From dataflow-based video coding tools to dedicated embedded multi-core platforms

Hervé Yviquel

▶ To cite this version:

Hervé Yviquel. From dataflow-based video coding tools to dedicated embedded multi-core platforms. Other [cs.OH]. Université de Rennes, 2013. English. NNT : 2013REN1S095 . tel-00939346

HAL Id: tel-00939346

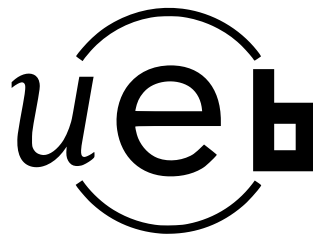
https://theses.hal.science/tel-00939346v1

Submitted on 30 Jan 2014

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

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

ANNÉE 2013



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Hervé Yviquel

préparée à l'unité de recherche IRISA (UMR 6074)
Institut de Recherche en Informatique et Systèmes Aléatoires
École Nationale Supérieure des Sciences Appliquées et de Technologie

---

From Dataflow-Based Video Coding Tools to Dedicated Embedded Multi-Core Platforms

Thèse soutenue à Lannion le 25 octobre 2013

devant le jury composé de :

Alain GIRAULT,
Directeur de recherche, Inria Rhône-Alpes
/ Rapporteur

Marco MATTAVELLI,
Maitre d'enseignement et de recherche, École Polytechnique Fédérale de Lausanne
/ Rapporteur

Tanguy RISSET,
Professeur des universités, Institut National des Sciences Appliquées de Lyon
/ Examinateur

Jarmo TAKALA,
Professeur, Tampere University of Technology
/ Examinateur

Emmanuel CASSEAU
Professeur des universités, Université de Rennes 1
/ Directeur de thèse

Mickaël RAULET
Ingénieur de Recherche, Institut National des Sciences Appliquées de Rennes
/ Co-directeur de thèse

Hervé Yviquel: *From Dataflow-based Video Coding Tools to Dedicated Embedded Multi-core Platforms*, © 2010-2013

Don’t loaf and invite inspiration;
light out after it with a club,

and if you don’t get it
you will nonetheless get something
that looks remarkably like it.

— Jack London

ABSTRACT

The development of multimedia technology, along with the emergence of parallel architectures, has revived the interest on dataflow programming for designing embedded systems. Indeed, dataflow programming offers a flexible development approach in order to build complex applications while expressing concurrency and parallelism explicitly. Paradoxically, most of the studies focus on static dataflow models of computation, even if a pragmatic development process requires the expressiveness and the practicality of a programming language based on dynamic dataflow models, such as the language included in the Reconfigurable Video Coding framework.

In this thesis, we describe a development environment for dataflow programming that eases multimedia development for embedded multi-core platforms. This development environment is built upon a modular software architecture that benefits from modern software engineering techniques such as meta modeling and aspect-oriented programming. Then, we develop an optimized software implementation of dataflow programs targeting desktop and embedded multi-core platforms. Our implementation aims to bridge the gap between the practicality of the programming language and the efficiency of the execution. Finally, we present a set of runtime actors mapping/scheduling algorithms that enable the execution of dynamic dataflow programs over multi-core platforms with scalable performance.

RÉSUMÉ

Le développement du multimédia, avec l'émergence des architectures parallèles, a ravivé l'intérêt de la programmation flux de données pour la conception de systèmes embarqués. En effet, la programmation flux de données offre une approche de développement suffisament flexible pour créer des applications complexes tout en exprimant la concurrence et le parallélisme explicitement. Paradoxalement, la plupart des études portent sur des modèles flux de données statiques, même si un processus de développement pragmatique nécessite l'expressivité et la practicité d'un langage de programmation basé sur un modèle flux de données dynamiques, comme le langage de programmation utilisé dans le cadre de Reconfigurable Video Coding.

Dans cette thèse, nous décrivons un environnement de développement pour la programmation flux de données qui facilite le développement multimédia pour des plates-formes multi-cœr embarquées. Cet environnement de développement repose sur une architecture logicielle modulaire qui bénéficie de techniques modernes de génie logiciel telles que la méta modélisation et la programmation orientée aspect. Ensuite, nous développons une implémentation logicielle optimisée des programmes flux de données ciblant aussi bien les ordinateurs de bureau que les plates-formes embarquées. Notre implémentation vise à combler le fossé entre la practicité du langage de programmation et l'efficacité de son exécution. Enfin, nous présentons un ensemble d'algorithmes de projection et d'ordonnancement d'acteurs qui permettent l'exécution de programmes flux de données dynamiques sur des plates-formes multi-cœur avec des performances extensibles.

ACKNOWLEDGMENTS

*Feeling gratitude and
not expressing it is like
wrapping a present and
not giving it.*

— William Arthur Ward

First, I would like to thank my advisors Pr Emmanuel Casseau and Dr Mickaël Raulet for their help and support during these three years. Working with both of you has been a very pleasant experience from a scientific point of view, as much as from a human relation point of view. Thank you for your trust in my work, the freedom you let me has been a great source of motivation. Emmanuel, thank you for all your consideration: Your wise advices as well as your ability to take a step back on my work are one of the reasons of the success of my PhD. Mickaël, Thank you for the close and unlimited support: Your expertise in video decoding has always been very helpful to make things working, and our long discussions have helped me to take the right decisions. In fact, working with both of you has been very enriching and I hope that our collaboration will be able to continue.

I would also like to give my thanks to Pr Alain Girault and Pr Marco Mattavelli for reviewing this thesis, and to Pr Tanguy Risset and Pr Jarmo Takala for participating to the jury. All your comments on my work were detailed and very encouraging. Marco, thank you for allowing the multiple collaborations with your team. Tanguy, thank you for you interest on my work. Alain, thank your again for your reviewing: Reading your comments on my thesis was a true pleasure. Jarmo, thank you for your involvement in my PhD: The few months I spent in visit at Tampere have been truly profitable for me.

I would like to extend my thanks to the people that I have had the pleasure to work with. Thanks to the former PhD students for introducing me to the world of Orcc: Matthieu Wipliez, Jérôme Gorin and Nicolas Siret. Matthieu, our never-ending debates have always been a pleasure for me. Thanks to all my collegues with who I have enjoyed working with: Antoine Lorence, Khaled Jerbi, Alexandre Sanchez, Maxime Pelcat, Jean-François Nezan. I would also like to thank Pekka Jääskeläinen for making me feel very welcome during my visit in Tampere. In fact, I would like to thank the Orcc and TCE communities as a whole for actively participating in the development of the tools which offers solid basements to this work. I would also give a special thanks to Angélique Le Pennec and Jocelyne Tremier for managing administrative tasks seamlessly.

Additionally, I would like to thank my family and friends for their support during all these years. Big thanks to my parents and sisters for their love. Thank you for accepting my craziness has it is. Many thanks to all my friends: Thank you for all the good time spent together, for all the incredible parties we have made.

Finally, I would like to give a special thanks to my hidden proofreader that has spent so many nights to fix and improve the English of this thesis.

CONTENTS

1 INTRODUCTION 1
1.1 Landscape of Embedded Computing . . . 1
1.1.1 Embedded Hardware . . . 2
1.1.2 Embedded Software . . . 2
1.1.3 Embedded System Design . . . 3
1.2 Our Approach and Contributions . . . 4
1.3 Outline . . . 5

i BACKGROUND 7
2 EMBEDDED PARALLEL PROGRAMMING 9
2.1 Parallelism is Everywhere . . . 9
2.2 Embedded Parallel Platforms . . . 10
2.2.1 Homogeneous versus Heterogeneous . . . 11
2.2.2 Memory Architecture . . . 12
2.2.3 Memory Hierarchy . . . 13
2.2.4 On-Chip Interconnection Network . . . 14
2.3 Parallel Programming Models . . . 15
2.3.1 General-Purpose Parallel Programming . . . 16
2.3.2 Assisted Parallel Programing . . . 17
2.3.3 High-level Parallel Programming . . . 18
2.4 Mapping and Scheduling . . . 20
2.5 Conclusion . . . 21
3 DATAFLOW PROGRAMMING 23
3.1 Definition of a Dataflow Program . . . 24
3.2 Dataflow Paradigm to Enhance Programming . . . 24
3.2.1 Modular Programming . . . 25
3.2.2 Parallel Programming . . . 25
3.3 Model of Computation . . . 27
3.3.1 Kahn Process Network . . . 27
3.3.2 Dataflow Process Network . . . 28
3.3.3 Static Dataflow Models . . . 29
3.3.4 Quasi-Static Dataflow Model . . . 29
3.4 Comparing Dataflow MoCs . . . 30
3.4.1 Characterization of Dataflow MoCs . . . 30
3.4.2 Taxonomy of Dataflow MoCs . . . 31
3.5 Dynamic Modeling Requires Dynamic Analysis . . . 32
3.5.1 Classification . . . 32
3.5.2 Critical Path Analysis . . . 33
3.6 Execution Models . . . 34
3.6.1 Multi-Threading . . . 34
3.6.2 Dynamic Scheduling . . . 35
3.6.3 Static Scheduling . . . 36
3.6.4 Multi-core scheduling . . . 37
3.7 Existing Dataflow-based Languages and Tools . . . 38
3.8 Conclusion . . . 39
4 RECONFIGURABLE VIDEO CODING 41
4.1 Limits of the Standardization Process . . . 41
4.1.1 Multiplication of the Standards . . . 41
4.1.2 Monolithic Specifications of the Standards . . . 43

4.2 An Innovative Development Framework 43
4.2.1 Dataflow to Enhance Multimedia Development 44
4.2.2 Towards the RVC Vision 45
4.3 Multimedia-Specific Languages 46
4.3.1 From Text to Visual Network Programming 46
4.3.2 Actor Programming Made Easy 48
4.4 Applications 51
4.4.1 Video Codecs 51
4.4.2 Other Applications 53
4.5 Existing Tools Supporting RVC 53
4.5.1 OpenDF 54
4.5.2 Orcc 54
4.6 Advances and Challenges of the RVC Framework 55
4.6.1 Tools Development 55
4.6.2 Applications Development 56
4.6.3 Platform Implementation 57
4.7 Conclusion 58

ii CONTRIBUTIONS 59

5 ADVANCED DEVELOPMENT ENVIRONMENT FOR DATAFLOW PROGRAMMING 61
5.1 Enhanced Dataflow-specific Compilation Infrastructure 61
5.1.1 Multi-Target Compilation Infrastructure 62
5.1.2 Model-driven Compilation Infrastructure 63
5.1.3 Unified Graph Library 64
5.1.4 Separation of Concerns 65
5.1.5 Procedural Aspect of the Intermediate Representation 65
5.1.6 Dataflow Aspect of the Intermediate Representation . 68
5.2 Architecture Model for Dedicated Embedded Multi-Core Platforms 70
5.2.1 Processor Architecture 71
5.2.2 Predefined Configurations of Processors 72
5.2.3 Dataflow-specific Memory Architecture 72
5.3 Dataflow Compiler for Embedded Multi-core Platforms . . . 74
5.3.1 Multi-stage Co-design Flow 74
5.3.2 Hardware Synthesis 76
5.3.3 Software Synthesis 77
5.3.4 Simulation Infrastructure 78
5.4 Conclusion 80

6 OPTIMIZED SOFTWARE IMPLEMENTATION OF DYNAMIC DATAFLOW PROGRAMS 81
6.1 Implementation of Dataflow Process Networks 81
6.2 Optimized Communications 82
6.2.1 To Be or Not To Be FIFO Channels 82
6.2.2 Software Circular Buffer 83
6.2.3 Control-Free Communications 84
6.2.4 Multi-rate Communications 85
6.2.5 Copy-Free Communications 86
6.2.6 Efficient Broadcasting of Communications 87
6.3 Optimized Scheduling 88
6.3.1 Scheduling Scheduling 89
6.3.2 Action Scheduling 89
6.3.3 Actor Machine 90

6.3.4 Quasi-Static Scheduling . . . 91
6.4 Study of RVC-based Video Decoders . . . 92
6.4.1 Experimental setup . . . 92
6.4.2 Analysis of Global Performance . . . 93
6.4.3 Analysis of Internal Communications . . . 94
6.4.4 Analysis of the Application Decomposition . . . 95
6.4.5 Comparison of the Scheduling Strategies . . . 99
6.5 Conclusion . . . 101
7 SCALABLE MULTI-CORE SCHEDULING OF DYNAMIC DATAFLOW PROGRAMS 103
7.1 Actors Mapping . . . 104
7.1.1 Definition of the metrics . . . 104
7.1.2 Evolutionary-based Actor Mapping . . . 105
7.1.3 Graph Partitioning problem . . . 106
7.1.4 Graph partition methodology . . . 107
7.1.5 Mapping Flow . . . 108
7.2 Actor Scheduling . . . 109
7.2.1 Distributed Scheduler . . . 109
7.2.2 Multi-core Scheduling Strategies . . . 109
7.2.3 Lock-Free Scheduling Communications . . . 110
7.3 Scalability Analysis of RVC-based Video Decoders . . . 111
7.3.1 Experimental setup . . . 112
7.3.2 Desktop Multi-core Implementation . . . 113
7.3.3 Embedded Multi-core Implementation . . . 116
7.4 Conclusion . . . 117
8 CONCLUSIONS AND OUTLOOK 119
8.1 Summary . . . 119
8.2 Perspectives . . . 120
8.2.1 An Even More Advanced Development Environment . 120
8.2.2 An Even More Optimized Software Implementation . 121
8.2.3 Towards a Platform Dedicated to RVC-based Video Decoders . . . 122

iii APPENDIX 125
A RÉSUMÉ EN FRANÇAIS 127
A.1 Systèmes embarqués . . . 127
A.1.1 Matériels embarqués . . . 128
A.1.2 Logiciels embarqués . . . 128
A.1.3 Conception de systèmes embarqués . . . 129
A.2 Approche et contributions . . . 129
A.3 État de l'art . . . 131
A.3.1 Programmation flux de données . . . 131
A.3.2 Reconfigurable Video Coding . . . 132
A.4 Environnement de développement dédié . . . 133
A.4.1 Infrastructure de programmation flux de données . . . 134
A.4.2 Modèle d'architecture dédié . . . 134
A.4.3 Co-conception de systèmes embarqués . . . 136
A.5 Implémentation logicielle des programmes flux de donnée . . 136
A.5.1 Implémentation optimisée . . . 137
A.5.2 Implémentation extensible . . . 138
A.6 Conclusion et perspectives . . . 138
A.6.1 Environnement de développement avancé . . . 139
A.6.2 Implémentation logicielle optimisée . . . 140

A.6.3 Plate-forme dédiée aux codecs vidéo RVC 141

PUBLICATIONS 152

BIBLIOGRAPHY 153

1 INTRODUCTION

To invent an airplane is nothing.
To build one is something.
But to fly is everything.

