should be modified. Consequently, clear and precise refactoring directions should be provided to the designer.
- **System integration**: when heterogeneous platforms are used as target architecture of

the application, the system integration can become one of the most tedious and error-prone stages of the development. System integration requires a working assembly and configuration of multiple components. DSE can be used to find feasible assemblies configurations that satisfy the design constraints and requirements.

Therefore, a designer must have a formal method, supported with a computer-aided framework for finding a feasible set of design alternatives, also referred to as **design points**, that meet the specification requirements. However, general and structured methodologies are lacking for designing application-specific architectures that are sufficiently modular and programmable. In fact, the current practice is to design application-specific architectures at a detailed level. The level of detail involved limits the number of design points that can be explored effectively. As a consequence, this may limit the freedom to make trade-offs between programmability, resource utilization and achievable performances. For this reason, a generic and DSE environment should encompass the following main **components**:

- **Application model**: a suitable representation of the design space is essential. This should be: formal (automated analysis and exploration techniques can be performed), general (the application should be platform-independent and retargetable), expressive (constraints imposed by the target platform can be captured and enforced).
- **Exploration and analysis**: the environment should provide a collection of computer-aided techniques for discovering potential design configuration candidates. Moreover, the framework should be able to tackle the challenge of solving, in a reasonable time frame, a large number of complex design constraints. As far as the user is concerned, the framework must provide a method for navigating through the set of interesting and distinctive solutions.
- **Performance estimation**: the environment should be able to estimate the application performance and directly test the different candidate solutions. As far as the user is concerned, testing the different configurations one by one without the possibility of estimating the performance can be an error-prone procedure since this may require several partial implementations of the application. Furthermore, good estimation also mean that the DSE analysis provides reliable results.

1.2 Motivation of this thesis

Dataflow MoCs are a promising practice in heterogeneous systems development. It has already been demonstrated how they can be efficiently used to support the portability by itself and the portability of the parallelism of an application. In domains such as signal processing and multimedia communication, where the scalability is also growing in interest as a fundamental requirement, dataflow MoCs have already become an accepted standard. However, the shift from a sequential and architecture-specific MoC to a dataflow MoC still uncovers several programming and development challenges. The Cal Actor Language (CAL) is a recently specified dataflow and actor-based language capable of concisely expressing complex and general purpose parallel applications. A subset of this language has also been standardized within the MPEG Reconfigurable Video Coding framework (RVC) where it is used to specify the standard Video Tool Library (VTL). However, design tools that support this language are generally not adequate to fully exploit its features and capabilities. Current design methodologies, where this language is used, severely restrict its MoC in order to reduce the DSE effort. The objective of this thesis is to provide a DSE methodology where all the features of CAL, and dynamic dataflow MoC, more generally, are completely exploited.

1.3 System development design flow

In this thesis the system development design flow and methodology illustrated in Figure 1.2 is used. The program functional behavior is taken separately from the architecture model. Program behavior is expressed using the CAL dataflow language, which is based on dataflow processing network principles. The architecture, together with its constraints, are modeled with a high-level abstraction used to describe the platform where the design is implemented. The architecture model is based on the notion of processing elements, media and links. A processing element defines the kind of platform, a medium defines the way that this platform is communicating, and a link defines a connection between processing elements and media. Constraints are applied within the architecture and the program and are used to define, for example, the maximal clocking frequency of each operator. The six different stages of this design flow are respectively:

- **Compiler infrastructure**: transforms the source code of a CAL program to an equivalent intermediate or representation. The compiler should provide the possibility of verifying the program behavioral correctness directly from the intermediate representation, without requiring any partial implementation or prototyping.

- **Profiling and analysis**: the design alternatives of the application are explored such that constraints and performance requirements can be satisfied. The design can also be statically or dynamically analyzed in order to evaluate its computational and communication costs. In the case where a design point satisfies the requirements, this is then used to drive the compiler infrastructure through a set of *compiler directives*. Otherwise, in the case where requirements cannot be satisfied, *refactoring directions* should be provided to the designer. These highlight which part of the design requires modification to allow requirement satisfaction.

- **Performance estimation**: performance of a given design point is evaluated without requiring any partial implementation of the program: only the high-level models of both the program and architecture are used. Results of the estimation are analyzed in order to reduce the design points that can satisfy requirements.

- **Code generation**: the CAL program is transformed to a low-level code representation. Software and/or hardware code is generated according to the mapping of the program to the target architecture.

- **Synthesis or compilation**: the software or hardware code is compiled or synthesized, respectively. Standard tools are used in order to obtain the software executables and the hardware binary files and netlists of the implementation.

- **Implementation**: when both the performances and constraints are satisfied, the design is implemented in the hardware or/and software architecture. If the implementation contains both hardware and software parts, then interfaces provided by the architecture should be automatically integrated into the design.

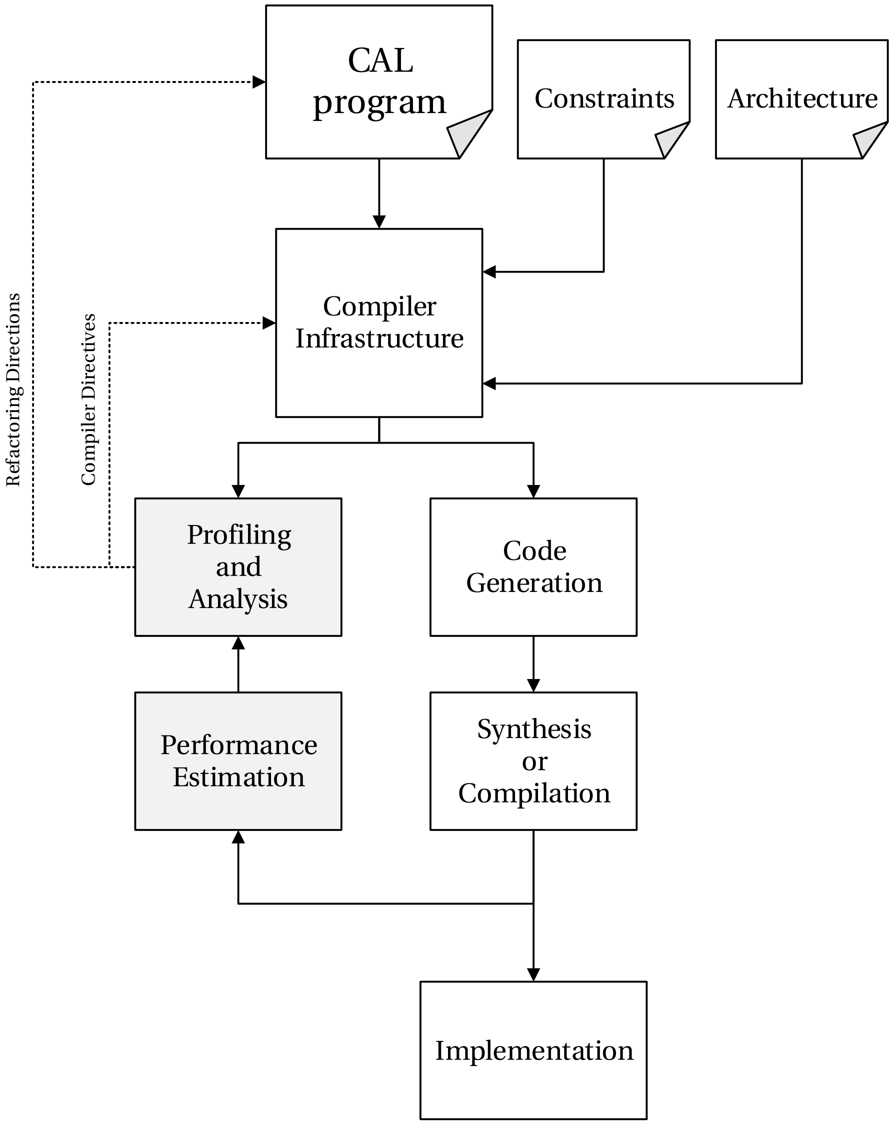


Figure 1.2: Heterogeneous system development design flow for CAL dataflow programs.

1.4 Research contributions of this thesis

This dissertation focuses on the profiling and analysis, and the performance estimation stages of a CAL program development design flow illustrated in Section 1.3. Those stages represent together the DSE of a CAL program. In this context, the main contributions of this thesis are:

(i) **Execution Trace Graph** [1, 2, 3, 4]: a graph-based representation of the program execution is formalized. This mathematical formalism can be used to model the execution of static, cyclo-static and dynamic dataflow programs. A collection of analysis and transformations are illustrated. As an example, it is demonstrated how it is possible to estimate the design performance by scheduling the execution trace graph post-mortem, or how this representation can be transformed to an event-driven system where advanced control technique methods can be used to explore the design points of the application.

(ii) **Profiling of dynamic dataflow programs** [1, 2]: a systematical methodology for profiling generic dynamic dataflow programs is formalized. This is based on the code interpretation, which does not require any partial implementation of the program. It is demonstrated how this methodology can be effectively used to extract the execution trace graph through a serial code interpretation.

(iii) **Design space exploration methodology**: a collection of heuristic methods, based on the analysis of the execution trace graph, is formalized for exploring the different design points of a generic dynamic dataflow program. In particular, the following problems have been addressed and solved:

  - **Design space critical path formalization** [1, 5]: the concept of design space critical path is formalized and used to bound the design points of an application. Furthermore, with this notion, the concepts of potential speedup, defined in the well known Amdahl's law, have also been revisited and adapted to the domain of dataflow programs.
  - **Hotspots analysis** [1, 5, 6, 7, 8, 9]: a methodology for highlighting the bottlenecks of the program and providing clear code refactoring directions is formalized.
  - **Buffer size configuration dimensioning** [1, 10, 11, 12]: a systematic methodology for solving the problem of bounding and optimizing the buffer size configuration of complex dynamic dataflow programs has been formalized and solved.
  - **Dynamic power dissipation minimization** [13, 14, 8, 15]: a systematic methodology for reducing the dynamic power dissipation of complex dynamic dataflow programs, implemented in multi-clock domain architecture, has been formalized and solved.

(iv) **Design space exploration environment** [16, 17, 18, 19, 20]: a computer-aided framework for exploring the design space of dynamic dataflow program has been implemented. This framework, called TURNUS, has been released as an open-source project that has already been integrated with other open-source CAL HW synthesis and SW code generation tools (i.e. called Xronos and Orcc, respectively). Its integration with these tools provides a complete systems design environment for CAL applications. The TURNUS's main functionalities and structure are discussed in this dissertation and used to prove the effectiveness of the illustrated design methodology and its heuristics algorithms.

1.5 Thesis organization

This dissertation is organized as follows:

- **Chapter 2** provides an overview of the main concepts of dataflow programming. An overview concerning the taxonomy classification and the different models of computation that can be defined is presented. General discussion about static, cyclo-static and dynamic dataflow programs are presented. Furthermore, an introduction to the CAL actor language is presented with a collection of examples.

- **Chapter 3** summarizes the possible profiling options of a dataflow program. Two main profiling axes are illustrated: computational load and memory utilization. Furthermore, a discussion concerning static and dynamic analysis of the code is presented. A focus is given on profiling the CAL actor language and how well-known profiling metrics, such as the Cyclomatic complexity and the Halstead metrics, are used in the context of this language.

- **Chapter 4** defines the design space of an application. It summarizes the concept of orthogonalization of concerns. In this direction, it is illustrated how the application can be modeled with a high-level of abstraction by defining the model of computation and the model of architecture. Furthermore, the concept of mapping configuration is defined together with the definition of design points. Different design space exploration strategies are presented. In this chapter the space for improvement on the context of CAL application exploration and optimization, that this thesis tries to gap, are also discussed.

- **Chapter 5** defines the concept of execution trace graph of a dataflow program. The main properties of this graph-based representation are illustrated and used to provide a formal definition of the design space of an application. Furthermore, how a dynamic dataflow program can be handled is demonstrated by the use of guard enable and disable dependencies concepts.

- **Chapter 6** illustrates the main functionalities and the design flow of the TURNUS dataflow exploration framework. This represents the tool implementation of the heuristic proposed throughout this thesis. The main software engineering structure, together with the formal design methodology, are presented.

- **Chapter 7** focuses on the TURNUS dataflow profile functionalities. The main advancement and improvement, compared to state of the art tools, are presented.
- **Chapter 8** focuses on the TURNUS design exploration and optimization functionalities. Design performance is estimated by the use of a post-mortem execution trace scheduler. It is illustrated how estimation results are used in order to guide the optimization heuristic during the exploration phases. Furthermore, the concept of design space critical path is defined and used as a primary metric of the optimization heuristics that are illustrated in this chapter. These are: the hotspots analysis, the buffer size dimensioning (i.e. minimization and optimization) and the partitioning (i.e. with a particular focus on the problem of minimizing the dynamic power dissipation on reconfigurable platforms).
- **Chapter 9** presents a collection of experimental results for video codec applications. More precisely, results obtained during the different stages of the design space exploration of video and image decoders (i.e. such as a MPEG4-SP, HEVC and JPEG) are presented and discussed.
- **Chapter 10** concludes the dissertation, highlighting possible future works and illustrating the open problems that this thesis has not yet solved.

Furthermore, some additional in-depth material is available in the appendixes of this dissertation. This additional material represents a quick reference guide for the reader. Appendixes are structured as follows:

- **Appendix A** illustrates the main concepts of discrete event systems and simulation. The formal definition of a Petri net is introduced. This is used in Section 5.5.3 when transforming the execution trace graph to an event-driven system is formalized. Furthermore, the formalism behind the concept of discrete event system specification and simulation is presented.
- **Appendix B** illustrates the main concepts and functionalities of the model predictive control. This receding horizon control technique is used in Section 8.4.3 where the problem of buffer size dimensioning is solved.

2 Dataflow programming

Stream processing is a widely used term in literature to describe a variety of systems. In fact, streaming applications are programs that process continuous data streams. These applications have become ubiquitous due to increased automation in signal and video processing, telecommunications, health care, transportation, retail, science, security, emergency response and finance. As a result, various research communities have independently developed programming models for streaming applications. While there are differences both at the language level and at the system level, each of these communities ultimately represent streaming applications as a graph of streams and operators, generally called **dataflow programs**. This chapter provides an overview about dataflow programming. Starting from the definition of dataflow program, different models of computations are illustrated. Successively, an overview about the Cal Actor Language is presented, as this language is used as a reference dataflow programming language in the remaining chapters of this dissertation.

2.1 Dataflow programs

In the context of this dissertation, a dataflow program is defined as directed graphs whose vertices are operators, called **actors**, and whose edges are streams. In general, stream graphs might be cyclic, though some systems only support acyclic graph. Dataflow programs implement streams as FIFO (first-in, first-out) queues, called **buffers**, sometimes with limited capacity, sometimes not. Conceptually, streams are infinite sequence of atomic data items, called **tokens**, and each actor consumes data items from incoming streams and produces data items on outgoing streams. The token is the atomic unit of communication in a dataflow program. One of the main properties of dataflow programs is that they have a **data-driven semantic**: it is the availability of tokens that enables an actor. One of the principal strengths of dataflow programs is that they do not over-specify an algorithm by imposing unnecessary sequencing constraints between actors. Instead, they only specify a partial order, where sequencing constraints are imposed only by data dependence and, since actors can run concurrently, dataflow programs inherently expose the application parallelism [21, 22].

In the following parts of this section, an overview of the different dataflow models of computations (MoCs) used within this dissertation is presented. These are: the Kahn Process Networks [23] that represent the underpinning representation for dataflow graphs, the Dataflow Process Networks [24] that are closely related to the Kahn Process Networks, and the Actor transition system [25] that extends Dataflow Process Networks with the notions of atomic step, priority and actor internal variables.

2.1.1 Kahn process networks

A Kahn process network (KPN) [23] is a network of **processes** that can communicate only through **unidirectional and unbounded buffers**. Each buffer carries a possible infinite sequence of tokens. Using the notation formalized in [24], each token's sequence is denoted with $X = [x_1, x_2, x_3, \ldots]$ where each x_i represents a token drawn from some set. A token is considered as an atomic data object that is written (produced) exactly once and read (consumed) exactly once. **Writes** to the buffers are **non-blocking**, in the sense that they always succeed immediately. **Reads** from buffers are **blocking**, in the sense that if a process attempts to read a token from a buffer and no data is available, then it stalls (waits) until the buffer has sufficient tokens to satisfy the read. Consequently, it is not possible to test the presence of input tokens.

Kahn process

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

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

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 (or is equal to) Y. For example, if $X = [x_1, x_2]$ and $Y = [x_1, x_2, x_3]$ then $X \sqsubseteq Y$ and it is common to say that X approximates Y, since it provides partial information about Y. The empty sequence, denoted with $\perp$, is a prefix of any other sequence.

The 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** [26] 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. A process F is said to be **monotonic** if:

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

Remark. *Monotonicity can be thought of as a form of causality that does not invoke time, in that "future input concerns only future output".*

A continuous process is monotonic. However, a monotonic process may be not continuous. A key consequence of this property is that a process can be computed iteratively [27]. This means that given a prefix of the final input sequences, it may be possible to compute 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 completion of an infinite input sequence). Networks of monotonic processes are **determinate**.

2.1.2 Dataflow process networks

Dataflow process networks (DPN) [24] formally establish a special case of KPNs, where the computational blocks are called **actors**. As for the KPN process, actors can communicate only through unidirectional and unbounded buffers which can carry possible infinite sequences of tokens. As for KPN, **writes** to buffers are **non-blocking**. On the contrary, **reads** from buffers are **non-blocking**, in the sense that an actor can test 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 suspended when it cannot read. This could introduce **non-determinism**, without requiring the actor to be non-determinate.

Actor with firings

DPN networks are a special case of KPN where each process consists of repeated **firings** of an actor [28]. An actor firing can be defined as an indivisible (atomic) quantum of computation. The firings themselves can be described as functions, and the invocation of these firings is controlled by some firing rules. Sequences of firings define a continuous Kahn process as the least-fixed-point of an appropriately constructed functional mapping, therefore formally establishing DPN as a special case of KPN [29].

An actor with m inputs and n output 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 an LUB.

The Kahn process F defined in Equation (2.1) 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 $(S^m \rightarrow S^n)$ the set of functional mapping S^m to S^n. It is possible to demonstrate that ϕ is both a continuous and monotonic function. The firing function f need not be continuous. In fact, it does not even need to be monotonic. It merely needs to be a function, and its value must be finite for each of the firing rules [29].

2.1.3 Actor transition systems

Actor transition systems (ATS) [25] describe actors in terms of labeled transition systems (LTS). The ATS extends the notion of 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. Moreover, the notion of priority allows actors to ascertain and react to the absence of tokens. This notion can make actors harder to be analyzed, and it may introduce unwanted non-determinism into a dataflow application.

Remark. *The state of an actor depends upon the value (state) of its internal variables, and not just on the sequence of tokens it has received.*

Let Σ denote the non-empty actor state space, u the universe 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 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 firing, in which tokens may be consumed and produced, and the internal variables may be updated.

Enabled transition and step of an actor

Intuitively, the priority relation determines that a transition cannot occur if some other transition is possible. This can be seen as the definition of a valid step of an actor, which is a transition such that two conditions are satisfied:

- The required input tokens must be present.
- There must not be another transition that has priority.

Given an n-to-m actor $(\sigma_0, \tau, \succ)$, a state $\sigma \in \Sigma$ and an input tuple $v \in S^n$, a transition $\sigma \xrightarrow{s \rightarrow s'} \sigma'$ is **enabled** if and only if:

$$\begin{cases} v \sqsubseteq s \\ \nexists \sigma \xrightarrow{r \rightarrow r'} \sigma'' \in \tau \ : \ r \sqsubseteq v \land \sigma \xrightarrow{s \rightarrow s'} \sigma' \succ \sigma \xrightarrow{r \rightarrow r'} \sigma'' \end{cases} \tag{2.6}$$

Hence, a **step** from state σ with input v is defined as any enabled transition $\sigma \xrightarrow{s \rightarrow s'} \sigma'$.

Actors composition

For any transition relation τ its set of **input ports** P_τ^{in} and its set of **output ports** P_τ^{out} are defined as the ports in which at least one transition consumes input from or produces output to:

$$\begin{cases} P_\tau^{in} = \{p \in P \mid \exists \sigma \xrightarrow{s \rightarrow s'} \sigma' \in \tau \ : \ \sigma(p) \neq \perp\} \\ P_\tau^{out} = \{p \in P \mid \exists \sigma \xrightarrow{s \rightarrow s'} \sigma' \in \tau \ : \ \sigma'(p) \neq \perp\} \end{cases} \tag{2.7}$$

where P is the set of input and output ports names. It is assumed that an input port with name p and an output port of the same name are in no way related. In order to express complex functionality, actors are composed into a **dataflow network**. As an example, Figure 2.6 depicts a dataflow network composed of five actors interconnected with five buffers. The structure of a network can be represented by a partial function from (input) ports to (output) ports, mapping each input port in its domain to the output port that connects to it. It must be noted that, this assumption implies the absence of fan-in (as every input port is connected to at most one output port), and it permits unconnected (open) input (and output) ports.

2.2 Dataflow paradigm

The emergence of massively parallel architectures, along with the difficulties to program these architectures, makes dataflow paradigm a more appealing alternative to an imperative paradigm [22, 30, 31, 32, 33, 34, 35]. 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. As a matter of 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 [36, 37]. Moreover, this paradigm explicitly exposes all the natural parallelism of a program [36, 22].

2.2.1 Modular programming

The decomposition of the program into processing blocks improves its maintainability by enforcing the encapsulation of the components. Such a decomposition naturally makes the program description modular. The main capabilities of a modular description are:

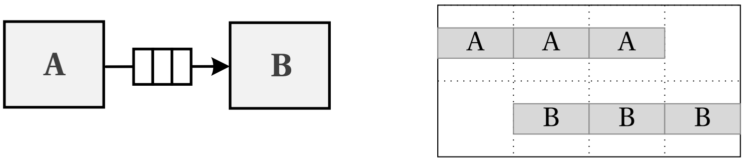
- **Reusability**: a single processing block can be used multiple times in the same dataflow network.
- **Reconfigurability**: a processing block can be easily replaced by another one when their input and output ports are (strictly) identical.
- **Hierarchical representation**: a processing block of the dataflow network may represent another dataflow network.

In the rest of this dissertation, a processing block that represents a hierarchical composition of processing blocks is referred to as a sub-network, otherwise it is simply referred to as an actor.

2.2.2 Parallelism flavors

It is worth summarizing the specific terminology for the various kinds of parallelism flavors among actors of a dataflow program. These are:

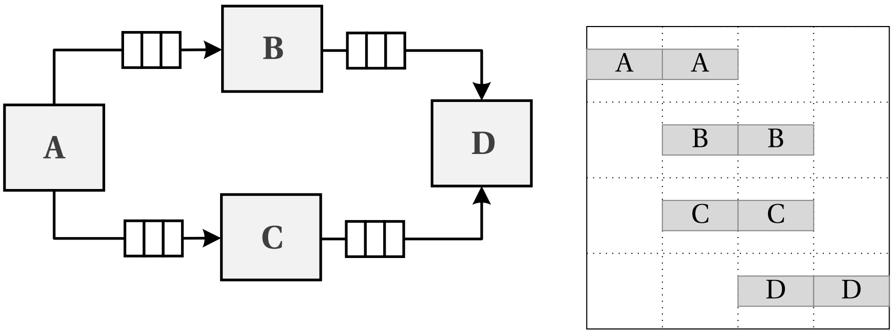
- **Pipeline parallelism** is inherent to a streaming execution model in case of a chain of actors. Pipelining does not enhance the throughput on one calculation, but the processing of a sequence of calculations. As an example, Figure 2.1 depicts the concurrent execution of a producer actor A with a consumer actor B. This parallelism flavor can be considered as a mixture of task and data parallelisms. Pipelining represents the separation of a computation in several stages that can be executed in parallel.



(a) Dataflow network (b) Parallel execution

Figure 2.1: Pipeline parallelism.

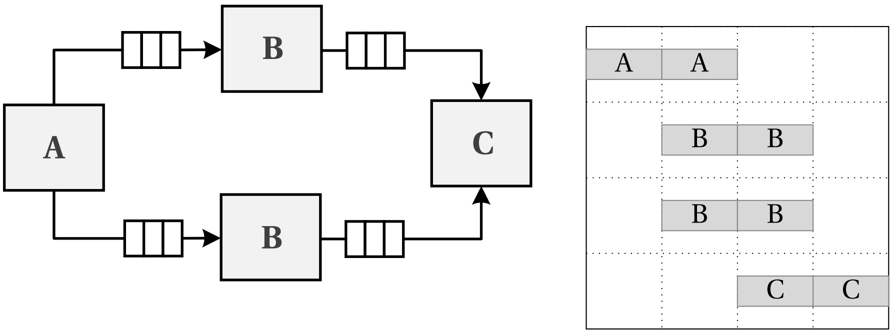
- **Task parallelism** refers to the parallelism between independent parts of an application. In a dataflow context, it appears when two or more actors do not have any dependency constraints. As an example, Figure 2.2 depicts the concurrent execution of different actors B and C, respectively, that do not constitute a pipeline.



(a) Dataflow network (b) Parallel execution

Figure 2.2: Task parallelism.

- **Data parallelism** refers to a unique computation performed on different sets of data. It can be applied by duplicating an actor when it processes several sets of data successively with no dependencies between them. Data parallelism is also sometimes characterized as SPMD (single program, multiple data). As an example, Figure 2.3 depicts the concurrent execution of multiple replicas of the same actor B on different portions of the same data.



(a) Dataflow network (b) Parallel execution

Figure 2.3: Data parallelism.

2.3 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 different parallelism kinds, to maximize the re-use or simply reduce the limited hardware resources available on the implementation platform. Figure 2.4 illustrates the three main dataflow MoC classes. The respective actor behavior that can be represented for each of them is discussed in this section.

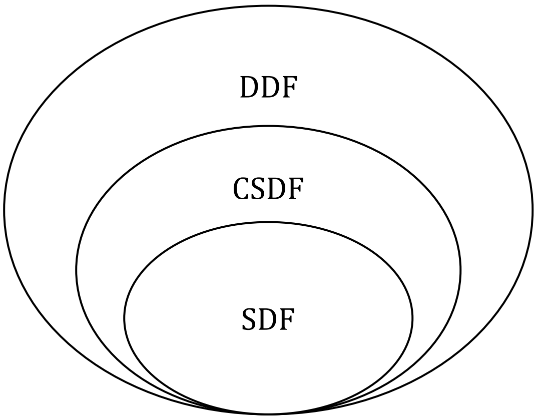


Figure 2.4: Dataflow MoCs classes.

2.3.1 Static dataflow programs

Static dataflow (SDF), sometimes also referenced as synchronous dataflow, is a special class of dataflow MoC where the number of tokens consumed and produced by each actor is fixed and known at compile time. Repeated firing of the same actor respects the same behavior. This is the less expressive class of dataflow programs, but it is also the one that can be more easily analyzed. In fact, its main property is its total **compile time predictability**, with respect to scheduling, memory consumption, and execution termination.

Static scheduling

In order to build a statical schedule, the compiler should construct a single cycle of a periodic schedule. The first step is then evaluating how many invocations of each actor should be included in each cycle. This can be easily obtained using the number of produced and consumed tokens for each actor firing. As depicted in Figure 2.5, 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:

$$p_{i,n}\, r_i = c_{j,n}\, r_j \tag{2.8}$$

In other words these equations ensure 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 in finding a schedule for an SDF graph is to solve a set of Equation (2.8) for the unknowns r_i.

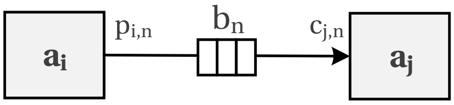


Figure 2.5: A dataflow graph with two actors, a_i and a_j, connected through the buffer b_n. $p_{i,n}$ defines the number of tokens produced on b_n during each firing of a_i. $c_{j,n}$ defines the number of tokens consumed from b_n during each firing of a_j.

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 the $i-th$ actor produces $p_{i,n}$ tokens on the $n-th$ buffer, and it contains 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, the dataflow graph shown in Figure 2.6 has the following topological matrix:

$$\Gamma = \begin{bmatrix} p_{A,1} & -c_{B,1} & 0 & 0 & 0 \\ p_{A,4} & 0 & 0 & -c_{D,4} & 0 \\ 0 & p_{B,2} & -c_{C,2} & 0 & 0 \\ 0 & 0 & p_{C,3} & 0 & -c_{E,3} \\ 0 & 0 & 0 & p_{D,5} & -c_{E,5} \end{bmatrix} \tag{2.9}$$

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

$$\Gamma \overrightarrow{r} = \overrightarrow{0} \tag{2.10}$$

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 (2.10) is usually referred to as the **balance equation** of the dataflow program.

Remark. *If an actor has a connection to itself (i.e. a self-loop) then 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 invoked. This difference needs to be zero for a correctly constructed graph. Hence, the entry describing a self-loop should be zero [38].*

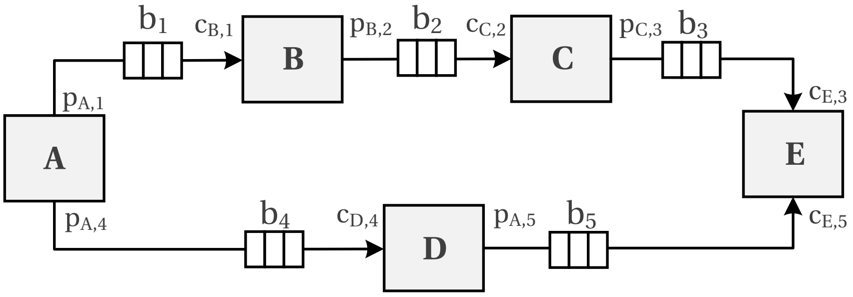


Figure 2.6: Dataflow graph example.

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 a periodic and infinite admissible sequential schedule. In [38] 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:

$$rank(\Gamma) = s - 1 \tag{2.11}$$

where s is the number of actors in the graph. In other terms, the null space of Γ should have dimension one. It is shown in [38] that when the rank is correct, there always exists a repetition vector $\overrightarrow{r}$ that contains only integers and relies in this null space. 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 in consistency** in 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 workable schedule for a single processing unit can be generated, then a workable schedule for a multi-processing units system can also be generated. The objective is then 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. For single processing unit targets, some reasonable scheduling objectives might include minimization of data or program memory requirements. For multi-processing unit targets, minimizing the throughput or maximizing flow-time are more likely objectives [38, 39, 40].

2.3.2 Cyclo-static dataflow programs

Cyclo-Static Dataflow (CSDF) generalizes the SDF MoC by defining cyclically-changing firing rules. 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 in 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 $\overrightarrow{\gamma}_{i,n}$. Because these patterns are periodic and predictable, it is still possible to statically construct periodic schedules using techniques based on those developed for SDF. State can be represented as an additional argument to the firing rules and firing function: in other words, it is modeled as a self-loop [41, 42].

Static scheduling

The topological matrix entries are defined such as:

$$[\Gamma]_{i,j} = t_{i,j}\ \frac{\sigma_{i,j}}{d_{i,j}} \tag{2.12}$$

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 $t_j = lcm(d_{i,j}, \forall i)$, 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 done for SDF, it is also possible for the CSDF programs to solve the balance equation (2.10) 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 $r_i\ t_i$.

2.3.3 Dynamic dataflow programs

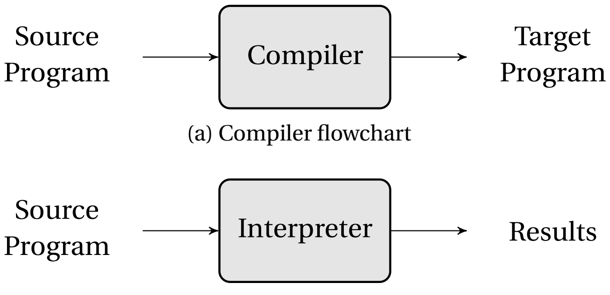
Although SDF and CSDF are adequate models for representing parts of many algorithms, they are rarely sufficient for expressing entire complex programs since they are not adequate to express data-dependent iterations, conditionals and recursion. For example, functionality that involves conditional execution of dataflow subsystems or actors with dynamically-varying production and consumption rates cannot be expressed in decidable dataflow models [43, 44]. 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 rate can vary according to the program input sequence.

Analysis techniques

The increased modeling flexibility and expressiveness power make DDF programs much harder to be analyzed. Due to their Turing-complete nature, many analysis problems may become undecidable [43]. 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 the context signal processing systems) [1, 11, 12]. 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 parts of it can be described in a more restricted way [45, 46, 47, 48]. A systematical and effective analysis methodology for DDF programs is illustrated in the following chapters of this dissertation.

2.4 Code interpretation and generation

The portability support of dataflow program onto different HW and SW platforms is provided by a compiler infrastructure capable to generate low-level from the high-level program representation at a system level. As illustrated in Figure 1.2, the compiler infrastructure is an essential part for enabling an effective DSE exploration and implementation of a dataflow program. In this section, the basic components of a dataflow compiler infrastructure are illustrated. These are extensively used in the rest of this dissertation when the profiling of a dataflow program is presented. Interpreters and compilers have much in common [49]. As illustrated in Figure 2.7, both have the source code of the input program as input. Moreover, both analyze and validate the input program and build an internal (i.e. intermediate) representation of it. However, the main difference is that a compiler generates a stand-alone machine code program, while an interpreter performs the actions described by the high-level input program description.



(a) Compiler flowchart

(b) Interpreter flowchart

Figure 2.7: Code compiler and interpreter flowcharts.

2.4.1 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 is not representing 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 correctness, the AST serves as the base for code generation. The AST is used to generate the intermediate representation for the code generation or interpretation.

2.4.2 Intermediate representation

Intermediate representation (IR) is a representation of a program partway between the input source and output target code. A well-structured IR is one that does not depend on both the input source code and the target architecture, so that it maximizes its ability to be re-used in a retargetable compiler.

2.4.3 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 the input program [50]. 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 flow of control. 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.

2.5 The Cal Actor Language

The Cal Actor Language (CAL) [51] is a domain-specific language that provides useful abstractions for dataflow programming with actors. CAL directly captures the features of ATS actors adding the notion of atomic action firings, also called *steps*. Figure 2.8 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**. CAL also includes the possibility of defining an explicit **finite state machine** (FSM). The FSM captures the actor state behaviour and drives the action selection according to its particular state, to the presence of input tokens and to the value of the tokens evaluated by other language operators called **guard functions**. 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. In the rest of this section, a basic overview is presented of the main concepts concerning the syntax, the semantics and the different MoC that can be represented with this language.

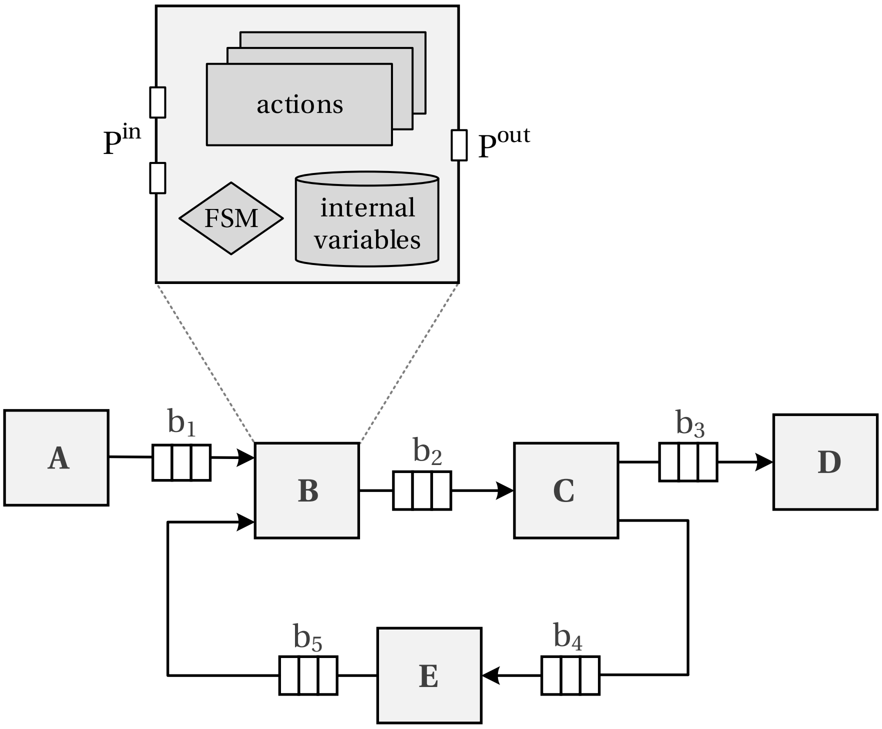


Figure 2.8: CAL network and actors structure.

2.5.1 CAL program

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

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

A CAL **actor-class** κ defines the program-code-template and the implementation behaviors of the actor (i.e. the CAL source code). Different actors can instantiate 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 $(\kappa, P^{in}, P^{out}, \Lambda, \mathcal{V}, \mathrm{FSM})$ 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 contrary, there are no limitations on how many buffers can be connected to an output port.

2.5.2 Execution model

For the purpose of this thesis, it is assumed that the firing of an action is performed by following the serial execution of the stages summarized in Figure 2.9. These are:

- **Wait for tokens** Q_{br}: the firing is waiting that 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 that all the required output tokens can be accommodated in the corresponding buffers.
- **Write output tokens** Q_w: the firing is producing the output tokens.

where the transition conditions are the following:

- **hasTokens**: the number of required input tokens is available from each corresponding input buffer.
- **hasSpace**: the number of output tokens space is available on each corresponding output buffer.

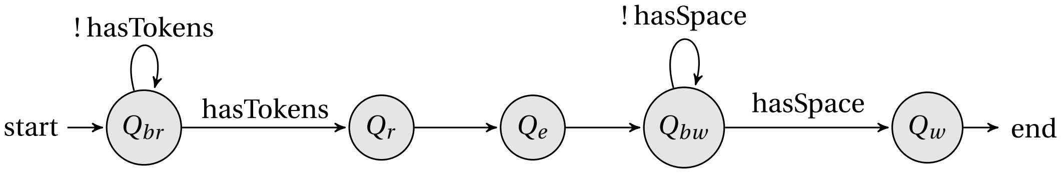


Figure 2.9: Action execution model according to Equation 5.10.

2.5.3 CAL syntax and semantics

In the following section, an overview concerning CAL is provided. The syntax and the semantic of this dataflow program are illustrated through simple but effective examples. The interested reader can refer to [51].

Lexical tokens

Lexical tokens help the user to understand the functionality provided by any language. A lexical token is a string of indivisible characters known as lexemes. The CAL lexical tokens, also summarized in Table 2.1, are described in the following:

- **Keywords** Keywords are a special type of identifiers. They are already reserved in the programming language by default. 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** Operators usually represent mathematical, logical or algebraic operations. Operators are written as any string of characters `!`, `%`, `^`, `&`, `*`, `/`, `+`, `-`, `=`, `<`, `>`, `?`, `~` and `|`.

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

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

Table 2.1: CAL lexical tokens.

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

Actions, input patterns and output patterns

The simplest actor that can be described using CAL is the `Inverter` actor defined in Listing 2.1. This actor consumes a token from its input port and produces a token on its output port. The actor header is defined in line 1, which contains the actor name, followed by a list of parameters contained inside the `()` construct (empty, in this case), and the declaration of the input and output ports. The input ports are those in front of the ==> construct and the output ports are those after it. In this case the input and output port sets are defined as $P^{in}_{\texttt{Inverter}} = \{\texttt{I}\}$ and $P^{out}_{\texttt{Inverter}} = \{\texttt{O}\}$ respectively. For each parameter and port, the data type is specified before the name (all defined with an `int` data type, in this case). This actor contains only one *action*, labeled as `invert` as defined in line 3. In this case, the action set is defined as $\lambda_{\texttt{Inverter}} = \{\texttt{invert}\}$. Action `invert` demonstrates how to specify token consumption and production. The part in front of the ==>, which defines the **input patterns**, specifies how many tokens to consume, from which ports, and what to call those tokens 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 the token is to be called `val` in the rest of the action. Such an input pattern also defines a condition that must be met for this action to fire: if the required token is not present, this action will not be executed. Therefore, input patterns do 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 by which tokens consumed by an action firing will be referred to within the action.

- They define a firing condition for the action, i.e. a condition that must be met for the action to be able to fire.

The **output patterns** of an action are those defined after the `==>` construct. They simply 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:[-v]` says that exactly one token will be generated at output port `O`, and its value is `-v`. It is worth noting that although syntactically the use of `v` in the input pattern `I:[a]` looks the same as the one in the output expression `O:[-v]`, their meanings are very different. In the input pattern the name `v` 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 `v` in the output expression uses that name.

Listing 2.1: Inverter.cal

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

    invert: action I:[val] ==> O:[-val] end

end
```

Guards

So far, the only firing condition for actions was that there be sufficient tokens for them to consume, as specified in their input patterns. However, in many cases, it is possible to specify additional criteria that need to be satisfied for an action to fire. Conditions, for instance, 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 `Split` 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`. Those actions require the availability of one token in `I`, however their selection is guarded by the value of the input token `val` read from `I`, and respectively defined in line 4 and line 7. In this example, if `val >= 0` then action `A` is selected, otherwise action `B` is selected.

Listing 2.2: Split.cal

```
1  actor Split() 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. These can be used when the same actor definition is used more then once in the same program definition. For example, the `ParametrizedProducer` actor, defined in Listing 2.3 uses the parameter `maxCounter`. This parameter, defined in line 1, is used as a guard condition by the (only) action `produce` as defined in line 7. This actor also defines the internal variable `counter` that is used and updated during each firing of the action as described in line 9.

Listing 2.3: ParametrizedProducer.cal

```
1   actor ParametrizedProducer(int maxCounter) ==> int O :
2
3       int counter := 0;
4
5       produce: action ==> O:[counter]
6       guard
7           counter < maxCounter
8       do
9           counter := counter + 1;
10      end
11
12  end
```

Priorities and State Machines

In the `PingPongMerge` actor, reported in Listing 2.4, a finite state machine schedule is used to force the action sequence to alternate between the two actions `A` and `B`. The schedule statement introduces two states `s1` and `s2`. On the contrary, in the `BiasedMerge` actor, reported in Listing 2.5, the selection of which action to fire is not only determined by the availability of tokens, but also depends on the priority statement.

Listing 2.4: PingPongMerge.cal

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

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

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

    schedule fsm s1:
        s1(A) --> B;
        s2(B) --> A;
    end

end
```

Listing 2.5: BiasedMerge.cal

```
actor BiasedMerge() 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
```

2.5.4 An example of a CAL program

In CAL it is possible to define a network of interconnected actors. Figure 2.10 depicts a CAL program composed by three actors `Producer`, `Filter` and `Consumer`, and by two buffers b_1 and b_2. Two different representation approaches are supported for defining the CAL network structure: the first one is based on a functional programming language called Functional unit Network Language (FNL), the second one is based on eXtensible Markup Language (XML) known as XML Dataflow Format (XDF).

As an example, the XDF and FNL network representations illustrated in Listings 2.8 and 2.7, respectively, both define a CAL program where the `Producer` actor instantiates the `ParametrizedProducer` actor-class defined in Listing 2.3, the `Filter` actor instantiates the `Inverter` actor-class defined in Listing 2.1, and the `Consumer` actor instantiates the `TokenConsumer` actor-class defined in Listing 2.6. It must be noted that, in this particular example, the `Producer` actor instantiates its actor-class using the parameter `maxCounter=3`. Supposing executing 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), the corresponding action firings are those summarized in Table 2.2.

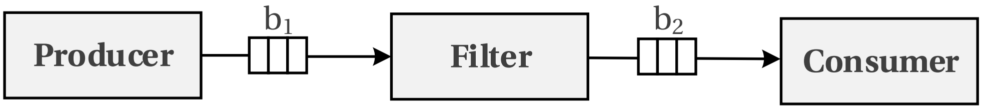


Figure 2.10: Basic dataflow program.

Listing 2.6: TokenConsumer.cal

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

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

end
```

Listing 2.7: BasicNetwork.nl

```
network BasicNetwork () ==> :

entities

    Producer = ParametrizedProducer(maxCounter = 3);
    Filter   = Inverter();
    Consumer = TokenConsumer();

structure

    Producer.O --> Filter.I
    Filter.O   --> Consumer.I

end
```

Listing 2.8: BasicNetwork.xdf

```
<?xml version="1.0" encoding="UTF-8"?>
<xdf name="BasicNetwork">
  <instance id="Producer">
    <class name="ParametrizedProducer"/>
    <parameter name="maxCounter">
      <expr kind="literal" literal-kind="integer" value="3"/>
    </parameter>
  </instance>
  <instance id="Filter">
    <class name="Inverter"/>
  </instance>
  <instance id="Consumer">
    <class name="TokenConsumer"/>
  </instance>
  <connection src="Producer" src-port="O" dst="Filter"   dst-port="I"/>
  <connection src="Filter"   src-port="O" dst="Consumer" dst-port="I"/>
</xdf>
```

Table 2.2: Firing of the CAL program described in Section 2.5.4.

Firing	Actor	Actor-class	Action
s_1 s_2 s_3	`Producer`	`ParametrizedProducer`	`produce`
s_4 s_5 s_6	`Filter`	`Inverter`	`invert`
s_7 s_8 s_9	`Consumer`	`TokenConsumer`	`consume`

2.5.5 RVC-CAL

CAL language has been expressly 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 the 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 [52, 53, 54, 55]. 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 under 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).

2.5.6 Compiler infrastructure

The RVC-CAL compiler infrastructure used in the context of this thesis is summarized in Figure 2.11. This is called open RVC-CAL compiler infrastructure (Orcc) [56, 57, 58]. 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 [59] such as the Eclipse modeling framework (EMF) [60, 61], Xtext [62] and Xtend [63]. The Orcc compilation flow can be summarized as follows:

- **Front-end**: the RVC-CAL code is parsed and translated into an Abstract Syntax Tree (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 allows the possibility of 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 optimization (i.e. IR to IR transformations) are performed before the code 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).

For the purposes of this dissertation, the framework taken in charge of generating RTL descriptions from a CAL code representation is Xronos [8, 64, 65]. This is the evolution of work presented in [66, 67] and it is fully integrated into the Orcc environment.

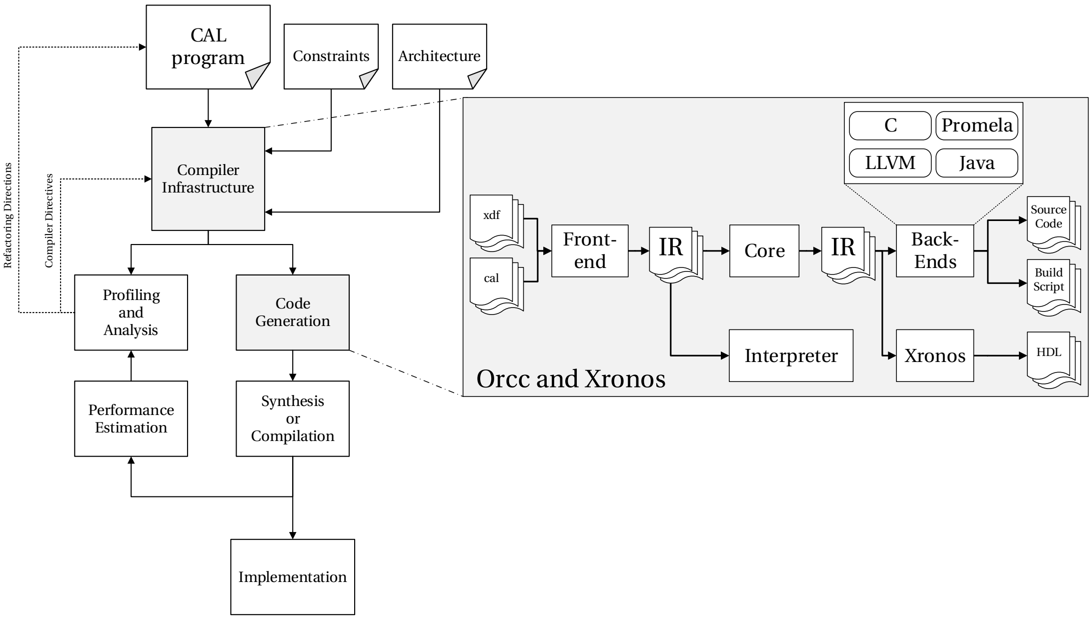


Figure 2.11: The RVC-CAL compiler and Xronos infrastructure integrated in the design flow presented in Figure 1.2.

2.6 Conclusions

In this chapter the notion of dataflow programming has been illustrated. Three different classes of dataflow graphs have been investigated. Those are notably the Kahn process network (KPN), the dataflow process network (DPN) and the actor transition system (ATS). For each one of these classes the main mathematical formalization has been provided and discussed. The notion of monotonicity has been introduced and used to illustrate the main analysis problematics that can arise when an operator (or actor) is not monotonic. Successively, the main features of modularity and the different parallelism flavors exposed by the dataflow MoC has been illustrated. Successively, the discussion has covered the taxonomic classifications of dataflow programs. The main properties of static (SDF), cyclo-static (CSDF) and dynamic dataflow (DDF) programs has been illustrated. It has been shown why the analysis of dynamic dataflow programs is considered a challenging task. Finally, the Cal Actor Language (CAL) has been introduced. Concepts like actor-class, actor, action, procedure, internal variable, ports, guards, internal state machine and priority have been illustrated through a collection of source code examples. Furthermore, the RVC-CAL standardized subset and its compiler infrastructure has been illustrated. It has been shown how starting from a CAL source code representation of the program it is possible to generate a low-level code representation suitable for implementing the program both in software and in hardware.

3 Profiling CAL programs

An appropriate complexity analysis stage is a fundamental step for any methodology aiming at the implementation of today's complex applications [68, 69]. Such a stage may have different final implementation goals such as defining a new architecture dedicated to a specific application under study, defining an optimal instruction set for a selected processor architecture, or guiding the software optimization process in terms of control-flow and dataflow optimization targeting a specific architecture. In this context, the term **complexity** is intended in a broader and more intuitive sense than its strict mathematical definition only considering the size of the minimal algorithm descriptions. More precisely, the various aspects and results of the run-time algorithm complexity metrics are investigated. Such metric results can hardly be evaluated from the algorithm code itself, because of its size or because the program behavior is data-dependent and therefore a more sophisticated analysis methodology should be used. In the following chapter, some methods to classify the behavior of an actor and successively different static and run-time analyses are illustrated.

3.1 Actor classification

Actor classification determines the behavior of a given actor in terms of production/consumption of tokens, patterns that may govern token exchanges, and possible acceptable token values. The final goal of this analysis is the detection of the class of each actor composing the network. Restricted dataflow classes represent different trade-offs between algorithmic expressiveness and execution predictability (see Section 2.3). In the simplest case, structural information of an actor is sufficient for the classification (e.g. the rules for an actor to be considered SDF only depend on the input and output patterns of actions). However, in more general cases, it is necessary to gather information from an actual execution of the actor [70, 71, 72, 73].

Using the set of dataflow classes illustrated in Section 2.3, it is possible to classify dynamic actors into a restricted dataflow class as follows:

- **Static behavior**: the classification tries to classify each actor within classes that are increasingly expressive and complex. The rationale behind this is that the more expressive (powerful) a class is, the more difficult it is to analyze. If an actor cannot be classified as a static actor, the method will try to classify it as CSDF. An actor is classified as static if and only if it conforms to the SDF class, which means that all its actions have the same input and output patterns. A one-action actor is by definition static.

- **Cyclo-static behavior**: an actor has to meet two conditions to be a candidate for CSDF classification: it must have a state and there must be a fixed number of data-independent firings that depart from the initial state, modify the state, and return the actor to its original state. Once the actor was identified as a valid CSDF candidate, abstract interpretation can be used to determine the sequence of actions characterizing its behavior, as well as its production and consumption rates [71, 72].

- **Dynamic behavior**: if not classified as SDF or CSDF, the actor is defined as DDF.

After being classified, the actors, as well as the network they compose, may be subject to additional analysis and optimizations that require the respect of more restricted dataflow classes.

3.2 Static analysis

The methods based on a static analysis of the source code range from simply counting the number of operations up to defining dependencies among the basic blocks. This information can be used during different optimization stages. For example, the lower and upper run-time of a given program on a given processing element can be directly evaluated from the operator count analysis [74, 75]. While this simple counting technique provides a very accurate evaluation of the operations, it cannot handle loops, recursion, conditional statements and data-dependent applications except for some particular cases. Explicit or implicit enumeration of program paths can handle loops and conditional statements and can yield bounds on best and worst case run-time [74, 75, 50]. The main drawback of these techniques is that the typical real processing complexity of many algorithms heavily depends on the input data statistics while static analysis can only detect upper and lower bounds. Restricted programming styles such as absence of dynamic data structures, recursion, and bounded loops are required in order to correctly perform a static analysis [74].

3.2.1 Source lines of code

Source lines of code (SLOC) is one of the most-used metric when dealing with program development complexity and maintainability. Using the definition proposed in [76], a *line of code* is a line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements in the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. However, the SLOC of a program can be strongly dependent on how the counting procedure is interpreted. For this reason, the number of lines of code should be used only as a crude complexity measure [77].

3.2.2 Operators count

As for the SLOC metric, the occurrence of each operator can be used as a crude complexity measure of the program. Table 3.1 reports the set of unary, binary, data handling and flow control operators available for the CAL language. However, basing the program complexity on the number of operator occurrences can be misleading as conditional blocks (e.g. `if` and `while`) are taken into account only once.