— Otto Lilienthal, Aviation pioneer

This thesis investigates pragmatic programming approaches of real-world applications for current and upcoming embedded systems. Actually, the programming experience is becoming a central problem for embedded computing. On the one hand, embedded devices are now complex hardware systems, known as *Multi-Processor System-on-Chip* (MPSoC), that include more and more heterogeneous components on a single chip in order to increase product functionalities and to meet expectations of the embedded market. On the other hand, the complexity of the software deployed on these devices keeps growing exponentially, because these are being used to solve more difficult technical problems. As a result, programmers have to implement increasingly complex applications for increasingly complex devices while respecting time-to-market and cost demand requirements.

This thesis aims at providing a toolkit to ease the development of real-world applications for MPSoC-based platforms from a pragmatic point of view. Thus, we propose to implement and evaluate a set of methodologies for designing embedded systems from the application specification to the platform implementation. In order to benefit from all parallelism present in the algorithms, applications are already specified in a decomposed form, called *Dataflow Process Network* (DPN), by way of a practical dataflow language inheriting from CAL Actor Language. They are latter mapped onto Very Long Instruction Word -like processors which are able to execute multiple operations at the same time. We evaluate the toolkit using state-of-the-art video decoders, including the emerging High Efficiency Video Coding (HEVC) standard.

Now, let us take a look of the landscape of embedded computing in order to understand the complexity of the problematic that this thesis faces.

1.1 LANDSCAPE OF EMBEDDED COMPUTING

Embedded systems are now widely used, much more than other computing systems with billions sold every year [182], flooding the market of general-purpose computers. Recent analyses have shown a drop in sales of desktop computers in favor of smartphones, tablets and other embedded devices. As opposed to general-purpose computers, embedded systems must meet quantifiable goals: real-time performance, restricted power/energy consumption and market cost. Thus, the design of embedded systems is entirely guided by these quantifiable goals which make it much more challenging than general-purpose computers design.

1.1.1 *Embedded Hardware*

Up till recent years, embedded devices were designed around a single processor associated with a set of peripherals and hardware accelerators. However, the increasing demand for flexibility from the embedded market has resulted in a migration from hardware to software. In other words, previously hardwired functionalities are now performed by programmable processors.

To handle increasingly demanding applications, the design of higher performance processors was achieved, until recent years, by increasing processor frequency. But, similarly to general-purpose computers, embedded systems have hit the *power wall* of the semiconductor technology, forcing chip manufacturers to look towards multi-core architectures to improve the overall system performance. As a result, embedded systems integrate more and more programmable processors, but contrary to general-purpose computers, most of these processors are tailored to specific tasks in order to bridge the gap between hardware efficiency and software flexibility.

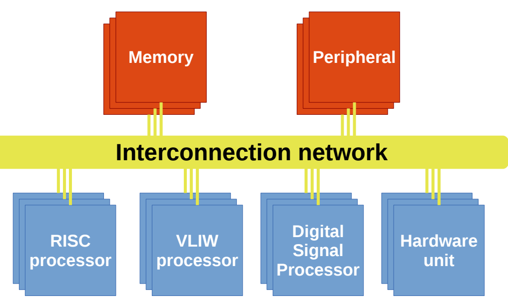


Figure 1: Generic MPSoC-based platform

Embedded devices are now complex heterogeneous multi-core platforms with an increasing number of processor cores so as to meet the performance requirement (Figure 1). For example, commercial many-core platforms like Intel SCC, Tilera TILE or STMicroelectronics SThorm [22] contain already hundred of programmable cores. The increasing number of processor cores has however raised new questions about hardware designs, such as the memory organization and the interconnection network, and about the way to program such a complex architecture.

1.1.2 *Embedded Software*

Early forms of embedded software were small programs usually written in assembly to get maximal performance. They can be now complex applications containing multiple algorithms [183]. Moreover, the nature of the computations performed in different parts of the application can vary widely (types of operations, memory requirement, parallelism, etc). As a matter of fact, this variability matches well with heterogeneous architectures. As an example, considering the structure of modern video decoders [150], the motion compensation has clearly the largest requirement in memory space and bandwidth, while the residual decoding and the intra prediction are mostly computational.

The embedded market is currently driven by user application demands increasing the complexity of embedded software. For instance, on the one hand, the new video compression standard namely HEVC reduces bit-rate requirement by 50% with same picture quality as its predecessor, and thus

allows higher-definition video. On the other hand, HEVC standard increases the computational complexity by 1.6x compared to its predecessor [171]. Complex applications are often limited to certain application domains like multimedia and communication. For example in a 3G phone, above 60% of the power and over 90% of the available performance are consumed by radio and multimedia applications [170].

Beyond the heterogeneity and the complexity of the applications, targeting multi-core platforms raises new questions concerning embedded software, such as the application decomposition in parallel tasks as well as the mapping and scheduling of these tasks on the multi-core platform.

1.1.3 *Embedded System Design*

Today, embedded computing is confronted to a fast technology evolution and a great variety of computing systems. Therefore, highly flexible design processes are required. As a matter of fact, the design of embedded systems can be decomposed in three aspects (architecture, application and methodology) as illustrated in Figure 2.

Since software and hardware are tightly coupled in embedded system design, embedded designers have to consider all architectural aspects including the organization of the hardware components (processors, memories, interconnections), the decomposition of the software in tasks in order to benefit as much as possible from the parallelism, and the mapping between the hardware and the software to get the best performance. Additionally, designers have to deeply understand their applications to take advantage of all possible optimizations. Finally, methodologies are central for successful embedded system design. Modeling provides higher-level of abstractions that are necessary to handle the growing complexity of embedded systems. As regards to the difficulty of analyzing and debugging hardware platforms, simulation and analysis are necessary to determine the efficiency and the cost of the design. Model-based design requires synthesis tools translating high-level specifications into optimized implementations. Moreover, automatic verification processes are also essential to achieve the required reliability level with minimal cost.

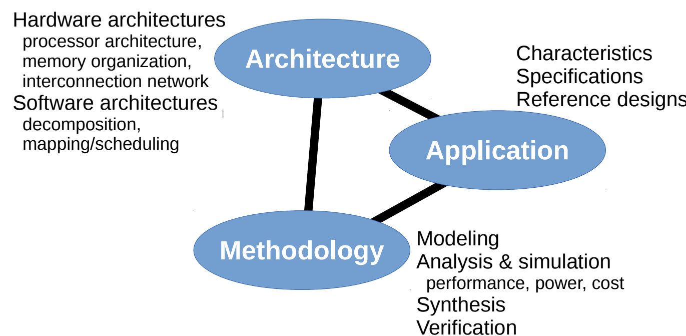


Figure 2: Aspects of embedded system design (adapted from Wolf's analysis [182])

In conclusion, tools are particularly important in embedded system design. Tools allow rapid design of embedded systems to deal with time-to-market pressure while achieving their high constraints of efficiency and reliability.

1.2 OUR APPROACH AND CONTRIBUTIONS

The emergence of massively parallel architectures, along with the need for modularity in software design, has revived the interest in dataflow programming. Indeed, dataflow programming offers a flexible development approach which is able to build complex applications while expressing concurrency and parallelism explicitly. Paradoxically, most of the studies stay focused on static dataflow programming, even if a pragmatic development process requires the expressiveness and the practicality offered by dynamic dataflow programming.

MPEG has however introduced an innovative framework, called *Reconfigurable Video Coding* (RVC), that can be considered as the first large-scale experimentation on dynamic dataflow programming to our knowledge. RVC has been initially introduced to overcome the lack of interoperability between the various video codecs deployed in the market. The framework allows the development of video coding tools, among other applications, in a modular and reusable fashion thanks to the inclusion of a subset of CAL programming language.

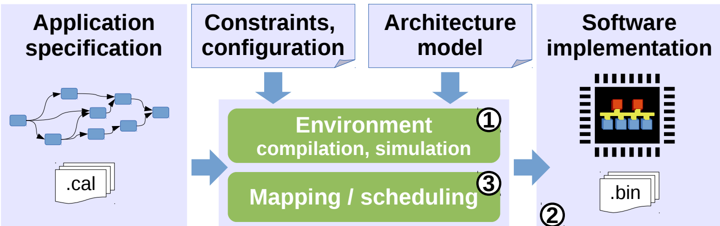


Figure 3: Contributions of this thesis on dataflow-based embedded system design

All along this thesis, we study all steps of the development of RVC-based video decoders (Figure 3), from their specification based on the dataflow paradigm to their implementation on embedded multi-core platforms. This thesis makes the following contributions:

- **Contribution 1** [5, 7, 2, 3]: An entire co-design flow to develop RVC-based applications for embedded multi-core platforms. The co-design flow relies on a advanced simulation process and a dedicated architecture model. Additionally, the multi-target compilation infrastructure underlying our co-design flow has been enhanced by the way of modern software engineering techniques such as *Model-Driven Engineering* (MDE).

- **Contribution 2** [8, 1]: An optimized software implementation of dynamic dataflow programs based on efficient communication techniques that limit the accesses to the memory, and based on advanced scheduling strategies that reduce the overhead of the scheduling.

- **Contribution 3** [4, 6]: A set of actor mapping/scheduling algorithms executable at runtime in order to handle the unpredictable behavior of dynamic dataflow programs, and to achieve scalable performance over multi-core platforms, either desktop multi-core processors or embedded multi-core platforms.

In addition to the specification of our dataflow-based development process, we evaluate the efficiency of this contribution using a set of video

decoders, including a decoder based on the HEVC standard, that were implemented within the RVC framework.

All this work has been implemented within two open-source software: a dataflow-based development environment known under the name of Open RVC-CAL Compiler (Orcc) [134], and a co-design toolkit using Transport-Trigger Architecture (TTA) as the architecture template called TTA-based Co-design Environment (TCE) [166].

1.3 outline

This thesis is decomposed in two distinct parts as follows. Part i contains an introduction to the global notions and research problems discussed in this thesis, and details also the previous works that lead to our work. Chapter 2 explores the existing programming models of embedded multi-core platforms including the influence of the hardware architecture. Then, Chapter 3 focuses on dataflow programming and shows the pragmatism underlying the dynamic dataflow model for software development. Finally, Chapter 4 introduces the Reconfigurable Video Coding framework, and deeply inspects its current state for highlighting the open challenges of the approach.

Part ii contains the contributions and proposed techniques of this thesis. Chapter 5 starts by introducing the tool flow. Chapter 6 details our optimized software implementation of dynamic dataflow programs. Chapter 7 closes this part by proposing a set of actor mapping/scheduling algorithms in order to obtain scalable performance on multi-core platforms.

Part I

BACKGROUND

2 EMBEDDED PARALLEL PROGRAMMING

We have seen that computer programming is an art,

because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [114]

Embedded computing has emerged in early in the history of computer sciences. Already in 1951, a group of researchers from the MIT have built the Whirlwind computer, the first real-time computer, which was adopted by the U.S. Air Force to integrate its air defense system. Latter in 1966, the Apollo Guidance Computer was designed at the MIT to integrate the navigation system that controls the spacecraft of the Apollo program. While the Whirlwind can hardly be considered as an embedded system because of its impressive dimensions, the Apollo Guidance Computer was embedded in the aircraft modules of several lunar missions requiring more compact dimensions. Since this time, embedded systems have massively gained in popularity, and are now everywhere.

Similarly, processor programming has moved well beyond the early days of assembly programming of 8-bit micro-controllers. The advances in compilation have allowed the translation of high-level language programs into efficient machine code. As a consequence, the developer simply writes his application keeping its attention on software aspects, and lets the compiler in charge of code optimization to reach the expected performance.

The physical limitations of current semiconductor technology have made it increasingly difficult to achieve frequency improvements in processors. Thus, hardware designers have organized computers, including embedded systems, into multi-core architecture to achieve the performance required for current applications. This raise of parallel computing has however introduced new challenges to both programmers and compiler developers. On the one hand, the programmers have to describe their application in such a way that the compiler is able to decompose it in parallel computations. On the other hand, the compiler has to translate the application to machine code that can exploit all the parallel ability of the executing platform.

2.1 PARALLELISM IS EVERYWHERE

The execution of an algorithm onto multiple processing units involves parallelism which can be defined as the *decomposition* of the computation into multiple pieces that can be executed in parallel [169].

Let us differentiate the *granularity* of the decomposition (fine-grain and coarse-grain) from the *form* of the decomposition (task, data and pipeline parallelisms). On the one hand, the granularity of the parallelism describes the amount of computation in relation to communication [156]. During this thesis, we distinguish only fine and coarse granularities:

- **Fine-grain parallelism** refers to *instruction-level parallelism* that has been typically exploited by Very Long Instruction Word (VLIW) and super-scalar processors.
- **Coarse-grain parallelism** refers to *task-level parallelism*, also called *thread-level parallelism*, that has been typically exploited by multi-processor platforms.

On the other hand, the form of the parallelism describes the decomposition of the computation into multiple chunks of smaller entities:

- **Data parallelism**, requiring *data decomposition*, is the simultaneous execution of the same computation across different datasets.
- **Task parallelism**, requiring *functional decomposition*, is the simultaneous execution of different computations across the same or different datasets.
- **Pipeline parallelism**, requiring *temporal decomposition*, is the simultaneous execution of multiple stages of a given computation. To achieve pipelining, the computation must be applied repeatedly and must be subdivided into a sequence of subtasks. The subtasks can then be performed in an overlapped fashion.

Combining all forms and granularities in *multi-level parallelism* (Table 1), let us fully exploit the potential parallelism of the application. The challenge is then to leverage the multi-level parallelism according to the processing capabilities of the platform.

	Fine-grain *Instruction, Loop*	**Coarse-grain** *Procedure, Subprogram*
Task	VLIW instruction	Concurrent thread
Data	Vectorization	Kernel function
Pipeline	Software pipelining	Thread pipelining

Table 1: Parallelism is multi-form and multi-level

2.2 EMBEDDED PARALLEL PLATFORMS

Computer systems have traditionally been characterized by their Instruction Set Architecture (ISA). The ISA provides the visible interface to program the processor: the operations available, the number of operands necessary for a given operation, as well as the type and the size of the operands, and the storage locations that are available. Computer systems are then categorized depending on the complexity of their instruction sets, known as Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC). The development of specialized processors such as Application-Specific Instruction-Set Processor (ASIP) and Digital Signal Processor (DSP), and the emergence of heterogeneous and hybrid systems make this classification outdated.

The most popular taxonomy of parallel systems has been introduced by Flynn in 1972 [77]. Flynn classifies computer organization according to the multiplicity of the hardware resources provided to the instruction and data streams :

Single Instruction Single Data streams (SISD) computer organization represents traditional serial processors. Instructions are executed sequentially but may be overlapped in their execution stages.

Single Instruction Multiple Data streams (SIMD) computer organization represents most Graphics Processing Units (GPUs) available today. Multiple processing elements execute simultaneously the same instruction but operate on different data sets from distinct data streams.

Multiple Instruction Single Data streams (MISD) computer organization represents systolic arrays. Multiple processing units execute distinct instructions over the same data stream and its derivatives. The results of one processing unit become the input of the next processing unit in the array.

Multiple Instruction Multiple Data streams (MIMD) computer organization represents multi-core platforms. Multiple processing elements execute distinct instructions which are operating over distinct data streams.

Obviously, the last three classes of computer organization are the classes of parallel computers.

2.2.1 *Homogeneous versus Heterogeneous*

Knowing that multi-core platforms are by definition composed of multiple cores, they can be classified as *homogeneous* or *heterogeneous* (Figure 4):

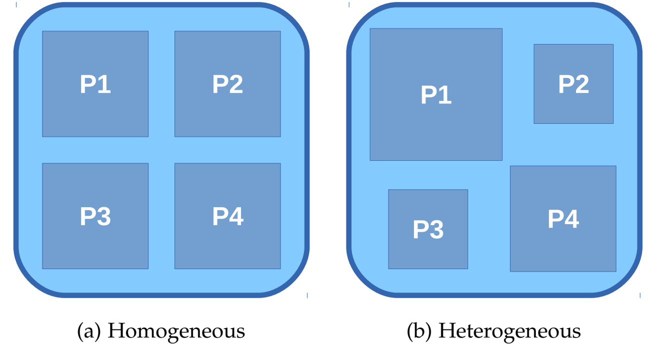
- **Homogeneous platforms** are composed of identical cores (Figure 4a). For instance, Texas Instruments commercializes a set of homogeneous multi-core platforms (excluding hardware accelerators), known as KeyStone TMS320C667x family, that targets a large range of application domains including machine vision and software defined radio.

- **Heterogeneous platforms** are composed of different types of cores (Figure 4b) that make their programming more challenging. The heterogeneity can however refer to several aspects of the processor cores (frequency, ISA, etc). For instance, the big.LITTLE MP is an heterogeneous architecture developed by ARM coupling a low power Cortex-A7 processor with a more powerful Cortex-A15 processor, both sharing the same ISA [92]. Nvidia commercializes another type of heterogeneous Multi-Processor System-on-Chips (MPSoCs), known as the Tegra's family, that are composed of an ARM processor and a Geforce GPU.

In fact, most of modern MPSoCs are heterogeneous and embed in the same chip General-Purpose Processors (GPPs), DSPs and hardware accelerators, but homogeneous organizations can still be present locally. As an example, the STHORM multi-core platform developed by STMicroelectronics is composed of homogeneous clusters of processors [22].

Heterogeneity is obviously the solution to handle all forms and levels of parallelism that are present inside the algorithms. Heterogeneity makes however the implementation process of applications exponentially harder by adding new constraints to the application mapping and scheduling, and by requiring multiple compilation flows.

Multi-core platforms are not only characterized by the architecture of their processor cores but also by their memory architecture.



(a) Homogeneous (b) Heterogeneous

Figure 4: Processor architectures characterizing multi-core platforms

2.2.2 *Memory Architecture*

Multi-core systems are traditionally based on shared-memory architectures, i.e. multiple processors fully connected to one or multiple memory modules through an interconnection network with no access restriction. As the number of processors grows (many-core processors already comprise hundreds of cores), it quickly becomes impossible for a centralized memory system to meet the bandwidth needs of the processors. Then, distributed memory system becomes a necessity [147].

Memory architectures are usually classified as follow [185]:

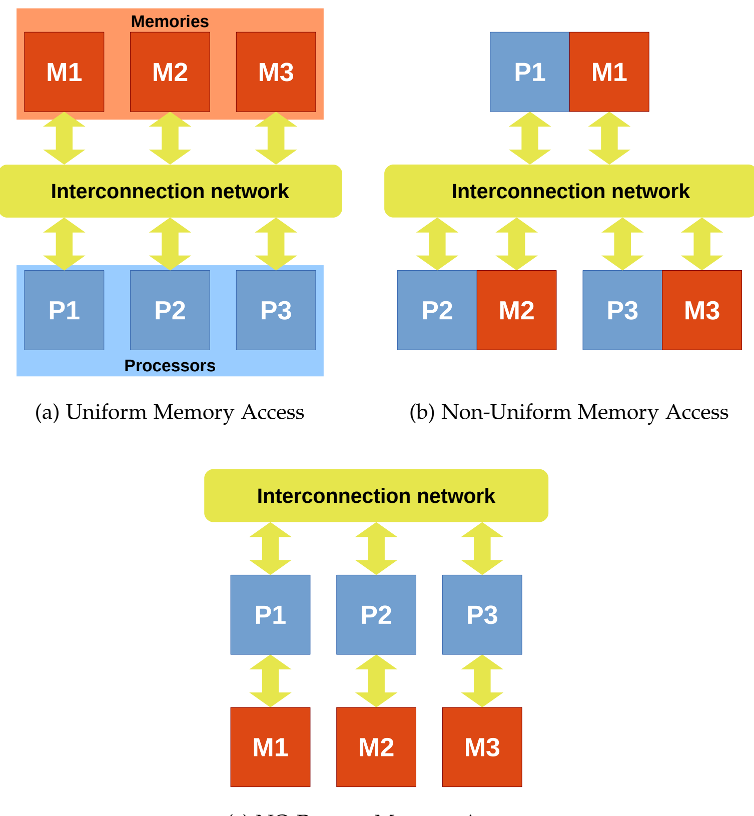
- **Uniform Memory Access (UMA):** The memory is shared and can be accessed uniformly by all processors (Figure 5a), in other words the processors have an equal access and access times to any memory location. In UMA, all the processors are tightly-coupled with the memory components. Such a centralized memory architecture is commonly used in GPP such as the ARM Cortex A-15.

- **Non-Uniform Memory Access (NUMA):** The memory is shared and can be similarly accessed by all processors but not with an equal access time (Figure 5b). In NUMA, processors and memory components are clustered making the memory accesses across the clusters slower but still possible. For instance, in Figure 5b the processor P1 can access both memories M1 and M2, but accessing M1 will be much faster than accessing M2 because P1 and M1 are tightly-coupled.

- **NO Remote Memory Access (NORMA):** The memory is not shared by the processors, i.e. distributed memory, but the processors can communicate directly through the interconnection network by the way of a dedicated communication protocol (Figure 5c). IBM's Cell processors are popular NORMA-based commercial platforms [109].

The memory architecture of multi-core platforms impacts directly the programming of processors. Therefore, the programming of shared-memory platforms and distributed-memory platforms is usually very different.

On the one hand, the programming of shared-memory systems stays closely similar to the programming of uniprocessor systems, which makes it easier to be handle by the developers. In fact, communication among processors is simply achieved by writing to and reading memory locations since all processors are allowed to see data written by any processor. But, multiple



(a) Uniform Memory Access

(b) Non-Uniform Memory Access

(c) NO Remote Memory Access

Figure 5: Memory architectures characterizing multi-core platforms

processors may access the same memory location concurrently and cause unexpected behaviors, usually called *race conditions*. Thus, the synchronization among processors is required and usually achieved using locks and barriers.

On the other hand, the programming of distributed-memory systems requires additional interfaces to handle communication among processors. Since each processor has its own memory system, a processor must communicate explicitly with the other processors to exchange data using dedicated methods such as message passing. Thus, programming distributed-memory systems is not as natural as shared-memory systems. For instance, IBM's Cell processors suffer from the complexity of their programming.

2.2.3 *Memory Hierarchy*

Similarly to uniprocessor, designing multi-core platform on top of hierarchical memory organization (Figure 6), brings several advantages:

- Reduction of the latency of memory accesses by replicating data in small and fast memory components.
- Reduction of the demands of external memory bandwidth by exploiting *spatial locality* and *temporal locality* of data accesses within algorithms.

As we have seen, shared-memory systems do not scale well when the number of processors increases: the interconnection network connecting the

processors to the shared memory becomes a bottleneck when too many processors are trying to access it simultaneously. Therefore, the introduction of a private level to the memory hierarchy limits the contention on the interconnection network. But, the memory being shared, the replication of data at private-level introduces a problem of consistency between the memory components, when a data is replicated and goes out of synchronization with the original values where the modifications have taken place. Consequently, the systems have to ensure the coherency between the memory components.

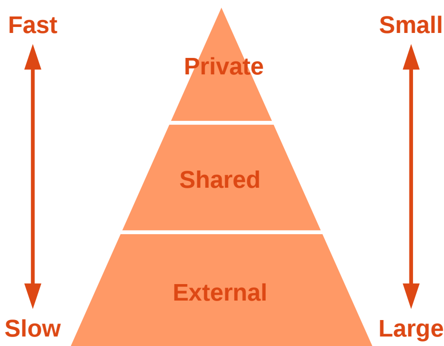


Figure 6: Hierarchical memory organization for multi-core platforms

In embedded systems, two types of hierarchical memory, known as cache and scrachpad memories, are dominating the market (Figure 7):

CACHE Data replication is controlled by the hardware that automatically loads the data when needed, providing transparent memory accesses to the developers but implying unpredictable access time [161] (Figure 7a). In fact, cache mechanism has been extensively used and studied in general-purpose computing.

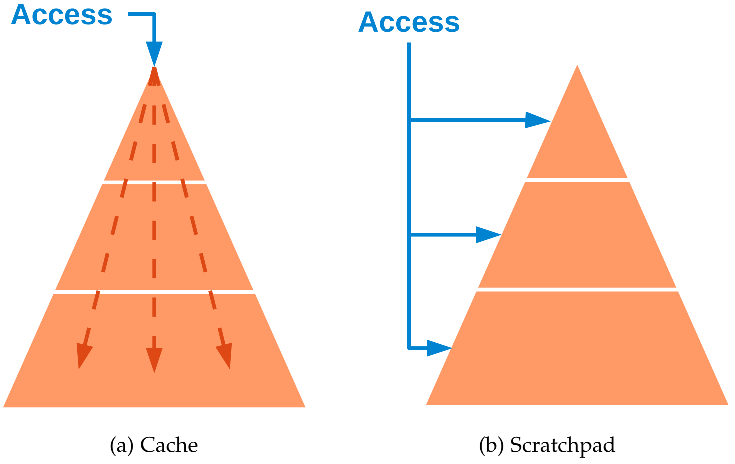
SCRATCHPAD Data replication is controlled by the software, done explicitly by the developer or added automatically by the compiler [19, 18] (Figure 7b). In fact, predictability is the key attribute of scratchpad memories [182]. All data movement being solved at compile-time, scratchpad memories have a smaller power consumption than caches.

Even if cache and scratchpad memories have different approaches to solve the *memory wall*, they may be mixed in the same system to benefit from all aspects.

2.2.4 *On-Chip Interconnection Network*

The interconnection network is responsible for the exchange of information between the components of the multi-core platform. Consequently, the global performance of the multi-core platform is directly affected by the efficiency of its interconnection network. Two types of exchange are possible:

- Simple accesses from processors to memories.
- Direct communication from a processor to another, to provide message passing facility.



(a) Cache (b) Scratchpad

Figure 7: Hardware versus software implementation of hierarchical memories

Two types of interconnection network are dominating for intra-chip communication, bus and Network-on-Chip (NoC). We sum up their properties in Table 2.

BUSES Bus interconnect has been the dominant architecture for intra-chip communications, but its lack of scalability makes it clearly not suitable for massively parallel systems. One the one hand, bus-based systems provide high-speed communication and large communication bandwitch while keeping low-cost implementation and easy testability. On the other hand, it suffers from a number of drawbacks such as a poor scalability, no fault tolerance, and a limited parallelism exploitation. Actually, even a well-designed bus-based system may suffer from data congestions that limit the global performance of the system. It is also not inherently scalable. As more and more modules are added to a bus, not only data congestion increases, but power consumption also rises due to the increased load presented to the bus driver circuits.

NETWORK-ON-CHIP NoC is another interconnection scheme that has the potential to overcome the limitations of bus-based interconnections. In a NoC-based system, the components are connected via a multistage switching system composed of point-to-point links. Such a multistage interconnection avoids the scaling issues of long wires. Since several paths may be available in the network, the switches dynamically route the data according to the communication traffic, similarly to networking theory. Consequently, the scalability of the interconnection network comes at the expense of the variability in memory access latencies (i. e.NUMA) as well as increased design complexity to guarantee correctness and fairness and to avoid deadlocks and starvations.