3.2.3 Cyclomatic complexity

The cyclomatic complexity analysis [78] is a quantitative measure of the complexity of programming instructions. It directly measures the number of linearly independent paths through the program source code. In other words, this is a software metric that equates complexity to the number of decisions in a program. Developers can use this measure to determine which modules (i.e. network, actor, action, procedure) of a program are overly complex and need to be re-coded. For each module, the metric can be calculated either from evaluating the CFG of the module (i.e. see Section 2.4.3) or from evaluating the program's statements. The cyclomatic complexity is defined as:

$$v = e - n + 2p \tag{3.1}$$

where e is the number of edges, n is the number of nodes, and p is the number of modules. It must be noted that this equation is based on the assumption that the CFG is a strongly-connected graph. The cyclomatic complexity of a module also gives the maximum number of linearly independent paths through it. In other words, it can be evaluated by counting the branch conditions in a module. Hence, Equation (3.1) can be redefined such as:

$$v = b + 1 \tag{3.2}$$

where b represents the number of simple branch conditions. The formulation defined in Equation 3.2 is convenient because it allows developers to calculate the cyclomatic complexity of a program without having to use graph analysis. However, this only applies to individual modules in such that they only contain single-entry and single-exit, structured, blocks of code [79].

Table 3.1: Profiled executed operators and statements.

Kind	Symbol	Name
Unary	~	binary not
	!	logical not
	−	unary minus
	#	number of elements
Binary	&	bit and
	|	bit or
	∧	bit xor
	==	equal
	!=	not equal
	≥	greater than or equal
	>	greater than
	≤	less than or equal
	<	less than
	&&	logical and
	∥	logical or
	−	minus
	+	plus
	∗	times
	/	division
	div	integer division
	∗∗	exponentiation
	%	modulo
	<<	shift left
	>>	shift right
Data Handling	`ASSIGN`	assign
	`CALL`	call
	`LOAD`	load
	`STORE`	store
	`LIST_LOAD`	list load
	`LIST_STORE`	list store
Flow Control	`if`	*if then else* statement
	`while`	*while, do while* and *for* statements

3.2.4 Halstead metrics

Halstead metrics [80] are used to deduce a program production and quality based on the numbers of operands and operators used in the source code. Halstead metrics are based on the following set of parameters:

- n_1 the number of distinct operators present.
- N_1 the total number of operators present.
- n_2 the number of distinct operands present.
- N_2 the total number of operands present.

In the context of a dataflow program, these parameters can be defined with different levels of granularity: they can be defined for the overall program or for each actor, action and procedure. Some of the most-used Halstead metrics are the following:

- **Program length**: describes the size of the abstracted program obtained by removing everything except operators and operands from the original source code. It is defined as:

  $$N = N_1 + N_2 \tag{3.3}$$

  Contrarily to the SLOC metric (see Section 3.2.1), Halstead length gives a clearer accounting of the overall statement complexity. In fact, SLOC does not tell anything about how complex the lines of code are.

- **Program volume**: models the number of bits required to store an abstracted program of length N. It is defined as:

  $$V = N \log_2(n1 + n2) \tag{3.4}$$

  With this formulation, it is supposed that both the operators and the operands are encoded as binary strings of uniform (and potentially non-integral) length.

- **Program level**: describes the ratio between the volume V of the current program and the *most compact* volume of the same algorithm implementation [80]. It is defined as:

  $$L = \frac{2}{n_1} \frac{n_2}{N_2} \tag{3.5}$$

  In other words, a longer implementation of an algorithm has a lower program level than a shorter implementation of the same algorithm.

- **Program difficulty**: is defined as the inverse of the program level, such as:

  $$D = \frac{1}{L} \tag{3.6}$$

  In other words, a longer implementation of an algorithm has a higher difficulty compared to a shorter implementation of the same algorithm.

- **Programming effort**: defines the effort required to develop (or understand) a program. It is defined as:

  $$E = D\ V \tag{3.7}$$

  In other words, the programming effort is proportional to both the difficulty and the volume of the program.

- **Programming time**: defines the time in seconds required to develop the program. It is defined as:

  $$T = \frac{E}{S} \tag{3.8}$$

  where the S value is the Stroud number, defined as the number of elementary discriminations performed by the human brain per second [81]. S ranges from 5 to 20 and its value for software scientists is generally set to 18.

3.3 Data-dependent analysis

The execution of DDF programs can vary according to a particular input stimulus (see Section 2.3.3). For this reason, complexity of a DDF program cannot be defined only through a static code analysis as the one illustrated in the previous section. In other words, in order to identify the program's basic structure and complexity with different levels of abstraction, the DDF program should be executed considering a statistically meaningful set of input sequences [1]. The different approaches that are generally used are:

- **Binary-code execution**: where a low-level code representation of the dataflow program is generated and successively profiled through an instrumented platform-dependent (host-)execution [33, 82, 83, 84, 85].

- **Code interpretation**: where the dataflow program IR is executed through a platform-independent code interpretation [1, 73, 86].

The main difference between these two approaches is how the program execution abstracts from the platform and how results are biased by low-level code optimizations. The complexity measure obtained through a binary code-execution can be dependent on the particular platform where the program is executed and can be biased by low-level code optimizations performed by the compiler. Contrarily, with a IR-code interpretation, the complexity measure is totally platform-independent and not biased by low-level code optimizations. During a data-dependent analysis it is possible to identify the program's basic structure and complexity with different levels of abstraction, independently of the approach. Two main axes are typically recognized: the computational load and the data-transfers and storage load.

3.3.1 Computational load

The computational load is expressed in terms of executed operators and control statements (i.e. comparison, logical, arithmetic and data movement instructions). It is possible to model the firing time of an action firing based on the number of its executed operands and control statements retrieved during the program code interpretation. For each action firing s_i this is defined such as:

$$w(s_i) = \sum_j c_j \; o(s_i)_j \tag{3.9}$$

where $o(s_i)_j$ represents the number of executions of the $j-th$ operator or control statements performed by the action firing s_i and c_j a weight for the respective operator or control statements. It must be noted that c_j can be defined according to a desired target architecture. As for the static analysis discussed in Section 3.2, Table 3.1 reports the set of operators and control statements that can be retrieved interpreting a CAL program.

3.3.2 Data-transfers and storage load

The data-transfers and storage load are expressed in terms of internal actor variable utilization, input/output port utilization, buffer utilization and token production/consumption. During the program code interpretation, some statistical information concerning the actor internal variables and the buffer utilization can be stored to evaluate the memory load and utilization.

Internal actor variables

During the program code interpretation, for each firing and each actor internal variable the following information can be collected:

- **Writes**: number of writes that each firing has made on an internal actor variable.
- **Reads**: number of reads that each firing has made on an internal actor variable.
- **List writes**: number of writes that each firing has made on an internal actor list variable.
- **List reads**: number of reads that each firing has made on an internal actor list variable.

Tokens and buffers

During the program code interpretation, the following information can be collected for each firing and each buffer:

- **Writes**: number of tokens written on a buffer.
- **Reads**: number of tokens read from a buffer.
- **Peeks**: number of peeks (i.e. test of tokens presence) made by each firing on the respective input buffers.
- **Read miss**: number of unavailable tokens on the input buffers that made the selected action not fireable.
- **Write hit**: number of unavailable token places on the output buffers that made the selected action not fireable.

Furthermore, for each buffer, the **maximal occupancy** can be considered as a measure of an initial space estimation of the buffer size requirement.

3.4 Conclusions

In this chapter the main requirements of the profiling of a dataflow program have been summarized. It has been shown how actors can be analyzed and their behavior classified as static, cyclo-static and dynamic. Successively, the different static code analysis metrics have been illustrated. These are the elementary count of source lines of code, the operator count but also the more complex cyclomatic and Halstead metrics. Successively, data-dependent analysis for DDF programs has been discussed. The concepts of computational load and data-transfer and storage load have been introduced.

4 Exploring the design space of dataflow programs

Complex software systems may have many design points in terms of selection of software components and hardware architectures for implementation. These point choices create a large space of possible design solutions called the design space. The design process requires exploring through this design space to find design solutions before the actual implementation. The aim of the design space exploration (DSE) is to find design solutions that satisfy functional performance constraints and/or optimize portions of the system. In addition, the heterogeneity of modern parallel architectures and the diverse requirements of target applications greatly complicate modern systems design. Developing efficient programs for this kind of platform requires design methodologies that can deal with system complexity and flexibility. This has lead to the notion of system-level design, where key roles are played by aspects such as high-level modeling and simulation, and separation of concerns [87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97]. In this context, the exploration of the design space becomes an essential step when implementing applications to heterogeneous and parallel platforms. This is due to the combinatorial explosion of design options when dealing with multiple concurrent processing units. In order to have an efficient implementation and integration process, the design has to be sufficiently modular and portable, without the need of any or partial implementation and manual rewriting.

4.1 Orthogonalization of concerns

Orthogonalization of concerns is a well-established design paradigm [98]. Alternative solutions of the design space can be efficiently evaluated through design performance estimations. One of the main features of this design methodology is the separation between:

- Functional behavior and architecture.
- Communication and computation.

According to [89, 91], a formal model of a design is defined by the following components:

- A functional specification, given as a set of explicit or implicit relations which involve inputs, outputs and possibly internal state information.
- A set of properties that the design must satisfy, given as a set of relations over inputs, outputs, and states, that can be checked against the functional specification.
- A set of performance indexes that evaluate the quality of the design (e.g. in terms of cost, reliability, speed, size) given as a set of equations involving inputs and outputs.
- A set of constraints on performance indexes, specified as a set of inequalities.

The functional specification fully characterizes the operation of a system, while the performance constraints bound the cost. In other words, target points of the design space can be formulated in terms of minimization problems where the objective functions are defined as performance indexes and constraints as inequalities of the problem. In the following, the concept of *orthogonalization of concerns* is illustrated using the formalism described in [98], where the notions of model of computation, model of architecture and mapping are used.

4.1.1 Model of computation

The **Model of Computation** (MoC) is a formal representation of the operational semantics of networks of functional blocks describing computation [99, 98]. Depending on the modeling perspective, MoCs can be classified as an abstract or executable description [100]. Abstract models are used to define the application workload without executing the specification. On the other hand, executable specifications allow different abstraction levels: it can directly represent the application or, for example, a discrete-event performance model of the application itself. In the context of this thesis, only abstract dataflow MoCs are analyzed; more precisely, MoCs where the taxonomy can be described as illustrated in Section 2.3.

4.1.2 Model of architecture

The **Model of Architecture** (MoA) is a formal representation of the operational semantics of networks of functional blocks describing architectures [90, 98, 101, 102]. Depending on the modeling perspective, a MoA can be classified as an abstract or an executable architecture description [100]. Abstract models are used to represent performance in a symbolical manner. For example they associate the required latency in clock cycles with each operation without actually executing any hardware description. On the other hand, executable specifications allow to more precisely model state-dependent behavior, such as the timing of caches and pipelines. In the context of this thesis, only abstract dataflow MoCs are analyzed as the ones illustrated in [101, 102].

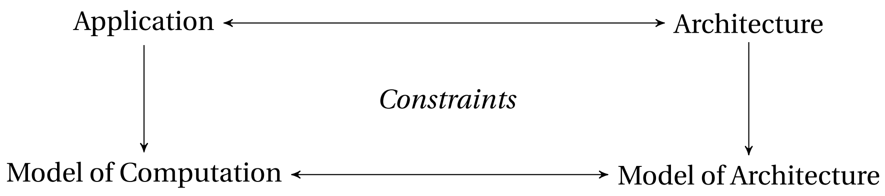


Figure 4.1: Mapping from an application to an architecture. Constraints represent the feasible regions of the design space.

4.1.3 Mapping

The mapping involves defining which part of the program is executed on a particular processing element, and which part of the communication structure is assigned to a particular media. In the context of hardware-software co-design the problem to be solved is coordinating the design of the parts of the system to be implemented as SW and the parts to be implemented as HW blocks [103]. The main requirement is to avoid HW/SW integration problems that can arise when heterogeneous platforms are used. As such, a set of **constraints** should be imposed and respected. Figure 4.1 depicts this process: the application is mapped into a target architecture if the set of constraints is fully satisfied. Constraints can be defined in terms of data type [90, 98] (e.g. an application that makes use of floating points can be mapped only in an architecture that supports this numeric representation) but also in terms of memory allocation, power utilization or clock frequency.

4.2 The design space of a dataflow program

The design space describes the different mapping configurations that can be defined among the application and the target architecture. However, there may exist many design alternatives that implement a given system specification. Each of these expose the design to different qualities of the design itself [104]. As such, these different implementations have to be explored and judged for their quality so that a designer can make a decision on which configuration has to be implemented. Consequently, during DSE many design alternatives have to be evaluated. Each design alternative may consist of different configuration choices with different levels of parameters: for example from the choice of the partitioning of an application block to a processing element, to a lower-level design parameter such as clock frequency or bus widths. Hence, the DSE objective is to evaluate one or more mapping configuration so that design objectives are satisfied. These objectives can be formulated in terms of real-time constraints, throughput, resource efficiency and utilization. This list can easily be extended by, for example, introducing requirements on the power consumption and silicon area utilization. The problem can be defined as efficiently finding a **feasible** design configuration so that requirements are fully satisfied.

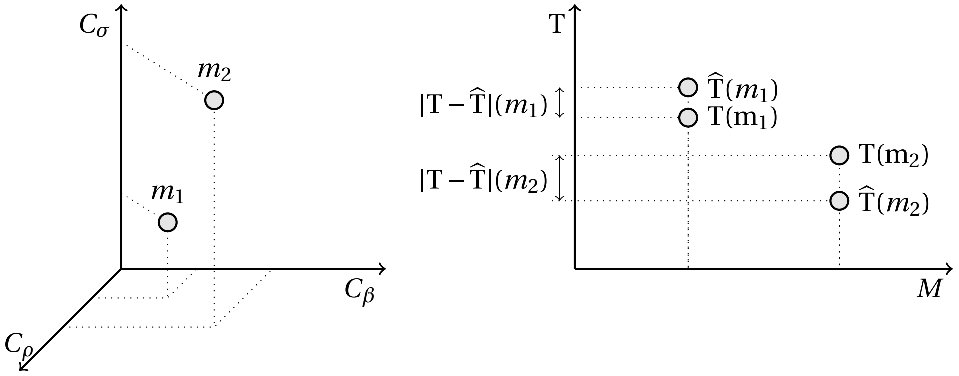


Figure 4.2: The design space $M = C_\rho \times C_\sigma \times C_\beta = \{m_1, m_2, \ldots, m_{n_M}\}$ and the corresponding performance $T(m)$ and estimated performance $\widehat{T}(m)$.

4.2.1 Design space and design points

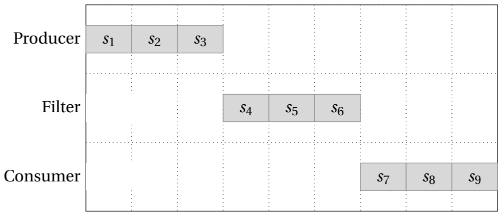
The evaluation of design points is one of the fundamental steps of the DSE. Its objective is to define the design space in terms of a set of independent parameters so that performance and requirements can be evaluated. The set of parameters is defined according to the abstraction level used for modeling both the application and the architecture. As such, when dealing with an abstract MoC and MoA, these parameters are defined in terms of partitioning, scheduling and buffer size configurations. In the following, the set of available partitioning, scheduling and buffer size configurations are referenced as C_ρ, C_σ and C_β, respectively. Hence, a **mapping configuration point** is defined as a 3-tuple $m = (\rho, \sigma, \beta)$ where:

- $\rho \in C_\rho$ defines a partitioning configuration of the network (i.e. actors and buffers mapped on the available processing elements and media, respectively).
- $\sigma \in C_\sigma$ defines a scheduling configuration of each partition.
- $\beta \in C_\beta$ defines a size configuration of each buffer.

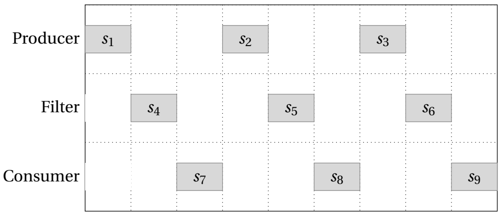
The **design space** of a dataflow program is then defined as the set of those independent configurations such as:

$$M = \{m_1, m_2, \ldots, m_{n_M}\} \subseteq C_\rho \times C_\sigma \times C_\beta \tag{4.1}$$

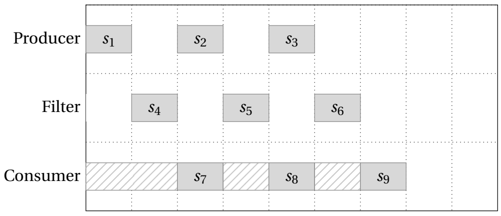
Consequently, the DSE problem is to efficiently find a mapping configuration point $m^* \in M$ so that the design objectives are met. As an example, let's consider the dataflow program presented in Section 2.5.4. Its network configuration is depicted in Figure 2.10 and it is composed of 3 actors: `Produce`, `Filter` and `Consume`. Supposing executing this program with the different mapping configurations illustrated in Table 4.1, hence corresponding execution Gantt charts are depicted in Figure 4.3.



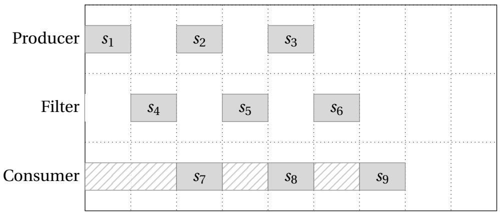
(a) Mapping configuration m_1



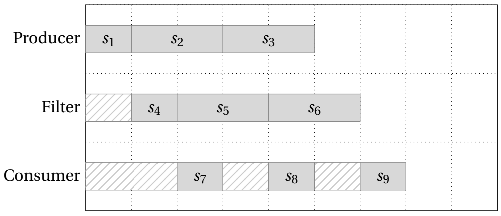
(b) Mapping configuration m_2



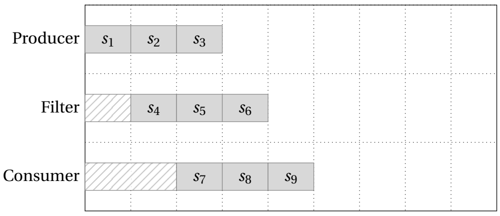
(c) Mapping configuration m_3



(d) Mapping configuration m_4



(e) Mapping configuration m_5



(f) Mapping configuration m_6

Figure 4.3: Platform independent simulation of the CAL network depicted in Fig. 2.10 with the mapping configurations described in Table 4.1. The execution of each action is supposed to take at least one (abstract) clock cycle (when there are no blocking output buffers), the overhead introduced by the action selection and buffer access overheads are both neglected. In gray the actor execution with the corresponding action firing. In striped-gray the actor execution is postponed due to the unavailability of a token (i.e. blocking reading).

Table 4.1: Mapping configurations for the dataflow network illustrated in Figure 2.10. For brevity, the actors Producer, Filter and Consumer are denoted with P, F, C, respectively. The partitioning of the buffers is not considered.

Mapping $\mathbf{m_i}$	**Partitions ρ_i**	**(static) Scheduler σ_i**	**Buffer size β_i**
m_1	$\rho_1^1 = \{P, C, F\}$	$\sigma_1^1 = \{P, P, P, C, C, C, F, F, F\}$	$\beta_1^1 = 512$ $\beta_1^2 = 512$
m_2	$\rho_2^1 = \{P, C, F\}$	$\sigma_2^1 = \{P, F, C, P, F, C, P, F, C\}$	$\beta_2^1 = 512$ $\beta_2^2 = 512$
m_3	$\rho_3^1 = \{P, F\}$ $\rho_3^2 = \{C\}$	$\sigma_3^1 = \{P, F, P, F, P, F\}$ $\sigma_3^2 = \{C, C, C\}$	$\beta_3^1 = 1$ $\beta_3^2 = 1$
m_4	$\rho_4^1 = \{P, F\}$ $\rho_4^2 = \{C\}$	$\sigma_4^1 = \{P, F, P, F, P, F\}$ $\sigma_4^2 = \{C, C, C\}$	$\beta_4^1 = 512$ $\beta_4^2 = 512$
m_5	$\rho_5^1 = \{P\}$ $\rho_5^2 = \{F\}$ $\rho_5^3 = \{C\}$	$\sigma_5^1 = \{P, P, P\}$ $\sigma_5^2 = \{F, F, F\}$ $\sigma_5^3 = \{C, C, C\}$	$\beta_5^1 = 1$ $\beta_5^2 = 1$
m_6	$\rho_6^1 = \{P\}$ $\rho_6^2 = \{F\}$ $\rho_6^3 = \{C\}$	$\sigma_6^1 = \{P, P, P\}$ $\sigma_6^2 = \{F, F, F\}$ $\sigma_6^3 = \{C, C, C\}$	$\beta_6^1 = 512$ $\beta_6^2 = 512$

4.2.2 Exploration methods

Different DSE methodologies can be classified according only if single or multiple design-objectives are taken into account. In the latter, optimality is usually defined using the notion of Pareto-dominance [105]: a design point dominates another one if it is equal or better in all criteria and strictly better in at least one. In a set of design points, these are called **Pareto-optimal** which are not dominated by any other. With this notion in mind, the different DSE approaches can be characterized, as summarized in [106], so that:

- **Exploration by hand**: the selection of design points is done by the designer himself. The major focus is on how design performance can be efficiently estimated [107].
- **Exhaustive search**: all design points of a specified region are evaluated. Generally, this approach is combined with local optimization heuristics where one or multiple design parameters are evaluated in order to reduce the size of the design space [108].
- **Reduction to a single objective**: design points are selected by reducing the DSE problem to a set of single criterion problems. Manual or exhaustive sampling is done in one or several directions of the search space and a constraint optimisation (e.g. iterative improvement or analytic methods) is done in the other [109, 110, 111, 112].
- **Black-box randomized search**: design points are evaluated using a black-box optimisation approach. The design space is iteratively analyzed, where at each iteration the new design point is computed based on the priory information and by defining an appropriate neighborhood function. The properties of these new design points are estimated.

Examples of sampling and search strategies are Pareto-simulated annealing [113] and Pareto-tabu search [114] evolutionary multi-objective optimization [115, 116] or Monte Carlo methods improved by statistical estimation of bounds [117]. These black-box optimizations are generally combined with local search methods [118].

4.2.3 Performance estimation

Performance analysis always involves three issues: a modeling effort, an evaluation effort and the accuracy of the obtained results [119, 120, 121]. Very accurate performance numbers can be achieved, but at the expense of a lot of detailed modeling and long evaluation times. However, performance numbers can be achieved in a shorter time with modest effort for modeling but at the expense of loss of accuracy. Independently from the abstraction level used to model the application and the architecture, the objective for efficiently exploring the design space is to find an appropriate performance estimation of the application for each mapping configuration point. If the performance of each mapping configuration point $m = (\rho, \sigma, \beta) \in M$ of the design space is defined in terms of application throughput as:

$$\mathrm{T}(m) = f(m) \tag{4.2}$$

the approximated model can be defined as:

$$\widehat{\mathrm{T}}(m) = \widehat{f}(m) \tag{4.3}$$

hence, the objective is to reduce the accuracy error defined as:

$$\epsilon = ||\mathrm{T} - \widehat{\mathrm{T}}||_2 = \left(\sum \{ |\mathrm{T}(m_i) - \widehat{\mathrm{T}}(m_i)|^2 \ : \ m_i \in M \} \right)^{\frac{1}{2}} \tag{4.4}$$

where $||.||_2$ is the 2-norm operator.

4.3 Related work

In the following section, an overview of some design space exploration tools and frameworks is presented. For each one, the main functionalities and limitations are discussed.

CAL Design Suite

The CAL Design Suite [122, 33] is a set of tools for exploring and optimizing the design space of RVC-CAL applications. It represents the first functional attempt to provide a complete design flow for optimizing RVC codec specification to multi-core and heterogeneous platforms [30]. It is based on the analysis of the execution trace graph (ETG) of the program (i.e. see Section 7.1). However, their definition is limited since only internal variables and tokens dependencies are supported. Furthermore, the CAL design suite provides a very basic architecture model for heterogeneous platforms. A detailed discussion on how CAL programs are profiled is presented in Section 7.1.

COMPA

The COMPA project [123, 46] provides an analysis and optimization framework for RVC-CAL applications. The design space exploration is performed through a static analysis of the source code. Different trade-offs between parallelism, communication traffic cost, and memory size requirement are implemented as source to source transformations.

Daedalus

Daedalus [124, 125, 126] provides a unified environment for rapid system-level architectural exploration, high-level synthesis, programming and prototyping of multimedia MPSoC architectures. The Daedalus framework is an automatic design flow for KPN networks. The application is modeled using a C/C++ imperative specification which is then automatically converted into a KPN using the KPNgen tool [127]. Because of the nature of KPN models, modeling of interrupts is difficult and inefficient. The design space exploration is performed using the Sesame system-level simulation framework.

MAPS

The MPSoCs Application Programming Studio (MAPS) [93, 94, 95, 97] is a DSE framework for KPN programs. Both the performance estimation and the design space is performed through an ETG analysis. ETGs are obtained by profiling and are augmented with timing information via performance estimation. However, their definition is limited since only internal variables and tokens dependencies are supported. Several heuristics for buffer sizing, mapping, and scheduling are available within the framework. For fast and functional validations, MAPS is fully integrated with the High-Level Virtual Platform (HVP) simulator [128]. Furthermore, MAPS is equipped with a pioneering multi-application analysis component that performs composability analysis in order to assess if a set of applications may run simultaneously, on the same platform, without interfering with each other.

Mescal

Mescal project [129, 130] aims at designing heterogeneous, application-specific, programmable (multi) processors. The goal is to allow the programmer to describe the application in any combination of models of computation that is natural for the application domain. The goal is also to find a disciplined and correct by construction abstraction path from the underlying micro-architecture to an efficient mapping between application and architecture. The micro-architecture description including the memory subsystem is based on an architecture description language.

Metropolis

Metropolis [131] is a framework allowing the description and refinement of a design at different levels of abstraction and integrates modeling, simulation, synthesis, and verification tools. It provides an infrastructure based on meta-modeling with precise semantics that are general enough to support various model of computations. This meta-model can capture the functionality, the architecture and the mapping between the two different abstraction levels. The function of a system, such as the application, is modeled as a set of processes that communicate through media. Architectural building blocks are represented by performance models where events are annotated with the costs of interest. A mapping between functional and architecture models is determined by a third network that correlates the two models by synchronizing events (using constraints) between them. Non-deterministic behavior can be modeled and constraints can restrict the set of possible executions.

PeaCE

The PeaCE Environment [132] specifies the system behavior with a heterogeneous composition of three models of computation. These are an extended SDF model (called SPDF) for computation tasks, an extended FSM model (called fFSM) for control tasks, and a task model to describe the task interactions, respectively. The PeaCE environment provides seamless co-design flow from functional simulation to system synthesis, utilizing the features of the formal models maximally during the whole design process. This framework is based on the Ptolemy project [133]. However, when dealing with C/C++ specifications, the PeaCE approach does not provide an automatic procedure to transform this specification into dataflow graphs.

Preesm

Preesm [134, 135] is a rapid-prototyping framework for static dataflow applications that has been inspired by the algorithm architecture adequation matching methodology (AAM, also sometimes called AAA) [136]. Preesm makes uses of a parameterized and interfaced dataflow meta-model (PiMM) [137] representation of the application, together with a System-Level Architecture Model (S-LAM) for the high-level architecture description. It automatically generates functional code for heterogeneous multi-core embedded systems, optimizing the application scheduler by using the throughput as an optimization requirement.

Ptolemy

Ptolemy [133, 99] is a component-based heterogeneous modeling environment. It allows the hierarchical combination of different models of computations with a high level of abstraction.

It uses tokens as the underlying communication mechanism. Controllers regulate how actors fire and how tokens are sent between each actor. This mechanism allows different models of computation to be combined within the Ptolemy framework. The design space exploration is performed with third party environments (e.g. the PeaCE framework [132]).

SDF^3

SDF^3 [138] is a dataflow analysis tool that supports SDF and CSDF dataflow models of computations. SDF^3 is oriented towards model analysis and simulation without generating an executable prototype of the application.

Sesame

The Sesame system-level simulation framework [139] addresses the problem of finding a suitable and efficient target MP-SoC platform architecture. Sesame deploys separate application and architecture models: the application model describes the functional behavior of an application, while the architecture model defines architecture resources and captures their performance constraints. Sesame maps application models onto architecture models for cosimulation by means of trace-driven simulation, while using an intermediate mapping layer for scheduling and event-refinement purposes. This allows for evaluation of the system performance of a particular application, mapping, and underlying architecture. Essential in this methodology is that an application model is independent from architectural specifics and assumptions on hardware/software partitioning. The main disadvantage in this methodology is that only KPN application models can be used and analyzed.

Space Codesign

Space Codesign [140, 141] is a design environment that provides an interface for user-written SystemC modules that models application software to make calls to a real-time operating system kernel. It provides a cosimulation environment for user-written SystemC hardware modules. The environment also facilitates successive refinement through three software abstraction layers for hardware-software codesign suitable for embedded-system design. The first level focuses on the system design: the application is specified and functionality validated through the SystemC simulator. In the second layer the application is partitioned among different software and hardware modules. The hardware is modeled and emulated via the SystemC simulator, while the software is encapsulated in the SystemC-RTOS interface via an RTOS emulation process. At the third level, a more sophisticated architecture model is emulated with the support of cycle accurate simulation at a chosen processor frequency.

SPADE

The Stream Processing Application Declarative Engine (SPADE) [142, 143] is a stream processing application development framework for System S [144], which is a large-scale, distributed datastream processing middleware. As a front-end for rapid application development for System S, SPADE provides an intermediate language for composition of parallel and distributed dataflow graphs, together with a toolkit of type-generic, built-in stream processing operators, that support scalar as well as vectorized processing and can seamlessly inter-operate with user-defined operators. It provides a code generation framework to create optimized applications that run natively on the Stream Processing Core (SPC), the execution and communication substrate of Systems. Successively, an optimizing compiler automatically maps applications into appropriately-sized execution units in order to minimize communication overhead, while at the same time exploiting available parallelism.

SynDEx

SynDEx [145] is a graphical and interactive software implementing the Algorithm Architecture Adequation Matching methodology (AAM, also sometimes called AAA) [136]. Within this environment, the designer defines an algorithm graph, an architecture graph and system constraints. SynDEx is a Computer-Aided-Design software aiming at mapping an algorithm into an architecture. The architecture taken into account is only composed of several processors, and hardware logic, like FPGA, cannot be taken into account in this flow. The design space exploration is done according to one unique criteria: the application throughput. It provides the possibility of low-level code generation, but it is not actually provided within the distributed tools.

SystemCoDesigner

SystemCoDesigner [146, 147, 148] is an actor-oriented approach using a high-level language named SysteMoC, which is built on top of SystemC. It generates HW-SW SoC with automatic design space exploration techniques. The model is translated into a behavioral SystemC model as a starting point for HW and SW synthesis. During DSE, the design space is explored using state of the art multi-objective optimization algorithms. For each design alternative, performance is estimated by using performance models (which are generated automatically from the SystemC behavioral model) and the behavioral synthesis results. The HW synthesis is delegated to Forte Cynthesizer [149], a commercial tool which generates RTL code from a SystemC intermediate model.

4.4 Advances in design space exploration of CAL programs

In the previous sections it has been discussed how the design space of an application can be modeled and explored. Moreover, a list of available design exploration tools has been presented. In the following section the main improvement and advancements concerning the exploration and optimization of CAL dataflow programs are illustrated.

4.4.1 Space for improvement

- **Dynamic program analysis**: dynamic program analysis is not supported by the tools available for CAL programs. They limit their analysis to static and cyclo-static MoC classes. Even though they can provide guarantees on the system performance and requirements (e.g. deadlock-free execution), complex dynamic programs (e.g. video codecs) can be analyzed only under strong assumptions and limitations on the design cases.

- **Design space modeling**: as mentioned before, the design space can be modeled only for restricted classes of dataflow programs. Moreover, performance estimation methodologies are specifically targeted for restricted sets of architectures.

- **Bottlenecks and refactoring directions**: except for the CAL design suite, bottleneck and design refractory directions are not provided. As such, the designer should implement its application on the target architecture and profile with an additional third-party profiler for the resulting implementation. Relations between profiling results of the application and the corresponding CAL source code is done by hand. However, sometimes this relation cannot be univocally obtained (e.g. due to code-inlining optimization done by compilers).

- **Automatic mapping and code generation**: automatic mapping and code generation is partially driven over a limited set of architectures. Moreover, tools does not provide a uniform and interoperable methodology to provide the mapping configuration.

4.4.2 New requirements

Consolidated design space exploration methodologies for static and cyclo-static dataflow programs are hardly extensible to dynamic dataflow programs. In fact, they make use of analytical models of the application MoC. For dynamic programs, this leads to possible non-linear and difficult-to-solve formulation. Consequently, this formalism should be extended in order to make the design exploration and performance analysis possible using a unique mathematical tool-set. For this reason, the concept of an execution trace graph of a dataflow program has been formalized. This is a graph-based representation of the program where nodes represent a single action firing and directed arcs represent dependencies among couples of actions firing. In the next chapter it is demonstrated how using this formalism is possible to efficiently explore the design space of dynamic dataflow applications (and also by consequence, in the case of static and cyclo-static programs).

4.5 Conclusions

In this chapter the main requirements for a design space exploration (DSE) environment have been summarized. The notion of orthogonalization of concerns has been introduced. The main features of these design methodologies are the separation between functional behavior and architecture and between communication and computational load. Furthermore, the notions of high-level models of computation (MoC) and models of architecture (MoA) have been presented. The design space and design points (i.e. design alternatives) of an application have been formalized. Each design point has been defined as a particular mapping configuration of the design defined in terms of partitioning, scheduling and buffer size configuration. Successively, different DSE analysis and performance estimation methods have been illustrated together with an overview of the current available frameworks. A discussion about possible space for improvements of the methodology and tools in the context of dynamic dataflow programming, and more precisely for the CAL dataflow language, has been presented at the end of the chapter.

5 Execution trace graph

In Chapter 3 we discussed how the execution of a dataflow program consists of a sequence of action firings. In this chapter how those firings can be correlated in a novel graph-based representation, called the **execution trace graph**, in order to model the execution behaviour of the program is illustrated. The graph is an acyclic directed graph where each node represents an action **firing** and each directed arc represents either a data or a logical **dependence** between two different action firings. A partial order of the fired actions can be obtained from the topological order of the graph. Hence, using the notions of partially-ordered space and directed-path developed in [150, 151, 152, 153], the effectiveness of analyzing a dataflow program starting from its behavioural execution is demonstrated.

5.1 Geometry of execution

Without the ambition to be complete, this section provides a brief introduction to the trace space theory formalized in [150, 151, 152, 153]. Looking at the geometry of dataflow program executions, it is possible to think of a concurrent execution of two actors `A` and `B` on two processing units pu_1 and pu_2 as a curve in $\mathbb{R}^2$. Points on this space have the local time on pu_1 taken to execute `A` on pu_1 as abscissa, and the local time on pu_2 taken to execute `B` on pu_2 as ordinate. Figure 5.1 depicts a possible **execution path** along the **execution space** of the program. The execution space of a program can be considered as the set of all possible increasing paths (as far as the time flow cannot be inverted) included in the square delineated by the interleaving of `A` and `B`.

5.1.1 Partially-ordered space

The geometric model which has already been implicitly used in Figure 5.1 is a partially-ordered space, also called a **po-space**. This is a topological space equipped with a partial order. In other words, a po-space is a topological space in which points are ordered globally through time. Formally, a **partial order** $\leq$ on a set U is a reflexive, transitive and antisymmetric relation.

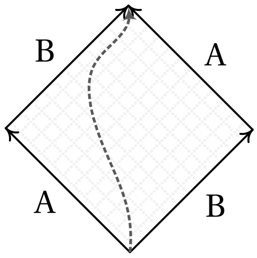


Figure 5.1: Execution space in $\mathbb{R}^2$ of two actors `A` and `B` mapped on two processing units pu_1 and pu_2, respectively. The dashed arrow represents a possible execution path of the program.

A partial order $\leq$ on a topological space X is said to be *closed* if $\leq$ is a closed subset of $X \times X$ in the product topology. In that case $(X, \leq)$ is called a po-space.

5.1.2 Execution trace

The dashed arrow that has already been intuitively used in Figure 5.1 represents an *execution trace* of the two actors `A` and `B`. In other words, it represents a directed path, also called **d-path**, in the execution space. Formally, a d-path $\overrightarrow{p}$ in a po-space $(X, \leq)$ is defined as:

$$\overrightarrow{p} : \overrightarrow{1} \rightarrow X \tag{5.1}$$

that is continuous and order-preserving, where $\overrightarrow{1} = [0,1] \subseteq \mathbb{R}$ represents the closed and directed unit interval. A d-path that is up to monotone reparametrizations is called **trace** and it is represented as $X(x_1, x_2)$, where $x_1, x_2 \in X$ such that $x_1 \leq x_2$. A po-space equipped with a notion of direction is defined as a directed topological space, also called a **d-space**. A d-space if formally defined as (X, dX) consists of a po-space X together with a set of dX of paths in X. In this case, it is possible to define a new partial order $\prec$ on X such that $x_1 \prec x_2$ if there is a d-path from x_1 to x_2 in X. This is a sort of **reachability** relation that is antisymmetric and coarser than the relation $\leq$ in the sense that $x_1 \prec x_2 \Rightarrow x_1 \leq x_2$.

5.1.3 Execution trace space

The concurrent execution of actors `A` and `B` depicted in Figure 5.1 might have several feasible traces. In other words, some equivalent traces might exist such that the corresponding executions end with the same result. Formally, two d-paths $\overrightarrow{p}_1$ and $\overrightarrow{p}_2$ are considered as equivalent when $\overrightarrow{p}_2$ can be obtained by continuously deforming $\overrightarrow{p}_1$ (or vice versa). This equivalence relation is called **dihomotopy**. Given two points x_1 and x_2 of a d-space (X, dX), then $E(X, dX)(x_1, x_2)$ identifies the **execution trace space** obtained from $X(x_1, x_2)$ by identifying all the dihomotopic equivalent paths. In particular, $E(X, dX)(x_1, x_2) \neq \emptyset$ if and only if there exists at least one directed path in X going from x_1 to x_2.

5.2 Execution trace graph

The execution of a dataflow program can be modelled as a directed acyclic graph (DAG) where each node represents a single action firing and each directed arc represents either a data or a logical dependence between two different action firings [1, 2, 33, 94, 154]. In Section 2 it has been shown how during each firing, an action can consume a finite number of input tokens, produce a finite number of output tokens, and modify the actor's internal variables. Hence, it can be observed that it is possible to identify the dependencies that arise among different firings. For example, if during a firing an action consumes some tokens, then it must rely on the execution of the action that produced those tokens. The same can be stated if the action, in the processing part of the firing, makes use of some of the internal actor variables that were previously modified or used by another action. Several other types of dependencies can be identified and used to characterize the execution of a dataflow program: these are summarized in Table 5.1 and presented in Section 5.2.2.

An **execution trace graph** (ETG) is formally defined as a DAG(S, D), where:

- S is the set of single action firings, defining the nodes of the graph.
- $D = S \times S$ is the set of dependencies, defining the directed edges of the graph.

Defining dependencies between action firings establishes a precedence order. If the firing $s_2 \in S$ depends on firing $s_1 \in S$, then s_1 has to be executed and completed before s_2 can be started. The dependency is then defined as $(s_1, s_2) \in D$. The transitive hull of the dependencies is the precedence relation $\leq$. So, S can be defined as a po-space $(S, \leq)$ and the precedence constraint among s_1 and s_2 can be expressed as $s_1 \prec s_2$.

Remark. *In this work it is assumed that the number of firings in S and the number of dependencies in S are finite and they will be denoted by the notation* $|S| < \infty$, $|D| < \infty$ *respectively.*

5.2.1 Firings

Each $s_i \in S$ represents a single action firing occurring during the execution of a dataflow program. In other words, if an action is fired n times, thus n nodes in S are used to represent each single firing.

A single action firing $s \in S$ 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.

5.2.2 Dependencies

Each $(s_i, s_j) \in D$ represents dependence between two fired actions s_i and s_j, such that $s_i \neq s_j$. Several kinds of dependencies can be defined during the execution of a dataflow program. As summarized in Table 5.1, these are: internal variable, finite state machine, guard, port and tokens. As illustrated in the following, each of these can be defined by a sub-kind and enhanced with some profiling parameters useful for a post-mortem analysis. Hence, more than one dependence can be defined between each couple s_i, s_j.

A dependency $(s_i, s_j) \in D$ is formally 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 dependence kind. As illustrated in the following, the kind can be: *internal variable*, *finite state machine*, *guard*, *port* or *tokens*.
- d is the dependence direction. As illustrated in the following, 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 such as:

$$\delta(s_i)^-_E = \{(s_n, s_m) : \forall (s_n, s_m) \in D, s_m = s_i\} \tag{5.2}$$

The set of firings which are the source of an incoming dependencies of s_i is called the **predecessor**, and is denoted as:

$$\delta(s_i)^-_S = \{s_j : \exists (s_j, s_i) \in D\} \tag{5.3}$$

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

$$S_{\emptyset^-} = \{s_i : \delta(s_i)^-_S = \emptyset\} \tag{5.4}$$

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

$$\delta(s_i)^+_E = \{(s_n, s_m) : \forall (s_n, s_m) \in D, s_n = s_i\} \tag{5.5}$$

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

$$\delta(s_i)^+_S = \{s_j : \exists (s_i, s_j) \in D\} \tag{5.6}$$

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

$$S_{\emptyset^+} = \{s_i : \delta(s_i)_S^+ = \emptyset\} \tag{5.7}$$

Internal variable

An internal variable dependency $(s_i, s_j) \in D$ 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 the 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, who wrote on v.
- **write/write**: when the action firing s_j has an intervening write operation on the internal variable v and s_i is the last action firing, previous to s_j, who wrote on v.
- **read/read**: when both the 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, who read from v.
- **write/write**: when both the action firings s_i and s_j wrote on the internal variable v and s_i is the last action firing, previous to s_j, who wrote on v.

Only the *write/read* is a data dependency. By contrast, the *read/write*, *read/read* and *write/write* express only memory utilization precedence between the two actions and such information could be useful if a memory optimization of the design is applied. The parameter that can be stored in this kind of dependency is variable v on which the dependency is related. Additional attributes retrieved from the profiling are the initial and final value of such a variable. In the following, the set of dependencies of this kind are denoted with $D_v \subseteq D$.

Finite state machine

An internal state machine dependency $(s_i, s_j) \in D$ connects two executed actions belonging to the same actor and related via its internal state scheduler. In other words, a dependency of this kind is defined when both the execution of the 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. In the following, the set of dependencies of this kind are denoted with $D_f \subseteq D$.

Guard

A guard dependency $(s_i, s_j) \in D$ occurs when an action firing s_i modifies the value of the guard which conditions the action firing s_j. The guard condition, which may be defined as a combination of state variable and token value, can be defined enabled or disabled by s_i by the modification of its variables or the production of particular token values. For this kind 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 not-executable (i.e. disabled).

The parameters that can be stored are the guard identifier on which the dependency is related and the appearance order on which this guard was enabled or disabled. In the following, the set of dependencies of this kind are denoted with $D_g \subseteq D$. It must be noted that in some design cases, uncovering these dependencies might have the side effect of letting the trace be dependent on both the buffer size and the scheduler configuration used during the program execution. A more detailed discussion about this kind of dependency is presented in Section 5.3.6.

Port

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

- **read/read**: when both the 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, who retrieved at least one token from p.
- **write/write**: when both the 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, who sent at least one token to p.

The parameter that can be stored in this kind of dependency is the port p (input of output) on which the dependency is related. In the following, the set of dependencies of this kind is denoted with $D_p \subseteq D$.

Tokens

A tokens dependency $(s_i, s_j) \in D$ connects the action firing that produces some tokens to the one that consumes at least one of them. In such cases, these actions may be in different actors, or they may be part of the same actor (i.e. in case of a direct dataflow feedback loop). The parameters that can be stored in this kind of dependency are the number of tokens that the consumer firing s_j consumed among the tokens produced by the producer firing s_i. Additional attributes retrieved from the profiling are the token values. In the following, the set of dependencies of this kind are denoted with $D_t \subseteq D$.

Table 5.1: Dependencies kinds, directions, parameters and additional attributes.

	Name	Direction	Parameters	Additional attributes
D_v	internal variable	read/read write/write read/write write/write	variable id	initial value final value
D_f	finite state machine			
D_g	guard	enable disable	guard id appearance order	
D_p	port	read/read write/write	port id	
D_t	tokens		output port id number of tokens	token values

5.2.3 Example of an execution trace graph

The dataflow program described in Section 2.5.4 is used in order to show the main structure of an ETG. The firing set S contains nine action firings $s = \{s_1, s_2, \dots, s_9\}$ which are summarized in Table 2.2. The firing set S can be divided in three sub-sets, one for each actor of the network, $S_P = \{s_1, s_2, s_3\}$, $S_F = \{s_4, s_5, s_6\}$ and $S_C = \{s_7, s_8, s_9\}$, such that $S = S_P \cup S_F \cup S_C$ and $S_{a_n} \cap S_{a_m} = \emptyset$ for each couple of actors $a_m \neq a_n$. Sets S_P, S_F and S_C contain the firings of `Producer`, `Filter` and `Consumer` respectively. The dependencies set D contains sixteen dependencies $D = \{e_1, e_2, \dots, e_{16}\}$ which are summarized in Table 5.1.

Even though the firing sequence of this program has already been illustrated in Section 2.5.4, it is worth re-describing part of it, and highlighting how the ETG of Figure 5.2 can be obtained. In this context, let's suppose the mapping configuration m_1 described in Table 4.1 is used. This particular mapping configuration defines a single partition configuration $\sigma_1 = \{P, F, C\}$ (i.e. all the actors are assigned to the same processing element), the scheduler configuration $\sigma_1 = \{P, P, P, F, F, F, C, C, C\}$ (i.e. predefined and static) and the buffer size configuration $\beta_1^1 = \beta_1^2 = 512$. The execution Gantt chart is depicted in Figure 4.3a, where each action firing takes one (abstract) clock cycle to conclude its execution. At time $t = 0$, the scheduler imposes to execute the actor `Producer` which fires the action `produce`. This single action firing is denoted with s_1. During its execution, s_1 updates the internal actor variable `counter`: the initial value is $\texttt{counter}_i = 0$ and the final value is $\texttt{counter}_f = 1$. Finally, the firing concludes by writing an output token $\tau_1 = 1$ on the output port `O`. At time $t = 1$ the scheduler selects again the actor `Producer` which fires again the action `produce`. This second firing is denoted with s_2. Also s_2 updates the internal actor variable `counter`: the initial value is $\texttt{counter}_i = 1$ and the final value is $\texttt{counter}_f = 2$. Finally, the firing concludes by writing an output token $\tau_1 = 2$ on the output port `O`. During the firing s_1 the internal state variable $\texttt{counter}_i$ has the value previously written by the firing s_1: hence an internal variable dependency between s_1 and s_2 can be defined. This is denoted with e_1. As both the firings wrote this variable, the

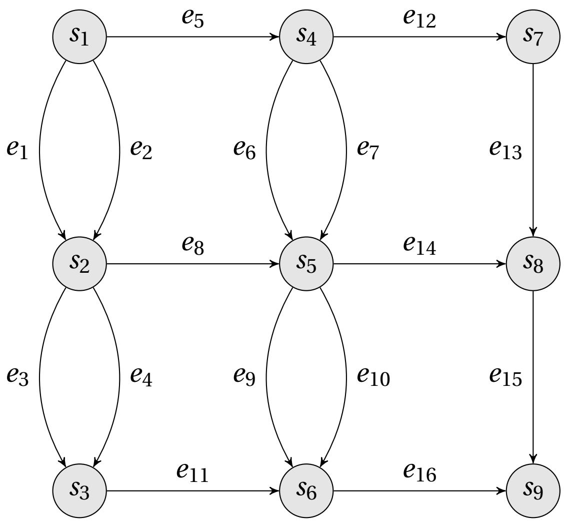


Figure 5.2: Execution trace graph obtained after the execution of the CAL program described in Section 2.5.4. The firing set S is summarized in Table 2.2, and the dependencies set D is summarized in Table 5.2.

dependency direction is *write/write*. Moreover, both s_1 and s_2 wrote a token on the same output port: hence a port variable, with direction *write/write*, can be defined. This is denoted with e_2. The same happens at time $t = 2$, when the same action is fired for the third time in a row. This new firing is denoted with s_3. Also in this case an internal variable and a token dependency can de defined with the previous step s_2: these are e_3 and e_4 respectively. At time $t = 3$, the scheduler imposes the execution of the actor `Filter` which fires the action `invert`. This action firing is denoted with s_4. During the firing, s_4 consumes the token τ_1 from its input port `I` and produces an output token τ_4 on its output port `O`. As 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: this is denoted with e_5. At time $t = 4$, the second execution of the actor `Filter`, imposed by the scheduler, again fires the action `invert`. This action firing is denoted with s_5. Also this firing read the token τ_2 from the input port `I` and wrote the token τ_5 on the output port `O`. As τ_2 was previously produced by s_2, a new token dependency can de defined: this is denoted with e_8. Furthermore, as 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: these are denoted with e_6 and e_7 which have as direction *read/read* and *write/write* respectively. The execution of the entire program continues till $t = 9$ and the same considerations can be made in order to build the remaining dependencies of the ETG.

Table 5.2: Dependencies set S of the execution trace graph depicted in Figure 5.2.

(s_i, s_j)	**Source**	**Target**	**Kind**	**Direction**	**Parameter**	**Attribute**
e_1	s_1	s_2	Variable	Write/Write	variable=`counter`	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=`counter`	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

5.3 Properties

The aim of this section is to illustrate and demonstrate the main properties of an ETG. In fact, as demonstrated in Chapter 6, these properties can be successfully exploited when exploring and optimizing the design space of a dataflow program.

5.3.1 Topological order

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

$$s_i \leq s_j \Rightarrow l(s_i) < l(s_j) \tag{5.8}$$

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 in the following of this section, an ETG can express the maximum potential parallelism of the program. This property is strictly related to the fact that a DAG generally admits several valid topological mapping functions.

Execution trace space

By recalling the notation introduced in Section 5.1, $(S, \leq)$ and (S, dX) represent the po-space and the d-space, respectively, of the program execution defined by the firings set S and the dependencies set D. The po-space $(S, \leq)$ refers to the collection of firings ordered by their dependencies. Similarly, the d-space (S, dX) refers to the collection of directed paths that can be defined among the firings by *following* their outgoing dependencies. Consequently, the ETG defines what is called the execution trace space of a program that has been formalized in Section 5.1.3.

5.3.2 Mapping independence

The mapping independence property can be demonstrated using the same example dataflow program described in Section 5.2.3 and analyzing the ETGs that are obtained using the different mapping configurations defined in Table 4.1. It must be noted that those considerations are valid only for deterministic actors, in the sense that the execution is not time dependent.

Scheduling independence

Let's consider the two mapping configurations m_1 and m_2, which differ only on how the scheduling configuration has been defined. The first was used in Section 5.2.3 to illustrate how the ETG depicted in Figure 5.2 has been obtained. In this case the firings set S has been obtained with the following order $S(m_1) = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\}$ as depicted in the

Gantt chart in Figure 4.3a. Using the scheduler configuration defined in m_2, the firing steps order changes. In this case, $S(m_2) = \{s_1, s_4, s_7, s_2, s_5, s_8, s_3, s_6, s_9\}$ as depicted in the Gantt chart in Figure 4.3b. Following the considerations made in Section 5.2.3, the two dependencies sets $D(m_1)$ and $D(m_2)$, respectively, remain the same. This leads to the same ETG as the one depicted in Figure 5.2, since the partial order of the firings defined on both $S(m_1)$ and $S(m_2)$ is the same. This demonstrates that the partial order of the firings set S defined only using the dependencies set D does not give any information about the scheduling policy used for constructing the ETG. Hence, the ETG does not depend on the scheduling policy. Additional edges should be introduced in the ETG in order to make it possible to define a more strict ordering of S and defining the scheduling configuration. If the objective is to model $S(m_1)$ and $S(m_2)$ such as $S(m_1) = \{s_1 < s_2 < s_3 < s_4 < s_5 < s_6 < s_7 < s_8 < s_9\}$ and $S(m_2) = \{s_1 < s_4 < s_7 < s_2 < s_5 < s_8 < s_3 < s_6 < s_9\}$, some additional edges should be introduced as depicted in Figure 5.3a and Figure 5.3b, respectively. Those additional edges are depicted with dashed arrows as they should not be confused with the dependencies defined in the previous section. In fact, those additional edges on the ETG are only used to model the constraints imposed by the scheduler.

Partitioning independence

The same considerations about the ETG scheduling independence can also be done for the partitioning configuration. Let's consider the two mapping configurations m_2 and m_4 defined in Table 4.1. In m_1 all the actors are mapped in one partition, contrary to m_4 where two partitions are defined. The scheduling and buffer size configurations of m_2 andm_4 are the same. Considering m_4, the firings set S has been obtained with the following order $S(m_4) = \{s_1, s_4, s_2, s_7, s_5, s_3, s_8, s_6, s_9\}$ as depicted in the Gantt chart in Figure 4.3d. Following the same considerations made in Section 5.2.3, since the dependencies set $D(m_4)$ is the same as D, in this case the corresponding ETG is also the one depicted in Figure 5.2. If the objective is to model the additional constraints imposed by the scheduling configuration, some additional edges should be introduced as depicted in Figure 5.3d. Those additional edges are depicted with dashed arrows as they should not be confused with the dependencies defined in the previous section. In fact, those additional edges on the ETG are only used to model the constraints imposed by the scheduler defined in each partition: σ_4^1 and σ_4^2, respectively. This leads to the following partial ordered set $S(m_4) = \{s_1 < s_4 < s_2 \leq s_7 < s_5 < s_3 \leq s_8 < s_6 < s_9\}$. When dependencies are satisfied, firings of actors mapped on ρ_4^1 can be executed in parallel to firings mapped on ρ_4^2. For example, $s_2 \leq s_7$ means that both s_2 and s_7 can be fired during the same clock cycle, as depicted in Figure 4.3d.