2.3 PARALLEL PROGRAMMING MODELS

Programming languages are historically sequential, making the programming of modern embedded platform a challenge. In their survey about MPSoC design methods and tools [138, 137], Park et al. classify the exist-

	Pros	Cons
Bus	High-speed Low-cost Large bandwidth	Scalability
NoC	High-speed Scalability Power efficiency	Design complexity Variable latency

Table 2: Comparison of interconnection networks

ing programming approaches of MPSoC based on the initial specification of the applications as follows:

- Compiler-based approaches that use compilation techniques to automatically extract the parallelism from the sequential description of an application,
- Language-extension approaches that ask the programmer to provide additional parallelism information into the description of the application using directives or Application Programming Interface (API) in order to help the compiler during the parallelizing stage,
- Model-based approaches that rely on Model of Computations (MoCs) to describe an application to abstract the parallel description of the application, and
- Platform-based approaches, an additional category introduced by Park as a model-based approach that embeds parallel information and architectural constraints.

Park's taxonomy clearly shows the challenge of programming multi-core platforms by exposing the variety of studies about MPSoCs programming but it suffers from few aspects. First, his classification does not highlight the tight link between the platform architecture and its programming. Then, the last category only is a bit artificial since it only includes his own work. As a consequence, we organized our overview of parallel programming approaches for embedded systems as follows: General-purpose programming, assisted programming, and high-level programming.

2.3.1 *General-Purpose Parallel Programming*

Programming languages are designed to create programs that control computers on which they are running. Knowing that, most general-purpose programming models directly inherit from an underlying processor architecture. Like imperative programming inherits from the Von Newman architecture, both ensure a sequential execution of the instructions according to the control flow. Similarly, parallel programming models inherit from the architecture of parallel platforms. For instance, multi-threading abstracts shared-memory architecture, message-passing abstracts distributed-memory, and stream processing abstracts GPU-like architecture.

MULTI-THREADING Definitely the most widely used parallel programming model, that is to say multi-threading, is based on the parallel execu-

tion of tasks, called threads, that share a single address-space. The shared memory makes multi-threading ideal to program platforms based on shared memory architecture. But, in the thread model, the programmer has to handle every low-level details, from the creation of the threads to their synchronizations. In fact, multi-threading requires synchronization primitives, such as semaphores, to handle concurrent memory accesses.

A well-know implementation of the thread model is the POSIX threading library *pthread*.

MESSAGE PASSING Another well-known parallel programming model is message passing in which the developer describes a set of concurrent processes that exchange data by sending and receiving messages. In contrast with multi-threading, message-passing systems have been introduced to program platforms based on distributed memory architecture. In message-passing systems, communications may be point-to-point, collective, synchronous, or asynchronous. Synchronous communications, also called *rendezvous*, require sender and receiver to wait for each other in order to transfer a message, synchronizing the processes and removing the need for buffering mechanism. Asynchronous communication simply delivers a message from the sender to the receiver without any assumption on the availability of the receiver.

Message Passing Interface (MPI) is a standardized message-passing system resulting of a joint effort of academician and industrial researchers [79]. While MPI has been heavily used in High Performance Computing (HPC), the emergence of NoC-based interconnection network has revived the interest of MPI for programming embedded multi-core platforms [75, 108].

STREAM PROCESSING More recently, the development of General-Purpose Processing on GPUs (GPGPU) has revived this interest for an SIMD-related programming model usually called stream processing. Instead of describing task parallelism, this model focuses on computations, usually called *kernels*, that can be applied simultaneously on a large panel of data. Data-parallel programming is usually built on top of an abstract architectural model composed of one *host* that controls the execution and a set of *devices* in which the kernels are executed. Because the kernels usually operate on huge amount of data, the developer has to manage precisely the data transfers between host and devices.

OpenCL and CUDA are well-know GPGPU-based implementations of stream processing principle. OpenCL is developed by Khronos as a standard for heterogeneous computing while CUDA is provided by Nvidia in order to give GPGPU facilities to their products. Brook [40] is another programming language, developed at Standford university, that inherits directly from stream processing.

2.3.2 *Assisted Parallel Programing*

General-purpose parallel programming is clearly error-prone and usually requires from the developer a real expertise about parallel computing to take care of low-level details. A large portion of applications is already written in a sequential way, and knowing that modifying an existing application is a well-known source of bugs, many efforts have been made to provide assisted parallelization methodologies.

AUTOMATIC PARALLELIZATION The ultimate goal, the *Holy Graal* of parallel computing, is the automated extraction of parallelism from the sequential description of the application. Early parallelization studies have focused on the extraction of instruction-level parallelism for VLIW processors. More recently, the extraction of fine-grain data parallelism, so-called *vectorization*, has been studied to take advantage of SIMD instructions such as Intel's MMX and ARM's NEON for general-purpose processors. However, embedded platforms require also a coarser decomposition to really benefit from multi-core architecture.

Most of coarse-grain parallelization techniques start by analyzing the program statically, to identify time-consuming portions and dependences, then try to decompose it in parallel tasks using heuristics. In order to overcome static analysis limitations, some studies propose to identify hidden parallelism, then expose it with the help of the developer [48]. Others studies used optimization techniques such as machine learning [47], integer linear programming [53, 54] or genetic algorithms [52] but they require execution profiling and may be time-consuming.

In fact, the efficiency of automatic parallelization remains limited on irregular applications. Generally speaking, automatic parallelization approaches suffer from a lack of knowledge about the application domain. Consequently, restricting general-purpose languages, or else extending the program with additional information, may improve the parallelization by making easier the analysis.

DIRECTIVE-BASED PROGRAMMING Directive-based programming aims at providing a rapid and easy-going parallelization methodology to the developer. Traditional programming languages are extended with a set of directives (such as *pragma* in C) that informs the compiler about the parallelism potential of certain portions of the program, usually loops but also parallel sections and even pipeline sections [122, 145].

```
1 #pragma omp parallel for
2 for (i = 0; i < N; i++) {
3   a[i] = 2 * i;
4 }
```

Listing 1: One simple directive to parallelize a loop

OpenMP [57] is a standardized language extension (Listing 1) which relies on multi-threading. OpenMP is probably the most popular directive-based programming interface. OpenACC [133] is an emerging standard that aims to make easier GPGPU, similarly to HMPP [62].

2.3.3 *High-level Parallel Programming*

Embedded computing usually relies on programming techniques that have been developed for general-purpose processors, but the specificities of embedded systems make their programming more challenging. Embedded systems have to meet memory footprint constraints, real-time performance constraints, as well as power consumption constraints. General-purpose parallel programming is error-prone and assisted parallelization of sequential programs is limited by nature. Thus, higher-level programming approaches aim to providing useful abstractions and optimization tools to efficiently program embedded systems.

ALGORITHMIC SKELETONS Algorithmic skeletons are a high-level programming model that takes advantage of common programming patterns to hide the complexity of parallel programming [51]. Similarly to software design patterns, algorithmic skeletons define general programming solutions to solve common occurring problems. But, while software design patterns focus on software design, algorithmic skeletons provide parallel resolution of computational problems.

Algorithmic skeletons aim to be an effective, generic and high-level approach for parallel programming. To do so, a skeleton is both composed of a generic interface that abstracts parallel computations, communications, or interactions, and of an implementation that handles all low-level details related to the parallelization. The synchronization between the parallel computations is hidden by the implementation of the skeleton, reducing the risk of errors and making the code more portable.

Dozens of patterns have been proposed since the introduction of algorithmic skeletons as a solution for parallel programming. These patterns can be classified in three families [85]:

- Data-based patterns: Map, Fork, Reduce.
- Task-based patterns: Pipe, For, While, If.
- Resolution-based patterns: Divide & Conquer, Branch & Bound.

Obviously, algorithmic skeletons are not the ultimate solution of parallel programming but provide a set of best practices that have been proven useful.

DATAFLOW PROGRAMMING Dataflow programming is another high-level programming method that describes parallel applications inherently [58]. A dataflow program is described as a directed graph composed of a set of computational units interconnected by communication channels. Dataflow programming being also graphical, it is a very natural way for describing parallel algorithms. And, contrary to general-purpose parallel programming, dataflow programming is not built upon an underlying computer architecture.

In fact, dataflow programming is related to message passing in the sense that it describes a set of concurrent processes interacting by explicit communications. But, contrary to message passing, dataflow programming is build upon a strict formalism that provides useful abstractions and enables advanced analysis. For instance, the developer has to describe explicitly the communication network, so two processes cannot communicate directly if they are not linked by a communication channel. Dataflow programming being the main subject of Chapter 3, no more details are given here.

DOMAIN-SPECIFIC LANGUAGE As defined by Van Deursen et al. [61], a Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. Thus, DSLs can provide knowledge about the application domain that is lacking in general-purpose languages, in order to allow automatic parallelization for example. Additionally, DSLs intend to be written by domain experts and not by experts in parallel programming.

For example, Streamit [168] and Halide [148] are DSL that inherit directly from stream processing. In contrast with general-purpose languages such as

OpenCL and CUDA, Streamit and Halide target especially signal processing applications and intend to simplify their development by providing a higher abstraction of the underlying architecture.

2.4 MAPPING AND SCHEDULING

In contrast with traditional compilers that stay focused on instruction scheduling, MPSoC-based compilation flows have to take advantage of the parallel processing capabilities of multi-core platforms to satisfy the performance constraints of the applications. Let us assume that the application has been previously decomposed into small tasks using automatic, assisted or manual techniques. Then, the tasks composing the application are mapped and scheduled onto the executing platform, two distinct processes that are intimately related:

- The **mapping** is the process of assigning a processor to a task, in other words *where* the task is executed, that involves the *partitioning* of the pool of tasks.
- The **scheduling** is the process of deciding the moment of the execution of a task, in other words *when* the task is executed, that involves the *ordering* of the tasks and the *timing* of their execution.

The mapping and scheduling problem is a NP-hard problem, equivalent to *Quadratic Assignment Problem* [81]. Thus, many heuristics aim to find a nearly optimal solution satisfying all the given constraints. The heuristics depend on both the application description, usually represented as a *task graph*, and the platform specification, based on an *architecture model* such as S-LAM [140] or AADL [74], as well as the architecture model used in Syndex [91]. Due to the inherent constraints of embedded systems, mapping and scheduling heuristics may target power consumption and reliability, additionally to execution performance.

As presented in Table 3, Lee and Ha have classified the different approaches according to the moment, **run-time** or **compile-time**, when the mapping, ordering and timing stages are performed [118]. On the one hand, fully dynamic approaches can efficiently balance the workload of any application over the processors by making all decisions at run-time, but they come at the cost of a larger execution overhead. On the other hand, fully static approaches determine all decisions at compile-time to minimize the execution overhead, but they are only suitable for a subset of applications that do not have dynamic behavior. Static-assignment approaches partition the tasks over the processors at compile time to reduce the execution overhead at the cost of the load-balancing, and self-timed approaches additionally determine the ordering of the task execution at compile-time without notion of the time. Additionally, compile-time approaches bring guarantees on the obtained schedule which is a central benefit for safety critical systems. Indeed, proving real-time constraints on dynamic scheduling is difficult.

Similarly, Singh et al. introduce additional criteria to characterized scheduling and mapping methodologies [155]:

- Mapping methodologies can target either **homogeneous** platforms, composed of identical cores, or **heterogeneous** platforms, composed of different types of cores, as introduced previously by Section 2.2.1. The heterogeneity of multi-core platforms imposes additional parameters to mapping heuristics.

	Mapping	Ordering	Timing
Fully dynamic	run	run	run
Static-assignment	compile	run	run
Self-timed	compile	compile	run
Fully static	compile	compile	compile

Table 3: Dynamism-based taxonomy of mapping and scheduling approaches, run-time or compile-time, from Lee and Ha's work [118]

- Mapping and scheduling methodologies performed at run-time require a management system responsible for taking the decisions. This system may be **centralized** and executed by in a single processor, dedicated or not, as well as **distributed** all over the platform.

Task mapping and scheduling have been extensively studied in both communities of general-purpose computing and embedded computing. A good overview of these techniques for embedded systems has been made by Singh et al. [155], and, as they say, the application mapping problem is still one of the most urgent problem to be solved for implementing embedded systems.

2.5 conclusion

All along this chapter, we have presented the variety of solutions for programming parallel platforms that clearly demonstrates its complexity. While expressing parallelism within an algorithm is already challenging, the architectural variability of parallel platforms in terms of component organization and interconnection makes the task infinitely harder.

While general-purpose computing stays often attached to conventional programming schemes, embedded computing must move towards higher-level programming approaches, like dataflow models and DSL, that provide the abstraction necessary to reach the efficiency and reliability that are required by embedded systems. While MoCs underlying dataflow programming provide formalism, DSLs provides domain knowledge by restricting the application description.

The next chapter deeply inspects dataflow programming, and we focus especially on practical aspects that enable the development of real-world applications. Then, Chapter 4 introduces the Reconfigurable Video Coding (RVC) framework, an innovating framework dedicated to the development of video coding tools that is built upon a dataflow-based DSL known as CAL Actor Language (CAL). After that, the last chapters describe the contributions of this thesis, that is to say the programming of embedded multi-core platforms thanks to the RVC framework.

3 DATAFLOW PROGRAMMING

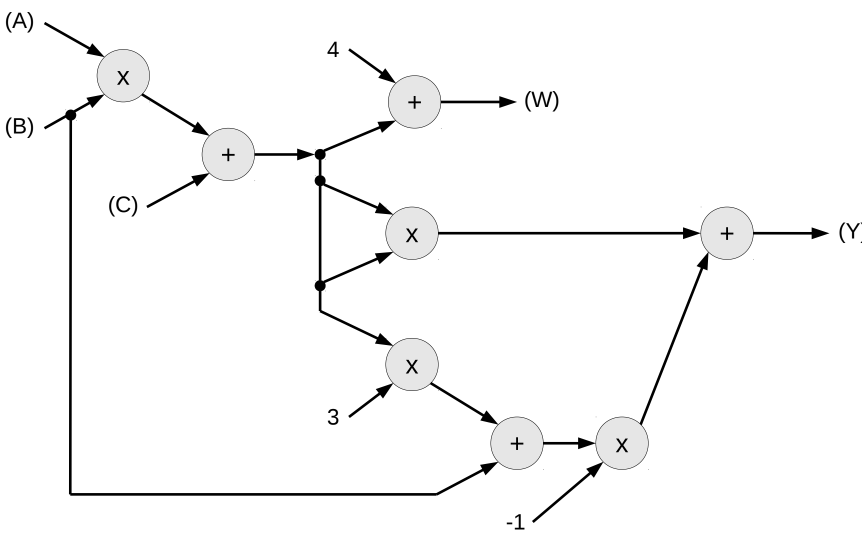
Πάντα ῥεῖ (Panta rhei), "everything flows".