Buffer size independence

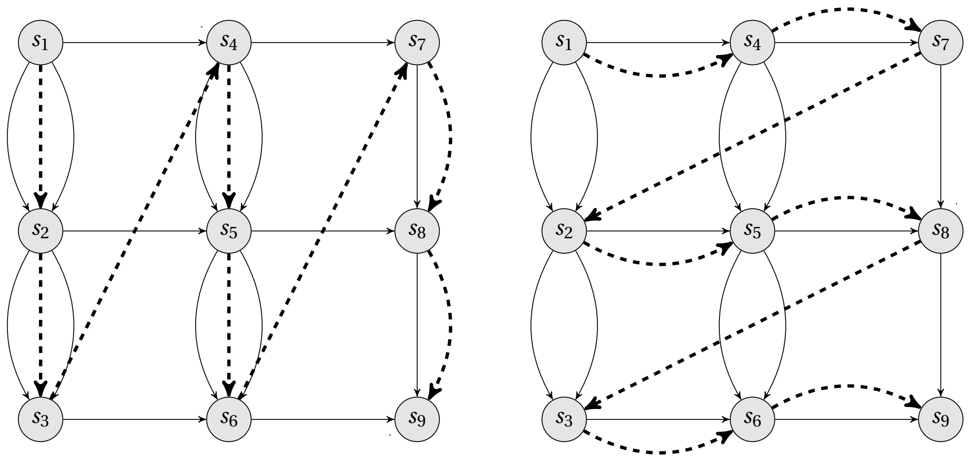
The same considerations about the ETG scheduling and partitioning independence can also be done for the buffer size configuration. Let's consider the two mapping configurations m_3 and m_4 defined in Table 4.1. In m_3 the buffer size configuration is defined as $\beta_3^1 = \beta_3^2 = 1$, contrary to m_4 where the buffer size configuration is defined as $\beta_3^1 = \beta_3^2 = 512$. The partitioning and scheduling configurations of m_3 andm_4 are the same. Considering m_3, the firings set S has been obtained with the following order $S(m_3) = \{s_1, s_4, s_2, s_7, s_5, s_3, s_8, s_6, s_9\}$ as depicted in the Gantt chart in Figure 4.3d. Following the same considerations made in Section 5.2.3, since the dependencies set $D(m_3)$ is the same as D, in this case the corresponding ETG is also the one depicted in Figure 5.2. This is because tokens are produced and consumed by the same firings. Hence no additional dependencies should be considered for $D(m_3)$, and as a consequence $D = D(m_3) = D(m_4)$. As such, changing the buffer size of a program (i.e. that is not time dependent) does not change the partial order of S imposed by D. This demonstrates that the ETG is independent from the buffer size used during the program execution.

5.3.3 Untimed

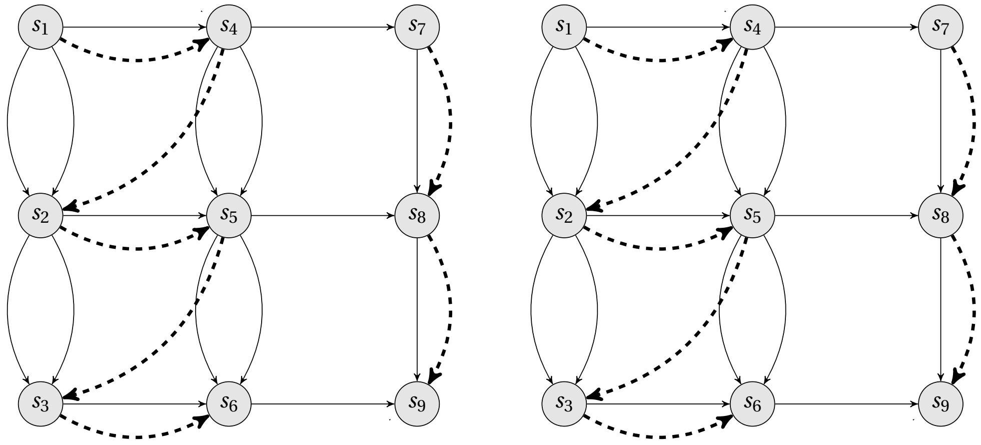
The ETG does not contain any information about timing of fired steps and dependencies. The only information that can be obtained is a partial ordering about firings. In other words, the dependency $(s_i, s_j) \in D$ defines only that s_j can only be fired after the complete firing of s_i. Let's consider the two mapping configurations m_5 and m_6, which differ only on how the buffer size configuration has been defined. As can be seen from the two Gantt charts depicted in Figure 4.3e and 4.3f, respectively, the firing of s_2 takes 2 clock cycles using the mapping configuration m_5 and 1 clock cycle using the mapping configuration m_6. However, this information is not defined in the ETG. Section 5.4 discusses how the ETG can be extended in order to define timing information for both the firings and dependencies.

5.3.4 Maximum parallelism

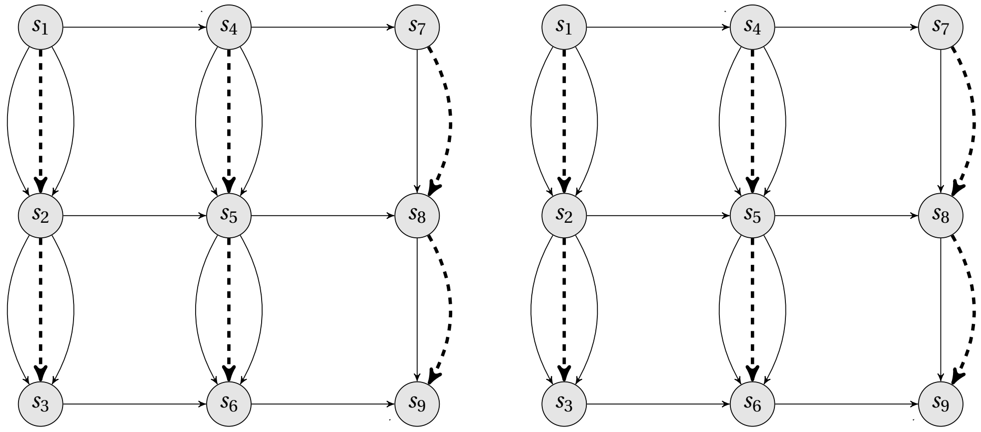
The ETG defines the maximum parallel execution that can be performed by the dataflow program. In fact, as described previously, it is completely independent from the mapping configuration. In other words, precedence relations about firings is imposed only by precedence about how data should be processed. For example a token dependency defines that the firing that consumes tokens can only be executed after the firing that produced those tokens has been fired. The same is for the other kind of dependencies. As such, the dependencies set D defines only a minimal information based on the data processing (i.e. tokens, internal variables) and resource utilizations (i.e. ports, guards) that should be respected in order to obtain a correct program execution. The constraints imposed by a particular mapping configuration can only be modeled by introducing additional edges as discussed in Section 5.3.2. The ETG without additional edges can be seen as the execution of the program using a fully-parallel mapping configuration (i.e. where each partition contains only one actor). Let's consider for example the mapping configuration m_6, where each actor is mapped in a separate partition. In this case the resulting ETG is depicted in Figure 5.3f where the additional edges imposed by the internal scheduler of the partition do not restrict the partial order of the ETG. This demonstrates that the ETG defined by S and D expresses the maximum parallelism of the application.



(a) Mapping configuration m_1 (b) Mapping configuration m_2



(c) Mapping configuration m_3 (d) Mapping configuration m_4



(e) Mapping configuration m_5 (f) Mapping configuration m_6

Figure 5.3: Execution Trace Graphs of the CAL network depicted in Fig. 2.10. Dashed lines represent additional edges that model a particular scheduling configuration defined within the mapping configurations described in Table 4.1.

5.3.5 Data dependent

The ETG can vary between two different program executions if this program contains at least one actor that is data dependent. For example, let's consider the CAL actor `Split` defined in Listing 2.2. This is composed of 2 actions: `A` and `B`, respectively. The firing conditions of both actions define that one input token should be available in the input port `I`. However, action `A` is fireable only if the token value is `val` ≥ 0 and action `B` if `val` < 0. Let's suppose that two input sequences are available in the input port `I`: $I_1 = \{0, 1, -10, -5\}$ and $I_2 = \{-1, -1, 0, -1\}$ respectively. Hence, the firing sequence $S = \{s_1, s_2, s_3, s_4\}$ of this actor defines different action firings as illustrated in Table 5.3. It must also be noted that the dependencies set D can change too. In this case, the program should be analyzed using different data sequences for generating representative ETG on which statistical analysis can be obtained (i.e. see Chapter 9 for an example on how stream applications that are data dependent are analyzed).

Table 5.3: Firings sequence of the CAL actor `Split` defined in Listing 2.2 when two input sequences are available in its input port `I`: $I_1 = \{0, 1, -10, -5\}$ and $I_2 = \{-1, -1, 0, -1\}$, respectively.

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

5.3.6 Modeling a dynamic program execution

The execution of dynamic actors such that the execution is mapping dependent can be modeled using the ETG. The CAL actor used to prove this property is the `GuardedInverter` actor which is illustrated in Listing 5.1. This actor is composed of 2 actions `A` and `B`, an input port `I`, an output port `O` and an internal actor variable `m`. The priority condition `B > A` is defined: this lead the action `A` to be fireable each time that the action `B` is not. It is important to note that action `B` is fireable each time there is at least one input token in its input port `I` and the guard condition `m > 0 and m <3` is satisfied. The state variable on which this guard is defined is modified only by the action `A`: consequently, only `A` can enable or disable the guard. Two possible execution paths are illustrated in Figure 5.4a and Figure 5.4b. The axis of abscissae defines the time flow for action `B`, similarly the axis of ordinates defines the time flow for action `A`. It is supposed that each firing of both `A` and `B` requires the same amount of time. The list of firings and the respective value assumed by the internal variable `m` and the guard condition are reported in Table 5.4a and Table 5.4b, respectively. For this example, two regions where the guard of `B` is enabled can be identified along the `A`-axis of the execution path: these are called the **guard enable window** $n = 1$ and $n = 2$ respectively. In those regions, `B` can be executed if there is at least one input token in its input port `B`. It is now clear how the execution path of a dynamic program can vary according to the mapping configuration used for the execution. In the following it is clarified how it is possible to model such kinds of enabling and disabling windows and make the entire analysis process unaware of the mapping configuration.

Listing 5.1: GuardedInverter.cal

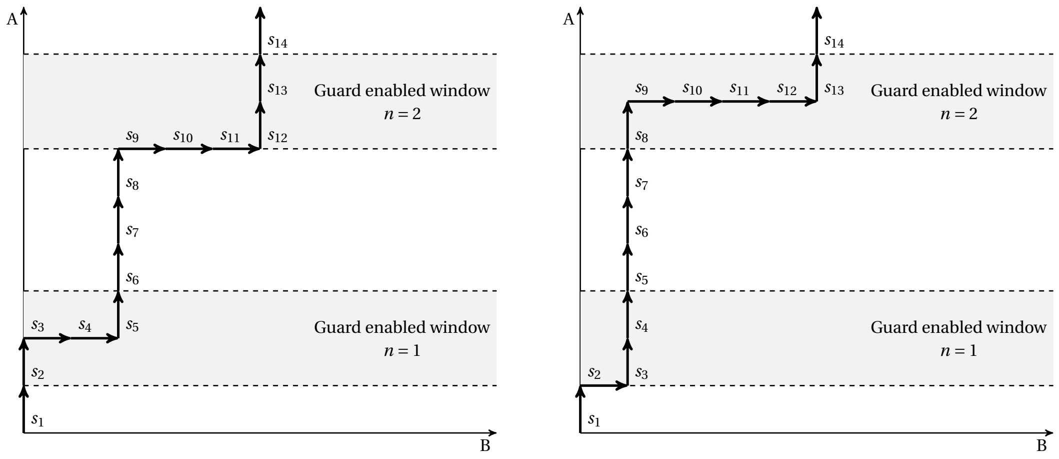
```
actor GuardedInverter() int I ==> int O :

    int m := 0;

    A: action  ==>
    do
        m := m + 1;
        if m = 5 then m := 0; end
    end

    B: action I:[val] ==> O:[-val]
    guard m > 0 and m < 3
    end

    priority:
        B > A;
    end

end
```

Using guard enable and disable dependencies

Even though the two previous paths are equivalent (i.e. they end at the same internal actor state configuration as illustrated in Table 5.4), not considering the enable and disable guard dependencies makes the ETG dependent to the mapping configuration used during the execution. This can be seen from the ETG obtained from the two execution paths depicted in Figure 5.4c and Figure 5.4d, where both guard enable and guard disable dependencies are not considered. Considering for example the first ETG depicted in Figure 5.4c, the firing s_3 can be executed when the guard is enabled. In other words it is possible to fire s_3 after the execution of s_1 and before the execution of s_5, but also after the execution of s_8 and before the execution of s_{13}. This can be argued for each firing of `B`. In other words, it is possible to identify two equivalent constraints on the partial order of the ETG: $s_1 < s_b < s_5$ and $s_8 < s_b < s_{13}$ where s_b identifies any firing of `B`. These two conditions can be modeled with the guard enable and disable dependencies as illustrated in Figure 5.5. Each guard enable and disable dependency is coupled with an appearance order that identifies which guard enabling window is modeled.

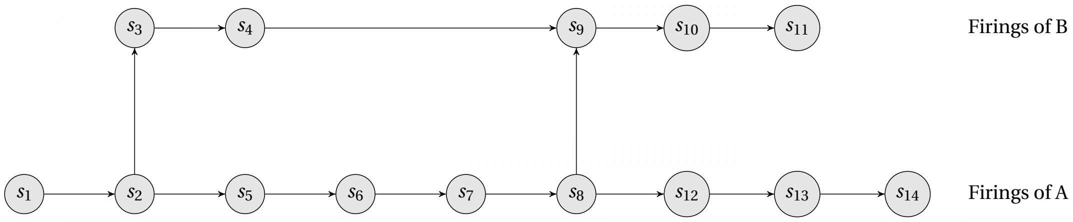
Removing cyclic paths

However, this kind of dependency cannot be considered as strict dependency, otherwise it would potentially cause the graph to become cyclic. Consequently, the ETG would not represent a po-space and by consequence a trace space. This happens when we consider a

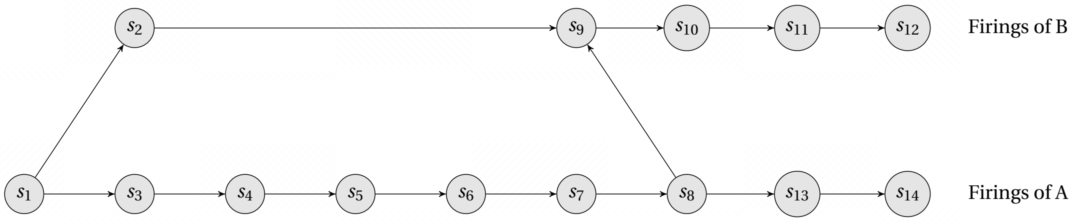


(a) A first possible execution path.

(b) A second possible execution path.


s_3 s_4 s_9 s_{10} s_{11}
Firings of B
s_1 s_2 s_5 s_6 s_7 s_8 s_{12} s_{13} s_{14}
Firings of A

(c) The execution trace graph corresponding to the execution path of Figure 5.4a without considering the guard enable and disable dependencies.


s_2 s_9 s_{10} s_{11} s_{12}
Firings of B
s_1 s_3 s_4 s_5 s_6 s_7 s_8 s_{13} s_{14}
Firings of A

(d) The execution trace graph corresponding to the execution path of Figure 5.4b without considering the guard enable and disable dependencies.

Figure 5.4: Two possible execution paths of the `GuardedInverter` actor illustrated in Listing 5.1. The corresponding execution trace graphs do not take into account the guard enable and disable dependencies.

Table 5.4: Firings with the corresponding internal variable and guard values for the execution trajectories and graphs depicted in Figure 5.4.

(a) Firings for the execution trajectory and graph depicted in Figure 5.4a and 5.4c, respectively.

Firing	**Action**	**`m` value**		**Guard status**	
		initial	**final**	**initial**	**final**
s_1	A	0	1	disabled	enabled
s_2	A	1	2	enabled	-
s_3	B	2	-	-	-
s_4	B	-	-	-	-
s_5	A	-	3	-	disabled
s_6	A	3	4	disabled	-
s_7	A	4	0	-	-
s_8	A	0	1	-	enabled
s_9	B	1	-	enabled	-
s_{10}	B	-	-	-	-
s_{11}	B	-	-	-	-
s_{12}	A	-	2	-	-
s_{13}	A	2	3	-	disabled
s_{14}	A	3	4	disabled	-

(b) Firings for the execution trajectory and graph depicted in Figure 5.4b and 5.4d, respectively.

Firing	**Action**	**`m` value**		**Guard status**	
		initial	**final**	**initial**	**final**
s_1	A	0	1	disabled	enabled
s_2	B	1	-	enabled	-
s_3	A	-	2	-	-
s_4	A	2	3	-	disabled
s_5	A	3	4	disabled	-
s_6	A	4	0	-	-
s_7	A	0	1	-	enabled
s_8	A	1	2	enabled	-
s_9	B	2	-	-	-
s_{10}	B	-	-	-	-
s_{11}	B	-	-	-	-
s_{12}	B	-	-	-	-
s_{13}	A	-	3	-	disabled
s_{14}	A	3	4	disabled	-

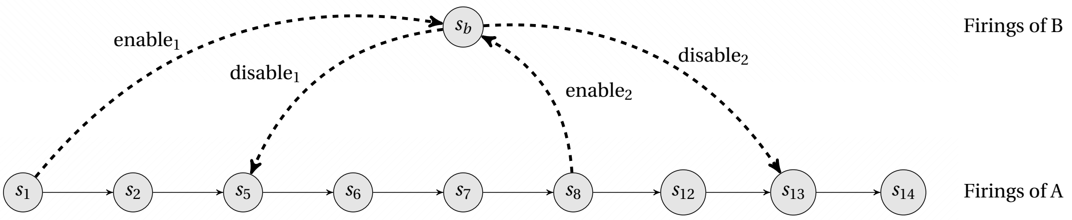


Figure 5.5: Guard enable and disable dependencies couples that model the guard enable windows $n = 1$ and $n = 2$ depicted in Figure 5.4. The firing s_b represents a generic firing of the action `B`.

path with a $(n+1)$ guard enable and a n guard disable (i.e. where n represents the appearance order of the enabling window). Consequently, when analyzing the dependency graph for each executed guarded action `B` only one of the available guard enable and disable couples with the same appearance order shall be taken into account (i.e. the others are discarded). For the previously described example, the two ETGs that model the two execution paths illustrated in Figure 5.4 are depicted in Figure 5.6.

It must be noted that, using this formalism lets the two ETGs illustrated in Figure 5.6 be defined as equivalent. In fact, both ETGs can model the first or the second execution path by choosing the appropriate guard enable and disable couple.

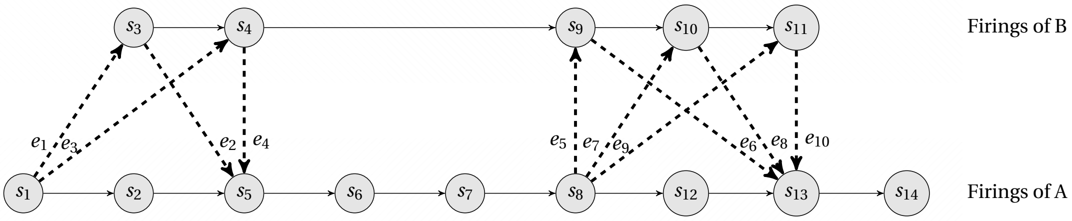
5.4 Timed execution trace graph

Time information is added to an ETG by defining for each firing and each dependency a corresponding time value. For this purpose, the ETG is transformed to a **weighted graph** which is a special type of labeled graph where labels are numbers (for this specific case, always positive) called **weights**.

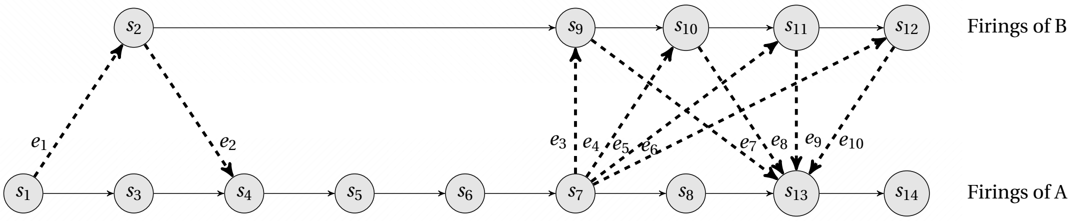
The **timed execution trace graph** (TETG) is formally defined extending the notation of the ETG as a DAG(S, D, Ψ_S, Ψ_D) where:

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

In other words, for each firing $s_i \in S$ is assigned a time value called *firing weight* and defined as $w(s_i) \geq 0$. Similarly, the *dependency weight* $w(s_i, s_j) \geq 0$ is defined for each dependency $(s_i, s_j) \in D$.



(a) The execution trace graph corresponding to the execution path of Figure 5.4a considers a couple of the guard enable and disable dependencies for each firing. The guard enable and disable couples (e_1, e_2) and (e_3, e_4) are used to model the firing of s_3 and s_4, respectively, on the guard enable window $n = 1$. The guard enable and disable couples (e_5, e_6), (e_7, e_8) and (e_9, e_{10}) are used to model the firing of s_9, s_{10} and s_{11}, respectively, on the guard enable window $n = 2$.



(b) The execution trace graph corresponding to the execution path of Figure 5.4b considers a couple of the guard enable and disable dependencies for each firing. The guard enable and disable couple (e_1, e_2) is used to model the firing of s_2 on the guard enable window $n = 1$. The guard enable and disable couples (e_3, e_4), (e_5, e_6), (e_7, e_8) and (e_9, e_{10}) are used to model the firing of s_9, s_{10}, s_{11} and s_{12}, respectively, on the guard enable window $n = 2$.

Figure 5.6: The ETGs related to the execution paths depicted in Figure 5.4a and Figure 5.4b where for each firing of `B` a couple of guard enable and disable has been considered in order to model the guard enabled windows $n = 1$ and $n = 2$.

5.4.1 Firing weight

The firing weight $w(s_i)$ models the time required for entirely executing the action firing s_i. In other words, using the action execution model discussed in Section 2.5.2, $w(s_i)$ should model the time required not only for executing the algorithmic part of the fired action, but also the time required for reading and writing the input and output tokens. Therefore, $w(s_i)$ can be de defined as the combination of five terms, that are respectively:

- **Wait for available input tokens**: models the waiting time of s_i for the availability of all its input tokens (i.e. blocking reading).
- **Read input tokens**: models the time required by s_i for reading all its input tokens.
- **Algorithmic part execution**: models the time required by s_i for executing the action algorithmic part.
- **Wait for available output space**: models the waiting time of s_i for the availability of the

necessary output token places (i.e. blocking writing).

- **Write output tokens**: models the time required by s_i for writing all its output tokens.

These terms can vary according to the mapping configuration chosen for the application implementation. Using the formalism illustrated in Section 4.2.3 where the mapping configuration has been defined as a 3-tuple (σ, ρ, β), $w(s_i)$ can be defined as:

$$w(s_i) = f(s_i, \rho, \beta) \tag{5.9}$$

where f is only a function of the partitioning and the buffer size configurations.

Linear model

The firing weight model of Equation 5.9 can be simplified as a linear combination of terms as:

$$w(s_i) = w(s_i)_{rd} + w(s_i)_r + w(s_i)_e + w(s_i)_{wd} + w(s_i)_w \tag{5.10}$$

where the meaning of each term is summarized on Table 5.5. Some examples of different techniques that can be used to measure or estimate these terms are discussed in Section 8.1.

Table 5.5: Firing weight parameters for the linear model of Equation 5.10.

	Parameter	Description
$w(s_i)_{rd}$	Wait for available input tokens	waiting time of s_i for the availability of all its input tokens (i.e. blocking reading)
$w(s_i)_r$	Read input tokens	time required by s_i for reading all its input input
$w(s_i)_e$	Algorithmic part execution	time required by s_i for executing the action algorithmic part
$w(s_i)_{wd}$	Wait for available output space	waiting time of s_i for the availability of the necessary output token places (i.e. blocking writing)
$w(s_i)_w$	Write output tokens	time required by s_i for writing all its output tokens

5.4.2 Dependency weight

The dependency weight $w(s_i, s_j)$ models the time required to make the dependency $(s_i, s_j) \in D$ available to the target firing step s_j after the execution of the firing s_i has been completely performed. Consequently, this value may depend on the particular mapping configuration

$m = (\sigma, \rho, \beta) \in M$ and it is defined as:

$$w(s_i, s_j) = f(s_i, m) = f(s_i, \sigma, \rho, \beta) \tag{5.11}$$

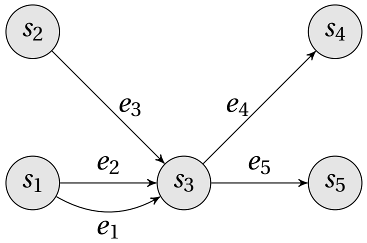
where f is a function of the scheduling, the partitioning and the buffer size configurations. Depending on the kind of (s_i, s_j), this weight may model different factors. For example, if $(s_i, s_j) \in D_t$ is a token dependency then $w(s_i, s_j)$ can model the time required by the buffer to receive and make the corresponding tokens available. The same considerations can be made for state variable dependencies $(s_i, s_j) \in D_v$ where the token is now a state variable and the buffer a local memory region. So, considering a read/write internal variable dependency, the weight corresponds to the time required for reading and storing the updated value of that internal variable. Similarly, if $(s_i, s_j) \in D_f$ is a finite state machine dependency then $w(s_i, s_j)$ defines the time required by the internal actor scheduler to select the specific action firing. Furthermore, when additional fictitious dependencies are introduced to model the scheduling configuration ρ, the weight of these fictitious dependencies models the time required for the partition scheduler to select the corresponding actor, as discussed in Section 8.1.

5.5 Transformations

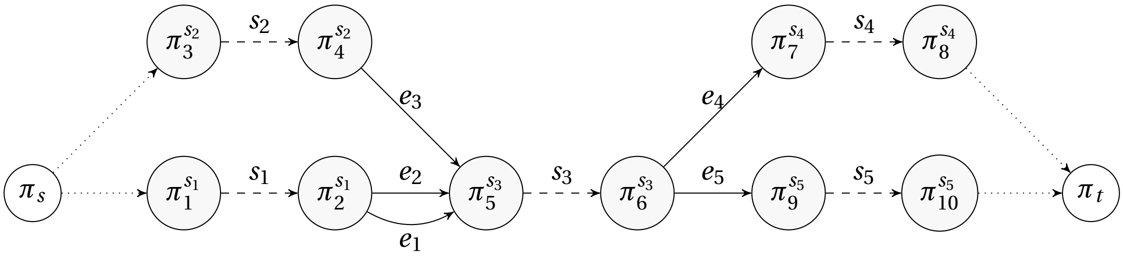
In the following some ETG transformations are discussed. These represent an overview of the main graph-based transformations that can be applied to an ETG. These are extensively used in the rest of this dissertation when the ETG is used to explore the design space of a dataflow program.

5.5.1 Firing expansion

The firing expansion of an ETG is a new DAG(V, E) where the set of vertexes is evaluated defining for each firing $s_i \in S$ two new vertexes $\pi^{s_i}_{2i-1} \in V$ and $\pi^{s_i}_{2i} \in V$, respectively, connected by a directed edge $(\pi^{s_i}_{2i-1}, \pi^{s_i}_{2i}) \in E$. Moreover, each dependency $(s_i, s_j) \in D$ is transformed to a new directed edge $(\pi^{s_i}_{2i}, \pi^{s_j}_{2j-1}) \in E$. As an example, Figure 5.7 depicts the transformation of an ETG. It can be seen how, for each firing $s_i \in S$, the corresponding $\pi^{s_i}_{2i-1} \in V$ inherits the incoming dependencies, similarly the corresponding $\pi^{s_i}_{2i} \in V$ inherits the outgoing dependencies. Furthermore, it is possible to define two new fictitious vertexes π_s and π_t called the source and sink vertex of $G(V, E)$, respectively. For each vertex such that $\delta(\pi^{s_i}_{2i-1})^-_S = \emptyset$ (i.e. that has no incoming edges) a new fictitious edge $(\pi_s, \pi^{s_i}_{2i-1}) \in E$ is defined. Similarly, for each vertex such that $\delta(\pi^{s_i}_{2i})^+_S = \emptyset$ (i.e. that has no outgoing edges) a new fictitious edge $(\pi^{s_i}_{2i}, \pi_t) \in E$ is defined.



(a) Initial execution trace graph.



(b) Expanded version.

Figure 5.7: Firings expansion of an execution trace graph.

5.5.2 Dependency amalgamation

When analyzing the ETG, it is possible that the only requirement is to know which are the set of predecessors and successors given firing (i.e. see Equation (5.3) and Equation (5.6), respectively). Consequently, all the information contained in the dependencies set D can be redundant as two firings s_i and s_j are related with more than one dependence. Let $D_A = \{e_1, e_2, \dots e_n\} \subseteq D$ any subset of dependencies having the same endpoints. The **multi-dependency amalgamation** (i.e. also called multi-edge amalgamation [155]) corresponding to D_A is an ETG that results from *merging* (amalgamating) all of the dependencies in D_A into a single and unlabeled dependency generally denoted with $e^\bullet = e_1 \bullet e_2 \bullet \dots \bullet e_n$. The set of amalgamated dependencies is denoted as $D^\bullet$. Informally, the amalgamation can be see as a non-minimal transitive reduction of a graph.

As an example, the ETG depicted in Figure 5.2, e_1 and e_2 have the same endpoints s_1 and s_2. Hence, e_1 and e_2 can be amalgamated as $e^\bullet = e_1 \bullet e_2$. In the same ETG other dependencies can be amalgamated as illustrated in Figure 5.8.

5.5.3 Event-driven system representation

This section illustrates a methodology for converting the ETG of a dataflow program into a discrete event system in the form of a Petri net (PN) [156, 157]. This conversion supports a more systematic development of design space exploration heuristics based on the application of automatic control methodologies (in this regard, see Chapter 6).

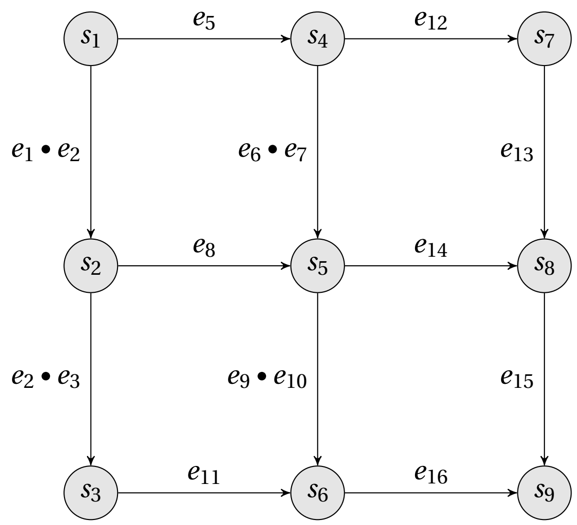


Figure 5.8: Amalgamation of the execution trace graph illustrated in Figure 5.2.

Petri nets

A PN is a particular kind of bipartite directed graph made up of three types of objects: **places**, **transitions**, and **directed arcs** (for a complete overview about PN see Appendix A.1). Directed arcs connect places to transitions or transitions to places. Each place can contain *tokens*: the presence or the absence of a token can indicate whether a condition associated with this place is true or false. Formally, a PN is defined as a tuple $N(P, T, I, O, M_0)$, where:

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions.
- $I : P \times T$ is the pre-incidence matrix that defines directed arcs from places to transitions.
- $O : T \times P$ is the post-incidence matrix that defines directed arcs from transitions to places.
- M_0 is the initial marking of places.

The execution of a PN is controlled by the number and distribution of tokens over the places. Similarly to the DPN with firings dataflow model, a PN also executes by firing transitions governed by enabling and firing rules. In a PN a transition t can be enabled if all its input places contain at least a number of tokens equal to the weight of the respective directed arcs. The firing of an enabled transition removes from each input place the number of tokens equal to the weight of the respective input directed arc and deposits in each output place a number of tokens equal to the weight of the respective directed output arc. Mathematically, firing the transition t at event k yields a new marking:

$$M(p,k) = M(p,k-1) - I(p,t) + O(t,p), \ \forall p \in P \qquad (5.12)$$

for any $p \in P$ at each firing instant $k \in \mathbb{N}$.

Why not transform the dataflow program directly to a PN?

When a dynamic dataflow program belonging to the DPN class is translated into a PN representation, it is in general required that the MoC of the resulting PN is modified accordingly [158]. This may imply, for instance, the use of a colored PN that allows tokens to carry values and that preserves the order in their respective places (see, in this regard, [159, 160]). However, a more effective approach is to directly transform the ETG into a PN. In such cases, the objective is to obtain a mathematical description of the behavior of an ETG similar to the one provided by Equation (5.12). A systematic approach to reach such objectives is to correlate the dependency constraints defined in the ETG with the firing rules of the PN.

ETG to PN transformation

Intuitively, an ETG action firing $s_i \in S$ can be represented as a PN transition $t_i \in T$ that can be fired only if there are enough input tokens at its incoming places $p \in P$. Similarly, each ETG dependency $(s_i, s_j) \in D$ can be represented as a PN place $p \in P$, for which the place weight $W(p,t)$ is defined as the number of tokens n_t (i.e. expressed by the tokens dependency) if $(s_i, s_j) \in D_t$ or unitary otherwise (i.e. $(s_i, s_j) \in D \backslash D_t$). Furthermore, defining $T_{\emptyset^-} \subseteq T$ as the set of transitions such that the respective ETG action firings are contained in $S_{\emptyset^-}$ (i.e. sources of the ETG, as defined in Equation (5.4)), an additional *fictive* incoming place with unitary weight must be defined for each of those transitions. The set of fictive transitions is referred to as $P_{\emptyset^-}$. In order to model the fact that only the transitions contained in $S_{\emptyset^-}$ are initially enabled (i.e. they do not depend on the firing of any other transition), one token is defined as the initial marking only for the places contained in $P_{\emptyset^-}$ (i.e. $M_0(p) = 0, \forall p \in P$ and $M_0(p) = 1, \forall p \in P_{\emptyset^-}$). It must be noted that the ETG amalgamation transformation illustrated in Section 5.5.2 can be applied in order to reduce the number of equivalent PN places. The only requirement is that token dependencies should not be amalgamated.

In conclusion, an ETG is formally transformed to its equivalent PN as follows:

$$\begin{array}{lcll} T & : & s_i \mapsto t_i \in T & \forall s_i \in S \\ P & : & (s_i, s_j) \mapsto p \in {}^\bullet t_i \cup t_j^\bullet & \forall (s_i, s_j) \in D \\ P_{\emptyset^-} & : & p \in {}^\bullet t_i & \forall t_i \in T_{\emptyset^-} \end{array} \tag{5.13}$$

where for each PN place the weight is defined such as:

$$W(p, t_i) = \begin{cases} n_t & \text{if } (s_i, s_j) \in D_t \\ 1 & \text{otherwise} \end{cases}$$

where n_t is the number of tokens defined in the token dependency. The initial PN marking is defined as:

$$M_0(p) = \begin{cases} 1 & \text{if } p \in P_{\emptyset^-} \\ 0 & \text{otherwise} \end{cases} \tag{5.14}$$

The just-introduced transformation allows the representation of the behavior of a dataflow program by means of an event-driven system. In this regard, Equation (5.12) can be revisited in order to describe the evolution of the variable of interest of the Petri net and, in turn, of the program. More precisely, introducing the **incidence matrix** of the net defined as $A(t,p) = O(t,p) - I(t,p)$, i.e. $A = O - I$, Equation (5.12) can be rewritten in the following more compact form (see e.g. [157]) as:

$$M(k+1) = M(k) + Au(k) \tag{5.15}$$

where $u(k)$ is a $n \times 1$ column vector with 1 as its i-th entry and 0 in the remaining $n-1$ positions denoting that only the i-th transition t_i fires at event k and $M(k)$ and $M(k+1)$ are, respectively, the marking vectors of the net before and after the firing occurrence. Equation (5.15) is usually referred to as the **state equation** of the net and can be augmented by an **output relation** of the form:

$$y(k+1) = CM(k+1) \tag{5.16}$$

in order to highlight suitable variables of interest which can be expressed as (linear) functions of the tokens actually in the net places. The state equation description of a Petri net and, more generally, of an event-driven system is needed when performance optimization of such systems has to be achieved through theoretic control approaches [161]. Thus, the use of the above-defined transformation can be regarded as an effective and systematic way to extend the use of such approaches to all the signal processing applications that can be casted within the considered dataflow programming framework.

Example

As an example, the just-introduced ETG to PN transformation can be applied to the ETG of Figure 5.8. The PN structure depicted in Figure 5.9 is obtained. It must be noted that, as required by Equation 5.13, token dependencies of the ETG have not been amalgamated. More over, the only fictitious place is $p_1 \in P_{\emptyset^-}$ where the initial marking is one token. Places that correspond to a token dependency (i.e. p_4, p_5, p_6, p_9, p_{10}, p_{11}) are denoted with a grey background.

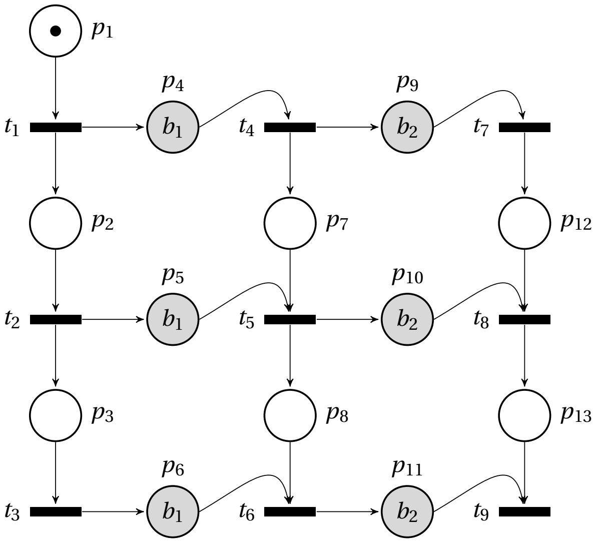


Figure 5.9: Petri net obtained from the execution trace graph depicted in Figure 5.2.

In this case, the incidence matrix A in Equation (5.15) is given by:

$$A = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

and the initial marking, according to Equation (5.14), is defined as:

$$M_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}'$$

where $[\cdot]'$ denotes the matrix transpose operator. Moreover, supposing that the variable of interest is the number of tokens stored in each buffer of the considered dataflow network (i.e.from Figure 2.5.4, b_1 and b_2, respectively), matrix C describing the output relation (5.16) is defined as:

$$C = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

5.6 Conclusions

In this chapter the notion of execution trace graph (ETG) of a dataflow program has been formalized. It has been shown how the ETG represents a graph-based structure of the execution of a dataflow program. The execution of a dataflow program has been modeled as a directed acyclic graph where nodes represent single action firings and edges represent (data or functional) dependencies between two different action firings. Notions of partially-ordered sets (i.e. po-sets) and directed paths (i.e. d-paths) have been adapted to this execution model. Different dependency kinds have been defined, notably the finite state machine dependencies, the internal variable dependencies, the port dependencies, the tokens dependencies and the guard dependencies. The importance of the guard (enable and disable) dependencies in the context of dynamic dataflow programs has been discussed. In fact, by the use of this kind of dependency it has been demonstrated how different execution trajectories can be modeled by using the ETG obtained through a serial program exception. Furthermore, the main properties of the ETG have been illustrated with some examples demonstrating how this graph-based representation is totally mapping independent and can be used to effectively estimate the design performance through a post-mortem analysis. Finally, some ETG transformations have been illustrated. For example, the transformation of the ETG to an event-driven system has shown how the DSE can be made by the use of advanced control techniques.

6 TURNUS: a design space exploration environment for CAL programs

In this section the main functionalities and the iterative design flow of TURNUS [16, 17, 19, 20] are presented. This is a DSE environment for dynamic dataflow programs. Compared to the state of the art exploration tools illustrated in Section 4.3, the novel features include both the possibility to estimate the design performance and to explore and optimize the design space based on the analysis of the ETG presented in Chapter 5. Moreover, it provides an application programming interface (API) to profile CAL programs that is usable by third-party dataflow compilers. In the following, the design flow together with the high-level models of the dataflow program and the architecture are illustrated. Furthermore, it is illustrated how this environment can be integrated with already existing dataflow environments.

6.1 Design flow features and capabilities

An overview of the TURNUS iterative design flow is depicted in Figure 6.1. This DSE environment is composed of two main blocks: the TURNUS profiler, and the TURNUS ETG post-mortem scheduling and analysis. The TURNUS profiler is used in the first stages of the DSE for evaluating both the ETG and the high-level profiling information of a CAL program. Successively, the ETG is post-mortem scheduled in order to estimate the design performance and explore and optimize the design space of the program. Results of the DSE can be used by the designer that is informed of which parts of the CAL program should be restructured, and by third-party tools that implement the program on the mapped target architecture.

6.1.1 Profiler

The TURNUS profiler is used on top of a CAL compiler infrastructure where the source code is interpreted. It provides a set of application programming interfaces (API) that are used to make a high level profiling analysis of the code. The profiler API is illustrated in Section 7.4. The only input of the profiler is the CAL program description. This contains the CAL program input description, which is defined by the CAL project and its collection of source code files. After the profiled simulation, a high-level profiling data file is generated. This contains profiling information concerning the workload and the buffer size utilization. Furthermore, using the high-level profiling information, the ETG file is generated. The collection of profiling data and how these are used to evaluate the ETG is illustrated in Section 7.2.

6.1.2 Execution trace graph post-mortem scheduling and analysis

The iterative DSE is performed by analyzing the ETG generated by the profiler. The ETG is used for both the design performance estimation and the exploration of the design space. At this step both the program and architecture model (i.e. defined as illustrated in Section 6.2) are used in order to estimate and define the possible design points. The performance estimation can be enhanced by using clock-accurate profiling information retrieved by third-party profilers (e.g. GnuProf, Valgring, ModelSim). The design space can be constrained using the constraints information provided by the designer. Examples of available design space optimization are the possibility to minimize and optimize the buffer size configuration, partition the program on many-cores, and reduce the dynamic power dissipation. At this stage, the following analysis can be done:

- **Performance estimation**: illustrated in Section 8.1, is used to estimate the design performance for a given mapping configuration. The ETG post-mortem scheduler is based on a discrete event simulator. The main functionality is to assign timing weight to each firing and dependency of the ETG. The timed ETG is then used by the underlying analysis provided by the framework.
- **Critical path evaluation**: illustrated in Section 8.2, is used to evaluate the critical path of an application. It defines what is called design space critical path, which is used to define bounds on the design space of the application.
- **Impact analysis**: illustrated in Section 8.3, is used to provide the code refactoring directions to the designer. It provides a list of actor and actions where code refactoring should be concentrated.
- **Buffer dimensioning**: illustrated in Section 8.4, provides a collection of heuristic algorithms for estimating a feasible bounded buffer size configuration. Furthermore, a solution for the problem of maximizing the application throughput and, at the same time, minimizing the buffer size configuration is presented.
- **Partitioning**: illustrated in Section 8.5, provides a collection of heuristic algorithms tailored for partitioning the application on multi-clock domains architectures. The main requirements are that the application throughput is maximized and, at the same time, the dynamic power dissipation is minimized.

The design space can be iteratively explored. For each iteration a mapping configuration file is generated. This is used to drive third-party dataflow tools (e.g. low-level code generators) during the design implementation stages. Section 6.3 illustrates a list of tools already integrated with this framework.

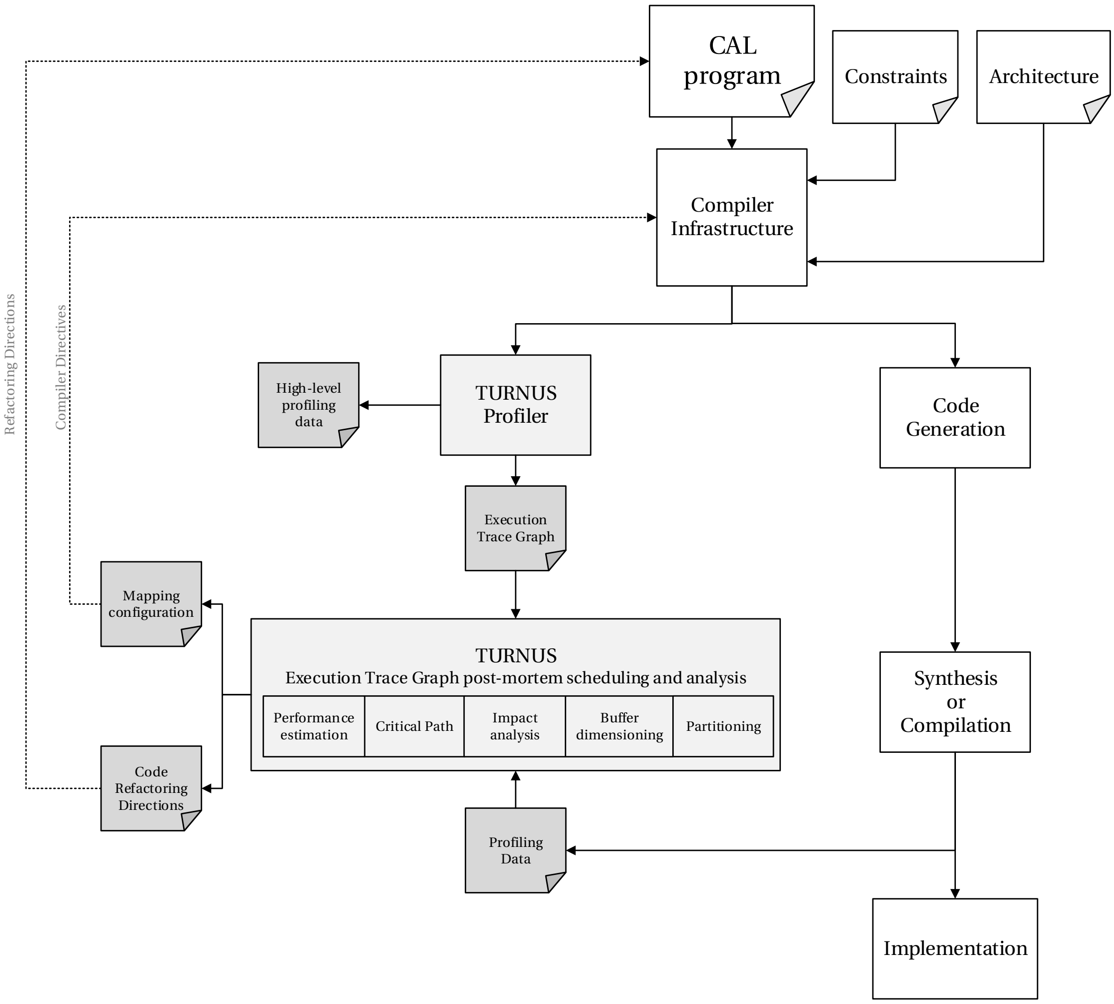


Figure 6.1: TURNUS design flow.

6.2 High-level models

In the following, the high-level models used in the framework to represent the CAL dataflow application, the target architecture, the ETG and the program profiling information are illustrated. These are presented using a (simplified) unified modeling language (UML) representation that respects the framework APIs available in [16].

6.2.1 CAL dataflow program

The CAL dataflow program model describes the basic structure of the program. A specific meta model representation is used in order to extend the interoperability of CAL tools already available. A CAL code compiler infrastructure that wants to make use of the TURNUS framework should wrap its intermediate representation and generate a consistent program model. The basic components of this representation are described in the following section. The formalism that is used is the same as the one illustrated in Section 2.5.

Network

The `Network` object is used to model a dataflow program network $N(A,B)$. As depicted in Figure 6.2, this object is defined by the following elements:

- **id**: `String` element that identifies the network under analysis.
- **sourceFile**: `String` attribute that contains the relative source file path of the network (i.e. the `.xdf` or `.nl` file name).
- **version**: `Version` element that contains the versioning information of the source file.
- **project**: `String` attribute that contains the name of the CAL project where the source file is stored.
- **classes**: list of `ActorClass` elements contained in the network.
- **actors**: list of `Actor` elements contained in the network.
- **buffers**: list of `Buffer` elements contained in the network.

Actor-class

The `ActorClass` object is used to model an actor-class $\kappa \in K$. As depicted in Figure 6.3, this object is defined by the following elements:

- **name**: `String` attribute that identifies the actor-class.

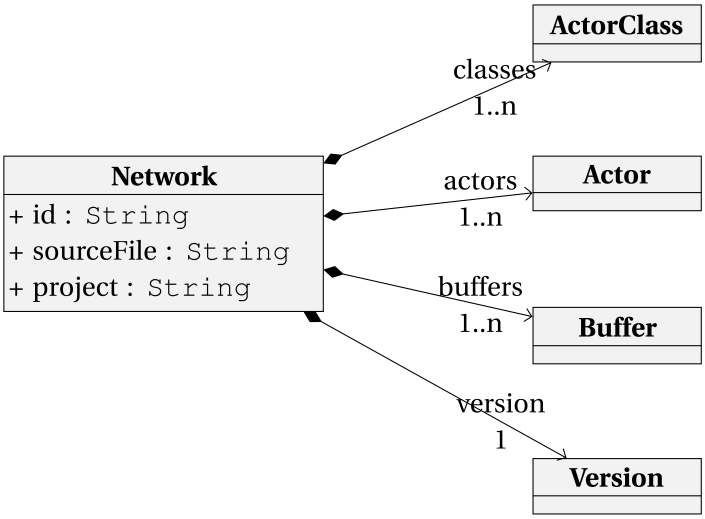


Figure 6.2: The `Network` object.

- **nameSpace**: `String` attribute used to represent the level of hierarchy of the source file.
- **sourceFile**: `String` attribute that contains the relative source file path of the actor-class.
- **version**: `Version` element that contains the versioning information of the source file.
- **Actions**: list of `Action` elements contained in the actor-class.
- **inputPorts**: list of input `Port` elements contained in the actor-class.
- **outputPorts**: list of output `Port` elements contained in the actor-class.
- **variables**: list of `Variable` elements contained in the actor-class.
- **procedures**: list of `Procedure` elements contained in the actor-class.

It must be noted that the concatenation of the name space and the name cannot be shared among actor-classes defined on the same `Network`.

Actor

The `Actor` object is used to model an actor $a \in A$. As depicted in Figure 6.4, this object is defined by the following elements:

- **id**: `String` attribute that identifies the actor. It must be noted that the same id cannot be shared among actors of the same network.
- **actorClass**: `ActorClass` element that is instantiated by the actor.

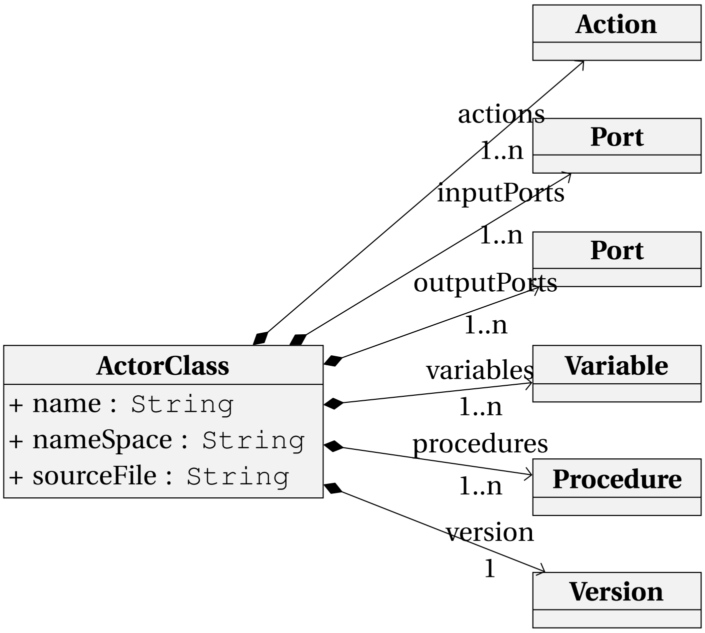


Figure 6.3: The `ActorClass` object.

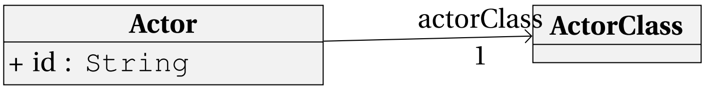


Figure 6.4: The `Actor` object.

Action

The `Action` object is used to model an action $\lambda \in \Lambda$. As depicted in Figure 6.5, this object is defined by the following elements:

- **id**: `String` attribute that identifies the action. It must be noted that the same id cannot be shared among actions of the same actor.
- **label**: `Qid` element that contains the qualifier identifier of the action.
- **guards**: list of the `Guard` elements used by the action.
- **procedure**: list of `Procedure` elements used by the action.
- **variables**: list of `Variables` elements used by the action.