— Heraclitus

The concept of dataflow representation was introduced by Sutherland in 1966 as a visual way to describe an arithmetic computation [165]. Sutherland represents a sequence of arithmetic statements as a dataflow graph (Figure 8b), in contrast with its mathematical form (Figure 8a), to demonstrate the advantages of the graphical form. In fact, the graphical form replaces each temporary variable by a simple edge that symbolizes the dependences between the computations. Inversely, the graphical form emphasizes the input and output variables.

$$
\begin{aligned}
Z &= A \times B + C \\
W &= Z + 4 \\
Y &= Z^2 - (3Z + B)
\end{aligned}
$$

(a) Written statement



(b) Graphical statement

Figure 8: The first dataflow representation, the graphical representation of an arithmetic computation, that was introduced by Sutherland in 1966 [165]

Later in 1974, Dennis has described formally the first dataflow programming language [58]. In this language, a program is modeled as a directed graph where the edges represent the flow of data and the nodes describe control and computation.

Following his work on a dataflow language, Dennis has introduced a novel hardware architecture on top of the dataflow model, known under

the name of *static dataflow architecture* [59]. This architecture differs from the traditional *Von Neumann* architecture by making an instruction executable when all its inputs are available. As a consequence, the instruction-level parallelism can be directly exploited by the processor. Similarly, Watson et al. have introduced in 1979 the tagged-token dataflow architecture [174] to overcome the limitation of static dataflow architecture. But, the too fine granularity of dataflow architectures prevents them from obtaining scaled performance on large programs.

3.1 DEFINITION OF A DATAFLOW PROGRAM

Thus, a dataflow program is defined as a directed graph (Figure 9) composed of a set of computational units interconnected by communication channels through ports:

- The communication corresponds to a stream of atomic data objects, called *tokens*, that follows the First-In-First-Out (FIFO) strategy.
- The computational units, usually called *processes* or *actors*, may first read some tokens from their input channels, may then process some internal computations, and may finally write some tokens to their output channels.

Conceptually, dataflow programming can be considered as the association of the component-oriented programming with message-passing communication.

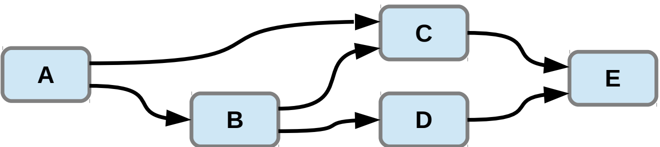


Figure 9: A dataflow network of five processes, the vertices named from A to E, that communicate through a set of communication channels, represented by the directed edges

3.2 DATAFLOW PARADIGM TO ENHANCE PROGRAMMING

Since the early studies on dataflow paradigm, dataflow programming has been mainly considered as an interesting approach for two domains of application, signal processing and parallel processing [107, 159].

During the last twenty years, dataflow programming has been heavily used for the development of signal processing applications due to its consistency with the natural representation of the digital signals processing. More particularly, dataflow programming gives the opportunity to use visual programming so as to describe the interconnection between its components. Such a graphical approach is very natural and makes it more easily understandable by programmers who can focus on *how things connect*.

The emergence of massively parallel architectures, along with the difficulties to program these architecture, make dataflow paradigm an alternative to the imperative paradigm thanks to its ability to express concurrency without complex synchronization mechanism. The internal representation of the application is a network of processing blocks that only communicate

through the communication channels. Consequently, the blocks are independent and do not produce any side-effect: This removes the potential concurrency issues that arise when the programmer is asked to manually manage the synchronization between the parallel computations.

3.2.1 *Modular Programming*

The decomposition of the program into *black boxes* enables the separation of concerns, improves **maintainability** by enforcing the encapsulation of the components, and makes the application description modular:

- **Hierarchical:** A component of the network may represent another network, such as the component B in Figure 10. The ability of an application to be specified in a hierarchical way is allowed by the strict separation between the modeling of the interconnection network and the behavior of the components.
- **Reusable:** A single component can be used to specify several applications, or can be used several time in the network which specifies the application, such as the components A and C in Figure 10 that are both reused by the subnetwork. This ability for a coded algorithm to be reused, is simplified by the strong encapsulation of the components in dataflow programming that makes them side-effect free. A feature also found in functional programming that highlights the strong links between functional and dataflow programming.
- **Reconfigurable:** A component can easily be replaced by another one while its interfaces (input and output ports) are strictly identical, such as the components D and G in Figure 10. Thus, the reconfigurability, i.e. the ability to switch the system into a new configuration in a timely and cost effective manner, is simplified by dataflow modeling.

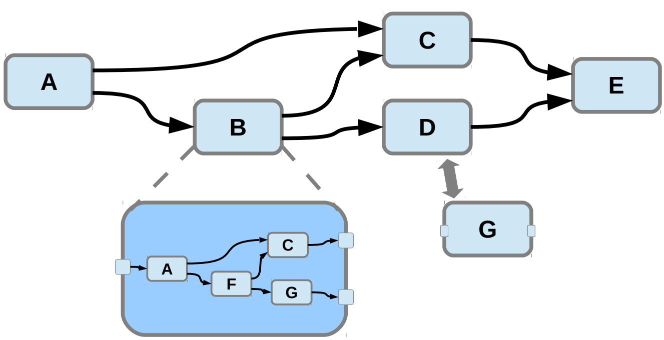


Figure 10: The dataflow representation is modular by offering hierarchical ability, re-usability and reconfigurability.

3.2.2 *Parallel Programming*

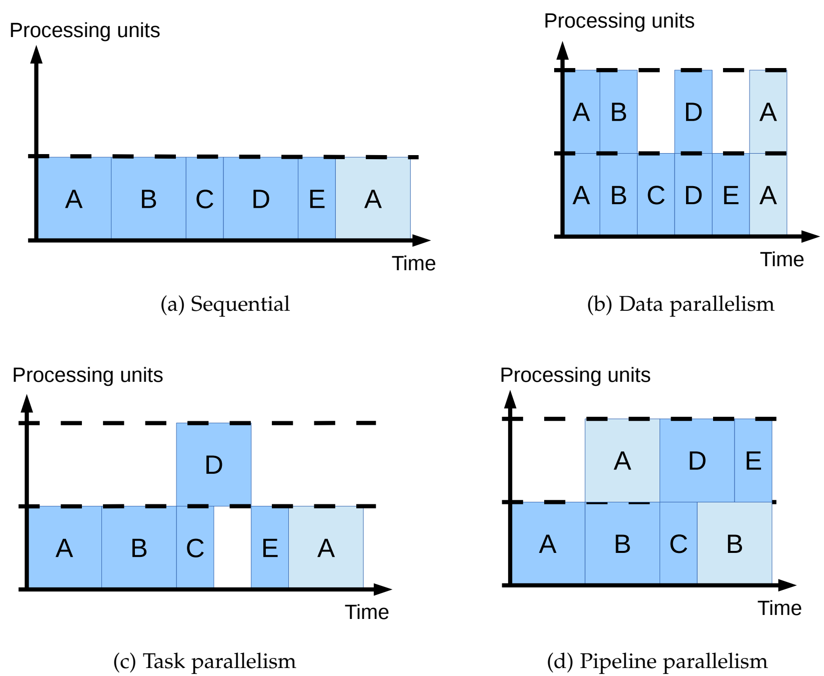
A dataflow program states an abundance of parallelisms thanks to the explicit exposition of the concurrency. In its structural view, the dataflow model presents three potential degrees of parallelism (task, data and pipeline) that can be applied to different granularities of description (Figure 11).

1. **Task parallelism** refers to the potential parallelism between the independent parts of an application. In a dataflow context, it appears when two or more components do not have any dependency constraints (Figure 11c).

2. **Data parallelism** refers to a unique computation performed on different sets of data. It can be applied by duplicating a given component when it processes successively several sets of data with no dependencies between them (Figure 11b).

3. **Pipeline parallelism** 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. This parallelism is inherent to streaming execution model in case of a chain of components (Figure 11d). Pipelining does not enhance the throughput on one calculation, but the processing of a sequence of calculation.

4. **Coarse-grain and fine-grain parallelisms** refer to the granularity of the decomposition of the application's algorithms into components, i.e. the ratio of computation to the amount of communication in a given component [156]. A fine-grain description is composed of small and atomic components that frequently exchange data, for instance a unique arithmetic operation such as presented in Figure 8. Conversely, a coarse-grain description is composed of larger components, which perform intensive computations and exchange a large amount of data in each firing. Consequently, a fine-grain description states higher degree of parallelism thanks to a higher number of actors, but has also a cost tied to the communication synchronization between actors.



(a) Sequential

(b) Data parallelism

(c) Task parallelism

(d) Pipeline parallelism

Figure 11: Parallelizing the dataflow program presented in Figure 9 from a sequential execution (11a) to parallel execution using different strategies (11b, 11d, 11c).

Additionally, these kinds of parallelism as well as the instruction-level parallelism, i.e. the potential overlap among instructions, can be potentially extracted from the internal algorithm of the components such as any procedural language.

3.3 MODEL OF COMPUTATION

A MoC is an abstract specification of how a computation can progress. A MoC is useful to define the semantics of a programming model, i.e. the type of components it can contain and the way they interact [152]. Classical examples of MoC are the Turing machine and Lambda calculus models. During the last twenty years, dozens of dataflow MoCs were studied due to the attractive use of dataflow programming for the development of signal processing applications.

Existing dataflow MoCs can be split into two main classes: The *static* ones allow a predictable behavior such as the scheduling can be done at compile time. The *dynamic* ones have a data-dependent behavior. Most of the studies on dataflow programming focus on the statically schedulable MoC because of the efficiency of synthesis techniques on such models due to their analyzability. Unfortunately, they do not take into consideration the flexibility and the expressiveness offered to the programmers by the dynamic dataflow MoC.

3.3.1 *Kahn Process Network*

A Kahn Process Network (KPN) [110] is represented as a graph $G = (V, E)$ such that V is a set of vertices modeling computational units that are called processes and E is a set of unidirectional edges representing unbounded communication channels based on FIFO principle. The behavior of this MoC can be described using the *denotational semantic* introduced by Kahn [110].

A FIFO channel $e \in E$ can be empty, deinputnoted as $\perp$, or can carry a possibly infinite sequence of tokens $X = [x_1, x_2, ...]$, where each x_i is an atomic data called a token. A sequence X that precedes a sequence Y, e.g. $X = [x_1, x_2]$ and $Y = [x_1, x_2, x_3]$, is denoted $X \sqsubseteq Y$. The set of all possible sequences is denoted S, while S^p is the set of p-tuples of sequences on the p FIFO channels of a process. In other words, $[X_1, X_2, ..., X_p] \in S^p$ represents the sequence consumed/produced by a process. The length of a sequence is given by $|X|$.

A KPN with m input ports and n output ports is a continuous and monotonic function denoted as:

$$F : S^m \rightarrow S^n \tag{1}$$

A process is triggered when the given sequences of tokens S^m appears on its input ports; it is activated iteratively as long as S^m exists. Conversely, the process is suspended when S^m does not exist on its input ports. In other terms, reading from a FIFO channel can be blocking for one process until S^m appears again.

The blocking reads ensure that every program following this model of concurrency is deterministic. However, it also implies to backup the current context of the blocked process before executing the next one when implementing KPN-based programs in a sequential environment.

3.3.2 *Dataflow Process Network*

Dataflow Process Network (DPN) [121], also known as Dynamic Data-Flow (DDF), is closely related to KPN. The DPN model is Turing-complete which means it can model any algorithm even non-deterministic ones.

In this model, an application is represented as a graph $G = (V, E)$ within the vertices/processes called *actors*. Additionally to the KPN model, it introduces the notion of firing. An actor firing, or action, is an indivisible quantum of computation which corresponds to a mapping function of input tokens to output tokens applied repeatedly and sequentially on one or more data streams. This mapping is composed of three ordered and indivisible steps: data reading, then computational procedure, and finally data writing. These functions are guarded by a set of firing rules R which specifies when an actor can be fired, i.e. the number and the values of tokens that have to be available on the input ports to fire the actor.

More formally, firings can be described using the *denotational semantic* extended by Dennis [58]. Every actor $a \in V$ is associated with its own set of firing function F_a, and firing rules R_a such that:

$$F_a = [f_1, f_2, ..., f_M] \tag{2}$$

$$R_a = [\mathbf{R}_1, \mathbf{R}_2, ..., \mathbf{R}_N] \tag{3}$$

Within each function $f_i \in F_a$ is associated to a given firing rule $R_i \in R_a$.

A firing rule $\mathbf{R}_i$ defines a finite sequence of patterns, one for each input m of the actor such as $\mathbf{R}_i = [P_{i,1}, P_{i,2}, ..., P_{i,m}] \in S^m$. A pattern $P_{i,j}$ is an acceptable sequence of tokens in R_i on one input j from the input m of an actor. It is satisfied if and only if $P_{i,j} \sqsubseteq X_j$ where X_j is the sequence of tokens available on the jth FIFO channel. The pattern $P_{i,j} = \perp$ designates any empty list where any available sequence on input j is acceptable. The pattern $P_{i,j} = [*]$ is acceptable for any sequence *containing at least one token*. The length of a pattern $P_{i,j}$ is denoted $|P_{i,j}|$. We abuse of this notation by using $|R_i|$ to express the consumption rate of the firing rule R_i and $|f_i|$ the production rate of the firing function f_i.

An actor can fire when at least one of its firing rules is satisfied, and, when several firing rules are satisfied at the same time, a single one is chosen and its corresponding firing function is executed. So that, DPN can describe nondeterministic algorithms which is not possible with the KPN model.

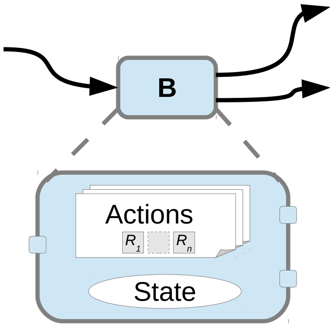


Figure 12: A self-contained actor with its own state, actions and firing rules

The strong encapsulation of the actors is described by Figure 12 that introduces the internal state of an actor. In fact, such an internal state is just

a more convenient representation since it is strictly equivalent to a feedback loop, so it only depends on the ability of the language syntax to describe state variables.

3.3.3 *Static Dataflow Models*

Static dataflow models can be seen as a simplification of the DPN model, in which tokens consumption and production follow a predictable behavior, which means no data-dependent behavior.

SYNCHRONOUS DATAFLOW The Synchronous Data-Flow (SDF) model is a simple static dataflow model, in which an actor consumes and produces a constant number of tokens at each firing. It may have a single firing rule, which is valid for any sequence S^m of a certain size on its inputs [119]. In the case where an actor has several firing rules, an actor is SDF if all its firing rules have the same consumption, which mean for $\mathbf{R}_A \in R$ and $\forall \mathbf{R}_B \in R$:

$$|\mathbf{R}_A| = |\mathbf{R}_B| \tag{4}$$

All the firing functions of an SDF actor must also produce a fixed number of tokens at each firing, which means for $f_a \in F$ and $\forall f_b \in F$:

$$|f_a(s)| = |f_b(s)| \tag{5}$$

for any $s \in S^m$ and $s_b \in S^m$

CYCLO-STATIC DATAFLOW The Cyclo-Static Data-Flow (CSDF) MoC [30] extends SDF actors by allowing the number of tokens produced and consumed to vary cyclically. This variation is modeled with a state in the actor, which returns to its initial value after a defined number of firing.

Several other static models were studied to solve a variety of specific problems. For instance, the Interface-Based Synchronous Dataflow (IBSDF) model [143] is a hierarchical MoC based on SDF that can be analyzed hierarchically in order to extract additional information that can be relevant for the processing.

3.3.4 *Quasi-Static Dataflow Model*

Dataflow modeling is the question of striking the right balance between expressive power and analyzability: On the one hand, synchronous and cyclo-static dataflow limit the algorithms to be modeled as graphs with fixed production and consumption rates for their predictability and their strong properties that allow powerful optimizations to be applied. On the other hand, dynamic dataflow offers a large expressiveness, until Turing-completeness, able to describe complex algorithms with variable and data-dependent communication rate that makes their analyze and optimization ultimately harder.

The need for a trade-off between expressiveness and predictability has brought the definition of so-called "quasi-static" dataflow models. Quasi-static dataflow differs from dynamic dataflow in that there are techniques that statically schedule as many operations as possible so that only data-dependent operations are scheduled at runtime [41, 42, 26, 60, 83, 80, 21].

BOOLEAN DATAFLOW Buck's Boolean Data-Flow (BDF) model [41, 42] extends the SDF model with production/consumption rates that depend on a *control* port with a consumption rate statically fixed at one token by firing. Basically, the rate of a given port p of an actor can be controlled by its associated *control* port C_p, which means that the actor consumes a token from C_p and the value of this token varies the consumption/production rate of p. The fundamental dynamic actors of the BDF model are the *Switch* and *Select* that simply choose one of its two inputs or outputs according to the *control* token. The BDF model has been proven Turing-complete [41] but it implies a restrictive coding style that is not useful for practical cases.

PARAMETERIZED DATAFLOW Parameterized dataflow presented by Bhattacharya et al. [26], as well as the Parameterized and Interfaced Dataflow Meta-Model (PiMM) [60], are both a higher-level approach to model quasi-static behavior by extending the semantic of existing dataflow models using parameters modifiable at runtime. For example, Parameterized synchronous dataflow (PSDF) [26], Parameterized and Interfaced Synchronous Dataflow (πSDF) [60], Schedulable Parametric Dataflow (SPDF) [80] and the Boolean Parametric Data-Flow (BPDF) [21] are all a generalization of the initial SDF model that allows the expression of quasi-static behavior. In fact, PiMM can be considered as an evolution of the parameterized dataflow, that intends to reach a faster propagation of the parameters, a lighter runtime overhead and a more friendly modeling of the application.

SCENARIO AWARE DATAFLOW Scenario Aware Dataflow (SADF), introduced by Theleen et al. [167], is a generalization of the SDF model where the dynamism is modeled by a collection of static behaviors, called *scenarios*. The scenarios may differ in their computation and production/consumption rates, but, are all extended by a probability of occurrence. The switch between the scenarios is made by specific actors, called detectors, that can reconfigure other actors by sending them specific tokens sequentially through control channels. A restricted version, known as Heterochronous Dataflow (HDF) [83] or FSM-SADF [163], increases the analyzability by modeling the dynamism using a state machine.

3.4 COMPARING DATAFLOW MOCS

The dataflow MoCs, presented in Section 3.3, have been designed to solve a wide range of practical issues. In fact, the design of such a formalism involves a trade-off between several properties, from expressiveness to predictability including practical details.

3.4.1 *Characterization of Dataflow MoCs*

Since the MoC is the underlying structure of a programming language, we have to consider formal properties as well as practical properties, that may be hard to formalize. Here is a non-exhaustive list of criteria that characterize dataflow MoCs.

EXPRESSIVENESS The expressiveness of a formalism is defined as its theoretical expressive power, regardless of the ease [73]. For instance, a MoC is said Turing-complete when it can model any calculation without any assumptions about the effort to achieve it.

PRACTICALITY Practicality is defined informally by the *ease to describe* [73]. In fact, practicality differs from formal expressiveness in the sense that it deals more with the idea of expressing a given system *concisely*, *intuitively* and *readily*. As an example, a model, proved to be Turing-complete, can imply a restrictive coding style that is not useful for practical cases.

ANALYZABILITY The analyzability of a formalism deals with the *availability* of automated processes able to analyze its behavior, for instance *termination* and *boundedness*. A high analyzability offers larger degrees of freedom for optimizations. In fact, the analyzability is directly related to the expressiveness: The more a formalism can express, the less it can be analyzed.

EFFICIENCY This criterion is related to the theoretical efficiency of the implementation. In fact, the implementation efficiency can be measured with the metrics usually involved in the evaluation of an algorithm: *speed*, i.e. the execution time, and *space*, i.e. the memory, that are needed by the algorithm to perform a certain number of computations.

3.4.2 *Taxonomy of Dataflow MoCs*

Figure 13 presents a classification system of the dataflow MoCs according to the evaluation of the criteria introduced below, extending the system used by Stuijk et al. to demonstrate the interest of scenario-aware dataflow modeling [163], by separating the practicality of the modeling from the theoretical expressiveness. This separation is made in order to better reflect the usability of the models to describe real-world applications. The taxonomy reflects that the theoretical expressive power is progressively restricted, from DPN towards SDF, while the analysis become more amenable, as well as the efficiency in general. For example, the proof of termination with bounded memory consumption is decidable for SDF and CSDF models but undecidable for BDF, DPN and KPN models [139]. However, the practicality of the modeling does not obey to the same rules.

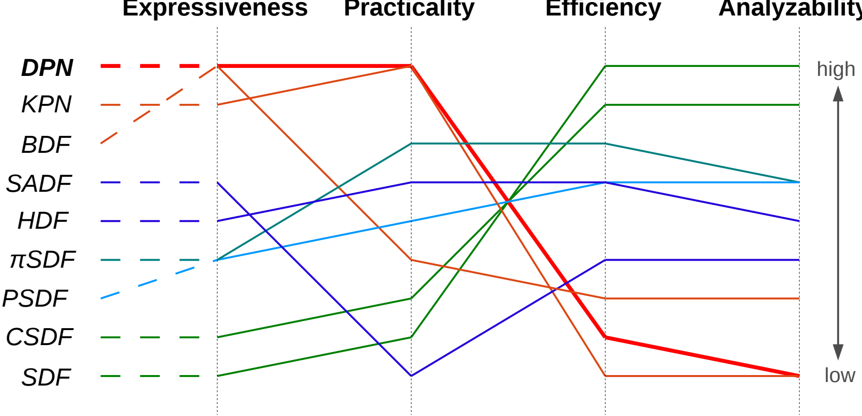


Figure 13: Comparison of dataflow MoCs, extending the classification system introduced by Stuijk et al. [163], which shows that DPN is the most suitable model for a practical programming language

DPN is considered more expressive and efficient than KPN, because it allows non-determinate behavior and does not rely on blocking mechanism. BDF is as expressive as DPN but its restrictive syntax makes it not practical.

SADF, although quite expressive, specifies the execution time of the actors for each scenario, which offers the efficiency but clearly not the practicality. HDF, PSDF and πSDF are intermediate trade-offs between these criteria.

In a practical point-of-view, DPN is well suited to model real-world programs, that become increasingly complex, by offering Turing completeness while also keeping an intuitive description. Yet, the implementation needs to be carefully set up in order to ensure the required efficiency. One way to reach such a goal is the development of advanced analysis that could bridge the gap between practical development and efficient implementation.

3.5 DYNAMIC MODELING REQUIRES DYNAMIC ANALYSIS

In the last section, we have claimed that the DPN model enables expressiveness and practicality, as well as efficient implementation, at the expense of the analysis power. Yet, the limit of static analysis can be overcome by the use of advanced analysis techniques such as classification and dynamic analysis.

3.5.1 *Classification*

Classification is an analysis that determines the behavior of a given actor in terms of production/consumption of tokens, patterns that may govern token exchanges, and possibly acceptable token values. The goal of the classification is to detect the MoC of an actor. In fact, restricted MoCs represent different trade-offs between expressiveness and predictability.

In the simplest case, structural information of an actor is enough to classify it, for instance the rules for an actor to be considered SDF only depend on the input and output patterns of actions. In more complicated cases, it is necessary to gather information from an actual execution of the actor.

The literature introduces several algorithms [186, 173, 178, 179, 49] to classify dynamic actors into restricted MoCs that can be summed up as follow:

1. **Detection of time-dependent actors:** DPN places no restrictions on the description of actors, and as such it is possible to describe a time-dependent actor in that its behavior depends on the time at which tokens are available. This happens when a given action reads tokens from input ports which are not read by a higher-priority action, and when their firing rules are not mutually exclusive.

2. **Identification of static behavior:** Classification tries to classify each actor within models that are increasingly expressive and complex. The rationale behind this is that the more powerful a model 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 cyclo-static, and then as quasi-static. An actor is classified as static if and only if it conforms to the SDF MoC, which means that all its actions have the same input and output patterns. A one-action actor is by definition static.

3. **Finding cyclo-static behavior:** An actor has to meet two conditions to be a candidate for cyclo-static 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 cyclo-static

candidate, abstract interpretation [56] can be used to determine the sequence of actions characterizing its behavior, as well as its production and consumption rates [178, 179].

4. **Determining quasi-static behavior:** A quasi-static actor is informally described as an actor that may exhibit distinct static behaviors depending on data-dependent conditions. The algorithm is composed of two steps. First, the detection of the input FIFO channels used to control the behavior of the actor and their existing configuration. Then, the identification of static behavior for each configuration using abstract interpretation.

5. **If not** classified in a restricted MoC, the actor is defined as **dynamic**.

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 MoCs, such as static scheduling (Section 3.6.3).

3.5.2 *Critical Path Analysis*

As defined by the Amdahl's law and similarly to any program, the execution time of a dataflow program is constrained by the sequential portions contained in the program. Even if dataflow modeling exposes explicitly the parallelism inside an application, the execution of a program is still driven by the data dependencies between the actors, which are characterized by the communication channels.

The widest metric to evaluate this efficiency in dataflow programming is the **critical path**, i.e. the longest, time-weighted sequence of events from the start of the program to its termination regardless the availability of the hardware resources. Since the static analysis of programs based on dynamic dataflow MoC is limited due to their data-dependent behavior, some works [102, 36, 38] investigate the evaluation of the critical path from the execution trace obtained after the simulation of the execution. These methodologies are composed of the following steps:

1. First, the execution trace is built from the simulation of the execution of the program. An execution trace is formally described as a directed acyclic graph $G = (V, E)$ where $v_i \in V$ corresponds to the firing of an action and $e_j = (v_x, v_y) \in E$ corresponds to the functional dependency between two action firings, i.e. v_x produces some data that are consumed by v_y.
   The functional dependency between the two action firings imposes an order in their execution $v_x \prec v_y$ and, by extension, a partial order on the execution trace V. Thus, this execution trace describes an abstract execution of the application, which is independent from the executing platform, and independent from the actor mapping/scheduling.

2. Then, a weight w_{v_i} is assigned to each action firing $v_i \in V$ of the execution trace in order to determine the critical path. The weight of a given action firing corresponds to the time that is needed to execute this particular firing such as

$$w_{v_i} = \delta_{select_{v_i}} + \delta_{process_{v_i}} + \delta_{comm_{v_i}} \quad (6)$$

   where $\delta_{select_{v_i}}$ corresponds to the scheduling overhead introduced by the selection of the action associated to v_i, where $\delta_{process_{v_i}}$ cor-

responds to the time needed to process v_i, and where $\delta_{comm_{v_i}}$ corresponds to the time that is needed to read and write the data from the communication channels. These values can be estimated by instrumented simulation or evaluated precisely by instrumented execution using profiling tools such as Valgrind.

3. Next, a source node v_{source} and a sink node v_{sink} and their dependencies are added to the graph in order to ease the analysis. To do so, two types of dependencies are added to the graph: new edges from v_{source} to any node without incoming edge, and, respectively, new edges from all nodes without outgoing edges to v_{sink}.

4. Finally, the critical path corresponds to the weighted longest path from v_{source} to v_{sink}, which can be evaluated in linear time using the algorithms presented in [36].

The critical path can be associated to a more practical metric such as the throughput of the application, which is quite convenient for stream-based application. Increasing the critical path would ultimately reduce the throughput performance.

This runtime critical path analysis is dependent to the input stimulus, which may reduces the interest of the analysis. Consequently, sufficiently large input stimulus have to be used to overcome this limitation.

3.6 EXECUTION MODELS

Dataflow MoCs assume an ideal execution model, offering unlimited computation resources and unbounded communication channels, which enable the execution of all actors in parallel. But the practical limit of this assumption requires the definition of an execution model that enables the execution of a dataflow program on a processor.