It must be noted that, even though actions are defined in the `ActorClass`, these are always considered by the framework as a tuple (`Actor,Action`) when analyses are performed.

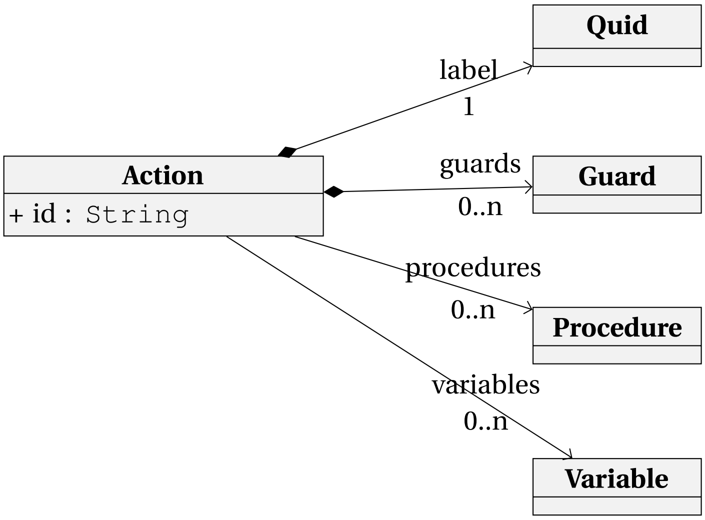


Figure 6.5: The `Action` object.

Qid

The `Qid` object is used to model a qualifier identifier, which is a sequence of identifiers separated by a dot. As depicted in Figure 6.6, this object is defined by the following elements:

- **ids**: array of `String` elements that contains the ordered sequences of identifiers.
- **size**: `Integer` attribute that defines the size of the identifier in terms of elements in the *ids* array.

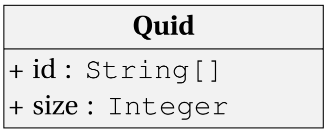


Figure 6.6: The `Quid` object.

Procedure

The `Procedure` object is used to model a procedure (or a function) defined in an actor-class and called by an action. As depicted in Figure 6.7, this object is defined by the following elements:

- **name**: `String` attribute that identifies the procedure. It must be noted that the same name cannot be shared among procedures of the same actor-class.
- **variables**: list of `Variable` elements used by the procedure.

It must be noted that, even though procedures are defined in the `ActorClass`, these are always considered by the framework as a tuple (`Actor`,`Procedure`) when analyses are performed.

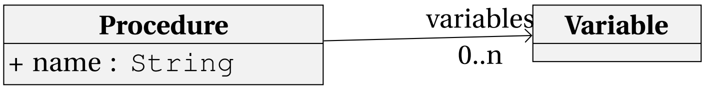


Figure 6.7: The `Procedure` object.

Internal actor variable

The `Variable` object is used to model an internal variable. As depicted in Figure 6.8, this object is defined by the following elements:

- **name**: `String` attribute that identifies the actor internal variable. It must be noted that the same name cannot be shared among variables of the same actor-class.
- **type**: `Type` element that contains the variable type.

It must be noted that, even though variables are defined in the `ActorClass`, these are always considered by the framework as a tuple (`Actor`,`Variable`) when analyses are performed.

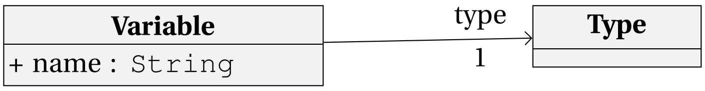


Figure 6.8: The `Variable` object.

Guard

The `Guard` object is used to model an action guard. As depicted in Figure 6.9, this object is defined by the following elements:

- **id**: `String` attribute that identifies the guard. It must be noted that the same id cannot be shared among guards of the same action.
- **variables**: list of `Variable` elements used by the guard.
- **ports**: list of input `Port` elements used by the guard.

It must be noted that, even though guards are defined in the `Action`, these are always considered by the framework as a tuple (`Actor`,`Action`,`Guard`) when analyses are performed.

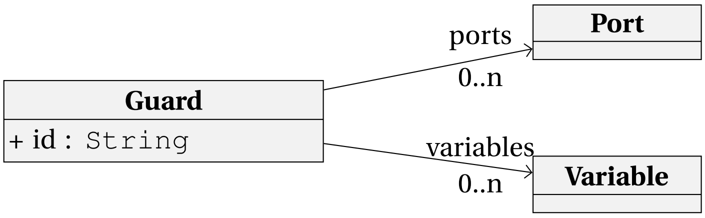


Figure 6.9: The `Guard` object.

Port

The `Port` object is used to model an input port $p_i^{in} \in P_a^{in}$ or an output port $p_j^{out} \in P_a^{out}$. As depicted in Figure 6.10, this object is defined by the following elements:

- **name**: `String` attribute that identifies the port. It must be noted that the same name cannot be shared among ports of the same kind (i.e. input or output) and of the same actor-class.
- **type**: `Type` element that contains the port type.

It must be noted that, even though ports are defined in the `ActorClass`, these are always considered by the framework as a tuple (`Actor`,`Port`) when analyses are performed.

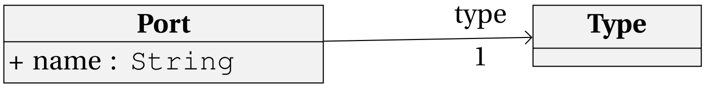


Figure 6.10: The `Port` object.

Buffer

The `Buffer` object is used to model a buffer $b \in B$. As depicted in Figure 6.11, this object is defined by the following elements:

- **sourceActor**: `Actor` element that contains the source actor.
- **sourcePort**: `Port` element that contains the source output port.
- **targetActor**: `Actor` element that contains the target actor.
- **targetPort**: `Port` element that contains the target input port.

Type

A basic type system is modeled using the `Type` object. As depicted in Figure 6.12, this object is defined by the following elements:

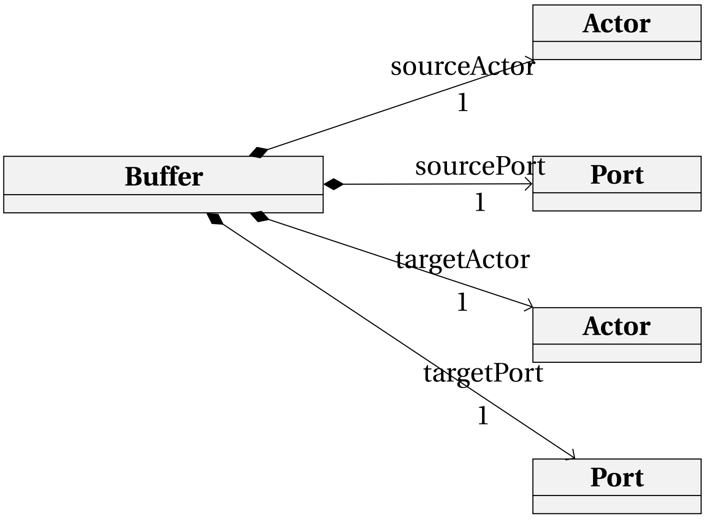


Figure 6.11: The `Buffer` object.

- **name**: `String` attribute that identifies the type.
- **size**: `Integer` attribute that defines the number of elements contained in a complex data type (e.g. elements of a list of elements of the same type).
- **bits**: `Integer` attribute that defines the number of bits required to represent the variable of the given data type.
- **subType**: `Type` element that contains the sub-type of a type, if any. This attribute is used to model complex data types (e.g. elements of a list of elements of the same type).

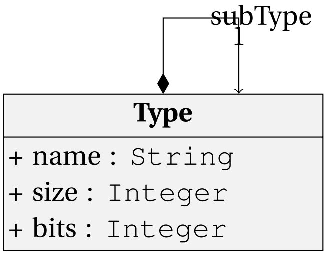


Figure 6.12: The `Type` object.

Version

The `Version` object is used to define a unique identifier of a file. It is used, for example, to track the code modification and refactoring that could be made on a network or in an actor-class. TURNUS supports a Git versioning system [162] and, as depicted in Figure 6.13, for this object defines the following elements:

- **date**: `String` attribute that contains the time stamp of the last local file modification.

- **revision**: `String` attribute that contains the commit hash identifier of the file.
- **repository**: `String` attribute that contains the Git repository URL of the file.

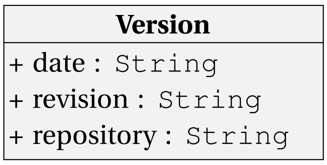


Figure 6.13: The `Version` object.

6.2.2 Architecture and constraints

The platform model describes the structure of the architecture where the dataflow program is implemented. A basic meta model representation is used in order to represent the available processing elements where actors are mapped and the media where buffers are mapped. The platform is modeled as a graph $G(PU, ME, L)$ where:

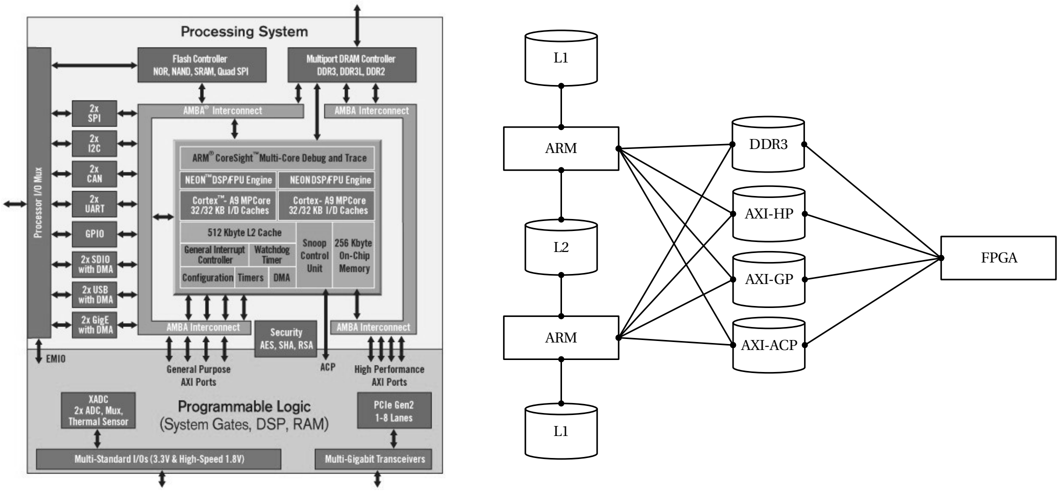
- $PU = \{pu_1, pu_2, \ldots, pu_{n_{PU}}\}$ is the set of processing elements.
- $ME = \{me_1, me_2, \ldots, me_{n_{ME}}\}$ is the set of media.
- $L = \{l_1, l_2, \ldots, l_{n_L}\}$ is the set of links between processing elements and media.

As an example, Figure 6.14 depicts the Xilinx Zynq-7 ZC702 evaluation-board [163] architecture model defined with this formalism. In this case, the set of processing elements PU is composed of three components: two ARMs and an FPGA, respectively. Each ARM has its own L1 memory. The two ARMs share a L2 memory between them and a DDR3 memory with the FPGA. Furthermore, the bus-interfaces AXI-HP, AXI-GP and AXI-ACP are modeled with three different media. Each one of these memories is modeled with a medium $me_i \in M$ and each interconnection between a processing element and a medium with a link $l_i \in L$. The basic components of this architecture high-level representation are described in the following section.

Platform

The `Platform` object is used to model a platform $G(PU, ME, L)$. As depicted in Figure 6.15, this object is composed of the following elements:

- **name**: `String` attribute that contains an identifier of the platform.
- **media**: list of `Medium` elements available in the platform.



(a) Xilinx architecture model.

(b) TURNUS architeture model.

Figure 6.14: Xilinx Zynq-7 ZC702 evaluation-board architecture model.

- **processingElements**: list of `ProcessingElement` elements available in the platform.
- **links**: list of `Links` elements available in the platform.

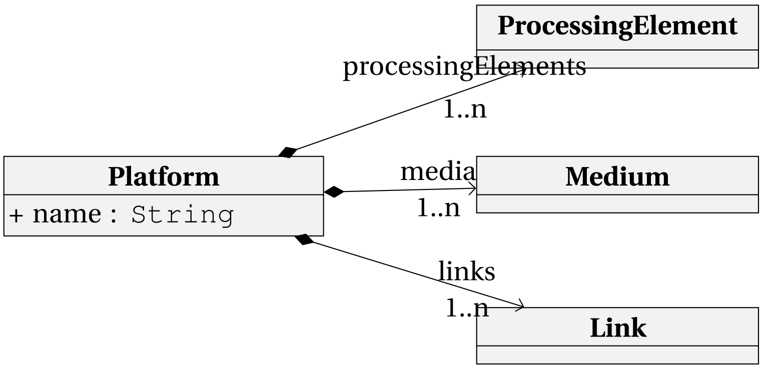


Figure 6.15: The `Platform` object.

Processing element

The `ProcessingElement` object is used to model a processing element $pu \in PU$. As depicted in Figure 6.16, this object is defined by the following elements:

- **name**: `String` attribute that contains an identifier of the operator. Operators of the same platform cannot share the same name.
- **family**: `String` attribute that contains the operator family identifier.

- **clock**: `Double` attribute that defines the period (in ns) of the operator clock cycle.
- **schedulers**: list of `Scheduler` elements available in the processing element.
- **supportedTypes**: list of `Type` elements supported by the processing element.

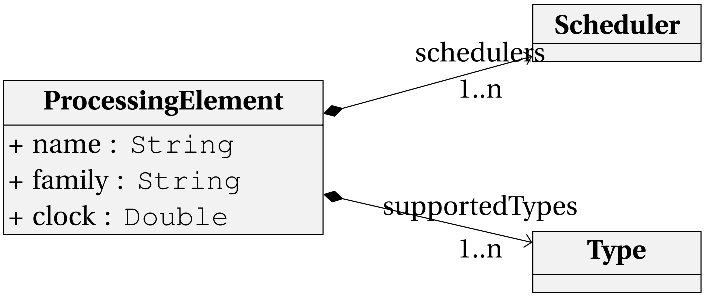


Figure 6.16: The `ProcessingElement` object.

Medium

The `Medium` object is used to model a medium $me \in ME$. As depicted in Figure 6.17, this object is defined by the following elements:

- **name**: `String` attribute that contains an identifier of the medium. Media of the same platform cannot share the same name.
- **family**: `String` attribute that identifies the medium family name.
- **schedulers**: list of `Scheduler` elements available in the medium.
- **inputClock**: `Double` attribute that defines the period (in ns) of the medium input clock cycle.
- **outputClock**: `Double` attribute that defines the period (in ns) of the medium output clock cycle.
- **maxSize**: `Integer` attribute that defines the maximum size (in bit) of the medium.
- **maxPush**: `Integer` attribute that defines the maximum number of bits that can be consumed by the medium during each input clock cycle.
- **maxPop**: `Integer` attribute that defines the maximum number of bits that can be produced by the medium during each output clock cycle.

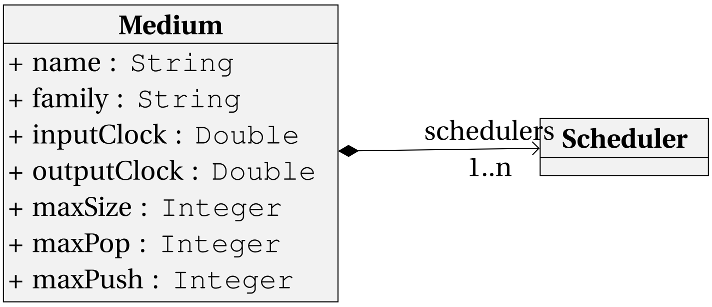


Figure 6.17: The `Medium` object.

Link

The `Link` object is used to model a link $l \in L$. As depicted in Figure 6.18, this object is defined by the following elements:

- **medium**: `Medium` element that defines the medium end-point of the link.
- **operator**: `Operator` element that defines the operator end-point of the link.

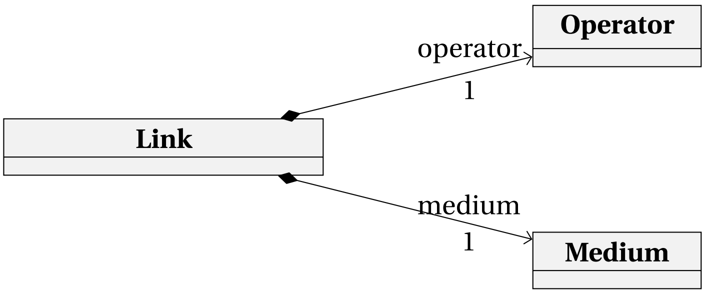


Figure 6.18: The `Link` object.

Scheduler

The `Scheduler` object is used to model a scheduling policy of an operator or a medium. As depicted in Figure 6.19, this object is defined by the following elements:

- **name**: `String` attribute that contains an identifier of the scheduler. Schedulers of the same medium or operator cannot share the same name.
- **selectionTime**: `Double` attribute that defines the time required for making a scheduling choice.

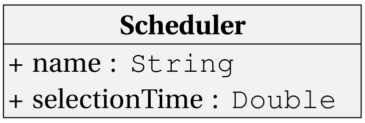


Figure 6.19: The `Scheduler` object.

6.2.3 Execution trace graph

The ETG model describes the graph-based structure illustrated in Chapter 5. A basic meta model representation is used in order to represent both the firings and dependencies sets. The basic components of this representation are described in the following section.

Trace

The `Trace` object is used to model an ETG $G(V, E)$. As depicted in Figure 6.20, this object is defined by the following elements:

- **firings**: list of `Firing` elements contained in the ETG.
- **Dependencies**: list of `Dependency` elements contained in the ETG.

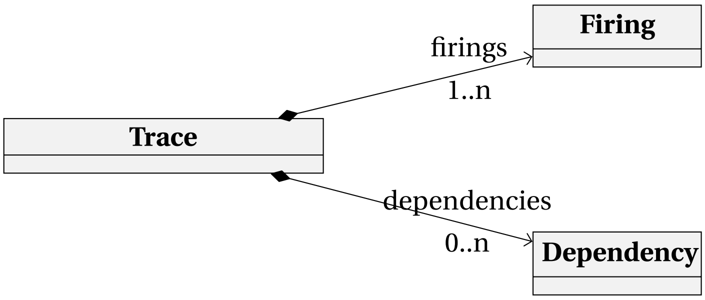


Figure 6.20: The `Trace` object.

Firing

Each firing $s_i \in S$ of the ETG is represented by a `Firing` object. As depicted in Figure 6.21, this object is defined by the following elements:

- **id**: `Long` attribute that defines the firing identifier i. Firings of the same trace cannot share the same id.
- **actorClass**: `String` attribute that contains the name of the `ActorClass`.
- **actor**: `String` attribute that contains the id of the `Actor`.
- **action**: `String` attribute that contains the id of the `Action`.

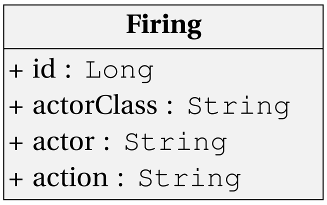


Figure 6.21: The `Firing` object.

Dependency

Each dependency $(s_i, s_j) \in D$ of the ETG is represented by a `Dependency`. As depicted in Figure 6.22, this object is defined by the following elements:

- **sourceFiring**: `Long` attribute that contains the source `Firing` identifier i.
- **targetFiring**: `Long` attribute that contains the target `Firing` identifier j.
- **kind**: `String` attribute that identifies the dependency kind k. Valid values are: *variable*, *fsm*, *guard*, *port* and *tokens*.
- **count**: `Integer` attribute that contains the number of tokens (required only for token dependencies).
- **port**: `String` attribute that contains the `Port` name (required only for port dependencies).
- **sourcePort**: `String` attribute that contains the source `Port` name (required only for token dependencies).
- **targetPort**: `String` attribute that contains the target `Port` name (required only for token dependencies).
- **direction**: `String` attribute that identifies the direction of a dependency. Valid names are: *read/read*, *read/write*, *write/read*, *write/write*, *enable* and *disable*. (required only for port, internal variable and guard dependencies).
- **variable**: `String` attribute that contains the `Variable` name (required only for internal variable dependencies).
- **guard**: `String` attribute that contains the `Guard` identifier (required only for guard dependencies).
- **appearance**: `Integer` attribute that contains the appearance order of a guard enable/disable window (required only for guard dependencies).

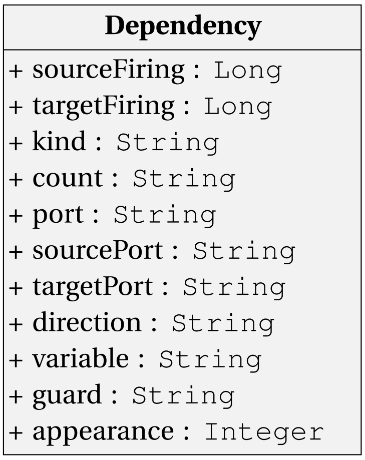


Figure 6.22: The `Dependency` object.

6.2.4 Profiling information

The profiling information represents the data collection provided by third-party profiles. This data set denoted with Θ contains clock-accurate profiling information for each actor and action of the dataflow program. The basic components of this data set are described in the following section.

Network profiling data

The `NetworkProfilingData` object contains the profiling information of a `Network` implemented and profiled on a specific `Operator`. As depicted in Figure 6.23, this object is defined by the following elements:

- **network**: `String` attribute that contains the `Network` name.
- **operator**: `String` attribute that contains the `Operator` name.
- **actionsData**: a list of `ActionProfilingData` elements that contains the profiling data for each tuple (`Actor,Action`) of the network.

Action profiling data

The `ActionProfilingData` object contains the profiling information of each tuple (`Actor,Action`). As depicted in Figure 6.23, this object is defined by the following elements:

- **actor**: `String` attribute that contains the `Actor` identifier.
- **action**: `String` attribute that contains the `Action` identifier.
- **max**: `Double` attribute that contains the maximum number of clock cycles.

- **min**: `Double` attribute that contains the minimum number of clock cycles.
- **average**: `Double` attribute that contains the average number of clock cycles.

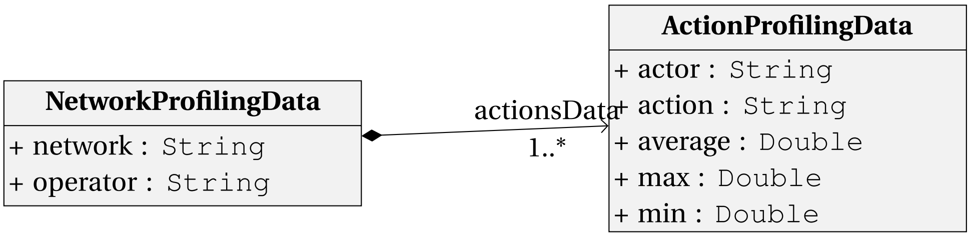


Figure 6.23: The `NetworkProfilingData` and `ActionProfilingData` objects.

6.3 Integration with third-party CAL dataflow environments

The Orcc compiler and the Xronos framework illustrated in Section 2.5.6 are two examples of CAL dataflow environments that are integrated into the TURNUS design flow. As depicted in Figure 6.24, both are used as compiler infrastructures. Additional CAL dataflow environments have been successfully integrated within the TURNUS design flow as illustrated in [164, 165, 166, 167, 168, 169, 170, 171]. In the following, it is discussed how both Orcc and Xronos interact with the TURNUS environment, as these two frameworks have been extensively used for the purpose of this dissertation.

Orcc

As depicted in Figure 6.24, the Orcc and the TURNUS design flow are integrated in the following parts:

- **Code interpretation**: TURNUS Orcc RVC-CAL profiler [172] provides an extension of the basic Orcc CAL interpreter functionalities, where the TURNS APIs illustrated in Section 7.4 have been integrated. This extended code interpreter and profiler is used both to generate the ETG and the high-level profiling data of the CAL program under analysis.
- **Profiling data**: Orcc supports the generation of C/C++ code where the performance application programming interface (PAPI) [173, 174, 175] is integrated. During the program execution, clock-accurate profiling information is retrieved and used, during the DSE performed by TURNUS, to enhance the architecture model.
- **Mapping**: mapping configuration file generated with TURNUS can directly be used in Orcc. In fact, for each Orcc back-end it is possible to drive the code compilation using the buffer size and partitioning configurations evaluated by TURNUS.

Xronos

As depicted in Figure 6.24, the Xronos and the TURNUS design flow are integrated in the following parts:

- **Profiling data**: Xronos provides a test-bench platform where it is possible to retrieve the exact number of clock cycles required for executing a CAL action. This clock-accurate profiling information is then used during the DSE performed by TURNUS to enhance the architecture model and the performance estimation.
- **Mapping**: mapping configuration file generated with TURNUS can directly be used in Xronos. It is possible to drive the code synthesis using the buffer size and partitioning configurations on multi-clock domain platforms.

6.4 Conclusions

In this chapter the DSE environment developed and used for demonstrating the effectiveness of the design methodology discussed in this dissertation has been introduced. Its main functionalities and structure have been illustrated. This DSE environment provides a complete DSE solution for dynamic dataflow programs implemented in heterogeneous and massively parallel architectures. The main functionalities are a collection of application programming interfaces (APIs) for profiling dataflow programs during their code interpretation. The main features of this profiler are the capability to generate an ETG and provide high-level profiling information both retrieved during a high-level code interpretation of the program. Furthermore, the main design space analysis and performance estimation capabilities have been illustrated.

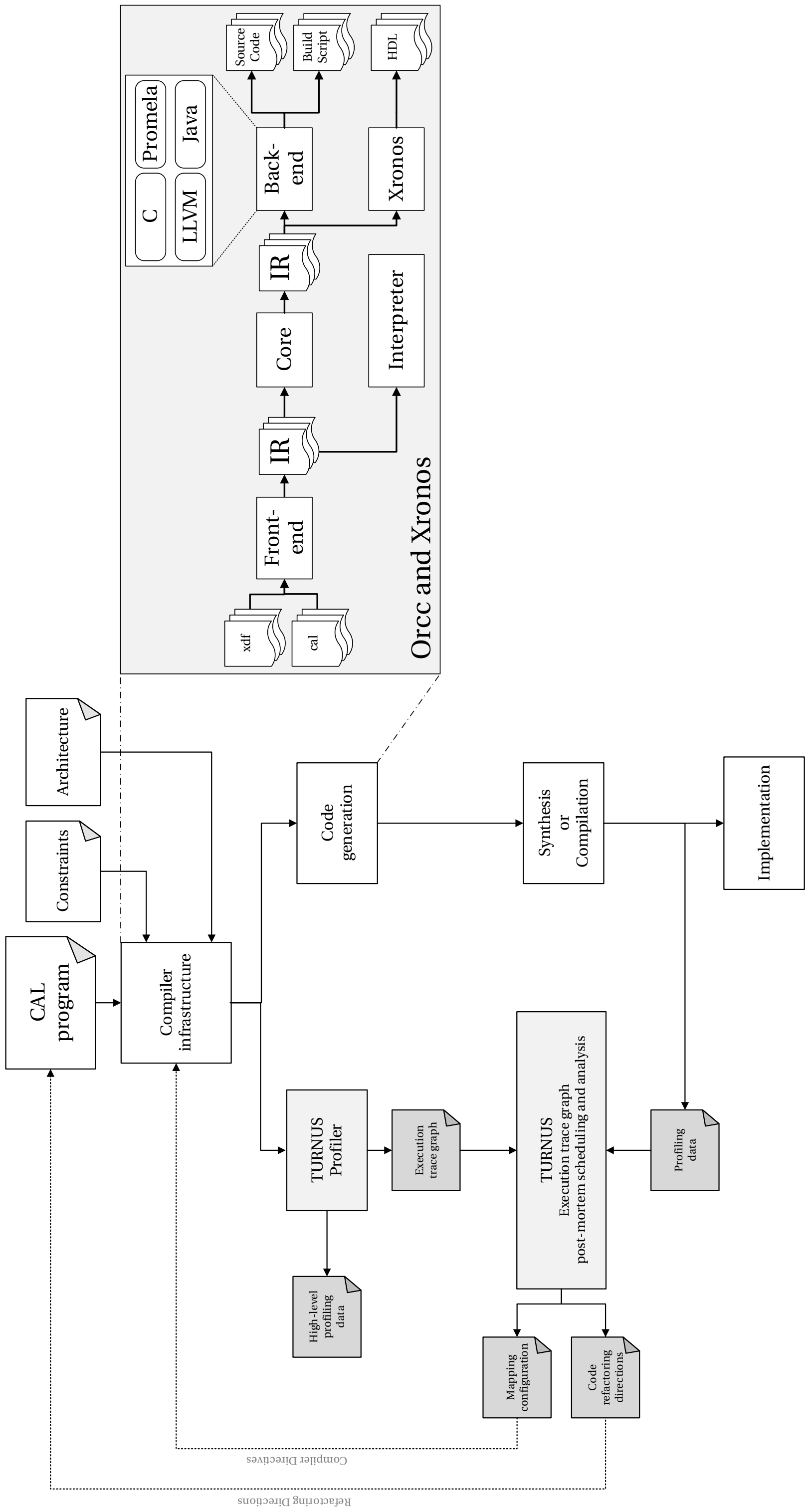


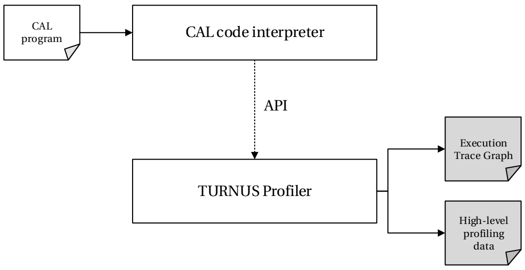
Figure 6.24: The open RVC-CAL compiler (Orcc) and Xronos infrastructure integrated in the TURNUS design flow.

7 Profiling CAL programs with TURNUS

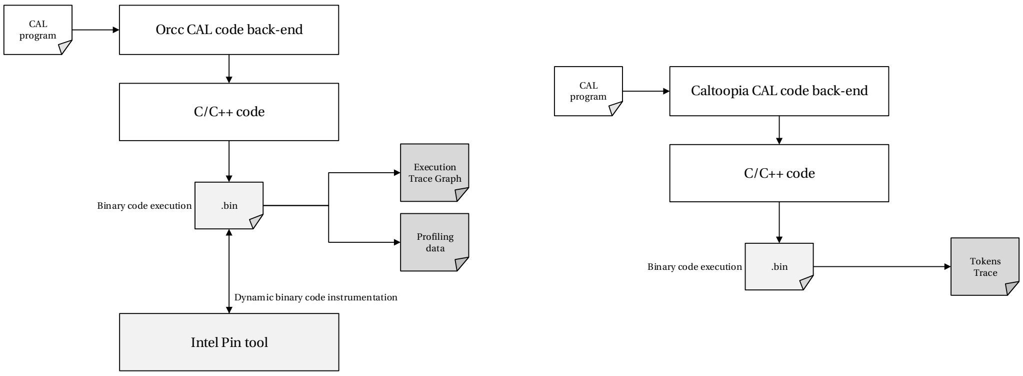
The TURNUS CAL profiler, as every program profiler, provides a statistical summary of the execution complexity of a CAL program. Information that can be retrieved is, for example, the number of operators (i.e. see Table 3.1) that each procedure, action and actor executed, the number of tokens produced and consumed by each action and actor and the buffer utilizations. Moreover, it provides a complete Java application programming interface (API) that can be plugged in a CAL code interpreter. One of its most powerful functionalities is the possibility to generate the ETG illustrated in Chapter 5 without generating any partial implementation of the code. Moreover, it does not depend on any third-party profilers and it can be integrated into an existing CAL code interpreter. At the time of writing this thesis, an integrated version of the TURNUS profiler is provided for the Orcc CAL code interpreter [56] and available as an open source product [16]. In this chapter the main functionalities and novelties that have been introduced are highlighted. Successively, the set of data that can be collected during the program interpretation is illustrated. Finally, it is described how this set of profiling data is used when building the ETG of the program execution.

7.1 Advances in profiling CAL programs

The DSE exploration based on the ETG post-processing can be effectively performed only if the ETG satisfies the requirements illustrated in Chapter 5. CAL profilers that are able to generate an ETG are available on both the CAL Design Suite [122, 33] and the Caltoopia [176] framework. However, as illustrated in Table 7.1, the ETG that these two profilers are able to evaluate does not completely satisfy all requirements. For example, the CAL Design Suite is not able to identify the internal variables and port dependencies. Caltoopia, on the other hand, can identify only tokens dependencies. Furthermore, its ETG is not untimed. Moreover, as illustrated in Figure 7.1, both profilers require a partial C/C++ implementation of the CAL program. Consequently, the ETGs are evaluated through a binary execution of the program. The CAL Design suite makes use of the Intel Pin tool [82, 177] in order to obtain a dynamic binary code instrumentation, while Caltoopia makes use of its native libraries. However, in both cases, the high-level profiling data can be biased by low-level code optimizations performed by the compilers (e.g. GCC [178], ICC [179]). Table 7.1 provides a summarized overview of the new functionalities introduced by the TURNUS CAL profiler. Contrary to the other two environments, the TURNUS CAL profiler provides a collection of APIs (i.e. see Section 7.4) that can be integrated in the available CAL code interpreters. Furthermore, the profiling of the CAL program is performed directly through a CAL code interpretation, without requiring any partial low-level implementation and binary code execution. The profiling information can be obtained with different levels of granularity. In fact, it is possible to obtain information for each actor-class, actor, action, procedure and buffer. It must be noted that the profiling data information is provided as a set of statistical data (i.e. see Section 7.2) where the call of each operator, the load and write of each variable and token are reported. The ETG generated by the TURNUS CAL profiler fully satisfies the requirements illustrated in Chapter 5.



(a) TURNUS CAL profiler.



(b) CAL Design Suite. (c) Caltoopia.

Figure 7.1: CAL profilers design flow.

Table 7.1: CAL profilers features.

(a) Environment and main features.

Tool	Environment	Static analysis	Dynamic analysis	API	Notes
TURNUS	Java 1.7	✓	Code interpretation	✓	
Cal Design Suite	Java 1.6	✓	C/C++ executable	-	(1)
Caltoopia	Java 1.6	-	C/C++ executable	-	

(b) Profiling granularity.

Tool	Actor-class	Actor	Action	Procedure	Buffer
TURNUS	✓	✓	✓	✓	✓
Cal Design Suite	-	✓	✓	-	✓
Caltoopia	✓	✓	✓	-	✓

(c) Profiling information.

Tool	Statistical data	Operator calls	Internal variables	Tokens	Buffers
TURNUS	✓	✓	✓	✓	✓
Cal Design Suite	-	✓	-	-	✓
Caltoopia	-	-	-	✓	✓

(d) Profiling information.

		Dependencies					
Tool	Untimed	FSM	Internal variables	Port	Tokens	Guard	Notes
TURNUS	✓	✓	✓	✓	✓	✓	(2)
Cal Design Suite	✓	✓	-	-	✓	-	
Caltoopia	-	-	-	-	✓	-	(3)

Notes: (1) requires Intel Pin [82, 177] as a third-party tool for instrumenting the binary program; (2) guard enable and disable dependencies analysis is an under-development functionality; (3) token production/consumption is logged during the program execution and successively used to build the ETG.

7.2 Data collection

In the following section, the set of profiling data that is collected during the code interpretation of a CAL program is illustrated.

7.2.1 Firing data

For each action execution, the TURNUS profiler generates a unique firing identifier (FID) stored in a `Long` variable. During the firing execution, the profiling information is stored in the `FiringData` object. As depicted in Figure 7.2, this object contains the following elements:

- **scheduledByFsm**: `Boolean` attribute that indicates if the firing has been scheduled by the internal actor FSM.
- **readVariables**: key-value map element that has a `Variable` as a key and an `Integer` as a value. It indicates how many times the firing has read the actor internal variable. In other words, depending on the variable `Type`, the value corresponds to the number of `Load` or `LoadList` operations that has been performed.
- **writeVariables**: key-value map element that has a `Variable` as a key and an `Integer` as a value. It indicates how many times the firing has written the actor internal variable. In other words, depending on the variable `Type`, the value corresponds to the number of `Store` or `StoreList` operations that has been performed.
- **calledOpcodes**: key-value map element that has an `Opcode` as a key and an `Integer` as a value. It indicates how many times the firing has called a specific operation code.
- **calledProcedures**: key-value map element that has a `Procedure` as a key and an `Integer` as a value. It indicates how many times the firing has called a specific `Procedure`.
- **enabledGuards**: list of `Guard` elements that contains the list of guards that has been enabled by the firing.
- **disabledGuards**: list of `Guard` elements that contains the list of guards that has been disabled by the firing.
- **consumedTokens**: key-value map element that has a `Buffer` as a key and a list of `Token` elements as a value. It contains the list of consumed tokens for each buffer.
- **consumedTokens**: key-value map element that has a `Buffer` as a key and a list of `Token` elements as a value. It contains the list of produced tokens for each buffer.

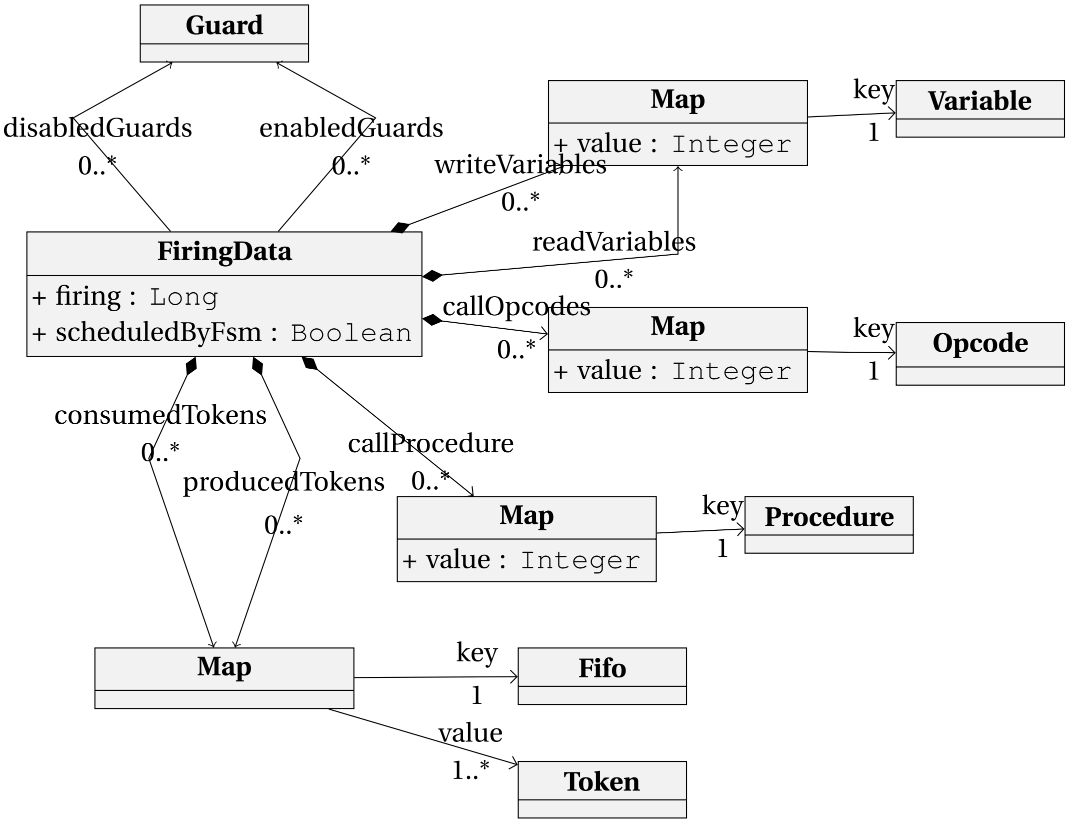


Figure 7.2: The `FiringData` object.

7.2.2 Action data

For each action $\lambda \in \Lambda$ a set of statistical data is collected during the entire program execution. This set of data is defined by the `ActionData` object. As depicted in Figure 7.3, this object contains the following elements:

- **readVariables**: key-value map element that has a `Variable` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of readings of a variable performed by all the firings of the action.
- **wroteVariables**: key-value map element that has a `Variable` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of writings of a variable performed by all the firings of the action.
- **calledOpcodes**: key-value map element that has an `Operand` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of calls of an operation code performed by all the firings of the action.
- **calledProcedures**: key-value map element that has a `Procedure` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of calls of a procedure performed by all the firing of the actions.

- **consumedTokens**: key-value map element that has a `Buffer` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of tokens consumed by all the firings of the action.
- **producedTokens**: key-value map element that has a `Buffer` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of tokens produced by all the firings of the action.

This data set is updated each time that a firing ends its execution: the firing's data is merged in the corresponding action's data.

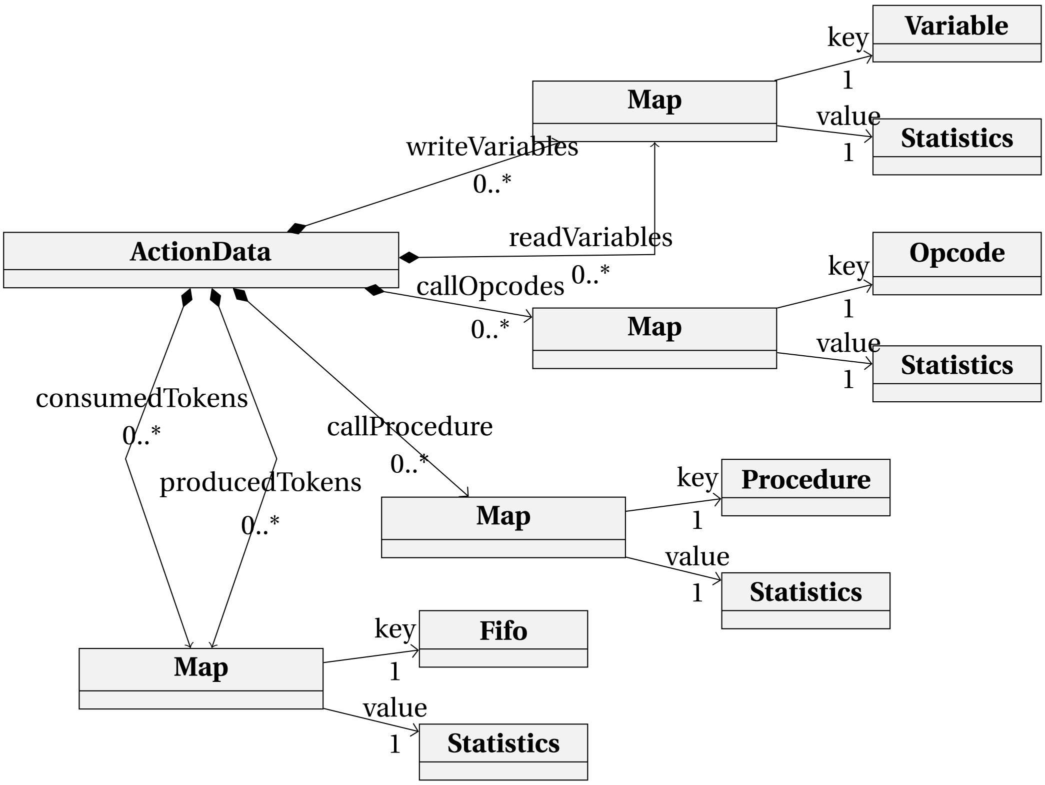


Figure 7.3: The `ActionData` object.

7.2.3 Actor data

For each actor $a \in A$ a set of statistical data is collected during the entire program execution. This information is stored in the `ActorData` object. As depicted in Figure 7.4, this object contains the following elements:

- **firedActions**: key-value map element that has an `Action` as a key and an `Integer` as a value. It contains the number of firings of each action contained in the actor.

- **readVariables**: key-value map element that has a `Variable` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of readings of a variable performed by all the firings of the actor.
- **wroteVariables**: key-value map element that has a `Variable` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of writings of a variable performed by all the firings of the actor.
- **calledOpcodes**: key-value map element that has an `Operand` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of calls of an operation code performed by all the firings of the actor.
- **calledProcedures**: key-value map element that has a `Procedure` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of calls of a procedure performed by all the firings of the actor.
- **consumedTokens**: key-value map element that has a `Buffer` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of tokens consumed by all the firings of the actor.
- **producedTokens**: key-value map element that has a `Buffer` as a key and a `Statistics` object as a value. It contains the statistical information concerning the number of tokens produced by all the firings of the actor.

Furthermore, an additional set of data is used to track which was the last action firing that used a resource (e.g. internal variable, input or output port). This information is stored in the `ActorTracingData` object. As depicted in Figure 7.5, this object contains the following elements:

- **lastFsmScheduled**: `Long` attribute that contains the FID of the last firing scheduled by the actor state machine.
- **lastVariableReader**: key-value map element that has a `Variable` as a key and a `Long` as a value. For each variable, it contains the FID, if it exists, of the last action firing that read the variable.
- **lastVariableReader**: key-value map element that has a `Variable` as a key and a `Long` as a value. For each variable, it contains the FID, if it exists, of the last action firing that wrote the variable.
- **lastGuardEnabler**: key-value map element that has a `Guard` as a key and a `Long` as a value. For each guard, it contains the FID, if it exists, of the last action firing that enabled the guard.
- **lastGuardDisabler**: key-value map element that has a `Guard` as a key and a `Long` as a value. For each guard, it contains the FID, if it exists, of the last action firing that disabled the guard.

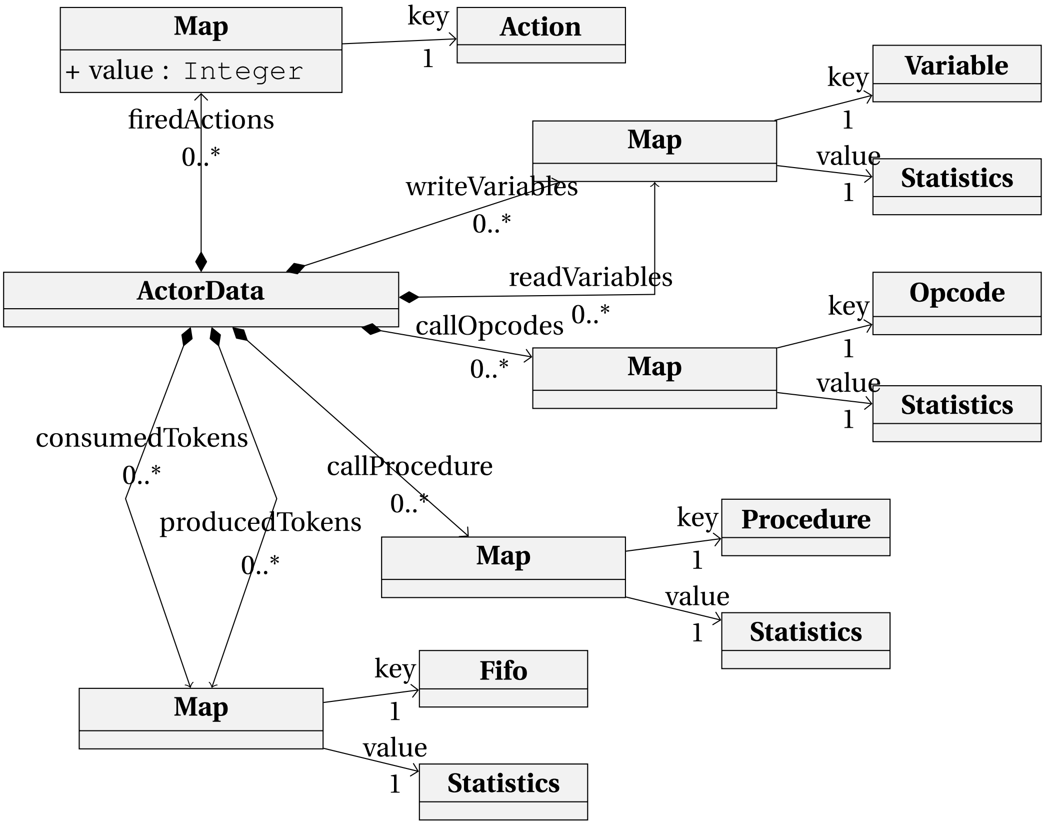


Figure 7.4: The `ActorData` object.

- **lastBufferReader**: key-value map element that has a `Buffer` as a key and a `Long` as a value. For each buffer, it contains the FID, if it exists, of the last action firing that consumed a token from this buffer.

- **lastBufferWriter**: key-value map element that has a `Buffer` as a key and a `Long` as a value. For each buffer, it contains the FID, if it exists, of the last action firing that produced a token on this buffer.

This data set is updated each time that a firing ends its execution: the `FiringData` are merged in the corresponding actor data. It must be noted that the data merging can be done only after the computation of the ETG dependencies has been performed as described in Section 7.3.

7.2.4 Buffer data

For each buffer $b \in B$ the following set of statistical data is collected during the entire program execution.This information is stored in the `BufferData` object. As depicted in Figure 7.6, this object contains the following elements:

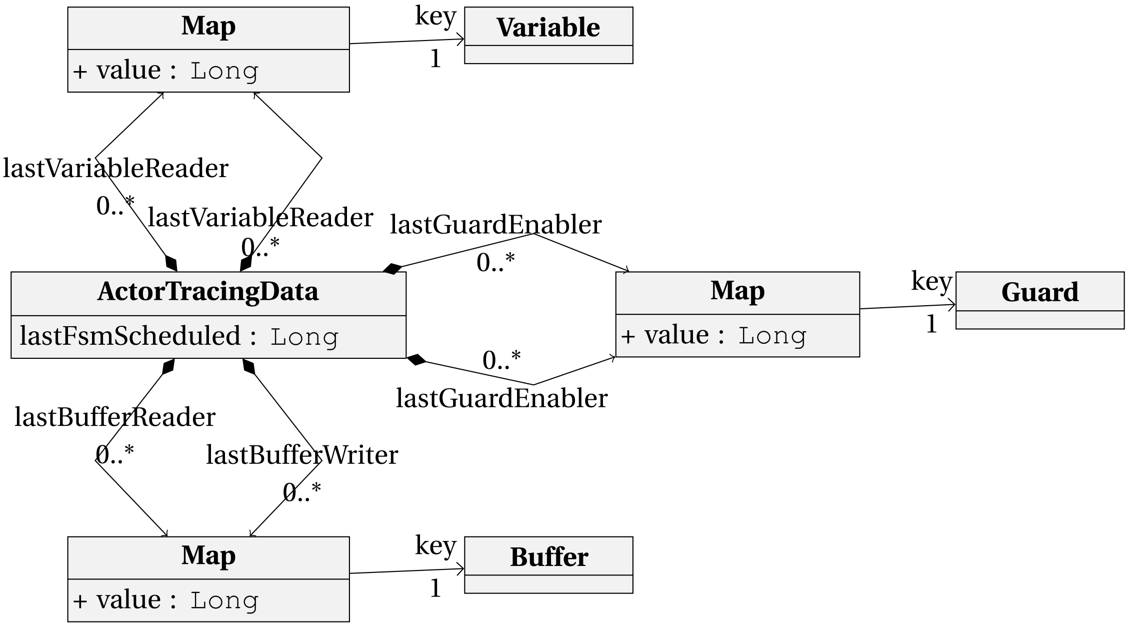


Figure 7.5: The `ActorTracingData` object.

- **consumedTokens**: `Statistics` element that contains the statistical information about the number of tokens that has been consumed from the buffer.

- **producedTokens**: `Statistics` element that contains the statistical information about the number of tokens that has been produced from the buffer.

- **maxOccupancy**: `Integer` attribute that contains the maximum number of stored tokens in the buffer.

- **readMisses**: `Integer` attribute that contains the sum of read misses.

- **readHits**: `Integer` attribute that contains the sum of read hits.

- **writeMisses**: `Integer` attribute that contains the sum of write misses.

- **writeHits**: `Integer` attribute that contains the sum of write hits.

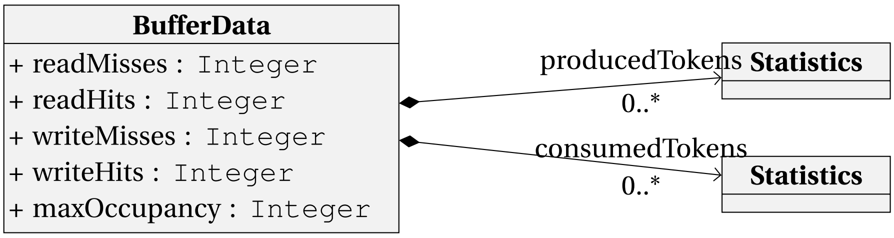


Figure 7.6: The `BufferData` object.

7.2.5 Statistical data

Summary statistics for a stream of data values are collected in a `Statistics` object. As illustrated in Figure 7.7, this object contains the following information:

- **min**: `Double` attribute that contains the minimum value of the data stream.
- **max**: `Double` attribute that contains the maximum value of the data stream.
- **average**: `Double` attribute that contains the average of the data stream.
- **variance**: `Double` attribute that contains the variance of the data stream.
- **count**: `Long` attribute that contains the number of elements stored in the data stream.

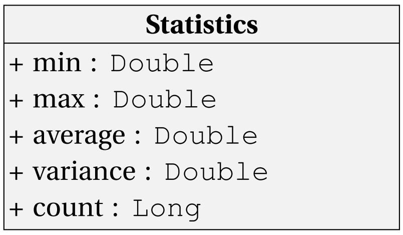


Figure 7.7: The `Statistics` object.