Fortunately, the strong encapsulation of dataflow components, on top of explicitly modeling the concurrency, lets the choice in a variety of execution models. But, the efficiency of these execution models stays strongly dependent to the dataflow MoC that they implement. In fact, this execution model is the formalization of task mapping and scheduling processes introduced in Chapter 2.

3.6.1 *Multi-Threading*

A natural approach for handling concurrent execution on a sequential environment is the use of threads, which can be seen as lightweight processes that share a single address space. Similarly to multitasking, multi-threading is based on a scheduler that organizes the concurrent execution of the threads using preemptive or cooperative strategies.

As a matter of fact, dataflow programs can be easily implemented on a multi-threading environment. Each component of the dataflow graph is executed in its own thread and we let the scheduler organize the execution. Multi-threading is commonly used to execute KPN-based programs on processors thanks to the blocking access to the FIFO channels [82, 94, 128].

However, thread-based implementations can lead to a large overhead when a large number of components are executed to the same processing unit [43]. That's why some works have studied lightweight thread implementations, such as Protothreads [94, 128].

3.6.2 *Dynamic Scheduling*

Instead of relying on threads managed by the operating system kernel, the DPN model allows a continuous execution of the operations of a graph thanks to a user-level scheduler [121]. This scheduler can sequentially test the firing rules from several actors, and fire an actor if a firing rule is valid. An efficient scheduling for dataflow programs consists in finding a, predefined or not, order of actor firings throughout the execution process capable of maximizing the use of all the processing units in one platform.

Since actors in a DPN may have data-dependent behaviors, and data are unknown in the system, determining an optimal schedule of a program is not possible at compile-time (equivalent to the halting problem [139]), i.e. the scheduling can be only done in the general case at run time. We present here two strategies of dynamic scheduling that has been extensively used for the implementation of dynamic dataflow program based on DPN MoC [177] [4].

ROUND-ROBIN This strategy is a simple scheduling strategy based on compile-time ordering of actor execution. The scheduler continuously goes over a static list of actors: The scheduler evaluates the firing rules of an actor, fires the actor if a rule is met and continues to evaluate the same actor until it no longer meets a firing rule. Then, the scheduler *switches* to the next actor. This scheduling policy guarantees to each actor an equal chance of being executed, and avoids deadlock and starvation. Contrary to classical round-robin scheduling, there is no notion of time slice so the timing is performed at run-time: an actor is executed until it cannot fire anymore in order to minimize the number of actor switching and consequently the scheduling overhead. The reason of this actor switching is that in practice the FIFO channels will finally be full or empty because of their bounded sizes.

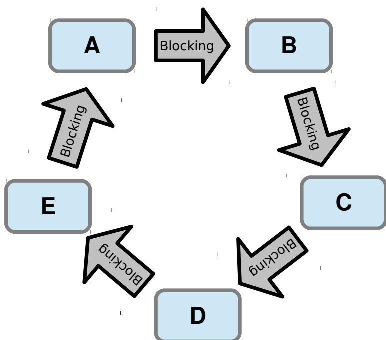


Figure 14: Round-robin scheduling of the actor of the dataflow network presented in Figure 9

Figure 14 shows an application of this round-robin scheduling on the example of dataflow graph presented in Figure 9. The scheduler executes the actors in a circular order i.e. the five actors A, B, C, D and E are successively executed then the scheduler starts again from A and so on.

DATA-DRIVEN / DEMAND-DRIVEN This strategy is a more advanced runtime scheduling strategy. Indeed, the round-robin strategy schedules actors unconditionally i.e. the firing rules of an actor could be checked even if

they are all invalid. In this case, the firing rules of the actor will be checked, but no computation will be performed: That is called a *miss*. As a result, the round-robin strategy becomes inefficient with complex applications containing hundred of actors and a lot of control communications.

Data-driven / demand-driven scheduling strategy is based on the well-known data driven and demand driven principles [139]. On the one hand, data-driven policy executes an actor when its input data have to be consumed to unblock the execution of the precedent actor. On the other hand, demand-driven executes an actor when its output is needed by one of its successor actor. Two types of events can cause the blocking of an actor execution, each one is implying a different scheduling decision:

- When an actor is blocked because an input communication channel is empty, demand-driven policy is applied and asks the scheduler to execute the predecessor of this channel.
- When an actor is blocked because an output communication channel is full, data-driven policy is applied and asks the scheduler to execute the successor of this channel.

Contrary to the round-robin algorithm, the ordering of the actor execution is made at run-time. Thus, a dynamic list is needed to store next schedulable actors. The behavior of this schedulable list is illustrated with Figure 15. When an actor is blocked during its execution, the empty or full FIFO channels are identified and their associate predecessors or successors are added to the schedulable list. The actor to be executed next corresponds to the next entry in the schedulable list.

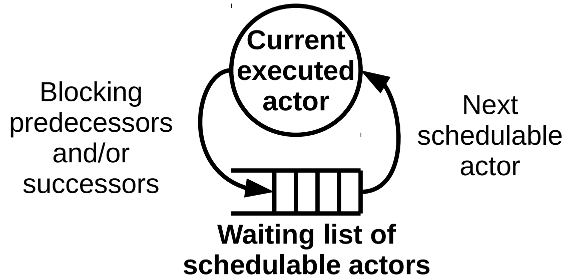


Figure 15: Behavior of the dynamic list of next schedulable actor used by data-driven / demand-driven scheduling

3.6.3 *Static Scheduling*

The main feature of static dataflow MoCs, i.e. SDF or CSDF, is their ability to be scheduled at compile-time, which allows optimizations that are not possible with dynamic MoCs. In fact, static scheduling aims to determine a valid schedule of a dataflow graph that can be applied periodically. A valid schedule consists in a finite sequence of actor firings that introduces no deadlock. When every actor appears just once in the valid sequence, we call the sequence a single appearance schedule.

Given a graph G=(V,E) with $|V| = n$ and $|E| = m$, a sequence of actor firing is defined by a *repeat* vector $\mathbf{q} = (q_{v_1}, q_{v_2}, \cdots, q_{v_n})$ where each q_{v_k} is the number of firing of the actor $v_k \in V$.

A valid sequence must respect the following equality:

$$\forall e = (v_x, v_y) \in E,\ \mathtt{prod}(e)q(v_x) - \mathtt{cons}(e)q(v_y) = 0 \quad (7)$$

This equation can be reformulated as an equivalent matrix equation:

$$\mathbf{\Gamma} = 0 \quad (8)$$

where $\mathbf{\Gamma}$ is the matrix of consumption/production, called *topology matrix*, whose entries are defined by $\mathbf{\Gamma} = (\gamma_{i,j})_{1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n}$ such as:

$$\gamma_{i,j} = \begin{cases} \mathtt{prod}(e_i) & \text{if } \mathsf{src}(e_i) = v_j \\ -\mathtt{cons}(e_i) & \text{if } \mathsf{dst}(e_i) = v_j \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

The resulting topology matrix defines a set of *balance equations* $\mathbf{q} = \vec{0}$. A valid schedule exists only if the set of balance equations admits a non-zero solution.

The problem of finding an optimal schedule being NP-complete, several heuristics have been studied [120, 27, 132, 142]. The main objective of static scheduling for single processor is the minimization of memory requirements. Knowing that the single appearance property guarantees the optimal code size of a static schedule, the studies focus on buffer minimization [120, 27, 132].

3.6.4 *Multi-core scheduling*

As seen before, multi-core scheduling of applications has been extensively studied over the last decade [155]. Considering especially the mapping of dataflow applications, most of the studies focus on static dataflow MoCs since they can more easily be analyzed [154, 72, 13, 142]. However, we can extract from the literature three categories of multi-core scheduling that allow dynamism within the application description.

SCENARIO-BASED APPROACHES These approaches handle the scheduling of dynamic applications using multiple static scenarios [162, 163, 153]. While scenario-based approaches have proved to be very efficient, they are not truly scalable as the number of scenarios increases exponentially with the complexity of the application. Moreover, describing the scenarios requires either additional work from developers, which is not very practical, or advanced analyses to detect them automatically, closely related to the classification described in Section 3.5.1.

TRACE-BASED APPROACHES These approaches handle the scheduling of dynamic applications using execution trace analysis [17, 45, 44]. In fact, the performance efficiency of dataflow programs is often characterized by the *makespan*, the length of the critical path obtained under scheduling constraints, due to its close relationship with the application throughput. Thus, several works perform actor mapping strategies as makespan minimization on the execution trace. While trace-based approaches are very efficient, they cannot be performed at runtime due to the complexity of execution trace analyses.

PROFILING-BASED APPROACHES These approaches handle the scheduling of dynamic applications by way of profiling [123]. The execution of the application is firstly profiled with an initial scheduling, then the scheduling system computes a better schedule from the profiling information. For example, Lucarz describes an algorithm [123] that assigns successively the actors to the available processors starting from the ones with higher workload. But, his approach focuses on the workload without taking into account other aspects such as the communications. Thus, Lucarz presents another algorithm based on Simulated Annealing that considers the communication with the cost of the algorithm complexity.

3.7 EXISTING DATAFLOW-BASED LANGUAGES AND TOOLS

Since Sutherland's preliminary work, dataflow programming has been heavily studied. In fact, dozens of languages, exploiting the dataflow paradigm, were designed to solve a wide range of problems, from digital signal processing to hardware design. In fact, dataflow languages can be described as DSL [61] that focus on *how things connect*.

Here is a non-exhaustive list of modern languages and tools that are based upon the dataflow paradigm:

LUSTRE is a synchronous dataflow language developed for programming real-time systems and describing hardware [95]. It was introduced in the early 1980s by a research project and is now used as the core language of the SCADE toolset, an environment with certified code generation dedicated to the programming of critical systems such as aircraft and nuclear plant.

SIGNAL is another synchronous dataflow language that was designed for programming real-time systems [117], similarly to Lustre. SIGNAL is supported by the open-source Polychrony toolset that provides a complete integrated development environment.

STREAMIT is a dataflow language based on SDF MoC, which was later extended with dynamic features. Streamit was initially developed as a support for research studies on dataflow programming at MIT [168]. Streamit is also a dedicated compilation infrastructure that includes compilation flow for several microprocessors, performing a set of domain-specific and architecture-specific optimizations.

DAEDALUS is a system-level design environment for MPSoC platform [130], which automatically parallelizes the C specification of an application using the Polyhedral Process Network (PPN) model, which is a special class of KPN that involves nested loops.

MAPS is another programming environment for MPSoC applications that is based on the KPN model [44, 45]. MAPS targets automatic parallelization of sequential application, scheduling of parallel application, as well as multi-applications scheduling.

PREESM is an open-source tool [141] that aims to generate efficient code for multi-core DSP thanks to a rapid prototyping approach based on static dataflow modeling.

C~ pronounced *c-flow*, is a KPN-based language targeting hardware development [181]. Synflow, the start-up company developing C~language

and the associated tools Synflow studio, intends to make hardware design more efficient and reliable using the high abstraction level offered by the dataflow paradigm.

CAL is a dataflow language based on the DPN model [67]. CAL was developed by Eker and Janneck to provide a practical language for the development of applications in a variety of domains, such as multimedia processing, control systems, network processing, etc. CAL has been designed as a part of the Ptolemy project, which studies model-based techniques for the development of real-time and embedded system, inside a tool called Ptolemy II [68].

This subset of dataflow languages and tools shows already the variety of the application domains where dataflow programming has been applied.

3.8 conclusion

All along this chapter, we introduce dataflow programming as a challenging programming paradigm that offers a flexible development approach to deal with the increasing complexity of the applications, and that offers a large degree of parallelism to exploit the massive parallel capabilities available in modern architectures. We also show that the use of a programming language based on the dynamic dataflow model, rather than static models, is a pragmatic choice to implement complex applications. Indeed, these dynamic dataflow languages offer a large expressive power along with a practical syntax that are both required for an industrial-scale development.

Before explaining our contribution to dynamic dataflow programming, the next chapter introduces the RVC framework: an innovative framework, introduced in order to improve the standardization process of video compression standards, that can be considered as the first large-scale experimentation on dynamic dataflow programming to our knowledge.

4 RECONFIGURABLE VIDEO CODING

Before you become too entranced with
gorgeous gadgets and
mesmerizing video displays,

let me remind you that
information is not knowledge,
knowledge is not wisdom, and
wisdom is not foresight.

Each grows out of the other, and
we need them all.

— Arthur C. Clarke, British writer

The growing popularity of multimedia has made digital video mainstream. Digital video is now used for a wide range of applications that are achievable with the advances in computing and communication technologies as well as video compression techniques. However, the deployment and adoption of these technologies were possible primarily because of the standardization process that offers the interoperability between the multimedia devices available on the market.

The standardization process of video compression formats is mainly driven by the following organizations that are themselves driven by the industry with participation and contributions from academia:

- The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) that have jointly formed the Moving Picture Experts Group (MPEG).
- The International Telecommunication Union (ITU) and more specifically its Video Coding Experts Group (VCEG).

Since 2001, both working groups are collaborating in the standardization of the major video formats. The standardization process is composed of several steps that start with a call for proposal asking for tools and technologies to solve a given problem. Then, the proposals are experimentally evaluated in order to choose the tools and technologies selected to take part of the standards and keep the standards relevant to the needs of industry.

4.1 LIMITS OF THE STANDARDIZATION PROCESS

Video compression standards have become extremely complicated systems, and are consequently long to specify. So, traditional standardization processes quickly show their limits as regards to the time-to-market pressure.

4.1.1 *Multiplication of the Standards*

Since the standardization of H.120, the first digital video coding standard (in 1984 by the ITU), the number of video coding standards has increased

in a linear way, as presented in Figure 16. Even if the H.120 standard was not widely used due to practical reasons, it represents the roots of digital video compression techniques, and most of recent video standards can be considered as its direct successors.