7.2.6 Profiled token

During the program execution each `Token` is treated as an object that contains, as depicted in Figure 7.8, both of the following information:

- **producer**: `Long` attribute that contains the FID of the firing that produced the token.
- **value**: generic `Object` attribute that contains the encapsulated value of the token.

It must be noted that, as illustrated in the next Section 7.3, the information concerning the producer is indispensable in order to evaluate the tokens dependencies of an ETG.

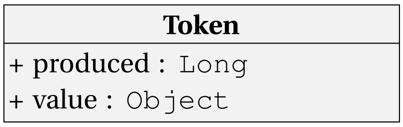


Figure 7.8: The `Token` object.

7.3 Building of the execution trace graph

At the end of each action firing, a `Firing` object is created by analyzing the corresponding `FiringData`. Each `Firing` object represents a single action firing $s_i \in S$, where the FID is evaluated such as $i =$ `FiringData.firing`. Furthermore, it is possible to compute the incoming dependencies set $\delta(s_i)^-_S$. This set is evaluated immediately after s_i ends its execution. The respective firing data and actor data contained in the `FiringData` and `ActorData`, respectively, are analyzed. In the following, it is illustrated how for each dependency kind described in Section 5.2.2, these data objects are be analyzed. By using this methodology, the ETG can be immediately streamed and stored in a file during the simulation process. It must be noted that the memory requirement of the profiler is limited and predictable: in fact, the size of the data sets is limited and predictable too.

Internal variable dependencies

The set of variable dependencies is evaluated by analyzing both the `FiringData` and the `ActorData` sets. For each `Variable` that has been read by the firing s_i, a read/read dependency $(s_j, s_i) \in D_v$ is defined if the read variables map of the `ActorData` contains an FID for the given `Variable`. Similarly a read/write dependency $(s_j, s_i) \in D_v$ is defined if the written variables map of the `ActorData` contains an FID for the given `Variable`. For each `Variable` that has been written by the firing s_i, a write/read dependency $(s_j, s_i) \in D_v$ is defined if the read variables map of the `ActorData` contains an FID for the given `Variable`. Similarly a write/write dependency $(s_j, s_i) \in D_v$ is defined if the written variables map of the `ActorData` contains an FID for the given `Variable`.

Finite state machine dependency

The internal state machine dependency is evaluated by analyzing both the `FiringData` and the `ActorData` sets. In fact, $(s_j, s_i) \in D_f$ can be defined if the firing s_i has been scheduled by the actor state machine and if the `ActorData` contains an FID j of a firing that has been previously scheduled by the internal state machine.

Guard dependencies

The set of guard dependencies is evaluated by analyzing both `FiringData` and the `ActorData` set. For each `Guard` that has been enabled by the firing s_i, an enable dependency $(s_j, s_i) \in D_g$ is defined if the last guard disabler map of the `ActorData` contains an FID for the given `Guard`. Similarly, a disable dependency $(s_j, s_i) \in D_g$ is defined if the last guard enabler map of the `ActorData` contains an FID for the given `Guard`.

Port dependencies

The set of port dependencies is evaluated by analyzing both the `FiringData` and the `ActorData` sets. For each `Buffer`, where at least one token has been consumed by the firing s_i, a read/read dependency $(s_j, s_i) \in D_p$ is defined if the last buffer reader map of the `ActorData` contains an FID for the given `Buffer`. Similarly, a write/write dependency $(s_j, s_i) \in D_p$ is defined if the last buffer writer map of the `ActorData` contains an FID for the given `Buffer`.

Tokens dependencies

The set of token dependencies is evaluated by directly analyzing the `FiringData` set. In fact, for each `Token` contained in the map of consumed tokens it is possible to identify a $(s_j, s_i) \in D_t$ where j is the `Token.producer` (i.e. which identifies the token producer).

7.4 Application programming interface

The collection of profiling APIs provided by TURNUS is evaluated in the following. These methods are used by a third-party CAL code interpreter. It must be noted that this API can be used under the assumption of a serial code interpretation. In other words, the code interpretation can be performed only by taking into account one single action firing at a time.

`Long startFiring(Actor` *`actor`*`, Action` *`action`*`, Boolean` *`sbfm`*`)`
This method is called when a new action can be fired. A new and empty `FiringData` object is created and associate to this new firing. The TURNUS profiler generates a new `firing` identifier.

`Void endFiring()`
This method is called when the current action firing has terminated its execution. After the call of this method the current firing is added to the ETG as illustrated in Section 7.3. Furthermore, both the `ActionData` and the `ActorData` data are updated as illustrated in Section 7.2.

`Void read(Variable` *`variable`*`, Object` *`value`*`)`
This method is called each time a `Variable` is read by the current action firing. The `readStateVariable` map defined in the `FiringData` is updated accordingly.

`Void write(Variable` *`variable`*`, Object` *`value`*`)`
This method is called each time a `Variable` is written by the current action firing. The `writeStateVariable` map defined in the `FiringData` is updated accordingly.

`Object[] produce(Buffer` *`buffer`*`, Object[]` *`tokens`*`)`
This method is called each time the current action firing writes a collection of tokens on the `Buffer`. It must be noted that the TURNUS profiler internally wraps each token `Object`

in a `ProfiledToken` object as discussed in Section 7.2. Both the `FiringData` and the `BufferData` are updated accordingly.

`Object[] consume(Buffer *buffer*, Integer *numTokens*)`
This method is called each time the current action firing consumes *numTokens* from a `Buffer`. It must be noted that the code interpreter receives the `Object` and it is unaware of the ProfiledToken object used to store the producer identifier (i.e. see Section 7.2). In other words, only the TURNUS profiler extends, internally, the concept of profiled token. Both the `FiringData` and the `BufferData` are updated accordingly.

`Void enable(Guard *guard*)`
This method is called each time the current action firing enables a `Guard`. The `FiringData` is updated accordingly.

`Void disable(Guard *guard*)`
This method is called each time the current action firing disables a `Guard`. The `FiringData` is updated accordingly.

`Void call(Procedure *procedure*)`
This method is called each time the current action firing calls (i.e. enter in) a `Procedure`. The `FiringData` is updated accordingly.

`Void endProcedure()`
This method is called each time the current action firing ends (i.e. exit from) a `Procedure`.

`Void call(Opcode *opcode*)`
This method is called each time the current action firing calls an `OpCode`. The `FiringData` is updated accordingly.

`Boolean hasTokens(Buffer *buffer*, Integer *numTokens*)`
This method is called each time the scheduler checks if there are enough tokens in the given `Buffer`. If the result is `true`, then a `readHit` is stored in the respective `BufferData`, otherwise it is a `readMiss`.

`Boolean hasSpace(Buffer *buffer*, Integer *numTokens*)`
This method is called each time the scheduler checks if there is enough space in the given `Buffer`. If the result is `true`, then a `writeHit` is stored in the respective `BufferData`, otherwise it is a `writeMiss`.

7.5 Conclusions

In this chapter, the main functionalities and structure of the CAL dataflow profiler available in the framework illustrated in Chapter 6 have been discussed. Compared to the available profiling tools for this dataflow language, the new functionalities are the possibilities to generate a complete ETG and to obtain statistical profiling information with different levels of abstraction. The number of executed and called operators and procedures as well as the internal actor variables and token utilization (i.e. read/write) for each actor-class, actor, action and procedure can be analyzed. Furthermore, buffer utilization statistics are provided in terms of token production/consumption rates, maximal occupancy, read hits and misses, as well as write hits and misses. Finally, the APIs that can be used by a generic third-party CAL code interpreter has been illustrated. An example of integration with an already-available CAL compiler has been discussed.

8 Design space exploration and optimization with TURNUS

In this chapter different DSE strategies are illustrated based on the analysis of the ETG. First of all, it is discussed how design performance can be estimated through a post-mortem scheduling of the ETG. How timing information are estimated and assigned both for each firing and each dependency is also illustrated. Then, some DSE analyses are illustrated and discussed. These heuristics are all based on the analysis of the design space critical path. Different problems, such as evaluating the design refactoring directions, minimizing and optimizing the buffer size configuration and minimizing the dynamic power dissipation of a design are also discussed.

8.1 Performance estimation

Performance of a program is estimated by an ETG post-mortem scheduling that takes into account a particular mapping configuration. Using the notions of architecture modeling, enhanced with clock-cycle accurate profiling information, Equation (4.3) can be defined as:

$$\widehat{\mathrm{T}}(m) = f(m,\ ETG(S,D),\ G(PU,ME,L),\ \Theta) \tag{8.1}$$

where $m = (\rho, \sigma, \beta) \in M$ is a mapping configuration point of the design space, $ETG(S,D)$ is the ETG of the program, $G(PU,ME,L)$ is the target architecture model and Θ is the set of clock-accurate profiling information retrieved by third-party profilers. Performance, in terms of throughput, is estimated introducing the timing information illustrated in Section 5.4 for each action firing $s_i \in S$ and each dependency $(s_i, s_j) \in D$. Recalling Equation (5.10), the algorithmic part execution time $w(s_i)_e$ can be obtained by third-party HW and SW profilers (e.g. GNU gprof, Valgrind, ModelSim). On the contrary, the other terms contained in $w(s_i)$ (e.g. the action selection time, read and write delays) and the dependencies weights $w(s_i, s_j)$ should be estimated. Furthermore, as discussed in Section 5.3.2, additional mapping dependencies might be introduced in D according to the particular mapping configuration. It must be noted that the partial order represented by the ETG should remain the same even after the post-mortem scheduling (i.e. locally in each actor and globally over the entire design network).

Following this section, the structure and the main functionalities of the ETG post-mortem scheduler used to estimate the design performance are discussed. Furthermore, it is clarified how the timing information can be estimated and used by the underling analyses that are illustrated in this chapter.

8.1.1 Post-mortem scheduler models

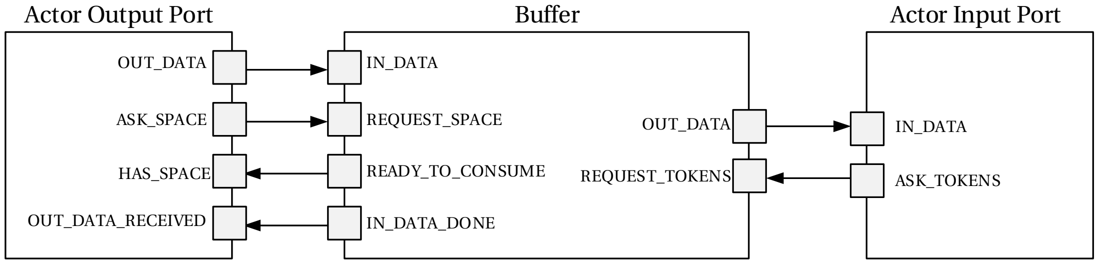
The ETG post-mortem scheduler is based on a discrete event system specification (DEVS) formalism [180, 181]. This is a modular, hierarchical and timed-event system which makes possible, among other things, the modeling and the analysis of discrete-event systems. The two basic elements that describe a DEVS model are the following:

- **`AtomicModel`**: the basic building blocks of a DEVS model. The behavior of an atomic model is described by its state transition functions (internal, external, and confluent), its output function, and its time advance function.
- **`PortValue`**: makes the communication possible between a pair of atomic models. Moreover it defines the template argument for the types of objects that can be accepted as input and produced as output.

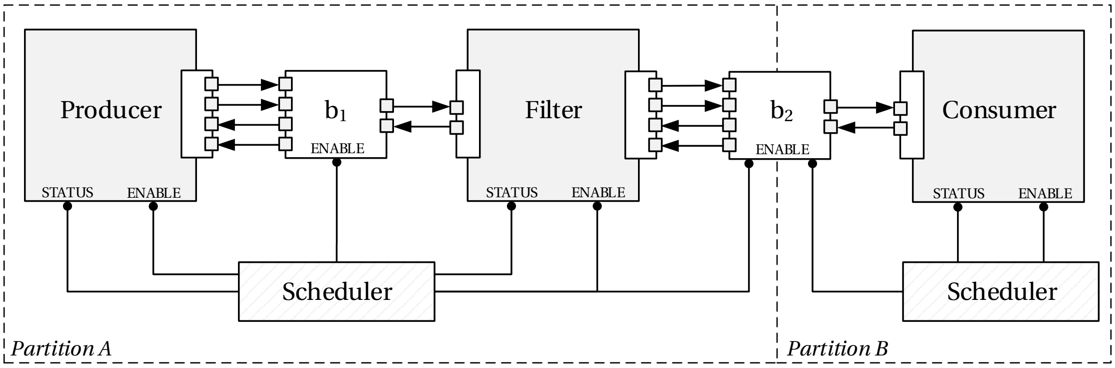
The state of an atomic model is realized by the attributes contained in the object that implements the model. The evolution of the state is modeled through the combination of the following functions and events:

- **Internal transition function** δ_{ext}: describes the model autonomous behavior (i.e. how its state evolves in the absence of input). These types of events are called internal events because they are self-induced (i.e. internal to the model).
- **Time advance function** δ_{a}: schedules these autonomous changes of state.
- **Output function** δ_{out}: describes the output of the model when an internal event occurs.
- **External transition function** δ_{ext}: describes how the model changes state in response to the input.
- **Confluent transition function** δ_{conf}: handles the simultaneous occurrence of both an internal and an external event.

In this section how the dataflow program and the target architecture are modeled using the DEVS formalism is described. See Appendix A.2 for a complete overview about DEVS.



(a) DEVS model of a buffer, actor input port and actor output port.



(b) DEVS model of the dataflow application, discussed in Section 2.5.4, mapped on two separate operator partitions.

Figure 8.1: Execution trace graph post-mortem scheduler: simulation models.

Actor model

Each actor $a \in A$ is modeled as an `AtomicActor` element which describes a DEVS atomic model. Each actor atomic model contains the subset of the actor firings $S_a \subseteq S$. Each actor output port $p_i^{out} \in P_a^{out}$ is modeled, as illustrated in Figure 8.1a, with the following four `PortValue` elements:

- **`OUT_DATA`**: used to send the produced tokens to the output buffer.
- **`ASK_SPACE`**: used to send the number of tokens that should be produced.
- **`HAS_SPACE`**: used for receiving an acknowledgment signal when the requested token space is available.
- **`OUT_DATA_RECEIVED`**: used for receiving an acknowledgment signal when all the produced tokens have been successfully received.

Similarly, each actor input port $p_i^{in} \in P_a^{in}$ is modeled with the following two `PortValue` elements:

- **`IN_DATA`**: used to receive the input tokens from the input buffer.
- **`ASK_TOKENS`**: used to send the number of tokens that should be consumed.

It must be noted that an input event is associated with each input `PortValue`. Similarly, an output event is associated with each output `PortValue`. Furthermore, as illustrated in Figure 8.1b, the additional **`ENABLE`** input `PortValue` element and the **`STATUS`** output `PortValue` element are defined for each actor. Both ports are used by the partition scheduler (i.e. see the following part of this section) to enable and disable the actor and to retrieve its status.

Buffer model

Each buffer $b \in B$ is modeled as an `AtomicBuffer` which describes a DEVS atomic model. Each buffer atomic model is modeled as an asynchronous receiver/transmitter (Rx/Tx). The following four `PortValue` elements, as illustrated in Figure 8.1a, are used to model the Rx interface with the source actor:

- **`IN_DATA`**: used to receive the tokens produced by the actor.
- **`IN_REQUEST_SPACE`**: used to receive the number of tokens that the actor wants to produce.
- **`READY_TO_CONSUME`**: used to send the acknowledgment signal when the requested token space is available.
- **`IN_DATA_DONE`**: used to send the acknowledgment signal when all the tokens produced by the actor have been received.

Similarly, the following two `PortValue` elements are used to model the Tx interface with the target actor:

- **`OUT_DATA`**: used to send the tokens requested by the actor.
- **`REQUEST_TOKENS`**: used to receive the number of tokens required by the actor.

It must be noted that an input event is associated with each input `PortValue`. Similarly, an output event is associated with each output `PortValue`. Furthermore, as illustrated in Figure 8.1b, the additional **`ENABLE_RX`** and **`ENABLE_TX`** input `PortValue` elements are defined for each buffer. These are used by the partition scheduler (i.e. see below) to asynchronously enable and disable the Rx and Tx interfaces, respectively, of the buffer.

Mapping model

Each partition is modeled as an `AtomicPartition` which describes a DEVS atomic model. As illustrated in Figure 8.1b, the scheduler of each actor and buffer partition is modeled as a controller that enables or disables the corresponding atomic objects. Each actor is enabled by sending a signal to the `ENABLE` port according to its status provided thought the `STATUS` port. Similarly, each buffer is enabled by sending a signal to the `ENABLE_RX` and `ENABLE_TX` ports. These ports can be used asynchronously in order to model buffers that are on the boundary of two actor partitions or buffers that are used in a multi-clock domain architecture. As an example, Figure 8.1b illustrates the post-scheduler model for the dataflow program discussed in Section 2.5.4. In this case the `Producer` and `Filter` actors are partitioned in the same partition *PartitionA*, and the `Consumer` actor is partitioned in partition *PartitionB*. Each of these partitions have an actors scheduler and a buffers scheduler. It must be noted that the buffer b_1 is modeled as a synchronous buffer (i.e. input and output interfaces are activated at the same time), and contrary to the buffer b_2, which is modeled as a asynchronous buffer (i.e. the activation of the input and the output interfaces is decoupled).

8.1.2 Execution trace graph post-mortem scheduling

Performance of a program is estimated by a post-mortem scheduling of the ETG using the DEVS simulator previously described. For each firing $s_i \in S$, the timing information illustrated in Section 5.4 is estimated. Additional dependencies are introduced in D according to the particular mapping configuration. Figure 8.2 illustrates how each firing $s_i \in S$ is post-scheduled by performing six different stages. These are respectively: *schedule firing*, *ask tokens*, *consume tokens*, *execute firings*, *ask space*, *produce tokens*. The starting and ending time of each stage is used, as described below, to evaluate both the firings and dependencies weights.

Schedule firing

During this stage the actor is selected by the partition scheduler by using the `ENABLE` signal. A new unprocessed firing $s_i \in S_a$ is selected. Considering s_j as the last firing already processed in the given partition, the additional dependency (s_j, s_i) should be added to the original dependencies set D as discussed in Section 5.3.2. The time required for performing this stage defines the dependency weight as:

$$w(s_j, s_i) = t(s_i)^{end}_{schedule} - t(s_i)^{start}_{schedule} \tag{8.2}$$

Time required for performing this step is estimated according to the architecture model and the scheduling policy.

Ask tokens

This stage is performed if there is at least one incoming token dependency of s_i that should be processed, hence the actor sends a token request to each corresponding input buffer through the corresponding `ASK_TOKENS` port. The time required for performing this stage defines

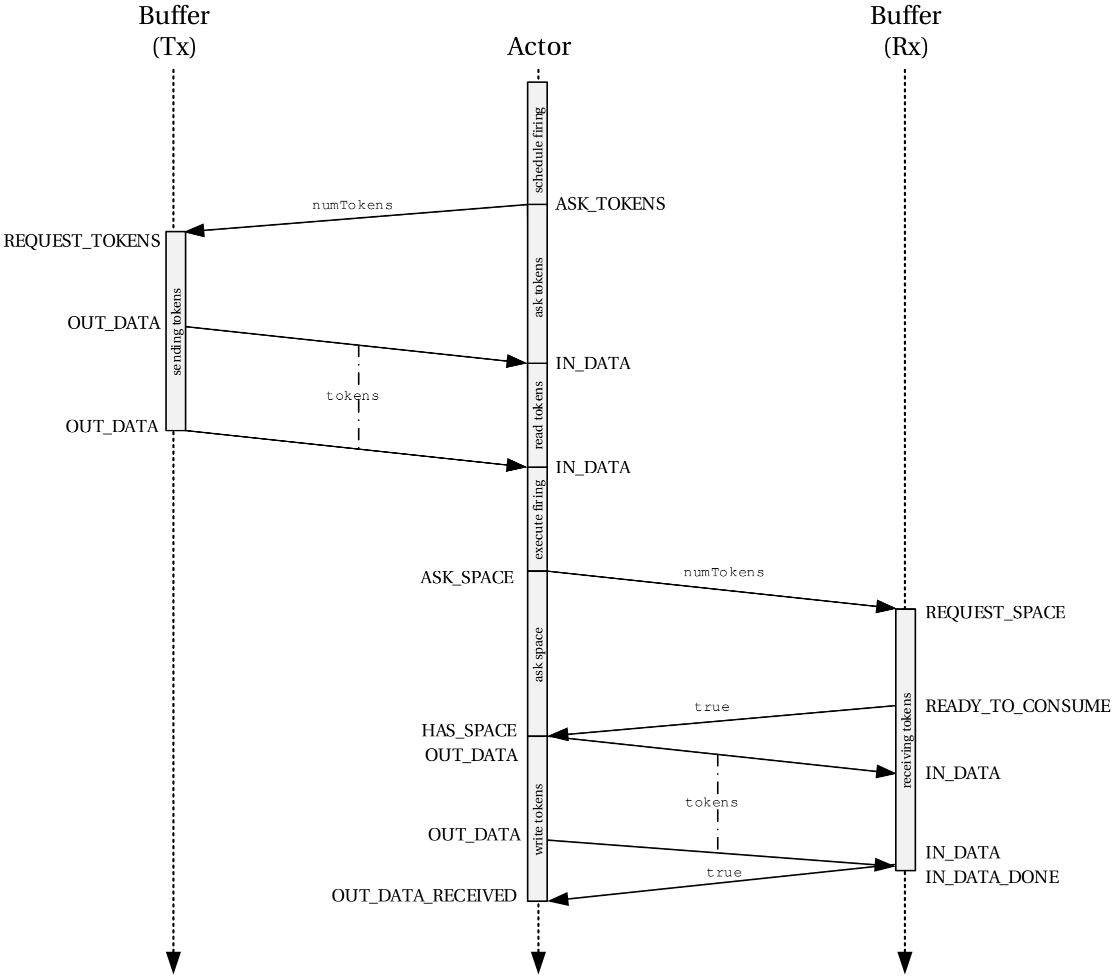


Figure 8.2: Sequence diagram for the DEVS atomic implementation of an actor.

the input wait time of the firing defined as:

$$w(s_i)_{rd} = t(s_i)^{end}_{askTokens} - t(s_i)^{start}_{askTokens} \tag{8.3}$$

The time required for performing this step is estimated according to the architecture model.

Consume tokens

During this stage the incoming token dependencies are processed and the corresponding tokens are consumed. Each token is retrieved from the corresponding `IN_DATA` port. The time required for performing this stage defines the read input token time of the firing defined such as:

$$w(s_i)_{r} = t(s_i)^{end}_{consumeTokens} - t(s_i)^{start}_{consumeTokens} \tag{8.4}$$

Furthermore, for each tokens dependency, the time when this stage is performed is associated and defined as $t(s_i)_{consume}$. The time required for performing this stage is estimated according to the architecture model.

Execute firing

During this stage the algorithmic part of the firing is executed. The time required to perform this stage defines the algorithmic part execution time of the firing defined as:

$$w(s_i)_r = t(s_i)^{end}_{execute} - t(s_i)^{start}_{execute} \tag{8.5}$$

It must be noted that the time required for performing this step can be obtained by third-party profiling information.

Ask space

This stage is performed if at least one outgoing tokens dependency of s_i that should be processed exists. In this case, the actor sends a space request to each corresponding output buffer through the corresponding `ASK_SPACE` port. The time required for performing this stage defines the output waiting time of the firing defined as:

$$w(s_i)_{wd} = t(s_i)^{end}_{askSpace} - tv^{start}_{askSpace} \tag{8.6}$$

Furthermore, for each token dependency, the time when this stage is performed is associated and defined as $t(s_i)_{askSpace}$. The time required for performing this step is estimated according to the architecture model.

Produce tokens

During this stage the outgoing token dependencies are processed and the corresponding tokens are produced. Each token is produced in the corresponding `OUT_DATA` port. The time required for performing this stage defines the write output token time of the firing defined as:

$$w(s_i)_w = t(s_i)^{end}_{produce} - t(s_i)^{start}_{produce} \tag{8.7}$$

Furthermore, for each token dependency, the time when this stage is performed is associated and defined as $t(s_i)_{produce}$. Consequently, the token dependency weight can be defined as:

$$w(s_j, s_i) = t(s_i)_{consume} - t(s_j)_{produce} \tag{8.8}$$

The time required for performing this step is estimated according to the architecture model.

8.1.3 Execution statistics

As initial computational load statistics, the overall **network workload** of the entire dataflow program is defined as:

$$w = \sum\{w(s_i) + \max\{w(s_j, s_i) : (s_j, s_i) \in \delta(s_i)^-_S\} : s_i \in S\} \tag{8.9}$$

For each actor-class $\kappa \in K$, the corresponding **actor-class workload** is defined as:

$$w(\kappa) = \sum\{w(s_i) + \max\{w(s_j, s_i) : (s_j, s_i) \in \delta(s_i)^-_{S_\kappa}\} : s_i \in S_\kappa\} \tag{8.10}$$

Similarly, for each actor $a \in A$ the **actor workload** is defined as:

$$w(a) = \sum\{w(s_i) + \max\{w(s_j, s_i) : (s_j, s_i) \in \delta(s_i)^-_{S_a}\} : s_i \in S_a\} \tag{8.11}$$

where $\delta(s_i)^-_{S_a}$ defines the incoming dependencies set that has a source firing that belongs to the same actor a. For each action $\lambda \in \Lambda$ of the actor, the **action workload** is defined as:

$$w(\lambda) = \sum\{w(s_i) + \max\{w(s_j, s_i) : (s_j, s_i) \in \delta(s_i)^-_{S_\lambda}\} : s_i \in S_\lambda\} \tag{8.12}$$

8.1.4 Analysis of a collection of execution trace graphs

As discussed in Section 2.3.3, the execution behavior of a dynamic dataflow program can change according to the input sequence. Hence, the analysis and the exploration should be performed using a collection of ETGs generated with different input sequences.

Considering a finite set of input sequences $I = \{I_1, I_2, \ldots, I_{n_I}\}$, the corresponding ETGs collection is defined as:

$$ETGs = \{ETG(S_1, D_1), ETG(S_2, D_2), \ldots, ETG(S_{n_I}, D_{n_I})\} \tag{8.13}$$

8.2 Design space critical path

Many metrics for dataflow programs have been developed with the aim of supporting designers to reduce the running time of their applications. The main requirement of such metrics is to provide a clear optimization objective by highlighting both problematic actors (or actions) and buffers that may reduce the design performance. Such possibilities are fundamental for applications whose complexity falls beyond the guess that a designer can make with success. The widest-used metric is the makespan which is defined as the start-to-end execution time of an application [182, 183]. Using the formalism of the ETG, the makespan can be seen as the execution critical path length: this can be defined as the longest, time-weighted sequence of events from the start of the program to its termination [1, 2]. In the context of RVC-CAL, a first attempt in defining a critical path analysis methodology was introduced in [33]. However, this approach makes the simplified assumption that all actors are executed in parallel with an unbounded buffer size configuration. As a result, only the computation load of each action is taken into account, whereas both the scheduler overhead and the buffer latencies are neglected. In other words, only the fully serial code portion of the program can be identified as the design bottleneck. Consequently, this approach severely restricts the design space exploration. In the following section, this methodology is improved defining the concept of **design space critical path**.

8.2.1 Critical path length

The critical path length (CPL) can be evaluated in different ways [32, 33, 184, 185]. Indeed, as demonstrated in [1, 2] the technique provided in [185] seems to be the most convenient both for the reduced complexity of the algorithm and for the additional profiling information that could be retrieved. The latter is illustrated herein below. For each action firing $s_i \in S$, four parameters should be evaluated. These are:

- **Early Start time** $ES(s_i)$ which defines its earliest possible starting execution time.
- **Latest Start time** $LS(s_i)$ which defines its latest possible starting execution time without extending the overall program completion time.
- **Early Finish time** $EF(s_i)$ which defines its earliest possible ending execution time.
- **Latest Finish time** $LF(s_i)$ which defines its latest possible ending execution time without extending the overall program completion time.

Moreover, an additional parameter called **slack** is introduced both for each action firing $s_i \in S$ and each dependency $s(s_i, s_j) \in E$ represented by $SL(s_i)$ and $SL(s_i, s_j)$, respectively. This is used in order to define the maximum delay that a fired action or a dependency can tolerate without impacting the overall completion time. The evaluation of the CP can be done in $O(|S| + |D|)$ by performing the Algorithms 1, 2 and 3, respectively. This evaluation can be summarized as follows. Firstly, for each $s_i \in S$ the early start time $ES(s_i)$ and the early finish time $EF(s_i)$ are evaluated by following any valid increasing topological order of S. It must be noted that for each source firing $s_j \in S_{\emptyset^-}$ (i.e. see Equation (5.4)) the preconditions $ES(s_j) = 0$ and $EF(s_j) = 0$ have been imposed. Secondly, for each $s_i \in S$ the latest start time $LS(s_i)$ and latest finish time $LF(s_i)$ are evaluated by following any decreasing topological order of S. It must be noted that for sink firings $s_j \in S_{\emptyset^+}$ (i.e. see Equation (5.7)) the preconditions $LS(s_j) = ES(s_j)$ and $LF(s_j) = EF(s_i)$ have been imposed. Hence, the slack value for both action firings $s_i \in S$ and dependencies $(s_i, s_j) \in D$ are evaluated. The set of **critical action firings** is defined as:

$$S_c = \{s_i : SL(s_i) = 0\} \subseteq S \tag{8.14}$$

Similarly, the set of **critical dependencies** is defined such as:

$$D_c = \{(s_i, s_j) : SL(s_i, s_j) = 0\} \subseteq D \tag{8.15}$$

Finally, the $\overrightarrow{CP}$ is evaluated by walking back the ETG as illustrated in Algorithm 3. At each iteration a new action firing is selected by following one of the incoming critical edges such that $\delta(s_i)^-_{S_c} = \{s_j : \exists(s_j, s_i) \in D_c\}$. The sets $S_{CP} \subseteq S_c$ and $D_{CP} \subseteq D_c$ contain respectively the fired actions and dependencies along this path. Similarly, the sets $K_{CP} \subseteq K$, $A_{CP} \subseteq A$ and $\Lambda_{CP} \subseteq \Lambda$ contain respectively the actor-classes, actors and actions that have at least one action firing along this path. The CP can be considered completely determinate only when one of the source firings $s_i \in S_{\emptyset^-}$ is reached. It must be noted that one such path always exists [186]. As a result, the **critical path length** is defined such as:

$$|\overrightarrow{CP}| = f(\sigma, \rho, \beta) = \sum\{w(s_i) : s_i \in S_{CP}\} + \sum\{w(s_i, s_j) : (s_i, s_j) \in D_{CP})\} \tag{8.16}$$

where $w(s_i)$ and $w(s_i, s_j)$ represent the action firing and dependency weights, respectively, as discussed in Section 5.4. For this reason the $|\overrightarrow{CP}|$ can be see as a function f of the scheduling, partitioning and buffer size configuration. This can be also evaluated as:

$$|\overrightarrow{CP}| = \max\{LF(s_i) : s_i \in S\} \tag{8.17}$$

As mentioned above, the main advantage of evaluating the critical path in such a way is that this can be done in linear time (i.e. $O(|S| + |D|)$). Moreover, all the critical actions or critical dependencies that are not along the CP can be highlighted through their slack value.

Remark. *More then one CP may exist for each weighted ETG. In this case each CP contains different action firings. However, the length of these paths is always* $|\overrightarrow{CP}|$.

Statistical distribution

The profiling clock weights illustrated in Section 6.2.4 makes it possible to model the execution time as a statistical value. In fact, for each action, it is possible to specify the average, the minimal and the maximal number of clock cycles required for the execution. Hence, it is possible to model the execution weight $w(s_i)_e$ as a statistical variable in the sense of a normal distribution with expected value and variance, respectively, defined as:

$$\begin{cases} E[w(s_i)_e] = \frac{1}{\alpha_2}\big(\min(s_i) + \alpha_1 \text{mean}(s_i) + \max(s_i)\big) \\ Var(w(s_i)_e) = \left(\frac{\max(s_i) - \min(s_i)}{\alpha_2}\right)^2 \end{cases} \tag{8.18}$$

where mean(s_i), min(s_i) and max(s_i) are the average, minimal and maximal execution time, respectively, defined in Figure 6.23. It must be noted that α_1 and α_2 are used to model the distribution shape and they are generally defined as $\alpha_1 = 4$ and $\alpha_2 = 6$ [187]. Furthermore, making the assumption that the execution time of each firing is independent and uncorrelated

Algorithm 1: Compute the set of parameters $ES(s_i), EF(s_i), LS(s_i), LF(s_i)$ for each $s_i \in S$.

```
Input: $(S, \leq)$ the firings po-set with size $H_s = |S|$
Result: $ES(s_i), EF(s_i), LS(s_i), LF(s_i)$ for each $s_i \in S$
// Initialize the source firings $S_{\emptyset^-}$
for $s_i \in S_{\emptyset^-}$ do
    $ES(s_j) \leftarrow 0$
    $EF(s_j) \leftarrow 0$
end
// Iterate S with an increasing topological order
$i \leftarrow 1$
while $i \leq H_s$ do
    if $s_i \notin S_{\emptyset^-}$ then
        $ES(s_i) \leftarrow \max\{EF(s_j) + w(s_j, s_i) : (s_j, s_i) \in \delta(s_i)^-_D\}$
        $EF(s_i) \leftarrow ES(s_i) + w(s_i)$
    end
    $i \leftarrow i + 1$
end
// Initialize the sink firings $S_{\emptyset^+}$
for $s_i \in S_{\emptyset^+}$ do
    $LS(s_j) \leftarrow ES(s_j)$
    $LF(s_j) \leftarrow EF(s_j)$
end
// Iterate S with a decreasing topological order
$i \leftarrow H_s$
while $i \geq 1$ do
    if $s_i \notin S_{\emptyset^+}$ then
        $LF(s_i) \leftarrow \min\{LS(s_j) - w(s_i, s_j) : (s_i, s_j) \in \delta(s_i)^-_D\}$
        $LS(s_i) \leftarrow LF(s_i) - w(s_i)$
    end
    $i \leftarrow i - 1$
end
```

Algorithm 2: Compute the slack value $SL(s_i)$ for each $s_i \in S$ and $SL(s_i, s_j)$, and the set of critical firings set S_c and critical dependencies set D_c.

```
Input: $S$ the firings set
Input: $D$ the dependencies set
Result: $S_c$ the critical firings set and $D_c$ the critical dependencies set
Data: $ES(s_i), EF(s_i), LS(s_i), LF(s_i)$ for each firing $s_i \in S$ evaluated using Algorithm 1
Data: $S_c = \emptyset$ and $D_c = \emptyset$

// Compute the critical firings set $S_c$
for $s_i \in S$ do
    $SL(s_i) \leftarrow LF(s_i) - EF(s_i)$
    if $SL(s_i) = 0$ then
        $S_c \leftarrow S_c \cup \{s_i\}$
    end
end

// Compute the critical dependencies set $D_c$
for $(s_i, s_j) \in D$ do
    $SL(s_i, s_j) \leftarrow LS(s_j) - EF(s_i) - w(s_i, s_j)$
    if $SL(s_i, s_j) = 0$ then
        $D_c \leftarrow D_c \cup \{(s_i, s_j)\}$
    end
end
```

Algorithm 3: Critical path extraction.

```
Input: $S$ the firings set
Result: $\overrightarrow{CP}$ the critical path
Data: $EF(s_i)$ for each firing $s_i \in S$ evaluated using Algorithm 1
Data: $S_c$ the critical firings set evaluated using Algorithm 2
Data: $\overrightarrow{CP} = \emptyset$

// Find the last CP firing
$s \leftarrow \text{argmax}\{s_i : LF(s_i) \geq LF(s_j), \forall s_j \in S\}$
while $s \neq \perp$ do
    $\overrightarrow{CP} \leftarrow s \oplus \overrightarrow{CP}$
    $s \leftarrow$ getCriticalPredecessor ($s$);
end

begin getCriticalPredecessor ($s_i$)
    for $s_j \in \delta(s_i)^-_S$ do
        if $s_j \in S_c$ then
            return $s_j$
        end
    end
    return $\perp$
end
```

to the others [188], it is possible to redefine the $|\overrightarrow{CP}|$ such as:

$$\begin{cases} E[|\overrightarrow{CP}|] = \sum\{E[w(s_i)] : s_i \in S_{CP}\} \\ Var(|\overrightarrow{CP}|) = \sum\{Var(w(s_i)) : s_i \in S_{CP}\} \end{cases} \tag{8.19}$$

where $E[.]$ and $Var(.)$ are the expected value and the variance operators, respectively. In this context, the variance value is used to define the accuracy of the $\overrightarrow{CP}$.

8.2.2 Algorithmic critical path

The algorithmic critical path (ACP) is evaluated neglecting for each action firing the time spent in waiting for the availability of input tokens and output space, and additional scheduling dependencies (i.e. see Section 5.3.2). In other words, the ACP is evaluated supposing that the outgoing dependencies of an action firing are immediately made available to its successors. Consequently, for each action firing $s \in S$ the corresponding weight is evaluated as:

$$\begin{cases} w(s_i) = \begin{cases} w(s_i)_e & \text{for heterogeneous architecture} \\ w(s_i)_r + w(s_i)_e + w(s_i)_w & \text{for homogeneous architecture} \end{cases} \\ w(s_i, s_j) = 0 \end{cases} \tag{8.20}$$

taking into account only the algorithmic part execution weight $w(s_i)_e$ (i.e. see Table 5.5) of each action firing. Other weights, both for firings and dependencies, are neglected. After that, the CPL is evaluated as illustrated in Section 8.2.1 and denoted as $|\overrightarrow{CP}|_{algo}$. This value can be considered as the lower bound for the CPL value of the entire design space such as:

$$|\overrightarrow{CP}|_{algo} \leq |\overrightarrow{CP}|(m), \ \forall m \in M \tag{8.21}$$

It must be noted that in Equation (8.20) the write and read token times, denoted respectively as $w(s_i)_r$ and $w(s_i)_w$, have been neglected when a heterogeneous architecture is considered. This choice is motivated by the fact that the writing and reading time of tokens in heterogeneous architectures may depend on the particular mapping configuration of the buffers. Contrarily, if these values are also considered then Equation 8.21 cannot be considered as a lower bound for the entire design space M.

Remark. *Neglecting the time spent in waiting for the availability of input tokens and output space and neglecting the additional scheduling dependencies corresponds to post-scheduling the ETG considering an unbounded buffer size configuration and a partitioning configuration where for each processing unit is assigned only one actor (i.e. fully-parallel execution).*

8.2.3 Throughput and design space critical path

An interesting property concerning the CP length $|\overrightarrow{CP}|$ is that it can be easily related to the application throughput T defined in Equation (4.2) as:

$$\mathrm{T} \propto \frac{1}{|\overrightarrow{CP}|} \tag{8.22}$$

In other words, by reducing the execution critical path length (or makespan) the throughput of the application increases [183]. This makes it possible to explore the design space in terms of $|\overrightarrow{CP}|$ in order to find trade-offs between performance and resource configuration and usage. From Equation (8.21) it is possible to define an upper bound of the potential achievable performance of the design as:

$$\mathrm{T} \propto \frac{1}{|\overrightarrow{CP}|} \leq \frac{1}{|\overrightarrow{CP}|_{algo}} \tag{8.23}$$

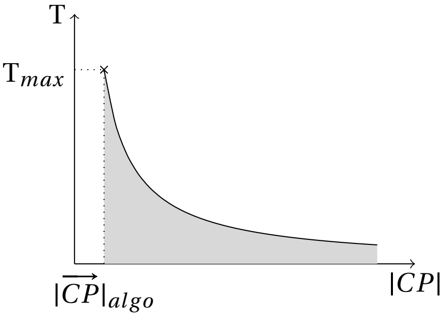
This latter equation defines what is called the **design space critical path** (DSCP) of an application, which is represented in Figure 8.3a. It must be noted that evaluating $|\overrightarrow{CP}|_{algo}$ can be considered as the first starting point of a design space exploration. In fact, in the event that performance does not meet established requirements with this optimistic design configuration, the computational load of the actions (or actors) along this critical path should be reduced. Successively, the design DSCP should be explored in order to find trade-offs between performance and resource configuration and usage as illustrated in Figure 8.3b. Several purpose-driven design optimization analyses can be performed in this direction as illustrated in the next sections of this chapter.

Remark. *Sometimes the throughput of a system is referred to as the production or execution rate of (a particular set of) actors. Throughput is usually measured in terms of bits per second or, for a dataflow program, in tokens per second. However, in a DDF program this rate can vary according to the input stimulus. Consequently, in order to compare its execution with different input stimuli, the throughput should be referred to as the rate at which each input stimulus is completely processed.*

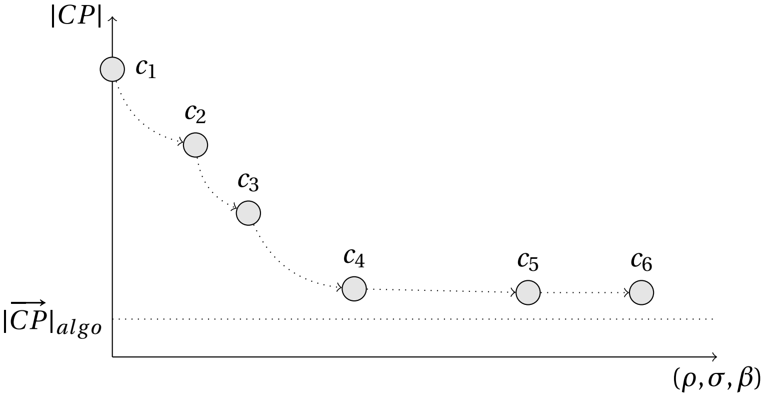
8.2.4 Potential speedup

The theoretical speedup of a program is a widely-used metric in the domain of parallel computing. This metric is used to predict the theoretical speedup for a program when parallel processing units are used (e.g. cores, threads of execution). The theoretical speedup is defined as:

$$\mathrm{S}(n) = \frac{t(1)}{t(n)} \tag{8.24}$$



(a) The relationship between T and the design throughput and the critical path length $|CP|$ as defined in Equation (8.23). The maximum achievable throughput T$_{max}$ is evaluated with the algorithmic critical path length $|CP|_{algo}$.



(b) The design space critical path. Points $\{c_1, c_2,, \dots c_6\}$ represent different mapping configuration points.

Figure 8.3: Design space critical path.

where $t(1)$ and where $t(n)$ are the program finishing times when mapped in one and n processing elements, respectively. Its value is generally estimated using the theoretical formulation proposed in [189], also known as Amdahl's law, defined as a relationship between parallelized implementation of an algorithm and its sequential implementation. Even though this formulation is widely used in computer science engineering, it has always been criticized for the assumption under which it has been formalized [190, 191]. The main criticisms are that it is assumed that the problem size remains the same when parallelized, that parallel portions of a program can hardly be estimated. Furthermore, this law is formulated assuming that the program MoC is sequential. In the context of dataflow programming, the theoretical speed $\mathrm{S}(n)$ can be formulated in terms of network workload w and critical path length $|\overrightarrow{CP}|$, defined in Section 8.1.3 and Section 8.2.1, respectively. In fact, it is possible to define as $t(n) = |\overrightarrow{CP}|(\rho_n)$, hence $t(1) = |\overrightarrow{CP}|(\rho_1) = w$ in the case of one processing unit. Consequently, Equation (8.24) can be redefined as:

$$\mathrm{S}(n) = \frac{w}{|\overrightarrow{CP}|(\rho_n)} \tag{8.25}$$

It must be noted that, as discussed in Section 8.2, $|\overrightarrow{CP}|(\rho_n)$ could be a non-linear function of the application mapping configuration. A lower bound can be evaluated using a simplified linear model, such as the one depicted in Figure 8.4 and defined as:

$$|\overrightarrow{CP}|(\rho_n) = \begin{cases} w - \frac{w - |\overrightarrow{CP}|_{algo}}{n_A - 1}(n-1) & \text{if } n \in [1, n_A] \\ |\overrightarrow{CP}|_{algo} & \text{if } n \geq n_A \end{cases} \tag{8.26}$$

where n_A is the number of actors $a \in A$. It must be noted that the model of Equation (8.26) only makes the assumption that the communication cost between different actor partitions remains constant and does not dominate the program execution. Consequently, defining as

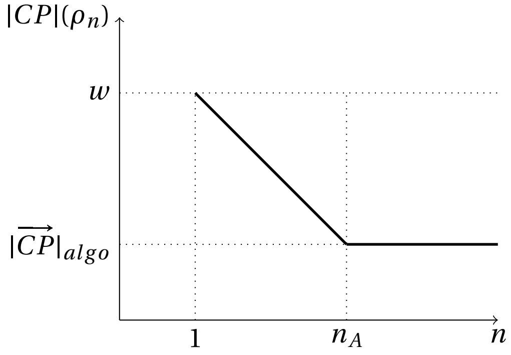


Figure 8.4: Critical path length linear model $|CP|(\rho_n)$.

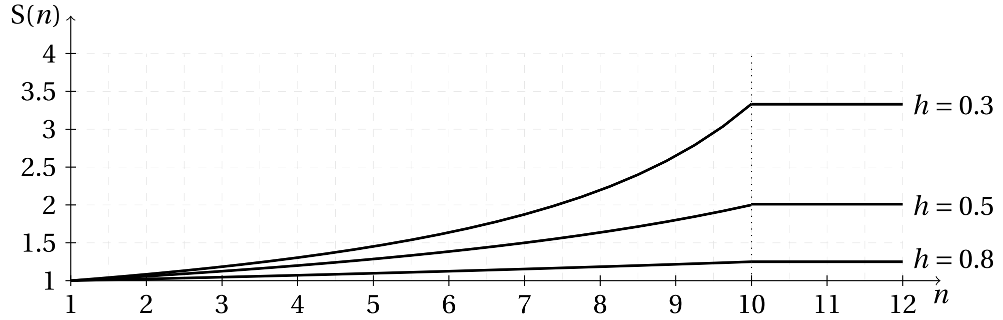


Figure 8.5: Theoretical speedup $\mathrm{S}(n)$ defined in Equation (8.27) for different values of $h = |CP|_{algo}/w \in [0,1]$ when $n_A = 10$.

$h = \frac{|\overrightarrow{CP}|_{algo}}{w} \in [0,1]$, the maximal theoretical speedup of a dataflow program can be defined as:

$$\mathrm{S}(n) = \begin{cases} w - \frac{1}{1+\frac{h-1}{n_A-1}(n-1)}n & \text{if } n \in [1, n_A] \\ \frac{1}{h} & \text{if } n \geq n_A \end{cases} \tag{8.27}$$

As an example, Figure 8.5 depicts this relation for different values of h when $n_A = 10$.

8.3 Hotspot analysis

When performance requirements in terms of $|\overrightarrow{CP}|_{algo}$ cannot be satisfied, the design should be refactored. In other words, the designer should reduce the algorithmic complexity of the actions that most contribute to the most serial part of the design. In the following, two different refactoring direction metrics, useful for the designer, are presented. The first, called critical actions ranking, is an ordered list of actions that most contribute to the overall $|\overrightarrow{CP}|_{algo}$. The second, called impact analysis, estimates which improvement margins can be obtained by refactoring a critical action.

8.3.1 Critical actions ranking

For each actor-class κ of the program, $w(\kappa)_c$ and $w(\kappa)_{CP}$ define the actor-class critical workload and the actor-class workload along the $\overrightarrow{CP}$, respectively. These values are evaluated as:

$$\begin{cases} w(\kappa)_c & = & \sum\{w(s_i) + \max\{w(s_j,s_i) : (s_j,s_i) \in \delta(s_i)^-_{S_{c,\kappa}}\} : s_i \in S_c \cap S_\kappa\} \\ w(\kappa)_{CP} & = & \sum\{w(s_i) + \max\{w(s_j,s_i) : (s_j,s_i) \in \delta(s_i)^-_{S_{CP,\kappa}}\} : s_i \in S_{CP} \cap S_\kappa\} \end{cases} \tag{8.28}$$

where, for a given firing s_i, $\delta(s_i)^-_{S_{c,\kappa}}$ and $\delta(s_i)^-_{S_{CP,\kappa}}$ represent the critical incoming edges and the incoming edges along the CP where source firings belong to the same actor-class κ, respectively. Similarly, for each actor a of the program, $w(a)_c$ and $w(a)_{CP}$ define the actor critical workload and the actor workload along the $\overrightarrow{CP}$, respectively. These values are evaluated as:

$$\left\{\begin{array}{lcl} w(a)_c & = & \sum\{w(s_i)+\max\{w(s_j,s_i):(s_j,s_i)\in\delta(s_i)^-_{S_{c,a}}\}:s_i\in S_c\cap S_a\} \\ w(a)_{CP} & = & \sum\{w(s_i)+\max\{w(s_j,s_i):(s_j,s_i)\in\delta(s_i)^-_{S_{CP,a}}\}:s_i\in S_{CP}\cap S_a\} \end{array}\right. \tag{8.29}$$

where, for a given firing s_i, $\delta(s_i)^-_{S_{c,a}}$ and $\delta(s_i)^-_{S_{CP,a}}$ represent the critical incoming edges and the incoming edges along the CP where source firings belong to the same actor a, respectively. Similarly, for each action λ of an actor, $w(\lambda)_c$ and $w(\lambda)_{CP}$ define the action critical workload and the action workload along the $\overrightarrow{CP}$, respectively. These values are evaluated as:

$$\left\{\begin{array}{lcl} w(\lambda)_c & = & \sum\{w(s_i)+\max\{w(s_j,s_i):(s_j,s_i)\in\delta(s_i)^-_{S_{c,\lambda}}\}:s_i\in S_c\cap S_\lambda\} \\ w(\lambda)_{CP} & = & \sum\{w(s_i)+\max\{w(s_j,s_i):(s_j,s_i)\in\delta(s_i)^-_{S_{CP,\lambda}}\}:s_i\in S_{CP}\cap S_\lambda\} \end{array}\right. \tag{8.30}$$

where, for a given firing s_i, $\delta(s_i)^-_{S_{c,\lambda}}$ and $\delta(s_i)^-_{S_{CP,\lambda}}$ represent the critical incoming edges and the incoming edges along the CP where source firings belong to the same action λ, respectively. Actor-classes, actors and actions can be ranked according to their value of critical workload and workload along the $\overrightarrow{CP}$. Consequently, it is possible to define the actor-class κ^*, the actor a^* and action λ^* that most contribute to the overall $|\overrightarrow{CP}|$ as:

$$\left\{\begin{array}{lcl} \kappa^*_{CP} & = & \mathrm{argmax}\{\kappa_i : w(\kappa_i)_{CP}\geq w(\kappa_j)_{CP}, \forall\kappa_j\in K_{CP}\} \\ a^*_{CP} & = & \mathrm{argmax}\{a_i : w(a_i)_{CP}\geq w(a_j)_{CP}, \forall a_j\in A_{CP}\} \\ \lambda^*_{CP} & = & \mathrm{argmax}\{\lambda_i : w(\lambda_i)_{CP}\geq w(\lambda_j)_{CP}, \forall\lambda_j\in \Lambda_{CP}\} \end{array}\right. \tag{8.31}$$

8.3.2 Impact analysis

If the maximum achievable design throughput T_{max} does not satisfy the design requirements (see Figure 8.3a), the exploration process should initially concentrate on the reduction of the algorithmic complexity of the design, and successively on finding an optimal mapping configuration. In [1, 2] it has been demonstrated how, when dealing with parallel designs, the information obtained exclusively from the evaluation of the $\overrightarrow{CP}$ (e.g. the critical ranking previously described) does not provide a reliable direction for refactoring. Hence, the analysis should be concentrated on estimating, and highlighting the action λ which requires the **less refactoring effort** in order to maximally reduce the $|\overrightarrow{CP}|_{algo}$ (i.e. maximally improve T_{max}, consequently). This estimation can be obtained using the impact analysis technique, which is summarized in Algorithm 4. The $\overrightarrow{CP}_{algo}$ and the set S_{CP} of actions along this path are initially evaluated as illustrated in Section 8.2.2. After that, for each single action $\lambda\in\Lambda_{CP}$ (i.e. that has at least one action firing along the CP) it is estimated how much the $|\overrightarrow{CP}_{algo}|$ can be reduced by reducing the algorithmic complexity of this action. The algorithmic complexity reduction-factor is defined as a value such that $r\in R_w=[1,2,\ldots,100]$. In other words, the $|\overrightarrow{CP}_{algo}|$ is iteratively computed for each $\lambda\in\Lambda_{CP}$ and $r\in R_w$ considering at each evaluation step the following weight configuration:

$$\begin{cases} w(s_i)_r = \begin{cases} \begin{cases} \frac{100-r}{100} w(s_i)_e & \text{if } s_i \in S_\lambda \\ w(s_i)_e & \text{if} s_i \not\in S_\lambda \end{cases} & \text{for homogeneous architecture} \\ \begin{cases} w(s_i)_r + \frac{100-r}{100} w(s_i)_e + w(s_i)_w & \text{if } s_i \in S_\lambda \\ w(s_i)_r + w(s_i)_e + w(s_i)_w & \text{if} s_i \not\in S_\lambda \end{cases} & \text{for heterogeneous architecture} \end{cases} \\ w(s_i, s_j)_r = 0 \end{cases} \tag{8.32}$$

In other words, r estimates the percentage of how much the algorithmic execution time $w(s_i)_e$ should be reduced in order to reduce the ACP length. For each iteration, the corresponding ACP length is denoted as $|\overrightarrow{CP}|(\lambda, r)_{algo}$ and the percentage decrease as:

$$\Delta|\overrightarrow{CP}|(\lambda, r)_{algo} = 100 \left(1 - \frac{|\overrightarrow{CP}|(\lambda, r)_{algo}}{|\overrightarrow{CP}|_{algo}}\right) \in [0, 100] \tag{8.33}$$

Finally, it is possible to clearly identify on which action the refactoring should be concentrated in order to reduce the $|\overrightarrow{CP}|_{algo}$ and, consequently, improve the maximum achievable design throughput T_{max}. As an example, Figure 8.6 depicts an example of impact analysis for three actions λ_1, λ_2 and λ_3, respectively.

Algorithm 4: Impact analysis for the set of critical actions Λ_{CP}.

```
Input: $S$ the firings set
Input: $\overrightarrow{CP}_{algo}$ the initial algorithmic critical path
Result: $\Delta|\overrightarrow{CP}|(\Lambda, R_w)_{algo}$ the ACP length reduction set
Data: $\Delta|\overrightarrow{CP}|(\Lambda, R_w)_{algo} = \emptyset$
for $\lambda \in \Lambda_{CP}$ do
    for $r \in R_w$ do
        // Set the action firing weights
        for $s_j \in S$ do
            $w(s_j) \leftarrow$ Equation (8.32)
        end
        // use Algorithms 1,2,3
        $\overrightarrow{CP}(\lambda, r)_{algo} \leftarrow$ computeCpLength()
        // evaluate the CP length reduction ratio
        $\Delta|\overrightarrow{CP}|(\lambda, r)_{algo} \leftarrow 100\left(1 - \frac{|\overrightarrow{CP}|(\lambda, r)_{algo}}{|\overrightarrow{CP}|_{algo}}\right)$
        $\Delta|\overrightarrow{CP}|(\Lambda, R_w)_{algo} = \Delta|\overrightarrow{CP}|(\Lambda, R_w)_{algo} \cup \Delta|\overrightarrow{CP}|(\lambda, r)_{algo}$
    end
end
```

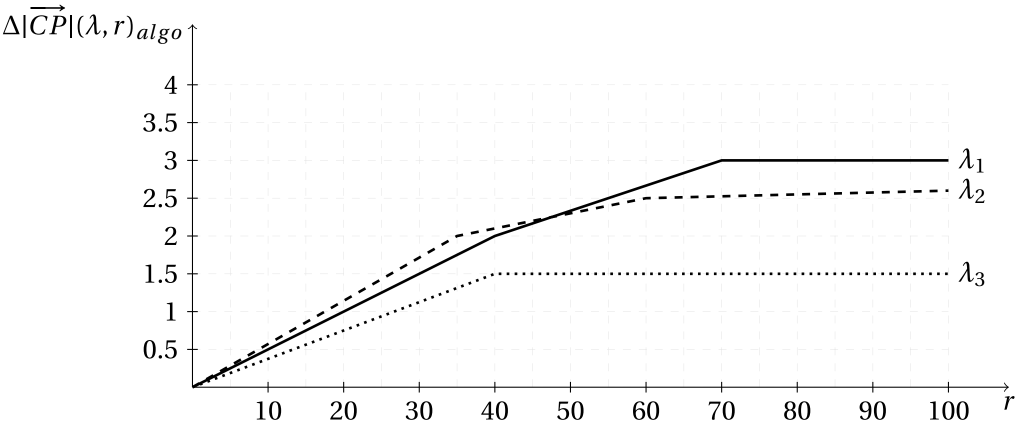


Figure 8.6: Example of impact analysis for three actions λ_1, λ_2 and λ_3.

8.4 Buffer size dimensioning

The total memory size requirement of an application implemented by a dataflow program consists of the sum of two contributions: the code size and the data buffer size. Minimizing the total buffer size can be a very important optimization objective in order to reduce cost of today's FPGAs that have severe embedded-memory limitations. In the domain of SDF, CSDF and DPN designs, which are typically implemented in memory-constrained hardware platforms, the buffer minimization problem is an NP-complete problem [192, 193, 194, 47, 45] which necessitates the use of heuristic algorithms.

8.4.1 Related work

One of the pioneering works on buffer minimization was presented in [195] where an algorithm for scheduling a KPN in-bounded memory was illustrated: while simulating the design using any scheduler and imposing an initial buffer size configuration, the buffer capacity is increased in case of system deadlock caused by buffer overflow. However, this approach is not guaranteed to find the minimum buffer size requirement. Since SDF is a special case of KPN (i.e. see Section 2.1.2), in [196] this approach has also been extended for SDF programs, where a backtracking search is added to the initial algorithm. Some other authors provides model-checking based techniques in order to obtain a close-to-optimal solution [197, 40, 196] by exploring the entire state space. However, the scalability of these techniques is limited by the capabilities of the state space exploration stage and can fail for large-scale systems. All of these heuristic algorithms are suitable only for SDF and CSDF designs and cannot be applied in a DDF context.

8.4.2 Deadlock and feasible regions

From the knowledge of the minimum buffer size configuration, guarantees on the minimum achievable throughput T_{min} of the design can be obtained. As depicted in Figure 8.7, for a given configuration of partitioning and scheduling (π^*, σ^*) the critical path design space can de divided in two different regions according to the buffer size configuration β. These two regions are the **deadlock region** and the **feasible region**, respectively. These are separated by the minimum buffer size configuration β_{min}. An upper and a lower bound for the $\overrightarrow{CP}$ length can be defined as:

$$|\overrightarrow{CP}|(\beta_{min}) \leq |\overrightarrow{CP}|(\beta) \leq |\overrightarrow{CP}|_{algo} \tag{8.34}$$

where $|\overrightarrow{CP}|(\beta_{min})$ defines the critical path length evaluated with the minimal buffer size configuration β_{min}.

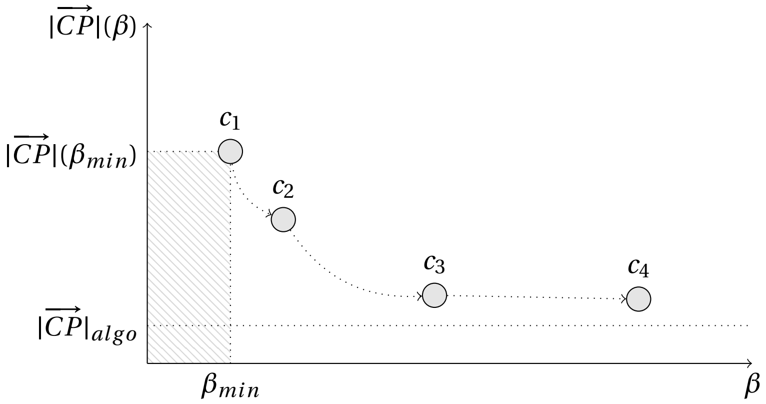


Figure 8.7: Critical path design space given different buffer size configurations.

8.4.3 Minimization by the use of a model predictive control approach

As previously discussed, the problem of bounding and minimizing the buffer size configuration of a dataflow program, without impacting the performance and guaranteeing at the same time a deadlock-free execution, has been proven to be an NP-complete problem. Consequently, it requires the use of heuristic algorithms. In this section, this problem is solved using ETG transformation and treating the program like a linear-discrete event system as illustrated in Section 5.5.3. Considering an ETG with $n_S = |S|$ firings, the problem of bounding (and minimizing) the buffer size configuration, guaranteeing at the same time a deadlock-free execution, can defined as:

$$
\begin{aligned}
\underset{u(k),u(k+1),\dots,u(k+n_S)}{\text{minimize}} \quad & J = \sum_{k}^{n_S} \sum_{j}^{b} y(k)_j \\
\text{subject to} \quad & \text{Equation (5.16)} \\
& y(k)_j \geq 0, \forall k \in \{1,2,\dots,n_S\}, \forall j \in \{1,2,\dots,b\} \\
& \sum_{i}^{n_S} u(k)_i = 1, \forall k \in \{1,2,\dots,n_S\} \\
& \sum_{k}^{n_S} u(k)_i = 1, \forall i \in \{1,2,\dots,n_S\}
\end{aligned}
\tag{8.35}
$$

where $y(k)_j$ represents the j-th component of $y(k)$, i.e. the number of tokens available on the j-th buffer, the constraint $\sum_i^{n_S} u(k)_i = 1, \forall k \in \{1,2,\dots,n_S\}$ requires that a firing $s_i \in S$ is executed at each event k, while the constraint $\sum_k^{n_S} u(k)_i = 1, \forall i \in \{1,2,\dots,n_S\}$ requires that a step must be fired only once (i.e. the deadlock condition is avoided because all the steps in S must be fired). In other words, executing only one firing at each event k, can also be seen as finding a topological order of S for which the sum of all the available tokens along the dataflow network is minimized during the entire execution. However, the problem defined in Equation (8.35) is an integer linear programming (ILP) problem, where the number of optimization variables and constraints can grow significantly according to the ETG size n_S. Consequently, a heuristic algorithm should be applied. For instance, find a feasible scheduling sequence of the ETG such that the buffer size is kept bounded and, if possible, minimized, guaranteeing a deadlock-free execution for all the action firings in S. When dealing with large-data graphs some well-known heuristics, such as graph-cutting or pattern recognition, can be successfully used to reduce the problem size. The heuristics that are illustrated in the following rely on both the formalism of the graph and automatic control theory to minimize the size and to find a sub-optimal solution to Problem (8.35). Model predictive control (MPC) [198, 199] is a receding horizon-control technique where at each event, an optimization problem is solved by predicting the future system behavior (i.e. see Appendix B). Bearing in mind the transformations discussed in Section 5.5.3, it is possible to define an MPC approach that makes use of the ETG, and where the prediction and control horizons are related to the graph-cut used for reducing the problem size.

Deadlock avoidance

As discussed in Section 5.3, one of the main properties of an ETG is that it is completely independent from any buffer size configuration. Moreover, an ETG can have different topological orders with different minimal buffer size requirements for admitting a deadlock-free execution. Therefore, the initial problem can be relaxed: sorting at each event $k \in \{1,2,3,\dots n_S\}$ only one firing, then the optimization problem of Equation (8.35) can be solved iteratively only for a limited set of firings. Consequently, this problem can be solved using the receding horizon control technique through the use of an MPC controller. In this case, the prediction horizon

H_p defines the number of firings of each ETG-cut. At each event k an ETG-cut $S(k)'_{H_p} \subseteq S$ is evaluated so that it contains only H_p unscheduled firings of S with the lowest available topological order. Then, according to the procedure just described, the optimization problem can be formulated as:

$$\begin{aligned} \underset{u(k|k),u(k+1|k),\dots,u(k+H_c-1|k)}{\text{minimize}} \quad & J(k) = \sum_{i}^{H_p}\sum_{j}^{b} y(k+i|k)_j \\ \text{subject to} \quad & y(k+j|k)_i \geq 0,\ \forall j,i \in \{1,2,\dots H_p\} \\ & \sum_{i}^{H_p} u(k+j|k)_i = 1,\ \forall j \in \{0,1,2,\dots H_c-1\} \\ & \sum_{i}^{H_p} u(k+j|k)_i = 0,\ \forall j \in \{H_c, H_c+1,\dots H_p\} \end{aligned} \tag{8.36}$$

where H_c (i.e. the control horizon) is the number of firings that can be executed (i.e. ordered) inside $S(k)'_{H_p}$. At each event k only the first selected firing defined by $u(k)^* = u(k|k)$ is executed. When firing the step defined by $u(k)^*$ the number of tokens inside each buffer is updated accordingly. The minimal buffer size configuration can be defined as the maximal token capacity of each buffer obtained during the entire execution only when all the firings have been executed. In other words, the bounded buffer size configuration is evaluated as:

$$\beta(b_i)_{min} = \max\{y(k)_i,\ \forall k \in \{1,2,3,\dots,n_S\}\} \tag{8.37}$$

Where $\beta(b_i)_{min}$ defines the minimal bounded size of each buffer $b_i \in B$ required for scheduling the ETG. Consequently, the minimal buffer size configuration which defines the borders between the deadlock region and feasible region depicted in Figure 8.7 can be defined as:

$$\beta_{min} = \{\beta(b_1)_{min}, \beta(b_2)_{min}, \dots, \beta(b_{n_B})_{min}\} \tag{8.38}$$

The flowchart of this approach is depicted in Figure 8.8. It must be noted that, if this analysis is performed on a collection of ETGs, then the minimal size value of each buffer is the maximal value obtained within the ETGs collection.

Deadlock recovery

In the previous approach, the problem of Equation (8.36) should be solved n_S times. Another approach, that reduces the number of time that this problem should be solved, is to schedule post-mortem the ETG using a dynamic buffer size configuration that is modified each time that a deadlock condition arises (i.e. the ETG has not action firing that cannot be scheduled because some buffers are full). This second approach can be considered as an improvement of the one introduced in [195], where the key idea is to recover only a blocked action from a deadlock execution that produces the highest number of tokens that could resolve the deadlock condition. However, it must be noted that in [195] any minimization cost function can be used based on the prediction of the program execution and buffer utilization. On the contrary, with this second approach, when a deadlock condition arises, a trace subgraph $S(k)'_{H_p}$ is evaluated as previously described. Successively, the problem of Equation (8.35) is solved in order to identify the next schedulable firings as done for the approach that avoids deadlocks. Hence, the found fired action is scheduled supposing, only at this time, an unbounded buffer size configuration. The new maximum token capacity of each buffer is then used from the successive scheduling, as a new buffer size configuration. It is worth noting that, initially the size of all the buffers can be set as 0 tokens. Only when all the action firings have been scheduled can the bounded buffer size configuration be defined as the maximal token capacity of each buffer obtained during the entire ETG post-mortem scheduling, as defined in Equation (8.37). The flowchart of this approach is depicted in Figure 8.9.

8.4.4 Optimization by the exploration of the design space critical path

When design performance in terms of throughput are not met by using the minimal buffer size configuration, the problem becomes how to increase the size in order to reduce the $|\overrightarrow{CP}(\beta)|$. The problem of exploring the design space, for a given scheduling and partitioning configuration, in order to find a suitable buffer size configuration that guarantees the throughput requirements, is depicted in Figure 8.7. This can be formulated such as:

$$\begin{aligned}
\text{minimize} \quad & J = \begin{cases} |\overrightarrow{CP}|(\beta) \\ \beta(b_i),\ \forall b_i \in B \end{cases} \\
\text{subject to} \quad & \beta(b_i)_{min} \leq \beta(b_i) \leq \beta(b_i)_{max},\ \forall b_i \in B \\
& \sum\{\beta(b_i),\ \forall b_i \in B\} \leq \beta(B)_{max}
\end{aligned} \tag{8.39}$$

where J is a multi-objective cost function over the critical path length $|\overrightarrow{CP}(\beta)|$ and the size $\beta(b_i)$ of each buffer $b_i \in B$. The constraints $\beta(b_i)_{min} \leq \beta(b_i) \leq \beta(b_i)_{max}$ impose that for each buffer the size should be equal or larger to the minimal size $\beta(b)_{min}$ evaluated in the previous section. Moreover, an upper bound on each buffer size $\beta(b_i)$ and on the overall buffer size configuration $\beta(B) = \sum\{\beta(b_i),\ \forall b_i \in B\}$ can be imposed. These additional constraints are necessary when the program is implemented in a severe memory-constrained platform (e.g. DSP, FPGA). Problem (8.39) can be demonstrated to be an NP-complete [10] and, therefore, it needs the use of efficient heuristics in order to find good approximate solutions.

Reducing the critical path length

From Equation (5.10) it can be seen how the overall execution time of each firing $s_i \in S$ is affected by the blocking writing overhead $w(s_i)_{wd}$ introduced by a buffer that cannot accommodate enough tokens. Hence, the objective becomes reducing $|\overrightarrow{CP}|$ by increasing the size of buffers that are responsible for introducing the highest total amount of writing overhead along the $\overrightarrow{CP}$. This can be formulated as an iterative procedure as illustrated in

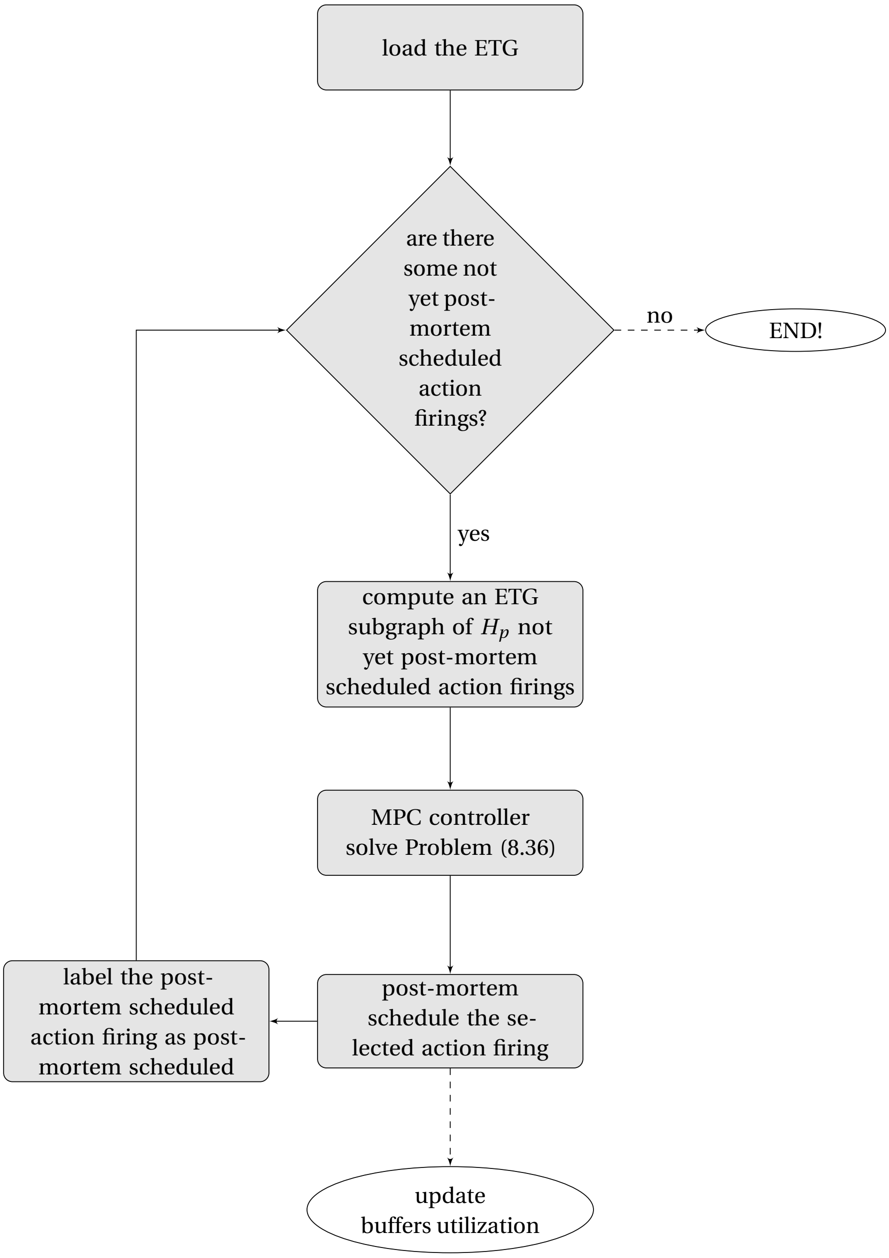

load the ETG
are there some not yet post-mortem scheduled action firings?
no
END!
yes
compute an ETG subgraph of H_p not yet post-mortem scheduled action firings
MPC controller solve Problem (8.36)
post-mortem schedule the selected action firing
label the post-mortem scheduled action firing as post-mortem scheduled
update buffers utilization

Figure 8.8: Bounded buffer scheduling with deadlock avoidance approach.

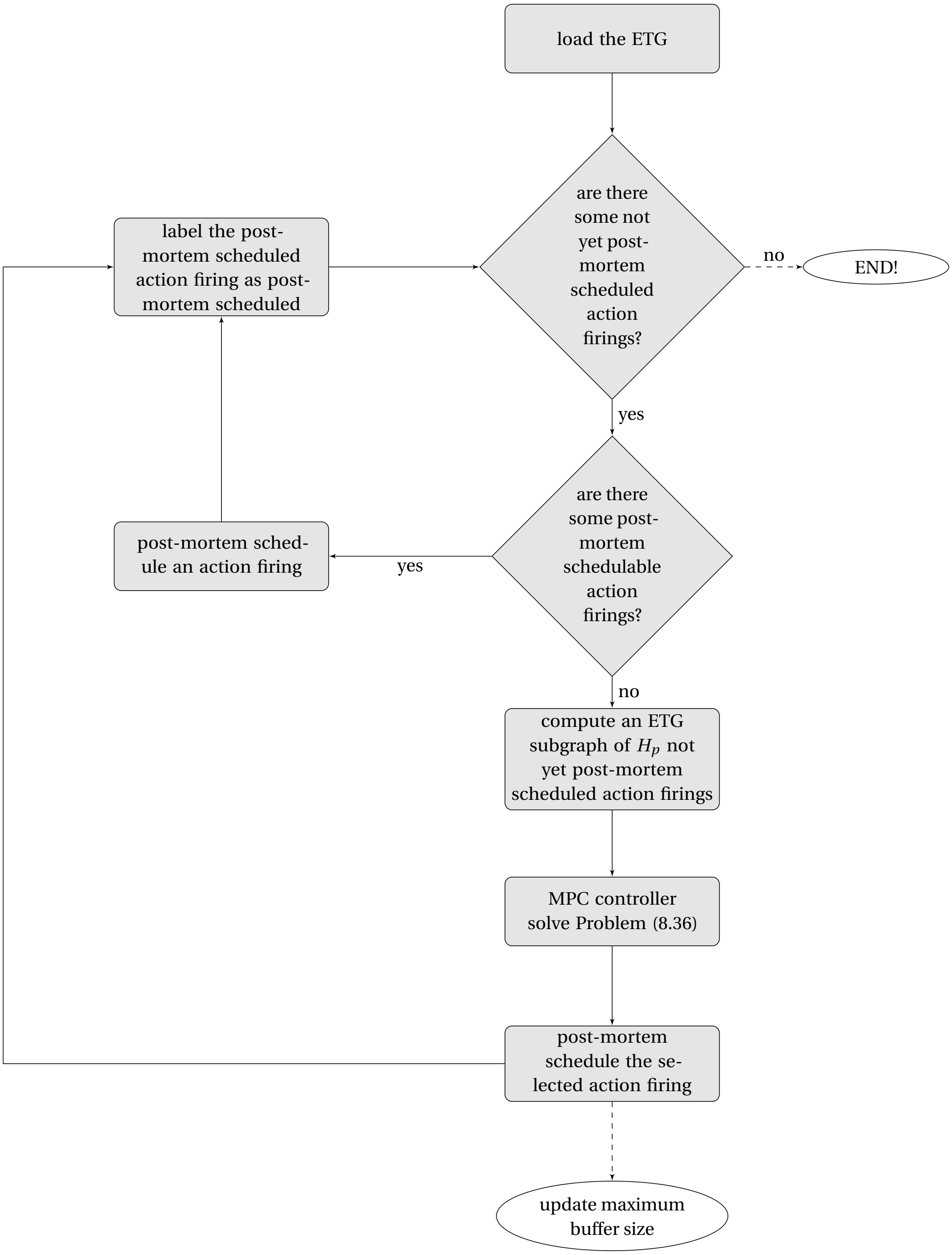

load the ETG
are there some not yet post-mortem scheduled action firings?
no
END!
yes
label the post-mortem scheduled action firing as post-mortem scheduled
are there some post-mortem schedulable action firings?
yes
post-mortem schedule an action firing
no
compute an ETG subgraph of H_p not yet post-mortem scheduled action firings
MPC controller solve Problem (8.36)
post-mortem schedule the selected action firing
update maximum buffer size

Figure 8.9: Bounded buffer scheduling with deadlock recovery approach.

Algorithm 5. At each iteration k, the $\overrightarrow{CP}(\beta(k))$ is evaluated by scheduling post-mortem the ETG according to the buffer configuration $\beta(k)$. At $k = 0$, the minimal buffer size β_{min} is used as the starting point of the algorithm. Successively, for each firing s_i along the $\overrightarrow{CP}(\beta(k))$, a tuple (s_i, b_i, d, τ) is computed for each buffer b_i that introduced a write delay (i.e. $w(s_i)(k)_{wd} > 0$). Each tuple contains the write delay time $d \leq w(s_i)(k)_{wd}$ introduced during the firing of s_i by the buffer b_i and the corresponding number of blocked tokens τ (i.e. that caused the delay of the execution because they could not be accommodated in b_i). Each tuple is then stored in the set $B(k)_{CP}$. When $B(k)_{CP}$ has been completely determinate, it is possible to obtain the following information for each buffer $b \in B$:

$$\begin{cases} d(b_i, k) & = & \sum\{d : n = i, \forall (s, b_n, d, \tau) \in B(k)_{CP}\} \\ \tau(b_i, k)_{max} & = & \max\{\tau : n = i, \forall (s, b_n, d, \tau) \in B(k)_{CP}\} \end{cases} \tag{8.40}$$

where $d(b_i, k)_d$ and $\tau(b_i, k)_{max}$ define the overall write blocking delay and the maximal number of tokens for each buffer b_i. Successively, the buffer that needs to be increased in size is defined as follow:

$$b_k^* = \operatorname{argmax}\{b_i : d(b_i, k) > d(b_j, k) \wedge \beta(b_i, k) + \tau(b_i, k)_{max} \leq \beta(b_i)_{max}, \forall b_j \in B\} \tag{8.41}$$

Once $b(k)_i^*$ has been found, the new buffer size configuration is modified as:

$$\beta(b_i, k+1) = \begin{cases} \beta(b_i, k) + \tau(b_i, k)_{max} & \text{if } b_i = b_k^* \\ \beta(b_i, k) & \text{otherwise} \end{cases} \tag{8.42}$$

A new iteration is then made following the same approach. The heuristic can conclude when the desired critical path length reduction has been achieved (or the maximal number of iterations k has been performed). It must be noted that, if this analysis is performed on a collection of ETGs, then the optimal size value of each buffer is the maximal value obtained within the ETGs collection.

8.5 Dynamic power dissipation minimization

Even though technological improvements in current VLSI design have led to higher clock frequencies, larger dies, and higher transistor density, they have created significant design challenges as a result of power consumption and the need for synchrony at higher speeds [200, 201]. As a result, the performance of applications does not necessarily increase at the same pace. The two main limiting factors are: the technological constraints (e.g. clock frequency caps imposed by wire delays and clock skew) and the requirement constraints (e.g. power consumption, low-noise and robustness). In order to address these issues, previous work has demonstrated that asynchronous circuits have the potential of achieving substantially higher performance compared to their synchronous equivalents. In addition to the elimination of clock skew and lower interconnection delays, asynchronous circuits have other advantages

Algorithm 5: Critical path length reduction by increasing the size of critical buffers.

```
Input: $S$ the firings set
Input: $\beta_{min}$ the minimal buffer size configuration evaluated as discussed in Section 8.4.3
Input: $|\overrightarrow{CP}|_{algo}$ the algorithmic critical path length
Result: $\beta_{opt}$ the optimal buffer size configuration
Data: $k = 0$ the iteration number
Data: $\beta(k)$ buffer size configuration at iteration $k$
// Find the last CP firing
$\beta(k) \leftarrow \beta_{min}$ do
    tracePostProcess($\beta(k)$)
    // use Algorithms 1,2,3
    $|\overrightarrow{CP}(\beta(k))| \leftarrow$ computeCpLength()
    $B(k)_{CP} \leftarrow$ getCriticalBuffers($\overrightarrow{CP}(\beta(k))$)
    $\beta(k) \leftarrow$ Equation (8.42)
    $k \leftarrow k+1$
while $B(k)_{CP} \neq \emptyset \wedge \frac{|\overrightarrow{CP}(\beta(k))|}{|\overrightarrow{CP}|_{algo}} > \epsilon$
$\beta_{opt} \leftarrow \beta k$
begin getCriticalBuffers($\overrightarrow{CP}$)
    $B_{CP} \leftarrow \emptyset$
    for $s_i \in S_{CP}$ do
        if $w(s_i)_{wd} > 0$ then
            $(s_i, b_i, d, \tau) \leftarrow$ getBlockingBuffers($s_i$)
            $B_{CP} \leftarrow B_{CP} \cup (s_i, b_i, d, \tau)$
        end
    end
    return $B_{CP}$
end
```

such as a higher tolerance to the influence of the external environment. On the other hand, the main drawbacks are the complexity of the implementation and the overall power consumption. In an asynchronous design process, performance evaluation, optimization and implementation are complicated by the presence of complex dependencies among concurrent events. While performance estimation for synchronous systems is based mainly on the static analysis of the critical path, the performance of an asynchronous design is related to several dynamic factors [201]. Moreover, performance estimation and design optimization of asynchronous systems are not supported by efficient and comprehensive automatic synthesis and optimization tools. An interesting trade-off between complete synchronous and asynchronous methodology is the globally asynchronous locally synchronous (GALS) clocking-style, supported by multiple-clock domain (MCD) architectures. The key features of a GALS system are the use of distinct local and independent clocks (i.e. with different frequencies and phases), rather than a global timing reference. In a typical GALS configuration, a GALS module (also called synchronous island) consists of a synchronous module, a clock generator and an asynchronous wrapper (i.e. that encapsulates the synchronous module). GALS modules communicate with each other through asynchronous interfaces. For GALS-based applications implemented on MCD architectures, the design objective is to optimize the mapping of the application into multiple clock domains, subsequently assigning a clock frequency to each clock domain in order to reduce the overall power consumption, while at the same time, meeting the design performance requirements.

8.5.1 Related work

The idea of using the dataflow representation for GALS-based applications was introduced in [202] where advantages and disadvantages of this approach are discussed. In the following, a one-to-one correspondence between hardware resources in the architecture and actors is assumed. Dataflow design modeling, exploration and optimization for GALS-based designs have been studied previously by several authors. For example in [203] a GALS design-partitioning method for high performance and very large VLSI systems is illustrated. The system is partitioned into an optimal configuration of synchronous blocks by exploring relationships between power consumption and the number of synchronous blocks which define the granularity of this approach. In this case, the main limitation is that the synchronous blocks have fixed sizes that cannot be changed during the optimization process. Moreover, this approach does not take system performance during the optimization process into account. In [202], a design and evaluation framework is provided for modeling application-specific GALS-based dataflow architectures for CSDF, where system performance (e.g. throughput) during optimization is taken into account. Similarly, in [204, 200], a method for automatic synthesis of asynchronous digital systems is discussed. However, both the approaches are developed for fine-grained dataflow graphs, where actors are primitives or combinational functions.

8.5.2 Multi-clock domain partitioning

The problem of partitioning an isomorphic GALS dataflow application into MCD architectures can be defined as finding a suitable *actor-clock* mapping configuration that employs the lowest clock frequencies that meet the overall design performance requirements. If $F = \{f_1, f_2, \ldots, f_{n_F}\}$ defines the set of available clock frequencies of a platform and $A = \{a_1, a_2, \ldots, a_{n_A}\}$ the set of actors (i.e. see Section 2.5), then the mapping (i.e. partitioning) function can be defined as:

$$\rho : A \to F \tag{8.43}$$

Consequently, the problem can be formulated as follows:

$$\begin{aligned} &\text{minimize} && J = \sum\{c_a \rho(a), \forall a \in A\} \\ &\text{subject to} && \mathrm{T}(\rho) \geq \mathrm{T}_{min} \end{aligned} \tag{8.44}$$

where c_a is a generic objective function weight and $\mathrm{T}(\rho)$ is the design performance function in terms of throughput. In other words, the goal is to find a partitioning configuration ρ that reduces the total dynamic power dissipation of the design without degrading the performance.

8.5.3 Linear programming formulation

Using the notion of ETG and critical path length, the problem of Equation (8.44) can be formulated as a linear programming (LP) [14, 13]. For this purpose, weights $w(s_i)$ of each firing $s_i \in S$ and $w(s_i, s_j)$ of each dependency $(s_i, s_j) \in E$ are evaluated by scheduling post-mortem the ETG where each actor has been mapped with the highest available clock frequency $f_{max} = \max\{f_i \in F\}$. Successively, the ETG dependency amalgamation transformation is used (i.e. see Section 5.5.2). For each amalgamated dependency $e_1 \bullet e_2 \bullet \ldots \bullet e_n$ the corresponding weight is evaluated as $w(e_1 \bullet e_2 \bullet \ldots \bullet e_n) = \max\{w(e_1), w(e_2), \ldots, w(e_n)\}$. Successively, the critical path $\overrightarrow{CP}$ and its length $|\overrightarrow{CP}|$ are computed as described in Section 8.2.1. Successively, the firing extension graph $G(V, E)$ of the ETG is computed (i.e. see Section 5.5.1). For each edge $e \in E$, weights are assigned such that $w(e) = w(s_i)$ if e corresponds to a firing $s_i \in S$, and $w(e) = w(s_i, s_j)$ if e corresponds to a dependency $(s_i, s_j) \in D$. It must be noted that for each fictitious edge $(\pi_s, \pi^{s_i}_{2i-1}) \in E$ and $(\pi^{s_i}_{2i}, \pi_t) \in E$ weights are defined as $w(\pi_s, \pi^{s_i}_{2i-1}) = 0$ and $w(\pi^{s_i}_{2i}, \pi_t) = 0$, respectively. Finally, the clock domains partitioning problem of Equation (8.44) can be defined as follows:

$$
\begin{array}{ll}
\text{maximize} & \sum\{c_a\gamma(a), \forall a \notin A_{CP}\} \\
\text{subject to} & \varphi(\pi_t) - \varphi(\pi_s) = |\overrightarrow{CP}| \\
& \varphi(\pi^{s_i}_{2i}) - \varphi(\pi^{s_i}_{2i-1}) = w(s_i),\ \forall s_i \in S_{CP} \\
& \varphi(\pi^{s_j}_{2j-1}) - \varphi(\pi^{s_i}_{2i}) = w(s_i, s_j),\ \forall (s_i, s_j) \in D_{CP} \\
& \varphi(\pi^{s_i}_{2i-1}) - \varphi(\pi_s) = 0,\ \forall s_i \in \{s_i \in S_{CP} : \delta(s_i)^-_S = \emptyset\} \\
& \varphi(\pi_t) - \varphi(\pi^{s_i}_{2i}) = 0,\ \forall s_i \in \{s_i \in S_{CP} : \delta(s_i)^+_S = \emptyset\} \\
& \varphi(\pi^{s_i}_{2i}) - \varphi(\pi^{s_i}_{2i-1}) \geq \gamma(a) w(s_i),\ \forall s_i \notin S_{CP} \\
& \varphi(\pi^{s_j}_{2j-1}) - \varphi(\pi^{s_i}_{2i}) \geq w(s_i, s_j),\ \forall (s_i, s_j) \notin D_{CP} \\
& \varphi(\pi^{s_i}_{2i-1}) - \varphi(\pi_s) \geq 0,\ \forall s_i \in \{s_i \notin S_{CP} : \delta(s_i)^-_S = \emptyset\} \\
& \varphi(\pi_t) - \varphi(\pi^{s_i}_{2i}) \geq 0,\ \forall s_i \in \{s_i \notin S_{CP} : \delta(s_i)^+_S = \emptyset\} \\
& \gamma(a) = 1,\ \forall a \in A_{CP} \\
& \gamma(a) \geq 1,\ \forall a \notin A_{CP}
\end{array}
\tag{8.45}
$$

where $\varphi(\pi)$ and $\gamma(a)$ are the unknown variables of this problem. One of the well-known LP techniques can be used to solve this problem. Once Problem (8.45) has been solved, the MCD mapping (i.e. partitioning) function defined in Equation (8.43) is obtained as follows:

$$
\rho(a) = \begin{cases} \frac{f_{max}}{\gamma(a)}, & \text{if } a \notin A_{CP} \\ f_{max}, & \text{if } a \in A_{CP} \end{cases}
\tag{8.46}
$$

In other words, the clock can be reduced only for *non-critical* actors. Additional constraints can be added in order to impose clock partitions among actors. The solution of this LP problem provides an optimal clock domain partition configuration. However, the number of constraints is at least $|S| + |D|$. Hence, the use of a heuristic algorithm must be considered when dealing with complex dataflow designs that require the exploration of *"large"* ETGs.

8.5.4 Heuristic approach

The following sections describe a heuristic approach, where the only assumption made is that the number of available clock frequency domains is known a-priori as $F = \{f_1 = f_{max}, f_2, \ldots, f_{n_F}\}$ where $f_i > f_j : \forall i < j,\ i = 1, 2, \ldots, n_F$. The heuristic is illustrated in Algorithm 6, where $\rho(f_i)^{-1}$ defines the set of actors such that $\rho(a) = f_i$. Initially, all the actors are assigned the highest available clock frequency (i.e. $\rho(f_1)^{-1} = A = \{a_1, a_2, \ldots, a_{n_A}\}$ and $\rho(f_i)^{-1} = \emptyset, \forall i > 1$). The $|\overrightarrow{CP}|$ is then calculated with respect to the performance constraints. Iteratively, at each k-step the maximum reduction set R^k is defined as $R^k = \{a : a \notin A^{k-1}_{CP} \wedge a \in \rho(f_{k-1})^{-1}\}$. $\rho(f_k)^{-1} \subseteq R^k$ is then calculated so that the $|\overrightarrow{CP}|$ does not increase. It must be noted that this approach does not claim to find the optimal solution, but provides a practical approach to be applied to complex dataflow programs [13, 14]. It must be noted that, if this analysis is performed on a collection of ETGs, then the clock domain of each actor is the one with the highest frequency obtained within the ETGs collection.

Algorithm 6: Heuristic algorithm for solving the problem of the multi-clock domain partitioning defined in Equation (8.45).

```
Input: $G(S,D)$ the execution trace graph
Result: $\rho(a)$ the clock-actor mapping function
Data: $k=0$ iteration number
Data: $\rho(f_0)^{-1}=A$ and $\rho(f_i)^{-1}=\emptyset, \forall i>1$
// use Algorithms 1,2,3
$|\overrightarrow{CP}(0)| \leftarrow$ computeCpLength($\rho(f_1)^{-1}$)
do
    for $a \in R^k$ do
        $\rho(f_{k-1})_*^{-1} \leftarrow \rho(f_{k-1})^{-1} \setminus \{a\}$
        $\rho(f_k)_*^{-1} \leftarrow \rho(f_k)^{-1} \cup \{a\}$
        // use Algorithms 1,2,3
        $|\overrightarrow{CP}(k)| \leftarrow$ computeCpLength($\rho(f_1)^{-1}, \rho(f_2)^{-1}, \dots, \rho(f_{k-2})^{-1}, \rho(f_{k-1})_*^{-1}, \rho(f_k)_*^{-1}$)
        // if this is a feasible configuration
        if $|\overrightarrow{CP}(k)| = |\overrightarrow{CP}(0)|$ then
            $\rho(f_k)^{-1} \leftarrow \rho(f_k)_*^{-1}$
        end
        $k \leftarrow k+1$
    end
while $\rho(f_k)^{-1} \neq \emptyset$
```

8.6 Conclusions

In this chapter, different DSE functionalities based on the analysis of the ETG of a dataflow program have been illustrated. Firstly, it has been shown how the ETG can be processed (i.e. scheduled) post-mortem in order to evaluate the performance estimation for a given mapping configuration of the program. The methodology has been illustrated for modeling the target architecture and scheduling post-mortem accordingly the ETG in order to assign a timing information (i.e. weight) for each action firing and each dependency of the ETG. Secondly, the concept of design space critical path (DSCP) has been formulated and related with the throughput of a design. This formalism has been used to effectively restring the design space that should be analyzed. Furthermore, by considering the CP of the program, the definition of potential speedup formulated in the Amdahl's law has been revisited and adapted to dataflow programs. Based on the CP analysis of a program, the hotspots analysis has been introduced. This can be used to highlight to the designer which part of a CAL program should be refactored in order to meet performance throughput. The problem of bounding the buffer size configuration of a dynamic dataflow program has been solved by using advanced control techniques such as a model predictive controller. A heuristic, based on the analysis of the CP, has been used to evaluate a reasonable trade-off between design throughput and buffer size configuration. Lastly, the problem of reducing the dynamic power dissipation of a dataflow execution has been studied for multi-clock domain architectures. A linear problem (LP) formulation and a heuristic approach have been illustrated to find partitioning configurations such that the energy consumption is minimized by guaranteeing at the same time the same throughput performance of the design.

9 Experimental results

In this chapter, a collection of experimental results based on the analysis of image and video codec applications is presented. These applications are JPEG, MPEG4-SP, and an HEVC decoders. Applications have been implemented in different target architectures. These are a multi-core i7 desktop CPU, an SThorm many-core platform and a Xilinx Virtex-5 FPGA. In the following, the critical path design exploration and the code refactoring assisted by the impact analysis is illustrated using an HEVC video decoder implemented in a multi-core i7 desktop CPU as a design case. Successively, the bounded buffer size heuristic, based on the use of an MPC controller, is illustrated using both a JPEG and an HEVC decoder as design cases. The optimal trade-off between buffer size dimensioning and throughput performance is then discussed for an MPEG4-SP decoder implemented on an SThorm many-core platform. Finally, the dynamic power dissipation minimization heuristic is illustrated on an MPEG4-SP decoder implemented on a Xilinx Virtex-5 FPGA.

9.1 Design cases

In the following, three multimedia applications specified using the RVC-CAL formalism [52, 53, 54, 55] are illustrated and used in the rest of this chapter. These are respectively a JPEG image decoder, an MPEG4-SP video decoder and an MPEG HEVC video decoder.

9.1.1 JPEG decoder

The first example is a JPEG decoder described using the RVC-CAL formalism [205]. The top-level network of this design is depicted in Figure 9.1. This is composed of 8 subnetworks (actor/network composition), 8 actor-classes, and 8 actors. The main functional components are a JPEG Parser, Huffman decoder, inverse quantization (IQ) and inverse discrete cosine transform (IDCT) block, respectively. Input to the decoder is a compressed 4:2:0 bit-stream and output is the decoded image.

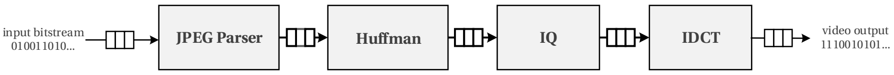


Figure 9.1: JPEG decoder.

9.1.2 MPEG4-SP decoder

The second design example is an MPEG-4 simple profile (SP) decoder described using the RVC-CAL formalism [31]. The top-level network of this design is depicted in Figure 9.2. This is composed of 8 subnetworks (actor/network composition), 27 actor-classes and 42 actors. The main functional components are a bit-stream parser, and for each deconding component (i.e. Y, U, V) a reconstruction, 2D-IDCT, frame buffer, and motion compensation block, respectively. The merging block makes a composition of the Y, U and V parts. Input to the decoder is a compressed 4:2:0 bit-stream and output is the decoded video sequence.

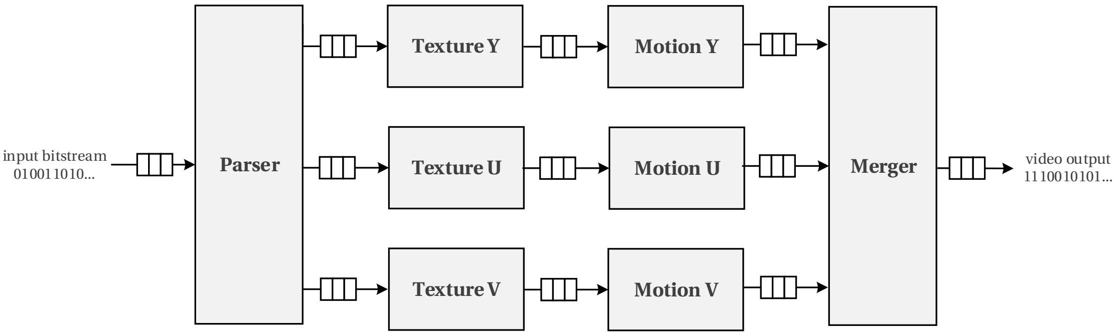


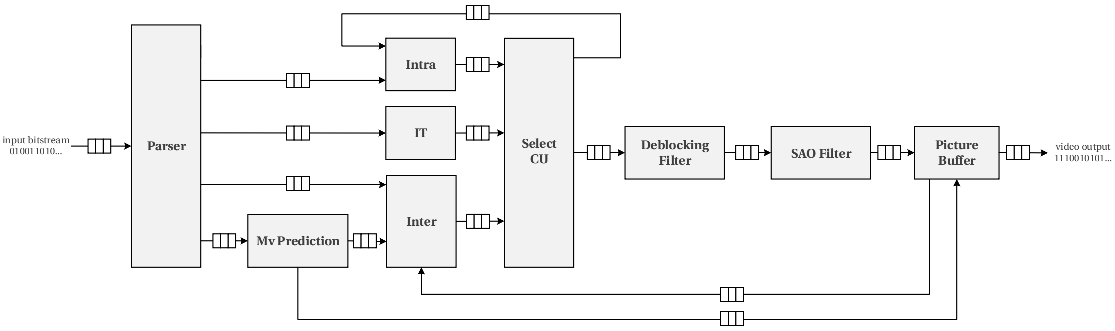
Figure 9.2: MPEG4-SP decoder.

9.1.3 MPEG-HEVC decoder

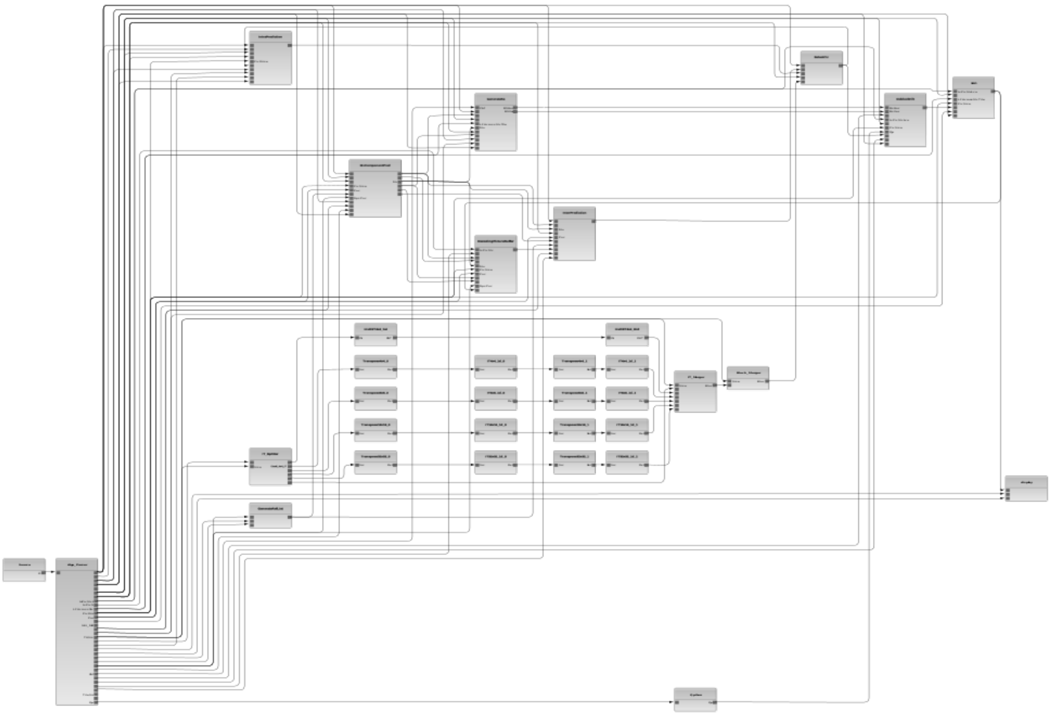
The third example is an MPEG-HEVC decoder described using the RVC-CAL formalism [9]. The top-level network of this design is depicted in Figure 9.3a. The basic version of this design is composed of 9 subnetworks (actor/network composition), 16 actor-classes and 32 actors. The main functional components are a bit-stream parser, moving prediction (MovPred), intra-prediction (Inra), inter-prediction (Inter), IDCT, reconstruct coding unit (RecCU), select coding unit (SelCU), deblocking filter (DebFilter), sample adaptive offset filter (SaoFilter) and decoding picture buffer (DecPicBuff) block, respectively. Input to the decoder is a compressed 4:2:0 bit-stream and output is the decoded video sequence.

9.2 CAL source code static and dynamic profiling

This section presents the experimental results on the high-level design profiling, illustrated in Chapter 3 and 7 respectively, for an MPEG-HEVC decoder. The design under study is the RVC-CAL standardized version [206] of the decoder illustrated in Section 9.1.3. The complete



(a) Top-level network with 9 subnetworks.



(b) RVC-CAL standardized version (without subnetworks).

Figure 9.3: HEVC decoder.

network topology (without subnetworks) is depicted in Figure 9.3b. This design is composed of 32 actors, 26 actor-classes and 112 buffers. In the following, this initial design configuration is referred to as *Ref-Standard*.

9.2.1 Source code static analysis

The first step of a high-level design profiling is the static source-code analysis illustrated in Section 3.2. Results of this analysis performed on the HEVC Ref-Standard design are summarized in Table 9.1 where the overall number of SLOC and Halstead metric values are reported.

Table 9.1: Static code complexity of the MPEG-HEVC decoder.

(a) Network composition in terms of actors, buffers, internal variales, actor-classes and SLOC.

	Actors	**Buffers**	**Internal variables**	**Classes**	**SLOC**
Ref-Standard	32	112	745	26	$1.46\ 10^4$
Shared-Memory	13	96	800	13	$1.37\ 10^4$

(b) Halstead complexity metric results.

	n1	n2	n	N1	N2	N
Ref-Standard	$3.20\ 10^2$	$2.35\ 10^3$	$2.67\ 10^3$	$3.29\ 10^4$	$3.33\ 10^4$	$6.62\ 10^4$
Shared-memory	$4.25\ 10^2$	$2.45\ 10^3$	$2.87\ 10^3$	$3.19\ 10^4$	$3.36\ 10^4$	$6.55\ 10^4$

	V	D	E	T	B	I
Ref-Standard	$7.54\ 10^5$	$2.24\ 10^3$	$1.69\ 10^9$	$9.38\ 10^7$	$3.33\ 10^{-4}$	$3.37\ 10^2$
Shared-memory	$7.52\ 10^5$	$2.76\ 10^3$	$2.07\ 10^9$	$1.15\ 10^8$	$3.33\ 10^{-4}$	$2.73\ 10^2$

Table 9.2: Actor memory requirements for the initial HEVC design.

	Internal variables	**Bit size**
Ref-Standard	745	12.25MB
Shared-memory	800	732.0kB

The overall design SLOC is 14658, which is smaller compared to the approximatively 110000 C/C++ SLOC of openHEVC implementation [207, 208]. According to Halstead development time T, the HEVC Ref-Standard design is a 37.8 man-months (i.e. $T = 9.38\ 10^9$s) project when developed using RVC-CAL as the programming language.

9.2.2 Memory requirements and utilization

The overall memory requirements, in terms of internal variables and buffer utilization, of a design can be estimated through a static and dynamic high-level code analysis. The overall amount of bits required for the actors' internal variables can be evaluated through a static code analysis. In fact, in RVC-CAL the dimension of each internal variable should be known at compile-time, as such no dynamic memory allocations can be made. This makes it possible to estimate exactly the static memory requirement for each internal variable. As summarized in Table 9.2, the memory requirement for the HEVC *Ref-Standard* design is 12.25MB. On the contrary, a dynamic code analysis is required for estimating the memory requirements in terms of tokens passed on each buffer because of the dynamic dataflow MoC of this design. As illustrated in Section 3.3, this analysis can be performed with a high-level code interpretation of the CAL source code. Table 9.3a summarizes the buffer utilization and requirement when

Table 9.3: Buffer utilization profiling data of the MPEG-HEVC decoder.

(a) Tokens passed through the network (i.e. produced and sucessively consumed).

	Total	**Bits**
Ref-Standard	1000824	16.91MB
Shared-Memory	213586	1.37MB

(b) Buffer bandwidth in terms of produced and consumed tokens for each firing.

	Produced-tokens/Firing			**Consumed-tokens/Firing**		
	Average	**Min**	**Max**	**Average**	**Min**	**Max**
Ref-Standard	49.74	1	4096	49.75	1	4096
Shared-Memory	3.16	1	8192	3.16	1	8192

(c) Write and read hits on each buffer.

	Write hits				**Read hits**			
	Total	**Average**	**Min**	**Max**	**Total**	**Average**	**Min**	**Max**
Ref-Standard	1000824	8935.9	8	400210	1052894	9400.8	8	400210
Shared-Memory	213586	2224.8	8	16082	235764	2455.8	8	16694

the 8-frame 416x240 `MERGE_B_TI_3` HEVC conformance bit-stream [209] is used as input for the design. In this case, the number of tokens passed through the design buffers is approximatively 10^6 which require 16.91MB to be represented. Furthermore, during the dynamic code interpretation it is possible to evaluate the number of write and read accesses that are performed on each buffer. Table 9.3b and 9.3c illustrate, respectively, the number of tokens produce/consumed each time a firing makes use of a buffer and the number of write/read accesses that have been performed on each buffer.

9.2.3 Execution trace graph

The structure of the ETG evaluated during the high-level code interpretation described in the previous section is summarized in Table 9.4. This is composed of approximatively $2\ 10^6$ firings (nodes of the graph) and $2.2\ 10^7$ dependencies (directed arcs). As can be seen from Table 9.6b, the ETG is (in general) a low-connected graph. In fact, for this particular case the incoming and outgoing degree is approximatively 11.64. Figure 9.4 depicts the rendering of a small portion (i.e. approximatively 80000 action firings and 350000 dependencies) of this ETG made with the Gephi graph-visualizer [210, 211].

9.2.4 Initial design-refactoring directions

From the high-level profiling information obtained during the analysis described in the previous sections, it is possible to identify the buffer utilization as a critical point of the HEVC

Table 9.4: Execution trace graph configuration of the Ref-Standard MPEG-HEVC decoder.

(a) Size.

Action firings	**Dependencies**
1932226	22490814

(b) Action firings incoming and outgoing degree.

	Average	**Min**	**Max**	**Var**
$\|\delta(s_i)^-\|$	11.64	0	200	54.63
$\|\delta(s_i)^+\|$	11.64	0	61530	12143.96

(c) Dependencies set.

		Direction					
Kind	**Total**	**input**	**output**	**rr**	**rw**	**wr**	**ww**
FSM	8.2%	-	-	-	-	-	-
Port	8.6%	54.2%	45.8 %	-	-	-	-
Internal variable	77.8%	-	-	34.4%	16.2%	32.6%	16.8 %
Tokens	5.4%	-	-	-	-	-	-

Ref-Standard design. In fact, the overall number of exchanged tokens between actors is approximatively 16.91MB. This issue can be effectively coped with an extension of the CAL MoC by introducing the notion of shared-memory between actors. If correctly used, this insight enables reducing the overall number of exchanged tokens, without introducing (possible) race-conditions. In other words, the shared-memory approach should only be used if an actor modifies the value of an internal variable where the values are used without any modification (i.e. read only access) by a second actor. Without this approach, each time that the first actor modifies an internal variable then the new value should be sent as a token to the second actor. In the case where the processing result of an actor does not depend on the arrival time of the token and its value is not locally modified, then the shared-memory approach can be used. The consumption of a token is transformed to a read-access of a variable. This is the case, for example, for the `DecPicBuffer` actor. This actor receives the decoded picture as a stream of tokens. However, the decoded picture can be shared among actors without requiring the use of tokens by the use of a shared-memory approach. More over, additional internal variables, necessary to store the token data values, can be removed. Following these considerations, the HEVC Ref-Standard design has been modified by supporting the shared-memory MoC. This new design configuration, summarized in Table 9.1 and referred to as *Shared-Memory*, is composed of 13 actors and actor-classes and 96 buffers. The same static and dynamic code analysis illustrated before has also been performed for this new design configuration. Table 9.5 summarizes the internal variables that can be shared among the actors of the HEVC Shared-Memory design. It is possible to see how the overall size of the shared-memory is

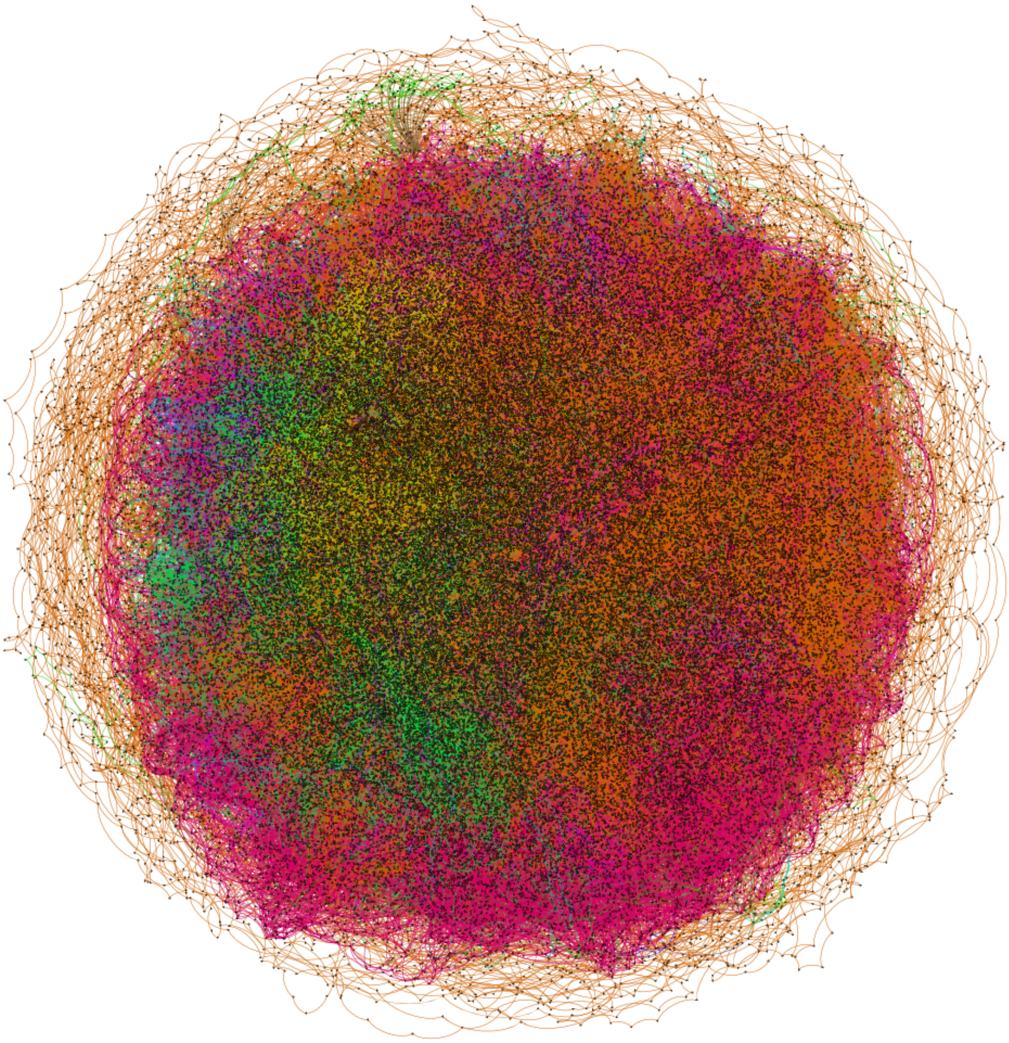


Figure 9.4: The rendering of a small portion (i.e. approximatively 80000 action firings and 350000 dependencies) of the execution trace graph described in Table 9.4. Action firings are colored according to the corresponding actor.

22kB, However, the real memory requirement reduction is obtained by removing internal actor variables that store redundant data and by reducing the overall number of exchanged tokens. In Table 9.3a and 9.3, respectively, it can be seen how the overall internal memory of the Shared-Memory design is 732kB (94% smaller compared to the Ref-Standard version) and the overall amount of exchange tokens is 1.37MB (92% smaller compared to the Ref-Standard version). The ETG structure obtained with this new design configuration is summarized in Table 9.6. This ETG is used in the following, where the hotspots analysis is performed.

Table 9.5: Memory requirement for the actor internal variables of the Shared-Memory MPEG-HEVC decoder.

		Shareable	
Internal variables	**Bit size**	**Variables**	**Bits size**
811	754KB	11	22.0kB

Table 9.6: Execution trace graph configuration of the Shared-Memory MPEG-HEVC decoder.

(a) Size

Action firings.	**Dependencies**
493551	6124379

(b) Action firings incoming and outgoing degrees.

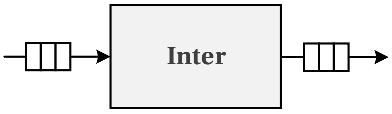
Degree	**Average**	**Min**	**Max**	**Var**
$\lvert\delta(s_i)^-\rvert$	12.40	0	84	146.02
$\lvert\delta(s_i)^+\rvert$	12.40	0	38643	9180.69

(c) Dependencies set.

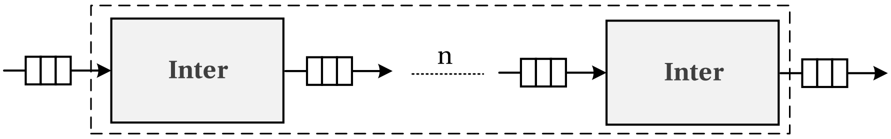
		Direction					
Kind	**Total**	**input**	**output**	**rr**	**rw**	**wr**	**ww**
FSM	8.0%	-	-	-	-	-	-
Port	6.0%	63.2%	36.8 %	-	-	-	-
Internal variable	82.0%	-	0	31.4%	18.9%	27.7%	22.0 %
Tokens	4.0%	-	-	-	-	-	-

9.3 Design refactoring

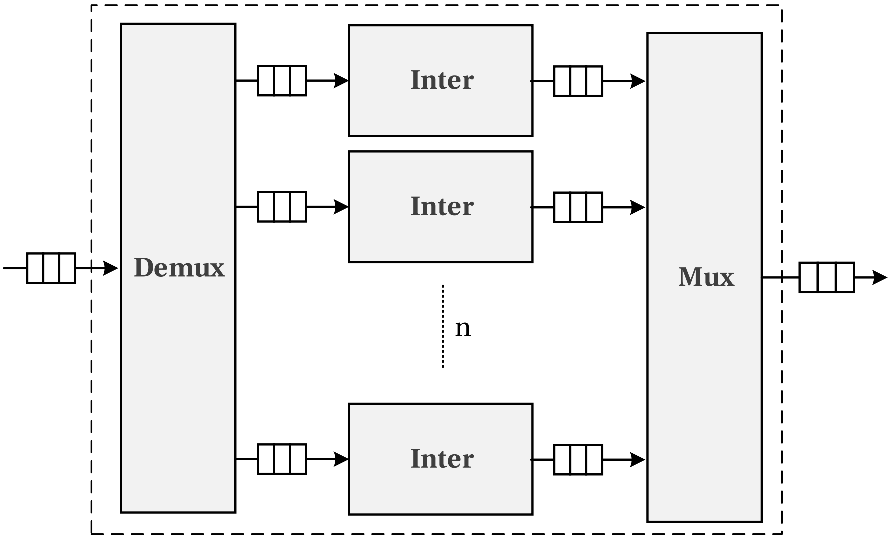
This section presents the experimental results of the design space critical path exploration and hotspots analysis, illustrated in Section 8.2 and 8.3 respectively, for an MPEG-HEVC decoder. The design under study is the decoder illustrated in Section 9.2.4, referred to as HEVC Shared-Memory, specified using the RVC-CAL dataflow formalism extended with the notion of shared-variables. The target platform for this implementation is a desktop computer equipped with an Intel i7-3770 3.40GHz processor and 8GB of memory. The objectives of this analysis are two-fold: improve the throughput performance of the initial design, increase the potential speedup $S(n)$ in order to fully exploit the 4 cores. To achieve both objectives, the designer has reduced the algorithmic critical path length $|\overrightarrow{CP}|_{algo}$ following the refactoring directions provided by the DSE framework.



(a) Initial version



(b) Pipeling recplication



(c) Data parallelism

Figure 9.5: Refactoring strategies for the `Inter-Prediction` actor.

9.3.1 Critical action ranking

As illustrated in Table 9.8b, the initial maximal speedup (i.e. achievable with $n = n_A$ processing elements) of this design is 2.43. The first step of the refactoring phase was to evaluate the critical action ranking. As illustrated in Section 8.3, the objective is to identify which actions contribute the most to the serial part of the design. Table 9.7a summarizes the list of the first 5 actions which contribute the most to the overall $|\overrightarrow{CP}|_{algo}$. It can be seen how the `interpolation` action, contained by the `Inter` actor, contributes by 45% to the overall $|\overrightarrow{CP}|_{algo}$. Hence, this action should be considered as the refactoring starting point. As summarized in Figure 9.5, this actor can be split in order to exploit task or data parallelism. In this case, as illustrated in Figure 9.5c, the actor has been replicated for each video component (i.e. Y, U, V). In this new design configuration, reported as *code optimization* in Table 9.8b, both the overall design complexity and the $|\overrightarrow{CP}|_{algo}$ have been reduced by 52%. However, the maximal potential parallelism has decreased to 2.23%. Consequently, a second round of this analysis has been performed in order to highlight the new most serial parts of the design. In this new design configuration, the new 5 most critical actions are summarized in Table 9.7b, where the $|\overrightarrow{CP}|_{algo}$ contributions of the `interpolation` action, contained by the `InterLuma200` actor, and the `addResAndClip` action, contained by the `SelCU` actor, are around 12.38% and 11.14%, respectively.

Table 9.7: Critical action ranking analysis of the MPEG-HEVC decoder. Results are for the initial and the full-parallel version, summarized for 5 different actions in Table 9.7a and 9.7b respectively.

(a) Critical action ranking for the initial version of the decoder.

Actor	**Action**	$w(a)_{CP}$
Inter	interpolation	45.86%
Inter	applyWeights	11.14%
DecPicBuff	expandBorders	6.68%
SaoFilter	getSaoTypeIdxDone	4.58%
DebFilter	filterEdges	4.00%

(b) Critical action ranking for the full parallel version of the decoder.

Actor	**Action**	$w(a)_{CP}$
InterLuma200	interpolation	12.38%
SelCU	addResAndClip	11.14%
SaoFilterLuma	getSaoTypeIdxDone	7.42%
DebFilter	filterEdges	7.28%
SaoFilterLuma	getSaoMerge	6.85%

Table 9.8: Description of different configurations of the HEVC decoder design and corresponding speedup, computational complexity and critical path length values.

(a) Design configurations.

	Design version	**Notes**
1	Shared-memory	Initial version
2	Code optimization	Reducing impacts of critical copies
3	InterPred Comp	Splitting luma and chroma
4	CompPipeline	The inter-prediction actors are pipelined
5	CompPipeline2x1	Splitting first part of the pipeline of the luma inter-prediction
6	Sao Comp	Splitting luma and chroma computation in the Sao filter
7	DPB Optim	Optimization of the action which expand the borders
8	Sao Split	Optimization of the Sao and parallelization of the luma part

(b) Potential speedup, computational complexity and critical path length.

	Design version	$S(n_A)$	Δw	$\Delta\lvert\overrightarrow{CP}\rvert$
1	Shared-memory	2.43	-	-
2	Code optimization	2.23	-52%	-57%
3	InterPred Comp	2.84	-55%	-47%
4	CompPipeline	3.65	-58%	-39%
5	CompPipeline2x1	3.84	-60%	-38%
6	Sao Comp	4.05	-60%	-36%
7	DPB Optim	4.18	-60%	-35%
8	Sao Split	4.46	-58%	-31%

9.3.2 Impact analysis

In this case the critical action ranking does not provide any clear direction as to what is the first action that should be refactored and what is the potential $\lvert\overrightarrow{CP}\rvert_{algo}$ reduction. Consequently, the impact analysis illustrated in Section 8.3 has been used. Results of this analysis, summarized in Figure 9.6, show that the two most critical actions highlighted in Table 9.7b are not the best refactoring candidates. In fact, by refactoring one of these two actions, the $\lvert\overrightarrow{CP}\rvert_{algo}$ can potentially be reduced by a maximum of 5%, compared to 7% by refactoring the `filterEdge` action, contained in the `DebFilter` actor, or the `geatSaoMerge` action, contained in `SaoFilterLuma` actor. According to this result, the `filterEdge` action has been refactored. Successively, other iterations have been performed in the same manner. Results are summarized in Table 9.8. At the end, the maximal potential speedup that has been obtained has been around 4.46% and the throughput improvement (i.e. in terms of $\lvert\overrightarrow{CP}\rvert_{algo}$) around 31%.

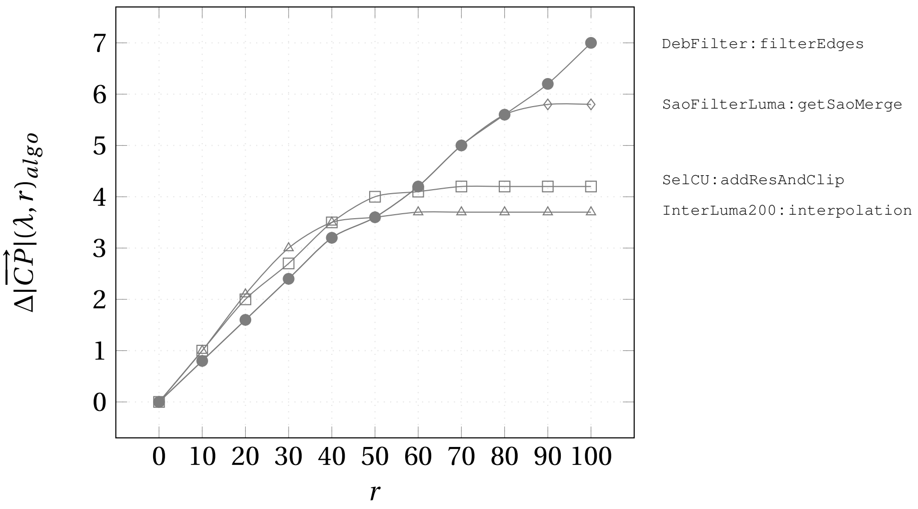


Figure 9.6: Impact analysis for the initial version of the Shared-Memory MPEG-HEVC decoder.

9.4 Bounded buffer size configuration

This section presents the experimental results on bounding and minimizing the buffer size configuration, illustrated in Section 8.4.3, for the JPEG decoder and the MEPG-HEVC decoders illustrated in Section 9.1.1 and 9.1.3, respectively. The number of actors and buffers of each specific design configuration is summarized in Table 9.9, together with the number of fired actions H_s contained in each ETG used for the analyses. Tables 9.10 and 9.11 report the results obtained using the deadlock avoidance and deadlock recovery approaches for the JPEG and HEVC decoders, respectively. The results have also been compared with what was obtained using a well-known state of the art method [195]. In the comparison $\Delta\text{bits}_{\%}$ and $\Delta\text{tokens}_{\%}$ represent the difference obtained with the new approach, respectively in terms of bits and token size savings. Moreover, for each configuration, the average time required for solving at each iteration, the problem formulated in Equation (8.36), is reported in terms of ms. The results have been obtained using a standard desktop PC with an i7-3770 3.40GHz processor and 32GB of memory. It can be observed that even with very small trace sub-graphs, such that $H_p = 1$ and a single, optimized fired step such as $H_c = 1$, the approach leads to a bounded buffer size configuration that is about 15% smaller compared to the well-established solution introduced in [195] (i.e. both in terms of tokens and bit savings). It is interesting to observe that if a buffer can contain only tokens of the same type (e.g. unsigned/signed integer, floats) where the number of bits for a single token is known (as in the case of these two examples) then the optimization objective $J(k)$ of Equation (8.36) can easily be formulated in terms of tokens by introducing a cost value for each component of the vector y.

Table 9.9: Design sizes: numbers of actors, buffers and action firings.

	Actors	**Buffers**	**Action firings**
JPEG	6	10	181739
HEVC (see Table 9.4a)	32	112	1932226

Table 9.10: Bounded buffer size configurations of the JPEG decoder using the MPC approach. Results are compared to state of the art approaches.

(a) Deadlock avoidance.

$\mathbf{H_p}$	1	2	2	4	4	4
$\mathbf{H_c}$	1	1	2	2	3	4
$\lvert S' \rvert$	6	12	12	24	24	24
bits (kB)	2.08	2.08	2.08	2.09	2.09	2.09
tokens	1961	1961	1961	1963	1963	1963
Δbits$_{\%}$	-15.8	-15.8	-15.8	-15.7	-15.7	-15.7
Δtokens$_{\%}$	-8.9	-8.9	-8.9	-8.8	-8.8	-8.8
solver (ms)	2.9	3.4	6.0	8.4	13.1	18.7

(b) Deadlock recovery.

$\mathbf{H_p}$	1	2	2
$\mathbf{H_c}$	1	1	2
$\lvert S' \rvert$	6	12	12
bits (kB)	2.07	2.07	2.07
tokens	1950	1950	1950
Δbits$_{\%}$	-16.2	-16.2	-16.2
Δtokens$_{\%}$	-9.4	-9.4	-9.4
deadlocks$_{\%}$	0.9	0.9	0.9
solver (ms)	3.0	3.5	4.7

Table 9.11: Bounded buffer size configurations of the MPEG-HEVC decoder using the MPC approach. Results are compared to state of the art approaches.

(a) Deadlock avoidance.

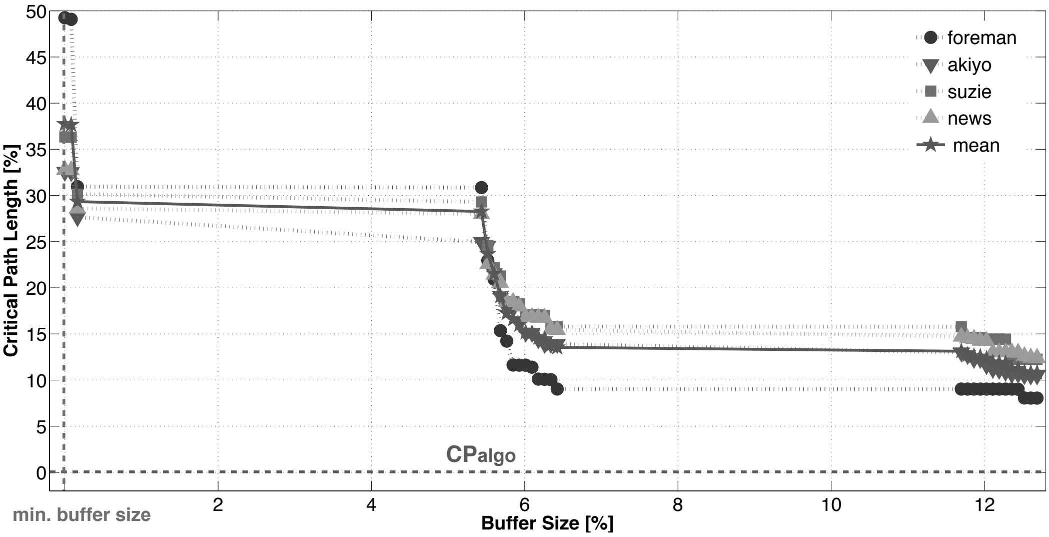
$\mathbf{H_p}$ $\mathbf{H_c}$	1 1	2 1	4 1	2 2
$\|S'\|$	32	64	128	64
bits (kB)	122.73	123.17	125.19	122.98
tokens	86411	86405	88285	86238
$\Delta\text{bits}_{\%}$	-14.1	-13.8	-12.4	-13.9
$\Delta\text{tokens}_{\%}$	-15.4	-15.4	-13.6	-15.6
solver (ms)	5.7	10.7	25.3	27.6

(b) Deadlock recovery.

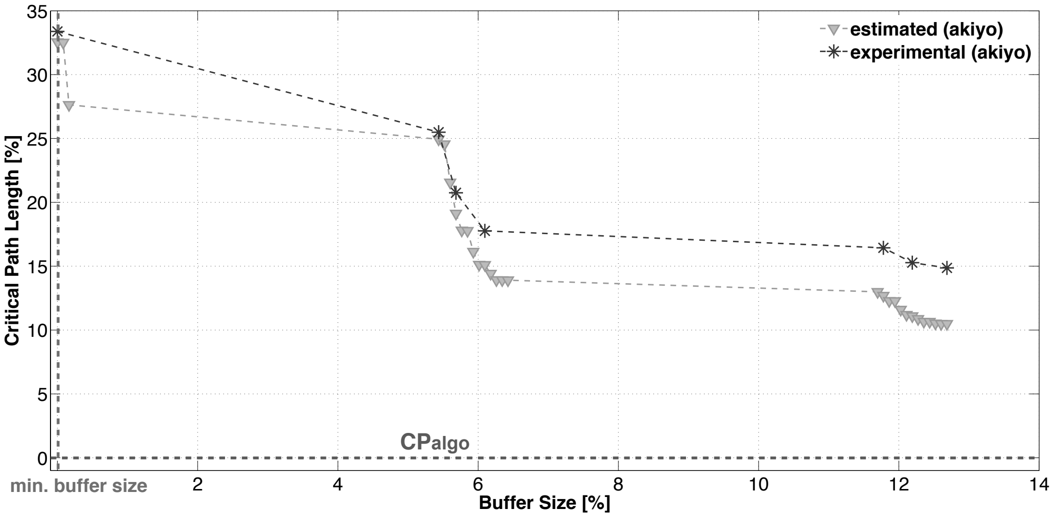
$\mathbf{H_p}$ $\mathbf{H_c}$	1 1	2 1	2 2	1 2
$\|S'\|$	32	64	64	32
bits (kB)	110.77	111.88	111.78	113.56
tokens	77663	78605	78554	80407
$\Delta\text{bits}_{\%}$	-22.5	-21.7	-21.8	-20.5
$\Delta\text{tokens}_{\%}$	-24.0	-23.0	-23.1	-21.3
$\text{deadlocks}_{\%}$	0.5	0.6	0.6	0.5
solver (ms)	8.2	12.5	29.0	15.2

9.5 Buffer size optimization

This section presents the experimental results on finding a trade-off between the buffer size configuration and the throughput performance, as illustrated in Section 8.4.4, of the MPEG4-SP decoder illustrated in Section 9.1.2. The target architecture is the ST Microelectronics STHorm platform [212]. This is an area and power-efficient, many-core platform based on multiple globally asynchronous, locally synchronous (GALS) clusters of processing elements. Clusters feature up to 16 processors and one control processor with independent instruction streams sharing a multi-banked L1 data memory (256 kB), multi-channel DMA engine, and specialized hardware for synchronization and scheduling. The fabric can be programmed in either OpenCL or standard C with the integration of a specific API, called native programming model (NPM), which is closely coupled to the platform and provides the highest level of control on application-to-resource mapping, at the expense of abstraction. For the purpose of this work, a C/NPM implementation of the RVC-CAL network synthesized using an STHorm specific extension of Orcc has been used. The results reported in the following are obtained using a software emulator running on Linux. In all the tests, the CAL actors composing the MPEG4-SP decoder were mapped into one processing element each, and a single STHorm cluster was used. Furthermore, the results presented here have been obtained with four different 10-frame QCIF bit-streams, which are commonly-used video test sequences also known as `Akyio`, `Foreman`, `Suzie` and `News`. Figure 9.7 illustrates both the estimated and the experimental results where the *Akiyo* test sequence has been used as a reference for evaluating both the minimal buffer size using the heuristic approach introduced in Section 8.4.3 and the different buffer size configuration using the heuristic algorithm described in Section 8.4.4. The other test sequences have only been used to validate this approach. Figure 9.7b depicts the estimated results (i.e. obtained post-processing the causation trace as illustrated in Section 8.1 and using the clock-accurate profiling information retrieved from the STM System Trace Module) and the experimental results (i.e. obtained from a cycle-accurate, but slower, design simulation). In the picture, the results obtained with the *Akiyo* test sequence are reported. It must be noted that the CP length with a minimal buffer size is roughly 35% higher compared to algorithmic critical path length $|\overrightarrow{CP}|_{algo}$. As it can be seen, a good trade-off can be achieved between performance improvement and resource utilization (i.e. in terms of memory utilization). In fact, increasing the buffer size from the minimal configuration by 6% leads to an overall throughput improvement of 30%. Compared to the experimental results, the performance estimation has 5% of inaccuracy in terms of absolute throughput and execution time values. This is probably related to the low precision of the STHorm scheduler model implemented in the DSE performance estimation engine used during the ETG post-mortem scheduling phase.



(a) Trade-off estimation.



(b) Experimental results and estimated values.

Figure 9.7: Buffer size optimization of the MPEG4-SP decoder implemented on an ST Microelectronics STHorm platform.

9.6 Dynamic power dissipation minimization

This section presents the experimental results on the dynamic power minimization, illustrated in Section 8.5, for the MPEG4-SP decoder illustrated in Section 9.1.2. The design under test contains 41 actors and it has been implemented on a Xilinx Virtex-5 FPGA. Performance profiling and low-level code synthesis has been performed with Xronos synthesizer. Two different MCD configurations have been tested: the first presents 2-clock domains with $F = \{50.0, 6.25\}$ MHz; the second has 4-clock domains with $F = \{50.0, 25.0, 12.50, 6.25\}$ MHz. In order to reduce the power dissipation related to memory access, the minimal buffer size configuration illustrated in Section 8.4.3 has been used. With this configuration, the results of the heuristic approach illustrated in Section 8.5.3 for the two MCD configurations are summarized respectively in Table 9.12 and Table 9.13. For each of these two configurations, the overall power consumption has been measured using the Xilinx XPE [213], enhanced with information retrieved during a post-place and route simulation using 10-frame QCIF bit-streams as input stimulus. Results of the different power contribution terms for both the 2-clock and 4-clock domain configurations are summarized in Table 9.12 and Table 9.13, respectively. Compared to the MCD configuration, where all the domains use the maximal available frequency, a significant overall power reduction can be achieved partitioning the design using the approach illustrated in this work. In fact, the overall power reduction ranges between 4% and 10% in the two cases.

Table 9.12: 2-Clock domains dynamic power minimization results of the MPEG-4 SP decoder implemented in a Xilinx Virtex-5 FPGA. Nominal: all the domains use the maximum available frequency; Optimized: with the clock frequencies illustrated in Table 9.12a. Δ% defines the percentage reduction between the nominal and optimized case of each contribution.

(a) Clock domains and partitioning.

	Clock Domain	**Actors**
f_1	50.0 MHz	21
f_2	6.25 MHz	20

(b) Experimental results.

Contribution	**Nominal** W	**Optimized** W	Δ%
Clocks	0.328	0.282	-14.0
Logic	0.069	0.06	-13.0
Signals	0.079	0.083	5.1
BRAMs	0.1	0.082	-18.0
Input/Output	0.005	0.005	0.0
Leakage	1.051	1.05	-0.1
Total	**1.632**	**1.562**	**-4.3**

Table 9.13: 4-Clock domains dynamic power minimization results of the MPEG-4 SP decoder implemented in a Xilinx Virtex-5 FPGA. Nominal: all the domains use the maximum available frequency; Optimized: with the clock frequencies illustrated in Table 9.13a. Δ% defines the percentage reduction between the nominal and optimized case of each contribution.

(a) Clock domains and partitioning.

	Clock Domain	**Actors**
f_1	50.0 MHz	12
f_2	25.0 MHz	5
f_3	12.5 MHz	8
f_4	6.25 MHz	16

(b) Experimental results.

Contribution	**Nominal** W	**Optimized** W	Δ%
Clocks	0.436	0.357	-18.1
Logic	0.07	0.041	-41.4
Signals	0.095	0.053	-44.2
BRAMs	0.106	0.075	-29.2
Input/Output	0.005	0.004	-20.0
Leakage	1.053	1.05	-0.3
Total	**1.765**	**1.58**	**-10.5**

9.7 Conclusions

In this chapter a collection of experimental results based on the analysis of video codec applications has been illustrated and discussed. The results obtained during the different stages of the design space exploration of video decoders, such as JPEG, MPEG4-SP, and HEVC decoders, have been presented and discussed. More precisely, the following cases of use have been discussed: the critical path design exploration and the code refactoring assisted by the impact analysis have been illustrated for the HEVC video decoder, implemented in a multi-core i7 desktop CPU. Successively, the bounded buffer size heuristic, based on the use of an MPC controller, has been used for both a JPEG and HEVC decoder. An optimal trade-off between buffer size dimensioning and throughput performance has been discussed for an MPEG4-SP decoder implemented on an SThorm many-core platform. Finally, the dynamic power dissipation minimization heuristic has been used for an MPEG4-SP decoder implemented on a Xilinx Virtex-5 FPGA.

10 Conclusions

This thesis addressed the problem of defining a DSE methodology for complex designs applications modeled with dynamic dataflow MoCs. Despite the increasing interest in massively and heterogeneous parallel platforms, a unified methodology for the specification and development of (complex) applications is far from being uniformly adopted. There are still too many approaches and methodologies: some give more emphasis to the re-use of legacy code and IP blocks, others require specific methodologies constrained to a given type of platform or technology. Very few approaches have as main objective the achievement of a true unified methodology capable to abstract from SW and HW. The work proposed in this thesis has tried to demonstrate how a unified HW and SW design methodology for complex designs can be successfully adopted. One of the major contributions of this work has been the formalization of DSE methodology for dynamic dataflow programs. In fact, dynamic dataflow MoCs have always been criticized with the argument that their behavior is hardly analyzable. The general approach for the implementation of dataflow programs has always been using expressive limited MoCs (e.g. static and cyclo-static) under the assumption that run-time performance can be guaranteed at compile-time. However, this approach severely limits the scalability of an application when a wide set of features is required (e.g. multimedia application). This work has shown how it is possible to efficiently explore the design space and estimate the performance of an application through the analysis of the ETG. The main advantage of this approach is that it does not limit the program expressiveness. In fact, it can be independently adopted from the dataflow MoC class (i.e. static, cyclo-static and dynamic). The effectiveness of this design methodology has been proven through the development and use of a DSE framework called TURNUS. The main research contributions of this thesis are:

(i) **Execution Trace Graph**: a graph-based representation of the program execution has been formalized and illustrated in Chapter 5. It has been shown how this mathematical formalism, called execution trace graph (ETG), can be used to model the execution of static, cyclo-static and dynamic dataflow programs as a directed acyclic graph (DAG). Nodes and edges of this DAG represent a single action firing and a (data or functional) dependency between two different action firings, respectively. Notions of partially-ordered sets (i.e. po-sets) and directed paths (i.e. d-paths) have been adapted to this execution model. Different dependency kinds have been defined, notably the finite state machine dependencies, the internal variable dependencies, the port dependencies, the tokens dependencies and the guard dependencies. Compared to similar graph-based execution models, the ETG model defines the concept of guard enable and disable dependencies. By the use of these kinds of dependencies it has been demonstrated how different execution trajectories can be modeled using the ETG. This can be obtained through a serial high-level program execution. The guard enable and disable dependencies also make it possible to model the execution of dynamic dataflow programs without requiring an MoC expressiveness reduction (i.e. to a static or a cyclo-static MoC) as done by previous approaches. Two interesting properties of the ETG are that design performance can be efficiently estimated through post-mortem scheduling (i.e. see Section 8.1) and that different analysis approaches can be used to find design configuration points that satisfy trade-off requirements between performance and resource utilization. As an example, LP methods (e.g. see 8.5.3) and advanced control technique approaches (e.g. see Section 8.4.3) can be efficiently used for every class of dataflow MoC.

(ii) **Profiling of dynamic dataflow programs**: a systematical methodology for profiling dynamic dataflow programs has been formalized and illustrated in Chapter 7. It has been shown how static and dynamic information concerning the program complexity can be retrieved during a high-level code interpretation of the program. Compared to previous approaches that mainly required a low-level code generation and binary-execution of the program, the proposed methodology is completely based on a serial and high-level code interpretation of the program. Furthermore, it has been shown how the profiling information is systematically used to evaluate the different dependency kinds of an ETG.

(iii) **Design space exploration methodology**: a unified SW and HW DSE methodology based on the post-mortem scheduling and analysis of the ETG has been formalized and illustrated in Chapter 8. A collection of heuristic methods used for efficiently exploring different design configurations of a dynamic dataflow program has been illustrated. Compared to previous approaches, this methodology makes the analysis of dynamic dataflow programs possible without limiting MoC expressiveness. The main research contributions are:

- **Performance estimation**: a unified performance estimation approach for both HW and SW, based on ETG post-mortem scheduling, has been introduced and illustrated in Section 8.1. Compared to other approaches that requires several partial low-level implementations and integrations of the program, the proposed approach makes use of a DEVS simulator. Additional dependencies and timing information are evaluated and enhanced by cycle-accurate profiling data obtained by third-party profilers. The use of the ETG makes the analysis of the application parallelism by itself and exploration of its available levels of parallelism possible. Furthermore, SW and HW platforms are modeled with a unified approach.

- **Design space critical path**: the concept of design space critical path has been formalized and illustrated in Section 8.2. This concept has been used to bound and limit the design configuration of an application that should be considered by the DSE optimization heuristics. Furthermore, the concept of potential speedup defined in terms of critical path length has been formalized using the notion of ETG critical path. Contrary to the state of the art, this definition clearly states what are the serial and parallel parts of a program by using the ETG's graph-based formalism.
- **Hotspots analysis**: the concept of hotspots of a dataflow program has been formalized and illustrated in Section 8.3. This metric provides clear source code refactoring information to the designer that can be employed to reduce the algorithmic complexity and improve the throughput of a program. Compared to previous methods, this metric takes data or functional dependencies defined by the ETG directly into account.
- **Buffer size configuration dimensioning**: a buffer size dimensioning approach for dynamic dataflow programs has been formalized and illustrated in Section 8.4. The design space of the program has been split into a deadlock and feasible region. These regions are separated by what is called minimal buffer size configuration. This thesis has proposed a methodology, based on the use of advanced control techniques, to find a close-to-minimal buffer size configuration and successively evaluate different trade-offs between throughput and memory usage. Compared to previous approaches that are suitable only for static or cyclo-static dataflow MoCs, this methodology is suitable for dynamic dataflow MoCs and it does not require any behavioral limitation of the program's MoC. It must be noted that for dynamic dataflow programs, the minimal buffer size configuration can vary according to the input sequence used. As such, a deadlock-free execution can be guaranteed only for the tested set of input sequences. In other words, a deadlock-free execution can be guaranteed only if representative input sequences are tested. This is a common practice in multimedia processing.
- **Dynamic power dissipation minimization**: a dynamic power dissipation minimization approach for dynamic dataflow programs as been formalized and illustrated in Section 8.5. It has been shown how the program can be mapped on a multi-clock domain architecture reducing its dynamic power dissipation without impacting the throughput performances. Compared to previous approaches that are suitable only for static or cyclo-static dataflow MoCs, this methodology is also suitable for dynamic dataflow MoCs.

(iv) **Design space exploration environment**: a DSE environment, called TURNUS, suitable for the analysis and optimization of CAL applications has been developed and provided as an open source project. The structure and main functionalities of this framework have been illustrated in Chapter 6. Along this thesis it has been shown how TURNUS has been integrated with other open-source CAL HW synthesis and SW code generation tools (i.e. called Xronos and Orcc, respectively). Its integration with these tools has provided a complete system design environment for CAL applications that was not available before this work.

A collection of experimental results based on the analysis of image and video codec applications has been illustrated and discussed in Chapter 9. The results obtained during the different stages of the design space exploration of video decoders, such as JPEG, MPEG4-SP, and HEVC decoders, have been presented and discussed. More precisely, the following cases of use have been discussed: critical path design exploration and code refactoring assisted by impact analysis have been illustrated for the HEVC video decoder, implemented in a multi-core i7 desktop CPU. Successively, the bounded buffer size heuristic based on use of an MPC controller has been used for both a JPEG and HEVC decoder. An optimal trade-off between buffer size dimensioning and throughput performance has been discussed for an MPEG4-SP decoder implemented on an SThorm many-core platform. Finally, the dynamic power dissipation minimization heuristic has been used for an MPEG4-SP decoder implemented on a Xilinx Virtex-5 FPGA. It must be noted how these results have been obtained using the same CAL programs implemented on a wide variety of parallel architectures. Although the unified SW and HW methodologies illustrated in this thesis has been validated, several challenging issues are still unsolved. These open problems are discussed in the following section.

10.1 Future work

Since we used the term orthogonalization of concerns, in this section the term orthogonalization of effort is forged. By orthogonalization of effort it is assumed that both theoretical and implementation works should be concentrated in order to improve the DSE methodology and the supporting framework.

Open theory problems

The following problems, as far as what concerns the theory behind the DSE methodology, need more investigation:

- **Hardware and algorithmic critical path**: the relation between the program algorithmic critical path and the hardware critical path should be investigated. The objective is to verify if it is possible to provide further information to the designer in order to scale the design over higher frequency.
- **Many-core partitioning**: the problem of many-core partitioning of dynamic dataflow programs is a must-do problem. This should also be provided within the DSE framework in order to provide a fully-functional co-design environment. Initial, but still unpublished, work has been concentrated on finding effective partitioning heuristics based on the analysis of the design space critical path. However, further investigation is required to validate and prove the effectiveness of this approach.

- **Pipelining analysis**: initial, but still unpublished, work is the identification through the ETG analysis of actions where execution can be pipelined. If these actions are along the algorithmic critical path, pipelining their execution would directly provide improvement of the design performance without requiring any program modifications.
- **Scheduling optimization**: another must-do problem is the analysis and optimization of the scheduling of dynamic dataflow programs. Solving this problem is essential in order to fully explore the design space defined as the configuration points of partitioning, scheduling and buffer size configurations.
- **Performance estimation**: the performance estimation methodology based on the post-mortem processing of the ETG should be verified and validated on a wider set of heterogeneous parallel platforms. An example of a not-yet tested platform is the emerging many-core Parallella platform [214].

Open implementation problems

The following problems, as far as what concerns the improvement of the TURNUS DSE framework, need more investigation:

- **Complex guard conditions**: guard enable and disable dependencies are the key roles to model different execution trajectories with a single ETG. Guard conditions, where more than one internal actor variable is involved, require the use of a satisfiability modulo theories (SMT) problem solver [215]. Currently, the detection of enabled and disabled guard conditions is required to be made by the code interpreter. As future work, a new functionality that should be integrated within the DSE framework is the possibility to model such complex guard conditions and directly detect when a guard has been enabled or disabled. In other words, enabling and disabling guard conditions should be directly identified when post-processing the ETG by analyzing the internal variable modifications performed by the firings.
- **Big data**: the size of an ETG rapidly grows as the number of action firings increase. For complex designs and where big input sequences are used as program stimulus, the corresponding ETG can contain millions, even billions of nodes and dependencies. This could become a problem if effective methods for handling such big data are not used. An initial experimental approach that has been tested within the DSE framework is the use of a graph database integrated in a Blueprints graph interface ecosystem [216, 217, 218]. However, further investigation is required to consolidate and improve the performance of this approach

A Discrete event system and simulation

A.1 Petri nets

A Petri net [156, 157] (PN) is a bipartite directed graph with two kinds of nodes, called *transitions* and *places*, where arcs are either from a place to a transition or from a transition to a place. Using the concept of *conditions and events*, places represent conditions, and transitions represent events. A transition (an event) has a certain number of input and output places representing the *pre-conditions* and *post-conditions* of the event, respectively. Contrary to KPN and DPN, where tokens are atomic data objects, in a PN, tokens are used to simulate the dynamic and the concurrent activity of the system. The presence of a token in a place is interpreted as holding the truth of the condition associated with the place. In another interpretation, n tokens are put in a place to indicate that n data items or resources are available.

A PN is formally defined as a tuple $\{P, T, E, W, M_0\}$, where:

- $P = \{p_1, \ldots, p_m\}$ is a finite set of places.
- $T = \{t_1, \ldots, t_n\}$ is a finite set of transitions.
- $E \subseteq (P \times T) \cup (T \times P)$ is a finite set of directed arcs connecting transitions to places and places to transitions.
- $W : E \rightarrow \mathbb{N}$ is a weight function which defines the weight assignment for each arc.
- $M_0 : P \rightarrow \mathbb{N}_0$ is the initial marking.
- $P \cap T = \emptyset$ and $P \cup T = \emptyset$.

According to the weight function W, arcs are labeled with positive integer numbers $W(p_i, t_j)$ and $W(t_i, p_j)$, representing respectively the weight of an arc from a place to a transition and the weight from a transition to a place. The two sets ${}^\bullet t = \{p \in P : (p, t) \in E\}$ and $t^\bullet = \{p \in P : (t, p) \in E\}$ define respectively the *pre-set* and *post-set* of a transition t.

Remark. *PN transitions are like KPN processes or DPN actors: they fire when sufficient input is available. However, tokens have no value and firing of a transition does not involve any computation on tokens. For a PN, a firing is just the act of moving tokens from one place to another. Moreover, contrary to KNP and DPN buffers, places do not preserve the token ordering.*

A.1.1 State

The *state* of a PN is described by a *marking* function $M : P \to \mathbb{N}_0$, which assigns a non-negative integer representing the number of tokens residing in that place to each place. Typically, the marking function M is described as a column vector $M = [M(p_1), \dots, M(p_m)]' \in \mathbb{N}_0^m$ whose generic entry $M(p_i), i = 1, \dots, m$ represents the number of tokens present in place p_i, and the symbol $[\cdot]'$ is the matrix transpose operator.

A.1.2 Transition firing

Each *firing* (or occurrence) of a transition produces an update of the net marking vector M. In order to be able to occur, a transition has to be *enabled*. A transition t is said to be enabled if it satisfies the following firing rule:

$$\forall p \in {}^{\bullet}t : M(p) \geq W(p,t) \tag{A.1}$$

Roughly speaking, the occurrence of a transition removes tokens from the pre-set of a transition and adds tokens to its post-set, according to the weights of the arcs connecting the places to the transition. The firing of a transition t in M results in a new marking $\tilde{M}$ defined as:

$$\tilde{M}(p) : p \mapsto \begin{cases} M(p) + W(t,p) - W(p,t) & \text{for } p \in {}^{\bullet}t \cap t^{\bullet} \\ M(p) + W(t,p) & \text{for } p \in t^{\bullet} \setminus {}^{\bullet}t \\ M(p) - W(p,t) & \text{for } p \in {}^{\bullet}t \setminus t^{\bullet} \\ M(p) & \text{otherwise} \end{cases} \tag{A.2}$$

The marking transition from M to $\tilde{M}$ can be concisely represented using the notation $M[t\rangle\tilde{M}$. Note that an enabled condition of Equation (A.1) guarantees that the resulting marking can never assign a negative number to a place. By this, the *run* of a PN can be defined as the marking sequence $M_0^k = \{M_i, i = 0, \dots, k\}$ obtained by firing the enabled transition sequence $t_1^k = \{t_i, i = 1, \dots, k, t_i \in T\}$, such that $M_{i-1}[t_i\rangle M_i$ for $i \in \{1, \dots, k\}$. It must be noted that the run of a PN is *non-deterministic*: when multiple transitions are enabled at the same time, any one of them may fire. A run of a PN can be effectively computed by means of elementary matrix operations (i.e. multiplications and additions) using the pre-incidence matrix I and the post-incidence matrix O, respectively defined as:

$$
\begin{aligned}
I = [q_{i,j}]_{\substack{i=1,\dots,m,\\ j=1,\dots,n}} \; q_{i,j} &= \begin{cases} W(p_i, t_j) & \text{for } (p_i, t_j) \in E \\ 0 & \text{otherwise} \end{cases} \\
O = [r_{i,j}]_{\substack{i=1,\dots,m,\\ j=1,\dots,n}} \; r_{i,j} &= \begin{cases} W(t_j, p_i) & \text{for } (t_j, p_i) \in E \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{A.3}
$$

In this way, the marking $M_k(p)$ obtained by firing transition t at event k, can be expressed as:

$$M_k(p) = M_{k-1}(p) - I(p,t) + O(t,p), \; \forall p \in P \tag{A.4}$$

A.2 Discrete event system specification

Discrete event system specification (DEVS) [180, 181] is a conceptual framework for specifying modular, hierarchical and timed event systems. Its formalism makes it possible to model and analyze general systems that can be discrete event systems (i.e. described by state transition tables), continuous state systems (i.e.described by differential equations), and hybrid continuous state and discrete event systems.

Discrete event systems are a generalization of discrete time systems that allow time to be continuous. The trajectories of a discrete-event system are functions from the time base $\mathbb{R} \times \mathbb{N}$ to its sets of input, output, and state. These trajectories change value only a finite number of times in any finite interval. This is the defining characteristic of a discrete event system: the events that cause these discrete changes give the class of systems its name.

A.2.1 The atomic model

An Atomic DEVS model is defined as a tuple $M = (X, Y, S, t_a, \delta_{ext}, \delta_{int}, \lambda)$ where

- X is the set of input events.
- Y is the set of output events.
- S is the set of sequential states (or also called the set of partial states).
- $t_a : S \rightarrow \mathbb{T}^{\infty}$ is the time advance function which is used to determine the lifespan of a state.
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function which defines how an input event changes a state of the system.
- $\delta_{int} : S \rightarrow S$ is the internal transition function which defines how a state of the system changes internally (i.e. when the elapsed time reaches the lifetime of the state).

- $\lambda : S \rightarrow Y^{\phi}$ is the output function where $Y^{\phi} = Y \cup \phi$ and $\phi \notin Y$ is a silent event or an unobserved event. This function defines how a state of the system generates an output event (when the elapsed time reaches the lifetime of the state).

where $Q = \{(s, t_e) : s \in S, t_e \in (\mathbb{T} \cap [0, ta(s)])\}$ is the set of total states, t_e is the elapsed time since the last event, $\mathbb{T}^{\infty} = [0, \infty]$ defines the extended time base that is the set of the non-negative real number plus infinity [180].

B Model predictive control

The key aspects of the Model Predictive Control (MPC) are presented in the following. However, for a more complete presentation, the interested reader can refer to [198, 199] where different domains of application are also illustrated. MPC, also known as receding horizon control, is a model-based form of control in which the control action is obtained by solving, at each sampling time, a finite-horizon open loop optimal control problem. Using the current state of the plant as the initial state of the problem, the optimization solution yields an optimal control sequence. The control loop is then closed by using the first control move obtained from the optimized sequence. This is the main difference from conventional control strategies (e.g. Proportional Derivative Integrator, Linear-Quadratic Regulator) which use a pre-computed control law.

One of the main features of MPC is the possibility to take hard system constraints directly into account in the optimization problem. The system evolution is predicted over H_p (i.e. prediction horizon) events k. During event k, using the standard notation of discrete event systems (e.g. see Equation (5.16)) the actual output is represented as $y(k)$, the predicted output and optimized output for the event $k+i$ are represented respectively as $y(k+i|k)$ and $u(k+i|k)$. At each event k, the MPC strategy calculates a set of $H_c \leq H_p$ (Control Horizon) values of the input $U(k)_o^{H_c} = \{u(k+i|k), \forall i \in \{0,1,\ldots,H_c-1\}\}$. The input is evaluated so that that the predicted outputs $\widehat{Y}(k)_o^{H_p} = \{y(k+i|k), \forall i \in \{1,2,\ldots,H_p\}\}$ reach the target point in an optimal manner. $U(k)_o^{H_c}$ is obtained by optimizing a linear or quadratic constrained objective function such as:

$$J(k) = f(x(k|k), u(k|k), u(k+1|k), \ldots, u(k+H_c-1|k)) \tag{B.1}$$

In other words, at each event k the objective is to minimize an objective function subject to additional constraints as:

$$
\begin{aligned}
\underset{u(k|k),u(k+1|k),\ldots,u(k+H_c-1|k)}{\text{minimize}} \quad & J(k) \\
\text{subject to} \quad & y_{min} \leq y(k+i|k) \leq y_{max},\ \forall i \in \{1,2\ldots H_p\} \\
& u_{min} \leq u(k+i|k) \leq u_{max}, \forall i \in \{0,2\ldots H_c-1\} \\
& u(k+i|k) = 0, \forall i \in \{H_c, H_c+1\ldots H_p\} \\
& g(u(k|k), u(k+1|k), \ldots, u(k+H_c|k)) \leq 0
\end{aligned}
\tag{B.2}
$$

It must be noted that during the prediction, the control is held constant after H_c control moves (i.e. $u(k+i|k)=0$). As mentioned before, a remarkable feature of MPC is its receding horizon approach: after evaluating the optimal input set $U(k)_o^{H_c}$ only the first move $u(k)^* = u(k|k)$ is actually implemented. Then, a new sequence is calculated at the next event and only the first input move is implemented again.

C A CAL esoteric example

This appendix is a fun dissemination example that can be used to explain to people without any notion of dataflow programming (and parallel programing, in general) what a dataflow program is. The example that follows does not want to be strictly scientifically correct. An example based on a chocolate cake recipe is presented. This is a tribute to my parents, Patrizia and Andrea, who are both hotel-keepers and usually prepare homemade cakes for their guests' breakfast [219]. Unfortunately for the reader, the recipe presented in the following is not the original chocolate cake that my parents prepare. This is because the original recipe can potentially be used by their competitors when they read this thesis.

C.1 A Chef chocolate cake

The `Hello World Cake with Chocolate sauce` [220] is an open source recipe. This recipe, reported in Listing C.1, is written using the Chef esoteric programming language [221]. Within the Chef formalism, a program looks like a recipe. A Chef program is composed by ingredients, mixing bowls and baking dishes. According to the definition presented in [221], the ingredients hold individual data values and a program has access to an unlimited number of mixing bowls and baking dishes. These contain ingredient data values. The ingredients in a mixing bowl or baking dish are ordered, like a stack of pancakes. New ingredients are placed on top, and if values are removed then they are removed from the top. If the value of an ingredient changes, its old value in the mixing bowl or baking dish does not change. The values in the mixing bowls and baking dishes also retain their dry or liquid designations. Considering the Chef recipe reported in Listing C.1 it can be seen in this case how the program is composed of two methods: the first (i.e. see line 18) describes how the cake is baked; the second (i.e. see line 47) describes how the chocolate sauce used to serve the cake is made. Inside each method how ingredients, mixing bowls and baking dishes are used is defined. For each method, a set of input ingredients is defined (i.e. see lines 3 and 40, respectively).

Listing C.1: Cake.chef

```
Hello World Cake with Chocolate sauce.

Ingredients.
33 g chocolate chips
100 g butter
54 ml double cream
2 pinches baking powder
114 g sugar
111 ml beaten eggs
119 g flour
32 g cocoa powder
0 g cake mixture

Cooking time: 25 minutes.

Pre-heat oven to 180 degrees Celsius.

Method.
Clean the mixing bowl.
Put chocolate chips into the mixing bowl.
Put butter into the mixing bowl.
Put sugar into the mixing bowl.
Put beaten eggs into the mixing bowl.
Put flour into the mixing bowl.
Put baking powder into the mixing bowl.
Put cocoa  powder into the mixing bowl.
Stir the mixing bowl for 1 minute.
Combine double cream into the mixing bowl.
Stir the mixing bowl for 4 minutes.
Liquify the contents of the mixing bowl.
Pour contents of the mixing bowl into the baking dish.
Bake the cake mixture.
Wait until baked.
Serve with chocolate sauce.



chocolate sauce.

Ingredients.
111 g sugar
108 ml hot water
108 ml heated double cream
101 g dark chocolate
72 g milk chocolate

Method.
Clean the mixing bowl.
Put sugar into the mixing bowl.
Put hot water into the mixing bowl.
Put heated double cream into the mixing bowl.
Dissolve the sugar.
Agitate the sugar until dissolved.
Liquify the dark chocolate.
Put dark chocolate into the mixing bowl.
Liquify the milk chocolate.
Put milk chocolate into the mixing bowl.
Liquify contents of the mixing bowl.
Pour contents of the mixing bowl into the baking dish.
Refrigerate for 1 hour.
```

C.2 From a sequential to a dataflow program specification

A Chef program can be seen as a sequential collection of operations made of ingredients, with mixing bowls and baking dishes used as containers. Now suppose that the cake must be prepared before a given time and in the least time possible. For example, my parents want to make the breakfast cake in no more than one hour. So, my parents can decide to cooperate together in the baking process. However, which part of the cake should be prepared by my mother and which part by my father? Furthermore, which part of the cake should be prepared before the other parts? The problem here is how to effectively find which parts of the cake can be made at the same time and which parts need to be made before the other parts. Considering the `Hello World Cake with Chocolate sauce` illustrated before, it can be very hard to identify which parts of the recipe can be prepared at the same time. For example, the fact that the chocolate sauce can be prepared at the same time with other parts of the cake and that this sauce is used for dressing the cake is not implicitly defined by the recipe. This problem can be solved by using a dataflow formalism for defining the cake recipe. Using a dataflow approach, reading the cake recipe becomes much more *understandable*. A basic dataflow representation of this recipe is depicted in Figure C.1. This is what is called a dataflow network where boxes are actors (i.e. computational kernels) interconnected by buffers that handle unbounded sequences of tokens (i.e. atomic data objects). As a consequence, each actor contains a single Chef method and each buffer models how ingredients flow from different mixing bowls or baking dishes. Each dataflow token represents a single ingredient unit (e.g. 1g of sugar). Using this formalism it is immediately clear which are the single parts of the recipe, how they are related and which part should be prepared before others.

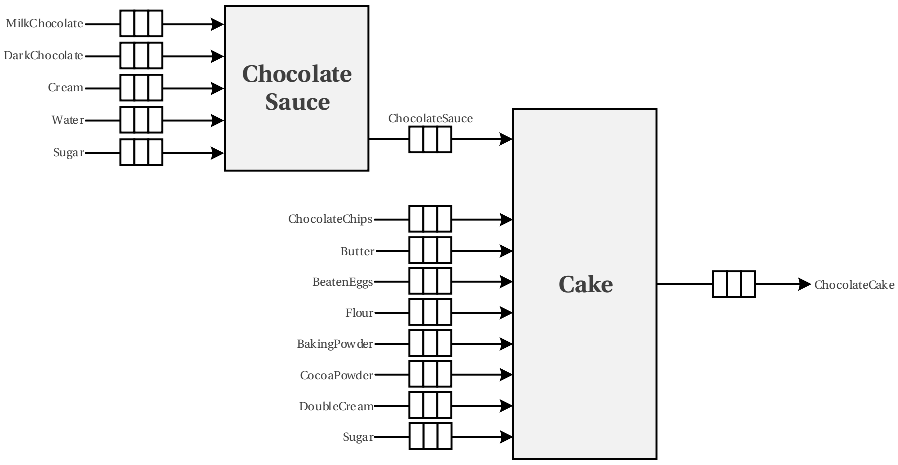


Figure C.1: A dataflow representation of the `Hello World Cake with Chocolate sauce` Chef program illustrated in Listing C.1.

C.3 The first CAL chocolate cake

In the previous section it has been shown how the recipe can be modeled as a dataflow program. However, it has not been described how the syntax and the semantic (i.e. how the program is written and what it describes) of Chef program can be translated to a dataflow program. This section provides a possible translation of a Chef recipe to an "equivalent" CAL dataflow program. Basically, each method is translated as an actor and each unitary quantity of an ingredient as a token. As described before, Figure C.1 illustrates the CAL dataflow representation of the Chef recipe reported in Listing C.1. This CAL program is composed by two actors: the `Cake` actor and the `ChocolateSauce` actor. The CAL code of these two actors is reported in Listing C.2 and C.3, respectively. For each actor the input tokens represent the required ingredients, while the output tokens represent the amalgamation result of the prepared (input) ingredients. The `Cake` actor is composed of seven actions, an actor internal state machine (FSM), and two action priority conditions are defined. Inputs and outputs of this actor are defined in Lines 3 and 4, respectively. Similarly, the `ChocolateSauce` actor is composed of six actions and an actor internal FSM. Inputs and output of this actor are defined in Lines 2 and 3, respectively. In both actors the respective Chef mixing bowl is modeled as an internal variable: the `cakeMixture` in the `Cake` actor, and the `chocolateSauce` in the `ChocolateSauce` actor. Activities like *liquify*, *stir*, *agitate* and *refrigerate* are modeled as CAL functions that can aggregate, modify or define the status of a single or a collection of ingredients.

Exploiting dataflow properties

The CAL program illustrated before can be used to show the powerfulness of a dataflow approach when dealing with parallel programs. As an example, lets consider the two actions `liquifyDarkChocolate` and `liquifyMilkChocolate` defined in Lines 25 and 35, respectively, of the `ChocolateSauce` actor. It is easy to see how liquefying (i.e. melting) both the dark and milk chocolate can be performed at the same time if there are at least two chefs (e.g. my parents). The dataflow approach allows to easily and explicitly model this condition as illustrated in Figure C.2: the Chef *liquify* activity can be modeled as a single CAL actor where the input is the solid ingredient and the output is the liquefied ingredient as illustrated in Listing C.4. Similarly, this approach can also be done for the other Chef activities like *stir*, *agitate* and *refrigerate*. In computer science, the fact that these activities can be performed at the same time by different chefs, is called **parallelism** (or more precisely task parallelism). Furthermore, it must be noted that this actor can be used for both milk and dark chocolate: in computer science this is called code-**reusability**. In order to exploit these properties the `ChocolateSauce` actor should be modified as illustrated in Listing C.5. Consequently, the new dataflow network representation is the one depicted in Figure C.3. In computer science, the property that an actor can be represented as a network of actors is called **modularity**. Furthermore, inside the actors network depicted in Figure C.3, the *refrigerate* Chef activity has been modeled with the `Refrigerate` CAL actor defined in Listing C.6. This

Listing C.2: Cake.cal

```
actor Cake()
    ChocolateChips Cc, Butter B, DoubleCream Dc, BakingPowder Bp, Sugar S, BeatenEggs Be,
    Flour F, CocoaPowder Cp, Cs ChocolateSauce
    ==>  CakeMixture Cm :

    CakeMixture cakeMixture;

    int stirMinutes;
    int stirstirMaxMinutesutes;

    clean: action  ==>
    do
        // initialize mixing bowl
        cakeMixture := 0;
        // set the stir timer
        stirMinutes := 0;
        stirMaxMinutes := 1;
    end

    add: action B:[butter] repeat 100,
                S:[sugar] repeat 114,
               Cc:[chocoChips] repeat 33,
               Be:[btnEggs] repeat 111,
                F:[flour] repeat 119,
               Bp:[bakingPwd] repeat 2,
               Cp:[cocoaPwd] repeat 32 ==>
    do
        cakeMixture := butter+ sugar + chocoChips + btnEggs + flour + bakingPwd + cocoaPwd;
    end

    stir1m: action ==>
    guard stirMinutes< stirMaxMinutes
    do
        stir1minute(cakeMixture);
        stirMinutes := stirMinutes+ 1;
    end

    combineCream: action Dc:[doubleCream] repeat 54 ==>
    do
        cakeMixture := cakeMixture + doubleCream;
        // set the stir timer
        stirMinutes := 0;
        stirMaxMinutes := 4;
    end

    liquify: action ==>
    do
        while(!isLiquified(cakeMixture)) do
            liquify(cakeMixture);
        end
        cakeMixture := mixingBowl;
        while(!isBaked(cakeMixture)) do
            bake(cakeMixture);
        end
    end

    bake: action ==>
    do
        while(!isBaked(cakeMixture)) do
            bake(cakeMixture);
        end
    end

    serve: action Cs:[sauce] ==>  Cm:[cakeMixture]
    do
        cakeMixture := cakeMixture + sauce;
    end

    schedule fsm s0 :
        s0(clean) --> s1;
        s1(add) --> s2;
        s2(stir1m) --> s2;
        s2(combineCream) --> s3;
        s3(stir1m) --> s3;
        s3(liquify) --> s4;
        s4(bake) --> s5;
        s5(serve) --> s0;
    end

    priority
        stir1m > combineCream;
        stir1m > liquify;
    end

end
```

Listing C.3: ChocolateSauce.cal

```
actor ChocolateSauce()
  Sugar S, HotWater Hw, HeatedDoubleCream Hdc, DarkChocolate Dc, MilkChocolate Mc
  ==>  ChocolateSauce Cs :

    ChocolateSauce chocolateSauce;

    clean: action ==>
    do
        chocolateSauce := 0;
    end

    add: action Hw:[water] repeat 108, Hdc:[cream] repeat 108  ==>
    do
        chocolateSauce :=  water + cream;
    end

    dissolveSugar: action S:[sugar] repeat 111 ==>
    do
        chocolateSauce := chocolateSauce + sugar;
        while(!isDissolved(chocolateSauce)) do
            agitate(chocolateSauce);
        end
    end

    liquifyDarkChocolate: action Dc:[darkChocolate] repeat 101 ==>
    var
        MeltedDarkChocolate mdc := 0
    do
        while(!isLiquified(darkChocolate)) do
            mdc := mdc + liquify(darkChocolate);
        end
        chocolateSauce := chocolateSauce + mdc;
    end

    liquifyMilkChocolate: action Mc:[milkChocolate] repeat 72 ==>
    var
        MeltedMilkChocolate mmc := 0
    do
        while(!isLiquified(milkChocolate)) do
            mmc := mmc + liquify(milkChocolate);
        end
        chocolateSauce := chocolateSauce + mmc;
    end

    refrigerate: action ==> Cs:[chocolateSauce]
    do
        foreach int t in 0 .. 60 do
            refrigerate1minute(chocolateSauce);
        end
    end

    schedule fsm s0 :
        s0(clean) --> s1;
        s1(add) --> s2;
        s2(dissolveSugar) --> s3;
        s3(liquifyDarkChocolate) --> s4;
        s4(liquifyMilkChocolate) --> s5;
        s5(refrigerate) --> s0;
    end

end
```

actor models a refrigerator where the number of minutes required for refrigerating a product is specified as a parameter. This parameter is specified along the recipe: for the chocolate cake example this value is 4 minutes. In computer science, specifying parameters in such a way is referred to as **compile-time** configuration of the program.

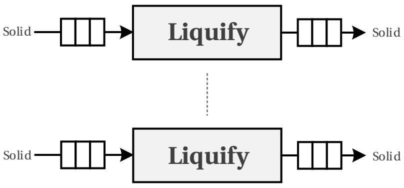


Figure C.2: The `Liquify` CAL actor defined in Listing C.4.

Listing C.4: Liquify.cal

```
actor Liquify(type Solid, type Liquid)  Solid S  ==> Liquid L :

    liquify: action  S:[solid] ==> L:[liquid]
    var
        Liquid liquid := 0
    do
        while(!isLiquified(solid)) do
            liquid := liquid + liquify(solid);
        end
    end

end
```

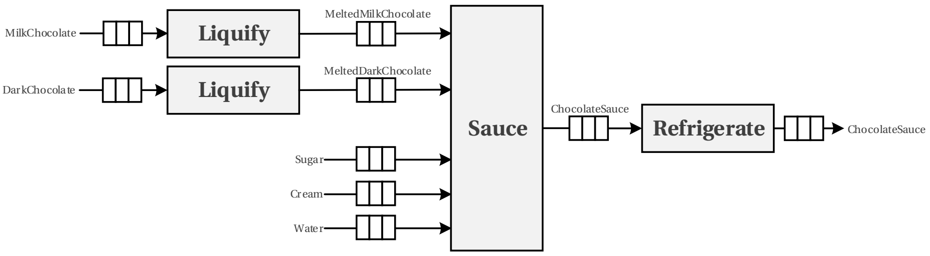


Figure C.3: The modified version of the `ChocolateSauce` CAL actor.

C.4 A dynamic refrigerator

Since the topic of this dissertation is the analysis of dataflow programs, this section illustrates a very basic example of a dataflow actor. Let's consider the `Refrigerate` actor defined in Listing C.6. The number of minutes required for refrigerating the input program is defined as a parameter. In other words, its value is specified before starting to prepare the recipe and cannot be changed (i.e. compile-time configuration as previously discussed). However, it is

Listing C.5: ModifiedChocolateSauce.cal

```
actor ModifiedChocolateSauce()
  Sugar S, HotWater Hw, HeatedDoubleCream Hdc, MeltedDarkChocolate Dc, MeltedMilkChocolate Mc
  ==>  ChocolateSauce Cs :

    ChocolateSauce chocolateSauce;

    clean: action  ==>
    do
        chocolateSauce := 0;
    end

    add: action S:[sugar] repeat 111,
               Hw:[water] repeat 108,
              Hdc:[cream] repeat 108,
              DCc:[darkChocolate] repeat 101,
               Mc:[milkChocolate] repeat 72   ==>
    do
        chocolateSauce := sugar + water + cream + milkChocolate + darkChocolate;
    end

    dissolve: action ==> Cs[chocolateSauce]
    do
        while(!isDissolved(chocolateSauce)) do
            agitate(chocolateSauce);
        end
    end

    schedule fsm s0 :
        s0(clean) --> s1;
        s1(add) --> s2;
        s2(dissolve) --> s0;
    end

end
```

Listing C.6: Refrigerate.cal

```
actor Refrigerate(type Product, int minutes) Product H ==>  Product C :

    refrigerate: action  H:[product] ==> C:[product]
    do
        foreach int t in 0 .. minutes do
            refrigerate1minute(product);
        end
    end

end
```

possible that chef can decide to increase the number of minutes required for refrigerating a product. This functionality of a refrigerator can be modeled as illustrated in the modified `Refrigerate` CAL actor described in Listing C.7. The number of minutes that the product should stay in the refrigerator is specified as an input token of the actor and can be modified while making the cake (i.e. program execution). In other words, (part of) the recipe can be prepared according to some chef's choices that are not predictable. In this example the number of minutes can vary according to the refrigerating status of the cake. In computer science, the execution of an actor that varies according to some input stimulus (i.e. the chef's choices) is referred to as **dynamism**.

Listing C.7: Refrigerate.cal

```
actor DynamicRefrigerator(type Product) Product H, int T ==>  Product C, int R :

    Product product;
    int remainingTime := 0;

    setTimer: T:[time] ==> R:[remainingTime]
    do
        remainingTime := remainingTime + time;
    end

    place: action H:[product] ==>
    do
        place := product;
    end

    refrigerate: action ==> R:[remainingTime]
    guard
        remainingTime > 0
    do
        refrigerate1minute(product);
        remainingTime := remainingTime - 1;
    end

    ready: action ==> P:[product], R:[remainingTime] end

    schedule fsm s0 :
        s0(place) --> s1;
        s1(refrigerate) --> s1;
        s1(ready) --> s0;
    end

    priority
        refrigerate > ready;
        setTimer > place;
        setTimer > refrigerate;
        setTimer > ready;
    end

end
```

C.5 Design space exploration of a kitchen

The design space exploration (DSE) of a dataflow program is one of the topic of this dissertation. Ok, but what is the DSE of a dataflow program? In order to easily explain what the DSE is, a similarity of a dataflow program implementation and the recipe is made in the following section. First of all, the recipe corresponds to a program. As illustrated in the previous sections, a dataflow program can be see as a cake recipe. Similarly, the kitchen corresponds to the target platform, where each processing unit can be seen as a chef. Thence, a massively parallel platform can be considered as a collection of *chefs* and *commis chefs*. So chefs are like processing units that can execute all kinds of complex operations, and commis chefs are like processing units that can execute a limited set of operations. Furthermore, the difference between a chef and a commis chef is how much they are payed per hour. So it is convenient to assign simple tasks of a recipe to a commis chef and complex task to a chef. In this way, constraints of an application implementation can be defined as the maximum time for cooking a cake and the maximum amount of money that should used to pay chefs and commis. Consequently, the design space of an application is the collection of design alternatives (i.e. mapping configuration) that define which parts of a recipe should be assigned to a chef or a commis chef (i.e. partitioning), the size of each mixing bowl and baking dish that should be used (i.e. buffer size configuration) and the operation order that each chef or commis chef should follow (i.e. scheduling). In this way, the DSE of a program can be seen as the analysis of a recipe and the results as the collection of rules for each available chef and commis chef in order to bake a cake with the given constraints. With these definitions, analysis and heuristics that have been presented in this dissertation can easily be adapted to the cooking domain. For example, the throughput of a system defined in terms of bit per second can be defined as cakes per hour, energy minimization can be see as money minimization.

Remark. *As an interesting similar example about the critical path analysis, that has been illustrated in this dissertation, and the cooking process of a recipe can be found in one of the Numb3rs TV series episode [222].*

Bibliography

[1] 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.

[2] S. Casale-Brunet, M. Mattavelli, and J. Janneck, “Profiling of Dataflow Programs Using Post Mortem Causation Traces,” in *Signal Processing Systems (SiPS), 2012 IEEE Workshop on*, pp. 220–225, Oct. 2012.

[3] S. Casale-Brunet, M. Mattavelli, C. Alberti, and J. Janneck, “Systems design space exploration by serial dataflow program execution,” in *Signals, Systems and Computers, 2013 Asilomar Conference on*, pp. 1805–1809, Nov. 2013.

[4] M. Casale-Brunet, S.and Mattavelli, C. Alberti, and J. Janneck, “Representing Guard Dependencies in Dataflow Execution Traces,” in *Computational Intelligence, Communication Systems and Networks (CICSyN), 2013 Fifth International Conference on*, pp. 291–295, 2013.

[5] S. Casale-Brunet, M. Mattavelli, C. Alberti, and J. Janneck, “Design Space Exploration of High-Level Stream Programs on Parallel Architectures,” *Conference: 8th International Symposium on Image and Signal Processing and Analysis (ISPA 2013), Trieste, Italy*, pp. 738–743, Sep. 2013.

[6] A. Ab-Rahman, R. Thavot, S. Casale-Brunet, E. Bezati, and M. Mattavelli, “Design space exploration strategies for FPGA implementation of signal processing systems using CAL dataflow program,” in *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pp. 1–8, Oct. 2012.

[7] A. Ab-Rahman, S. Casale-Brunet, C. Alberti, and M. Mattavelli, “Dataflow program analysis and refactoring techniques for design space exploration: MPEG-4 AVC/H.264 decoder implementation case study,” in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pp. 63–70, Oct. 2013.

[8] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. Janneck, “Synthesis and optimization of high-level stream programs,” in *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pp. 1–6, May 2013.

[9] D. de Saint-Jorre, C. Alberti, M. Mattavelli, and S. Casale-Brunet, "Exploring MPEG HEVC decoder parallelism for the efficient porting onto many-core platforms," in *Image Processing (ICIP), 2014 IEEE International Conference on*, pp. 2115–2119, Oct. 2014.

[10] S. Casale-Brunet, M. Mattavelli, and J. 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, May 2013.

[11] 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.

[12] 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.

[13] S. Casale-Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. Janneck, "Multi-clock domain optimization for reconfigurable architectures in high-level dataflow applications," in *Signals, Systems and Computers, 2013 Asilomar Conference on*, pp. 1796–1800, Nov. 2013.

[14] S. Casale-Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. Janneck, "Partitioning and optimization of high level stream applications for multi clock domain architectures," in *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pp. 177–182, Oct. 2013.

[15] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. Janneck, "Coarse grain clock gating of streaming applications in programmable logic implementations," in *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*, pp. 1–6, 2014.

[16] "TURNUS." http://github.com/turnus. Accessed: May 2015.

[17] S. Casale-Brunet, M. Mattavelli, and J. Janneck, "TURNUS: A design exploration framework for dataflow system design," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pp. 654–654, May 2013.

[18] S. Casale-Brunet, E. Bezati, C. Alberti, G. Roquier, M. Mattavelli, J. Janneck, and J. Boutellier, "Design space exploration and implementation of RVC-CAL applications using the TURNUS framework," in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pp. 341–342, Oct. 2013.

[19] S. Casale-Brunet, C. Alberti, M. Mattavelli, and J. Janneck, "TURNUS: A unified dataflow design space exploration framework for heterogeneous parallel systems," in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pp. 47–54, Oct. 2013.

[20] 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)*, 2014.

[21] E. Lee and A. Sangiovanni-Vincentelli, "Comparing models of computation," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pp. 234–241, IEEE Computer Society, 1997.

[22] 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.

[23] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing* (J. L. Rosenfeld, ed.), (Stockholm, Sweden), pp. 471–475, North Holland, Amsterdam, Aug. 1974.

[24] E. Lee and T. Parks, "Dataflow Process Networks," in *Proceedings of the IEEE*, pp. 773–799, 1995.

[25] 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, Mar. 2013.

[26] A. Grabowski, "Scott-continuous functions," *Journal of Formalized Mathematics*, vol. 10, 1998.

[27] D. McAllester, P. Panangaden, and V. Shanbhogue, "Nonexpressibility of Fairness and Signaling," *J. Comput. Syst. Sci.*, vol. 47, pp. 287–321, Oct. 1993.

[28] J. Dennis, "First Version of a Data Flow Procedure Language," in *Programming Symposium, Proceedings Colloque Sur La Programmation*, (London, UK), pp. 362–376, Springer-Verlag, 1974.

[29] E. Lee and E. Matsikoudis, "A Denotational Semantics for Dataflow with Firing," in *Memorandum UCB/ERL M97/ 3, Electronics Research*, 1997.

[30] 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.

[31] 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.

[32] 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.

[33] C. Lucarz, *Dataflow Programming for Systems Design Space Exploration for Multicore Platforms.* PhD thesis, EPFL - STI - EDIC, Lausanne, 2011.

[34] 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.

[35] S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, eds., *Handbook of Signal Processing Systems.* Springer, 2013.

[36] W. Najjar, E. Lee, and G. Gao, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, no. 13, pp. 1907–1929, 1999.

[37] E. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[38] E. Lee and D. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Comput.*, vol. 36, pp. 24–35, Jan. 1987.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] S. Bhattacharyya, E. Deprettere, and B. Theelen, "Dynamic dataflow graphs," in *Handbook of Signal Processing Systems*, pp. 905–944, Springer, 2013.

[44] M. Geilen and T. Basten, "Kahn process networks and a reactive extension," in *Handbook of Signal Processing Systems*, pp. 1041–1081, Springer, 2013.

[45] 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.

[46] 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.

[47] J. Ersfolk, "Scheduling Dynamic Dataflow Graphs with Model Checking," 2014. PhD Thesis, TUCS Dissertations.

[48] 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.

[49] L. Torczon and K. Cooper, *Engineering A Compiler*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd ed., 2011.

[50] F. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM*, vol. 19, pp. 137–147, Mar. 1976.

[51] 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.

[52] I. 23001-4:2011, "Information technology - MPEG systems technologies - Part 4: Codec configuration representation," 2011.

[53] 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.

[54] M. Mattavelli, "MPEG reconfigurable video representation," in *The MPEG Representation of Digital Media* (L. Chiariglione, ed.), pp. 231–247, Springer New York, 2012.

[55] 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.

[56] "The Open RVC-CAL Compiler, Orcc." http://github.com/orcc. Accessed: May 2015.

[57] M. Wipliez, *Compilation infrastructure for dataflow programs*. Theses, INSA de Rennes, Dec. 2010.

[58] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia Development Made Easy," in *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pp. 863–866, ACM, 2013.

[59] "Eclipse IDEs." http://eclipse.org/ide. Accessed: May 2015.

[60] "Eclipse modeling framework." http://eclipse.org/modeling/emf. Accessed: May 2015.

[61] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd ed., 2009.

[62] "Xtext: Language development made easy!." http://eclipse.org/Xtext. Accessed: May 2015.

[63] "Xtend: Modernized java." http://eclipse.org/xtend. Accessed: May 2015.

[64] "Xronos." http://github.com/orcc/xronos. Accessed: May 2015.

[65] E. Bezati, *High-level synthesis of dataflow programs for heterogeneous platforms: design flow tools and design space exploration.* PhD thesis, EPFL - STI - EDMI, Lausanne, 2015.

[66] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing Hardware from Dataflow Programs: An MPEG-4 Simple Profile Decoder Case Study," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 241–249, 2009.

[67] 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.

[68] M. Ravasi and M. Mattavelli, "High-abstraction level complexity analysis and memory architecture simulations of multimedia algorithms," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 15, pp. 673–684, May 2005.

[69] A. Abran, *Software Metrics and Software Metrology.* Wiley-IEEE Computer Society Pr, 2010.

[70] C. Zebelein, J. Falk, C. Haubelt, and J. Teich, "Classification of General Data Flow Actors into Known Models of Computation," in *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pp. 119–128, Jun. 2008.

[71] M. Wipliez and M. Raulet, "Classification and transformation of dynamic dataflow programs," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pp. 303–310, Oct. 2010.

[72] M. Wipliez and M. Raulet, "Classification of Dataflow Actors with Satisfiability and Abstract Interpretation," *IJERTCS*, vol. 3, no. 1, pp. 49–69, 2012.

[73] I. Chukhman, W. Plishker, and S. Bhattacharyya, "Instrumentation-driven model detection for dataflow graphs," in *System on Chip (SoC), 2012 International Symposium on*, pp. 1–8, Oct. 2012.

[74] Y. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," *SIGPLAN Not.*, vol. 30, pp. 88–98, Nov. 1995.

[75] P. Puschner and C. Koza, "Calculating the Maximum Execution Time of Real-time Programs," *Real-Time Syst.*, vol. 1, pp. 159–176, Sep. 1989.

[76] S. Conte, H. Dunsmore, and Y. Shen, *Software Engineering Metrics and Models.* Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1986.

[77] V. Shen, S. Conte, and H. Dunsmore, “Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support,” *Software Engineering, IEEE Transactions on*, vol. SE-9, pp. 155–165, Mar. 1983.

[78] T. McCabe, “A Complexity Measure,” *Software Engineering, IEEE Transactions on*, vol. SE-2, pp. 308–320, Dec. 1976.

[79] R. Prather, “Theory of program testing: An overview,” *Bell System Technical Journal*, vol. 62, pp. 3073–3105, Dec. 1983.

[80] M. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[81] P. Hamer and G. Frewin, “M.H. Halstead’s Software Science - a Critical Examination,” in *Proceedings of the 6th International Conference on Software Engineering*, ICSE ’82, (Los Alamitos, CA, USA), pp. 197–206, IEEE Computer Society Press, 1982.

[82] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” *SIGPLAN Not.*, vol. 40, pp. 190–200, Jun. 2005.

[83] L. Gao, J. Huang, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr, “TotalProf: a fast and accurate retargetable source code profiler,” in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 305–314, ACM, 2009.

[84] J. Eusse, C. Williams, and R. Leupers, “CoEx: A novel profiling-based algorithm/architecture co-exploration for ASIP design,” in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, pp. 1–8, IEEE, 2013.

[85] J. Castrillon and R. Leupers, “Parallel Code Flow,” in *Programming Heterogeneous MPSoCs*, pp. 123–164, Springer International Publishing, 2014.

[86] I. Chukhman and S. Bhattacharyya, “Instrumentation-driven framework for validation of dataflow applications,” in *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1–6, Oct. 2014.

[87] G. De-Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st ed., 1994.

[88] G. De-Micheli and R. Gupta, “Hardware/Software Co-Design,” *IEEE MICRO*, vol. 85, pp. 349–365, 1997.

[89] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli, “Design of Embedded Systems: Formal Models, Validation, and Synthesis,” in *PROCEEDINGS OF THE IEEE*, pp. 366–390, 1999.

[90] A. Kienhuis, *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools.* PhD thesis, Delft University of Technology, The Netherlands, Jan. 1999.

[91] G. De-Micheli, R. Ernst, and W. Wolf, eds., *Readings in Hardware/Software Co-design*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[92] A. Nandi and R. Marculescu, "System-level Power/Performance Analysis for Embedded Systems Design," in *Proceedings of the 38th Annual Design Automation Conference*, DAC'01, (New York, NY, USA), pp. 599–604, ACM, 2001.

[93] 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.

[94] 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," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 753–758, Mar. 2010.

[95] 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.

[96] J. Castrillon, W. Sheng, and R. Leupers, "Trends in embedded software synthesis," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pp. 347–354, IEEE, 2011.

[97] J. Castrillon, R. Leupers, and G. Ascheid, "Maps: Mapping concurrent dataflow applications to heterogeneous MPSoC," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 527–545, 2013.

[98] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level Design: Orthogonalization of Concerns and Platform-based Design," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 19, pp. 1523–1543, Nov. 2006.

[99] E. Lee, "Overview of The Ptolemy Project," Technical Memo UCB/ERL M98/71, Electronics Research Laboratory, University of California at Berkeley, Nov. 1998.

[100] M. Gries, "Methods for Evaluating and Covering the Design Space During Early Design Development," *Integr. VLSI J.*, vol. 38, pp. 131–183, Dec. 2004.

[101] M. Pelcat, J. Nezan, J. Piat, J. Croizer, and S. Aridhi, "A System-Level Architecture Model for Rapid Prototyping of Heterogeneous Multicore Embedded Systems," in *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, (nice, France), p. 8 pages, Sep. 2009.

[102] E. Bezati, R. Thavot, G. Roquier, and M. Mattavelli, “High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms,” *Journal of Real-Time Image Processing*, vol. 9, no. 1, pp. 251–262, 2014.

[103] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli, “Formal Models for Embedded System Design,” *IEEE Design & Test of Computers*, vol. 17, no. 2, pp. 14–27, 2000.

[104] R. Ernst, “Codesign of embedded systems: Status and trends,” *Design & Test of Computers, IEEE*, vol. 15, no. 2, pp. 45–54, 1998.

[105] K. Miettinen, *Nonlinear multiobjective optimization.* Kluwer Academic Publishers, Boston, 1999.

[106] S. Kunzli, *Efficient Design Space Exploration for Embedded Systems.* PhD thesis, ETH Zurich, Apr. 2006.

[107] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC.* Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[108] G. Palermo, C. Silvano, and V. Zaccaria, “Multi-objective Design Space Exploration of Embedded Systems,” *J. Embedded Comput.*, vol. 1, pp. 305–316, Aug. 2005.

[109] C. Chantrapornchai, E. Sha, and X. Hu, “Efficient design exploration based on module utility selection,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, pp. 19–29, Jan. 2000.

[110] A. Ghosh and T. Givargis, “Analytical design space exploration of caches for embedded systems,” in *In Design Automation and Test in Europe (DATE*, Press, 2003.

[111] K. Lahiri, A. Raghunathan, and S. Dey, “Design Space Exploration for Optimizing On-Chip Communication Architectures,” in *IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 952–961, 2004.

[112] S. Rajagopal, J. Cavallaro, and S. Rixner, “Design Space Exploration for Real-Time Embedded Stream Processors,” *IEEE Micro*, vol. 24, pp. 54–66, Jul. 2004.

[113] P. Czyzzak and A. Jaszkiewicz, “Pareto simulated annealing—a metaheuristic technique for multiple-objective combinatorial optimization,” *Journal of Multi-Criteria Decision Analysis*, vol. 7, no. 1, pp. 34–47, 1998.

[114] G. Agosta, G. Palermo, and C. Silvano, “Multi-objective Co-exploration of Source Code Transformations and Design Space Architectures for Low-power Embedded Systems,” in *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC ’04, (New York, NY, USA), pp. 891–896, ACM, 2004.

[115] T. Blickle, J. Teich, and L. Thiele, “System-Level Synthesis Using Evolutionary Algorithms ,” *Design Automation for Embedded Systems*, vol. 3, pp. 23–58, 1998.

[116] M. Eisenring, L. Thiele, and E. Zitzler, “Conflicting Criteria in Embedded System Design,” *IEEE Design & Test Of Computers*, vol. 17, pp. 51–59, 2000.

[117] D. Bruni, A. Bogliolo, and L. Benini, “Statistical design space exploration for application-specific unit synthesis,” in *Design Automation Conference, 2001. Proceedings*, pp. 641–646, 2001.

[118] N. Bambha, S. Bhattacharyya, J. Teich, and E. Zitzler, “Hybrid global/local search strategies for dynamic voltage scaling in embedded multiprocessors,” in *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, pp. 243–248, 2001.

[119] P. Bose and T. Conte, “Performance analysis and its impact on design,” *Computer*, vol. 31, pp. 41–49, May 1998.

[120] 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,” in *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*, pp. 279–284, Apr. 2007.

[121] M. Obaidat and G. Papadimitriou, *Applied System Simulation: Methodologies and Applications.* Springer Publishing Company, Incorporated, 2013.

[122] “CAL design suite.” http://sourceforge.net/projects/caldesignsuite/. Accessed: May 2015.

[123] “COMPA Project.” http://www.compa-project.org. Accessed: May 2015.

[124] “Daedalus: System-Level Design For Multi-Processor System-on-Chip.” http://daedalus.liacs.nl. Accessed: May 2015.

[125] 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.

[126] 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.

[127] 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.

[128] 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.

[129] 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.

[130] M. Gries and K. Keutzer, *Building ASIPs: The Mescal Methodology.* Springer Publishing Company, Incorporated, 1st ed., 2010.

[131] 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, Apr. 2003.

[132] 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.

[133] “Ptolemy project: heterogeneous modeling and design.” http://ptolemy.eecs.berkeley.edu. Accessed: May 2015.

[134] “PREESM: the parallel and real-time embedded executives scheduling method.” http://sourceforge.net/projects/preesm/. Accessed: May 2015.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

[139] 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.

[140] “Space Codesign Systems.” http://http://www.spacecodesign.com. Accessed: May 2015.

[141] 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.

[142] 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.

[143] 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.

[144] 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.

[145] “SynDEx.” http://www.syndex.org. Accessed: May 2015.

[146] “SystemCoDesigner.” http://www.mycodesign.com/research/scd. Accessed: May 2015.

[147] 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 (DAC0́8)*, (Anaheim, CA, USA.), pp. 580–585, Jun. 2008.

[148] 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.

[149] “Forte Synthesizer.” http://www.cadence.com/products/sd/cynthesizer/. Accessed: May 2015.

[150] A. Mazurkiewicz, “Trace theory,” in *Petri Nets: Applications and Relationships to Other Models of Concurrency* (W. Brauer, W. Reisig, and G. Rozenberg, eds.), vol. 255 of *Lecture Notes in Computer Science*, pp. 278–324, Springer Berlin Heidelberg, 1987.

[151] T. Kahl, “Relative directed homotopy theory of partially ordered spaces,” *Journal of Homotopy and Related Structures*, vol. 1, no. 1, pp. 79–100, 2006.

[152] L. Fajstrup, E. Goubault, and M. Rauben, “Algebraic Topology And Concurrency,” tech. rep., Theoretical Computer Science, 1998.

[153] L. Fajstrup, E. Goubault, E. Haucourt, S. Mimram, and M. Raussen, "Trace Spaces: An Efficient New Technique for State-Space Reduction," in *Programming Languages and Systems* (H. Seidl, ed.), vol. 7211 of *Lecture Notes in Computer Science*, pp. 274–294, Springer Berlin Heidelberg, 2012.

[154] J. Janneck, I. Miller, and D. Parlour, "Profiling dataflow programs," in *Multimedia and Expo, 2008 IEEE International Conference on*, pp. 1065–1068, Jun. 2008.

[155] J. Gross and J. Yellen, *Graph Theory and Its Applications, Second Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2005.

[156] J. Peterson, "Petri Nets," *ACM Comput. Surv.*, vol. 9, pp. 223–252, Sep. 1977.

[157] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr. 1989.

[158] J. Rocha, L. Gomes, and O. Dias, "Dataflow model property verification using petri net translation techniques," in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pp. 783–788, Jul. 2011.

[159] R. David and H. Alla, "Petri nets for modeling of dynamic systems: A survey," *Automatica*, vol. 30, no. 2, pp. 175–202, 1994.

[160] K. Jensen and L. Kristensen, *Coloured Petri Nets*. Springer Berlin Heidelberg, 2009.

[161] K. Ogata, *Modern Control Engineering*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 4th ed., 2001.

[162] D. Spinellis, "Git," *Software, IEEE*, vol. 29, pp. 100–101, May 2012.

[163] "Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit." http://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html. Accessed: May 2015.

[164] M. Arslan, J. Janneck, and K. Kuchcinski, "Partitioning and mapping dynamic dataflow programs," in *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, pp. 1452–1456, Nov. 2012.

[165] J. Ahmad, S. Li, R. Thavot, and M. Mattavelli, "Secure computing with the MPEG RVC framework," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1315 – 1334, 2013.

[166] 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.

[167] U. Mirza, F. Gruian, and K. Kuchcinski, "Design Space Exploration for Streaming Applications on Multiprocessors with Guaranteed Service NoC," in *Proceedings of the Sixth International Workshop on Network on Chip Architectures*, NoCArc '13, (New York, NY, USA), pp. 35–40, ACM, 2013.

[168] F. Palumbo, N. Carta, D. Pani, P. Meloni, and L. Raffo, “The multi-dataflow composer tool: generation of on-the-fly reconfigurable platforms,” *Journal of Real-Time Image Processing*, vol. 9, no. 1, pp. 233–249, 2014.

[169] C. Sau, L. Raffo, F. Palumbo, E. Bezati, S. Casale-Brunet, and M. Mattavelli, “Automated design flow for coarse-grained reconfigurable platforms: An RVC-CAL multi-standard decoder use-case,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pp. 59–66, Jul. 2014.

[170] J. Janneck, S. Casale-Brunet, and M. Mattavelli, “Characterizing communication behavior of dataflow programs using trace analysis,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pp. 44–50, Jul. 2014.

[171] D. Bhowmik, A. Wallace, R. Stewart, X. Qian, and G. Michaelson, “Profile Driven Dataflow Optimisation of Mean Shift Visual Tracking,” in *IEEE Global Conference on Signal and Information Processing (GlobalSIP), 2014 Conference on*, Dec. 2014.

[172] “TURNUS Orcc RVC-CAL Profiler.” http://github.com/turnus/profiler-orcc. Accessed: May 2015.

[173] “PAPI: Performance Application Programming Interface.” http://icl.cs.utk.edu/papi. Accessed: May 2015.

[174] 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.

[175] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, “A Portable Programming Interface for Performance Evaluation on Modern Processors,” *Int. J. High Perform. Comput. Appl.*, vol. 14, pp. 189–204, Aug. 2000.

[176] “Caltoopia.” http://www.caltoopia.org. Accessed: May 2015.

[177] “Pin, A Dynamic Binary Instrumentation Tool.” http://software.intel.com/en-us/articles/pintool. Accessed: May 2015.

[178] “GCC, the GNU Compiler Collection.” http://gcc.gnu.org. Accessed: May 2015.

[179] “Intel C++ Compiler.” https://software.intel.com/en-us/c-compilers. Accessed: May 2015.

[180] B. Zeigler, T. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. Orlando, FL, USA: Academic Press, Inc., 2nd ed., 2000.

[181] J. Nutaro, *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Wiley Publishing, 2010.

[182] E. Coffman, *Computer and Job Shop Scheduling Theory.* New York: John Wiley & Sons Inc, 1976.

[183] K. Ravindran, *Task Allocation and Scheduling of Concurrent Applications to Multiprocessor Systems.* PhD thesis, EECS Department, University of California, Berkeley, Dec. 2007.

[184] C. Yang and B. 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.

[185] C. Alexander, D. Reese, and J. Harden, "Near-Critical Path Analysis of Program Activity Graphs," in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, MASCOTS '94, (Washington, DC, USA), pp. 308–317, IEEE Computer Society, 1994.

[186] D. West, *Introduction to Graph Theory.* Prentice Hall, 2 ed., Sep. 2000.

[187] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* New York, NY, USA: Cambridge University Press, 3 ed., 2007.

[188] R. Walpole, R. Myers, S. Myers, and K. Ye, *Probability & statistics for engineers and scientists.* Upper Saddle River: Pearson Education, 8th ed., 2007.

[189] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[190] J. Gustafson, "Reevaluating Amdahl's Law," *Commun. ACM*, vol. 31, pp. 532–533, May 1988.

[191] S. Krishnaprasad, "Uses and abuses of Amdahl's law," *Journal of Computing Sciences in Colleges*, vol. 17, no. 2, pp. 288–293, 2001.

[192] S. Battacharyya, E. Lee, and P. Murthy, *Software Synthesis from Dataflow Graphs.* Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[193] P. Murthy and S. Bhattacharyya, *Memory Management for Synthesis of DSP Software.* CRC Press, 2006.

[194] S. Stuijk, M. Geilen, and T. Basten, "Exploring Trade-offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs," in *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, (New York, NY, USA), pp. 899–904, ACM, 2006.

[195] T. Parks, *Bounded scheduling of process networks.* PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1995. UMI Order No. GAX96-21312.

[196] 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*, CODES+ISSS '09, (New York, NY, USA), pp. 61–70, ACM, 2009.

[197] 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.

[198] C. Garcia, D. Prett, and M. Morari, "Model predictive control: Theory and practice - A survey," *Automatica*, vol. 25, no. 3, pp. 335–348, 1989.

[199] S. Qin and T. Badgwell, "A survey of industrial model predictive control technology," *Control Engineering Practice*, vol. 11, no. 7, pp. 733–764, 2003.

[200] B. Ghavami and H. Pedram, "High performance asynchronous design flow using a novel static performance analysis method," *Comput. Electr. Eng.*, vol. 35, pp. 920–941, Nov. 2009.

[201] P. Kudva, G. Gopalakrishnan, E. Brunvand, and V. Akella, "Performance analysis and optimization of asynchronous circuits," in *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, pp. 221–224, 1994.

[202] S. Suhaib, D. Mathaikutty, and S. Shukla, "Dataflow architectures for GALS," *Electronic Notes in Theoretical Computer Science*, vol. 200, no. 1, pp. 33–50, 2008.

[203] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D. Lundqvist, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style," in *Design Automation Conference, 1999. Proceedings. 36th*, pp. 873–878, 1999.

[204] T. Wuu and S. Vrudhula, "Synthesis of Asynchronous Systems from Data Flow Specification," Research Report ISI/RR-93-366, University of Southern California, Information Sciences Institute, Dec. 1993.

[205] 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.

[206] "Open RVC-CAL Applications." http://github.com/orcc/orc-apps. Accessed: May 2015.

[207] W. Hamidouche, M. Raulet, and O. Deforges, "Real time SHVC decoder: Implementation and complexity analysis," in *Image Processing (ICIP), 2014 IEEE International Conference on*, pp. 2125–2129, Oct. 2014.

[208] "Open HEVC decoder." http://github.com/OpenHEVC/openHEVC. Accessed: May 2015.

[209] “International Telecommunication Union (ITU) HEVC conformance bit-stream collection (draft).” http://wftp3.itu.int/av-arch/jctvc-site/bitstream_exchange/draft_conformance/. Accessed: May 2015.

[210] “Gephi Toolkit.” https://github.com/gephi. Accessed: May 2015.

[211] M. Bastian, S. Heymann, and M. Jacomy, “Gephi: An Open Source Software for Exploring and Manipulating Networks,” in *International AAAI Conference on Weblogs and Social Media*, 2009.

[212] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, “Platform 2012, a Many-core Computing Accelerator for Embedded SoCs: Performance Evaluation of Visual Analytics Applications,” in *Proceedings of the 49th Annual Design Automation Conference*, DAC ’12, (New York, NY, USA), pp. 1137–1142, ACM, 2012.

[213] “Xilinx Power Estimator (XPE).” http://www.xilinx.com/products/design_tools/logic_design/xpe.htm. Accessed: May 2015.

[214] “Parallella Board.” https://www.parallella.org. Accessed: May 2015.

[215] L. De-Moura and N. Bjorner, “Satisfiability modulo theories: An appetizer,” in *Formal Methods: Foundations and Applications*, pp. 23–36, Springer, 2009.

[216] “Blueprints: A Property Graph Model Interface.” https://github.com/tinkerpop/blueprints. Accessed: May 2015.

[217] M. Ciglan, A. Averbuch, and L. Hluchy, “Benchmarking traversal operations over graph databases,” in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pp. 186–189, IEEE, 2012.

[218] S. Jouili and V. Vansteenberghe, “An empirical comparison of graph databases,” in *Social Computing (SocialCom), 2013 International Conference on*, pp. 708–715, IEEE, 2013.

[219] “Hotel Bouton d’Or, Courmayeur Mont-Blanc.” http://www.hotelboutondor.com. Accessed: May 2015.

[220] “Baking a Hello World Cake.” http://www.mike-worth.com/2013/03/31/baking-a-hello-world-cake/. Accessed: May 2015.

[221] “Chef.” http://www.dangermouse.net/esoteric/chef.html. Accessed: May 2015.

[222] “Numb3rs: End of Watch (18 Dec. 2014), Season 3, Episode 8.” http://www.casalebrunet.com/phd/video/criticalPath.html. Accessed: May 2015.

Simone CASALE BRUNET

Doctoral Assistant

`casalebrunet@ieee.org` ● www.casalebrunet.com
skype: casalebrunet ● github: casalebrunet ● twitter: casalebrunet

Last update: May 2015

Summary Simone Casale-Brunet received a B.S. degree in Electrical Engineering (2008) and an M.S. degree in Mechatronics Engineering (2010), both with highest honours, from the Politecnico di Torino, Italy. In 2010 he joined the EPFL SCI STI MM group of the École Polytechnique Fédérale de Lausanne, Switzerland, where he is currently a Ph.D. candidate under the supervision of Dr. Marco Mattavelli. His research interests include design space exploration of heterogeneous parallel systems and advanced control theory. His research work is sponsored by the Fonds National Suisse pour la Recherche Scientifique.

Experience

December 2010 - June 2015 (exp.), École Polytechnique Fédérale de Lausanne
Doctoral assistant: in the SCI-STI-MM Multimedia Group, under the supervision of Dr. Marco Mattavelli. Design methodologies for software/hardware applications for digital signal processing and communication.

September 2010 - December 2010, Politecnico di Torino
Research assistant: in the Department of control and computer engineering. Development of hard real-time model predictive controller, system identification and optimization.

Education

2010 - present, Electrical Engineering Doctoral School
Main research topic: Design space exploration and optimization for high parallel heterogeneous systems using high-level dataflow representation.
Supervisor: Dr. Marco Mattavelli

2008 - 2010, MSc Mechatronics Engineering (Summa Cum Laude)
Advanced control theory, electronics engineering, computer science engineering, mechanical engineering, mathematical optimization.
Supervisor: Prof. Massimo Canale

2005 - 2008, BSc Electronics Engineering (Summa Cum Laude)
Electronics engineering, computer science engineering, control theory, business management
Supervisor: Prof. Massimo Canale

Skills

Foreign Languages
○ Italian *(Mother tongue)*
○ French *(Bilingual)*
○ English *(Fluent)*

Programming Languages
○ Java
○ C/C++
○ PhP
○ CAL

Operating Systems
○ MacOS
○ GNU/Linux
○ Windows

Collaborative Projects

- **(FP7) ICT-ALICANTE**
  Media Ecosystem Deployment through Ubiquitous Content-Aware Network Environments.
  http://www.ict-alicante.eu
- **(EUREKA's Eurostars) VAMPA**
  Embedded Video content Analysis on the STM STHORM Multicore Architecture.
  http://vampa.epfl.ch

Open Source Software

- **TURNUS**: (main contributor and maintainer)
  a computer-aided co-exploration framework that guides designers during the co-exploration and optimization process. Released under GPL3 licence.
  http://github.com/turnus
- **Open RVC-CAL Compiler (Orcc)**: (code interpreter contributor)
  an RVC-CAL compiler infrastructure that allow several languages (software and hardware) to be generated from the same description composed of RVC-CAL actors and XDF networks. Released under BSD licence.
  http://github.com/orcc

Grants and Sponsorships

- Fonds National Suisse pour la Recherche Scientifique, grant 200021.138214

Service to the Profession

- Session Co-Chair, Applications of Model Predictive Control
  12th European Control Conference (ECC13), Zurich, July 2013

Professional Memberships

- Member of the Institute of Electrical and Electronics Engineers (IEEE)
  and the IEEE Computer Society (CS)
  and the IEEE Circuits and Systems Society (CSS)
  and the IEEE Council on Electronic Design Automation (CEDA)
- Member of the Association for Computing Machinery (ACM)

References

- **Dr. Marco Mattavelli**
  École Polytechnique Fédérale de Lausanne, EPFL SCI STI MM, Switzerland
  marco.mattavelli@epfl.ch
- **Prof. Massimo Canale**
  Politecnico di Torino, Dipartimento di Automatica e Informatica, Italy
  massimo.canale@polito.it

Publications

Journals

2014

[J2] M. Canale and S. Casale-Brunet. A multidisciplinary approach for Model Predictive Control Education: A Lego Mindstorms NXT-based framework. *International Journal of Control, Automation and Systems*, 12(5):1030–1039, 2014

2012

[J1] 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.*, 28(10):1278–1294, Nov. 2013

Conferences

2014

[C23] 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

[C22] 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)*, 2014

[C21] 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*, pages 1–6, Oct. 2014

[C20] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. Janneck. Coarse grain clock gating of streaming applications in programmable logic implementations. In *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*, pages 1–6, 2014

[C19] J. Janneck, S. Casale-Brunet, and M. Mattavelli. Characterizing communication behavior of dataflow programs using trace analysis. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 44–50, Jul. 2014

[C18] A. Ab-Rahman, S. Casale-Brunet, C. Alberti, and M. Mattavelli. A methodology for optimizing buffer sizes of dynamic dataflow FPGAs implementations. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 5003–5007, May 2014

[C17] C. Sau, L. Raffo, F. Palumbo, E. Bezati, S. Casale-Brunet, and M. Mattavelli. Automated design flow for coarse-grained reconfigurable platforms: An RVC-CAL multi-standard decoder use-case. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 59–66, Jul. 2014

[C16] D. de Saint-Jorre, C. Alberti, M. Mattavelli, and S. Casale-Brunet. Exploring MPEG HEVC decoder parallelism for the efficient porting onto many-core platforms. In *Image Processing (ICIP), 2014 IEEE International Conference on*, pages 2115–2119, Oct. 2014

[C15] J. Janneck, G. Cedersjo, E. Bezati, and S. Casale-Brunet. Dataflow Machines. In *Signals, Systems and Computers, 2014 Asilomar Conference on*, Nov. 2014

2013 [C14] S. Casale-Brunet, M. Mattavelli, C. Alberti, and J. Janneck. Systems design space exploration by serial dataflow program execution. In *Signals, Systems and Computers, 2013 Asilomar Conference on*, pages 1805–1809, Nov. 2013

[C13] S. Casale-Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. Janneck. Multi-clock domain optimization for reconfigurable architectures in high-level dataflow applications. In *Signals, Systems and Computers, 2013 Asilomar Conference on*, pages 1796–1800, Nov. 2013

[C12] S. Casale-Brunet, M. Mattavelli, C. Alberti, and J. Janneck. Design Space Exploration of High-Level Stream Programs on Parallel Architectures. *Conference: 8th International Symposium on Image and Signal Processing and Analysis (ISPA 2013), Trieste, Italy*, pages 738–743, Sep. 2013

[C11] 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*, pages 1384–1387, May 2013

[C10] S. Casale-Brunet, M. Mattavelli, and J. Janneck. TURNUS: A design exploration framework for dataflow system design. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 654–654, May 2013

[C9] S. Casale-Brunet, E. Bezati, C. Alberti, G. Roquier, M. Mattavelli, J. Janneck, and J. Boutellier. Design space exploration and implementation of RVC-CAL applications using the TURNUS framework. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 341–342, Oct. 2013

[C8] S. Casale-Brunet, C. Alberti, M. Mattavelli, and J. Janneck. TURNUS: A unified dataflow design space exploration framework for heterogeneous parallel systems. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 47–54, Oct. 2013

[C7] M. Casale-Brunet, S.and Mattavelli, C. Alberti, and J. Janneck. Representing Guard Dependencies in Dataflow Execution Traces. In *Computational Intelligence, Communication Systems and Networks (CICSyN), 2013 Fifth International Conference on*, pages 291–295, 2013

[C6] 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. In *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pages 177–182, Oct. 2013

[C5] A. Ab-Rahman, S. Casale-Brunet, C. Alberti, and M. Mattavelli. Dataflow program analysis and refactoring techniques for design space exploration: MPEG-4 AVC/H.264 decoder implementation case study. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 63–70, Oct. 2013

[C4] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. Janneck. Synthesis and optimization of high-level stream programs. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pages 1–6, May 2013

[C3] M. Canale and S. Casale-Brunet. A Lego Mindstorms NXT experiment for Model Predictive Control education. In *Control Conference (ECC), 2013 European*, pages 2549–2554, Jul. 2013

2012 [C2] S. Casale-Brunet, M. Mattavelli, and J.W. Janneck. Profiling of Dataflow Programs Using Post Mortem Causation Traces. In *Signal Processing Systems (SiPS), 2012 IEEE Workshop on*, pages 220–225, Oct. 2012

[C1] A. Ab-Rahman, R. Thavot, S. Casale-Brunet, E. Bezati, and M. Mattavelli. Design space exploration strategies for FPGA implementation of signal processing systems using CAL dataflow program. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pages 1–8, Oct. 2012