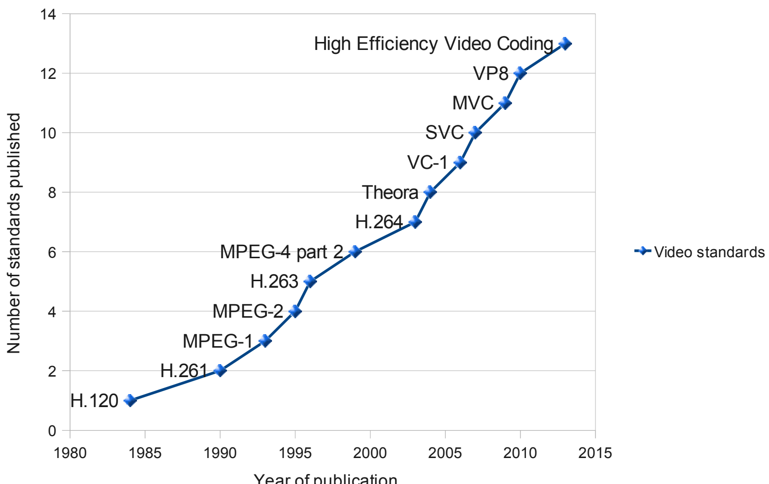


Figure 16: Multiplication of the video compression standards

In fact, the standardization organizations, ITU and ISO/IEC, have ratified most of the successful video standards: H.262 / MPEG-2 Part 2 in 1996 and H.264 / MPEG-4 Part 10 in 2003. However, the controversy about the patent licensing of these standards has led to the definition of royalty-free formats such as the Google's VP8/VP9, BBC's Dirac and Xiph's Theora/Dalaa. Various other video compression formats were standardized as well such as Microsoft's VC-1 included in the Blu-ray standard or the Chinese Audio Video Standard (AVS).

Apart from the question of licensing, a large choice of compression standards is offered to encode a video stream:

- The increasing demand in multimedia applications, requiring constantly higher compression rate, combined with the technological improvement of computing hardware, offering ever more computational capabilities, has led to the recurring development of additional video coding standards based on state-of-the-art algorithms. For instance, the new High Efficiency Video Coding (HEVC) standard has been designed to take advantage of the parallel processing capabilities offered by modern devices [164].

- Digital video compression techniques have expended their application domains according to the development of multimedia. Nowadays, video compression is involved in a large panel of applications such as streaming, conferencing, surveillance, storage, medical monitoring, etc. As a consequence, the ITU/ISO video standards are structured around the definition of several profiles, each one defining a subset of features that is relevant for the applications that they characterize. H.264 / Advanced Video Coding (AVC) has even been amended with additional profiles in order to provide scalable and multiview capabilities.

Unfortunately, a standard does not disappear when a new one is announced, so they have to coexist in most of the situations. In fact, the mi-

gration towards a newly adopted standard is a long-time process that often requires a large scale hardware update, which is much more difficult than updating a software. As an example, MPEG-2 part 2, yet released in 1995, is still being used by the television broadcast network in north America, although MPEG-4 part 10 is also proposed.

4.1.2 *Monolithic Specifications of the Standards*

Since the specification of MPEG-2, the textual reference of each standard is provided with an implementation of a video decoder complying with the ratified standard. This normative implementation, called *reference software*, has been up to now developed in monolithic fashion using regular C/C++ languages. Nevertheless, the structure of a software system impacts all its life, from the development to the deployment through the maintenance, and a monolithic architecture brings the following limitations:

- Software designers are often compelled to **rewrite** the decoder from scratch so as to design a new architecture that may be necessary not only to reach the performance expected, but also to adapt the software description to the design methodologies of the computing devices. For instance, it is impossible to translate automatically such a monolithic implementation into a synthesizable hardware description. Moreover, programming languages based on the imperative paradigm have in general a limited potential of parallelism due to their tendencies to over-specify the programs.

- Seeing that most of the new standards can be considered as an evolution of the last one, the monolithic implementation of the reference software has brought a lot of **redundancy** in video compression standards and their reference software. As an example, the deblocking filter is available in both AVC and HEVC standards but, because of their slight differences, it was entirely rewritten during the specification of HEVC. Unfortunately, the difficulty to extract the redundancies among the video standards limits the development of decoding systems supporting several standards.

4.2 AN INNOVATIVE DEVELOPMENT FRAMEWORK

To overcome the lack of interoperability between all the video compression standards deployed in the market, MPEG has introduced an innovating framework, called RVC [125, 29, 126], dedicated to the development of video coding tools in a modular and reusable fashion.

The MPEG RVC framework defines two standards that have been produced by the RVC working group:

- The codec configuration representation (ISO/IEC 23001-4 or MPEG-B pt. 4) [10] describes the format with which an RVC decoder can be defined as a network of computational blocks, as well as a textual language for the definition of video coding blocks (Section 4.3).

- A video tool library (ISO/IEC 23002-4 or MPEG-C pt. 4) [11] that standardizes actors needed to describe existing video coding standards (Section 4.2.2), currently MPEG-4 part 2 and MPEG-4 part 10.

In fact, RVC does not only provide a new standardization process that overcomes the limits of the current standardization process, but also introduces a framework that enhances multimedia development by offering all the advantages of dataflow programming with the pragmatism required by the development of complex applications.

4.2.1 *Dataflow to Enhance Multimedia Development*

The traditional imperative programming paradigm leads to the implementation of monolithic applications that are limited by nature. The RVC framework overcomes this limitation by exploiting the dataflow programming paradigm in order to propose a flexible development approach that produces modular, scalable and portable applications.

MODULARITY Multimedia applications become more and more complex. To handle this complexity, the development process has to be flexible enough to allow the writing of modular descriptions. The strong encapsulation of the components of a dataflow program offers the required modularity:

1. **Hierarchical ability** enables the organization of the components in subnetworks according to their concern, such as the motion prediction or the residual decoding in a video decoder.
2. In video compression standardization, **re-usability** is particularly relevant to describe the multiple profiles of a given standard that may share a large portion of their algorithms.
3. **Reconfigurability** enables adaptive execution, which is required by the ideal video decoder: A *universal* decoder which would be able to decode any video stream, independently from the compression standard to which it refers.

SCALABILITY Modern multimedia applications manipulate rich media contents, such as video stream, and consequently can be qualified as compute-intensive applications. As presented in Chapter 2, modern architectures offer massive parallel capabilities in order to achieve the real-time performance required by these compute-intensive applications. One of the advantages of dataflow programming, that we have emphasized in Section 3.2, is the explicit concurrency that simplifies the use of the parallelism compared to traditional imperative programming paradigm. As an example, some parallelisms are inherent to video processing:

1. The succession of processings on the data stream (filtering, transforming, etc), which composes a video codec, can be directly modeled by a set of interconnected boxes making **Pipeline-Level Parallelism** and **Task-Level Parallelism** straight-forward.
2. The independent processings of the image components, Luma and Chroma, are a good example of the **Data-Level Parallelism** that can be exposed within a dataflow description.

PORTABILITY The heterogeneity of multimedia devices makes portability an interesting property of video coding tools, and their implementation within the RVC framework enables their compatibility with hardware and

software platforms. In fact, these high-level descriptions aim to be transcompiled in lower-level languages in order to bridge the gap between existing programming models:

1. **Hardware synthesis:** Dataflow modeling simplifies the translation of the description of an application into Hardware Description Languages (HDLs) targeting Application-Specific Integrated Circuits (ASIC) and Field-Programmable Gate Arrays (FPGAs). In fact, the explicit concurrency is quite similar to the structural view of a hardware description, contrary to monolithic specifications that usually involve complex analysis such as the ones performed in High-Level Synthesis (HLS).

2. **Software programming:** When hardware synthesis is straight forward, the translation of dataflow descriptions into software programming languages usually requires an execution model to handle the scheduling of the dataflow components in a sequential environment (Section 3.6). But, the increasing parallel capability of modern processors takes advantage of explicit concurrency of dataflow descriptions. Furthermore, explicit communications simplify the execution on architectures that require advanced memory accesses such as DSP and GPU.

3. **Co-design for heterogeneous platform:** Describing the application using a set of interconnected components enhances its ability to be executed on an heterogeneous platform, containing a mixture of microprocessors and/or hardware processing units. Since the components can be translated independently into software or hardware, the codesign flow performs the mapping of the components onto the available computing resources according to the specification of the application, the executing platform, and the user-defined constraints.

4.2.2 *Towards the RVC Vision*

To do so, the RVC framework introduces the concepts of Video Tool Library (VTL) and Abstract Decoder Model (ADM) [125, 29, 126]:

- The VTL is a collection of algorithmic components composing video codecs, known as Functional Units (FUs), that are specified using a programming language called RVC-CAL. The RVC framework provides a normative VTL, the MPEG VTL, that contains all the FUs required to cover all MPEG standard specifications. Proprietary VTLs can also be proposed within the framework to provide extended collections of FUs that allow optimizations or additional features.

- The ADM is a generic representation of a decoder, specified using a dataflow network of coding tools, the FUs from the VTL. Several codecs can be specified by combining FUs together from a common VTL, and, for instance, a single FU can be involved in several specifications. This re-usability simplifies the development of multi-standard video decoding applications and devices by allowing software and hardware components to be reused across video standards.

Thus, as shown in Figure 17, a decoder description can be delivered along with the encoded video stream. The decoder description is used to configure the decoder engine to be able to decode the video stream. So, the decoder is instantiated from the dataflow graph representing the decoder description

and the components available in the VTLs (normative and/or proprietary). In other words, the decoder is constructed according to the video encoding format, as opposed to traditional rigidly-specified video decoders.

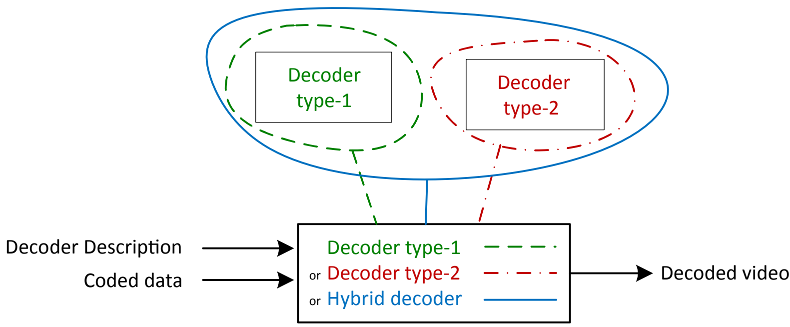


Figure 17: RVC vision

4.3 MULTIMEDIA-SPECIFIC LANGUAGES

The RVC framework uses two programming languages for both levels of the description [10]: A visual programming language, called Functional unit Network Language (FNL), to represent the interconnection network (Section 4.3.1), and a subset of CAL, known as RVC-CAL, to describe the component's behavior (Section 4.3.2).

Additionally, the framework also introduces the Bitstream Syntax Description Language (BSDL) which describes the structure of the bit-stream [9, 149], i.e the incoming video stream. This language aims to automatically generate the actor that can parse and decode the corresponding input stream. But, considering that no implementation has yet demonstrated the practicality of this approach, the BSDL will not be described deeper in this thesis.

4.3.1 *From Text to Visual Network Programming*

FNL is the programming language used to specify the interconnection network between all the actors. The main characteristic of this language is its ability to support a textual representation (Listing 2) as well as a graphical representation (Figure 18). Additionally, an equivalent representation based on eXtensible Markup Language (XML) has been standardized, known as XML Dataflow Format (XDF), to allow the interoperability between the tools [10] (Listing 3). As a matter of fact, each vertex or edge from the graphical representation corresponds to an element of the textual representation, as well as an element of their XML-based representation:

- A vertex represents one `Instance` of an entity. An entity can be an actor or a network, both being identified by their `Class` which is composed of the package name, i.e. the localization of the entity, and the name of the entity. An entity may also be parametrized to improve its reusability. By way of example, in Figure 2, the vertex `Display` (Line 4) represents the instantiation of an entity, named `Display` as well, which is located in the package `org.sc29.wg11.common`. For this instantiation, the parameter `BLK_SIDE` is set to `64`.
- An edge represents a `Connection`, i.e. a communication channel, between two entities, and, by extension, a data dependency. This edge

is connected to the port (**src-port**) of its source entity (**src**), and to the port (**dst-port**) of its destination entity (**dst**). A connection may also be parametrized with a specific channel size.

This simple example already demonstrates the practicality of visual programming: The verbosity of the textual representation emphasizes the clarity and the natural of the graphical representation. Even so, visual programming requires an advanced editor to be effective.

```
1  network Top_mpegh_part2_main():
2    entities
3      Source = org.sc29.wg11.common.Source();
4      Display = org.sc29.wg11.common.Display(BLK_SIDE=64);
5      Decoder = org.sc29.wg11.mpegh.part2.Decoder();
6    structure
7      Source.O --> Decoder.BYTE;
8      Decoder.VID --> Display.VID;
9      Decoder.DispCoord --> Display.DispCoord;
10     Decoder.PicSizeInMb --> Display.PicSizeInMb;
11 end
```

Listing 2: Textual representation of dataflow network

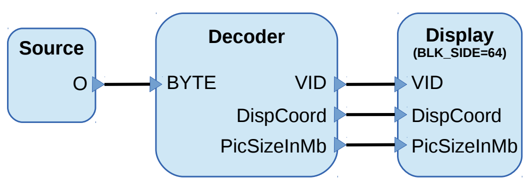


Figure 18: Visual representation of dataflow network

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <XDF name="Top_mpegh_part2_main">
3    <Instance id="Source">
4      <Class name="org.sc29.wg11.common.Source"/>
5    </Instance>
6    <Instance id="Display">
7      <Class name="org.sc29.wg11.common.Display"/>
8      <Parameter name="BLK_SIDE">x
9        <Expr kind="Literal" literal-kind="Integer" value="64"/>
10     </Parameter>
11   </Instance>
12   <Instance id="Decoder">
13     <Class name="org.sc29.wg11.mpegh.part2.Decoder"/>
14   </Instance>
15   <Connection dst="Decoder" dst-port="BYTE" src="Source"
         src-port="O"/>
16   <Connection dst="Display" dst-port="VID" src="Decoder"
         src-port="VID"/>
17   <Connection dst="Display" dst-port="DispCoord" src="Decoder"
         src-port="DispCoord" />
18   <Connection dst="Display" dst-port="PicSizeInMb" src="Decoder"
         src-port="PicSizeInMb"/>
19 </XDF>
```

Listing 3: XML-based intermediate representation of dataflow network

4.3.2 *Actor Programming Made Easy*

A subset of CAL, called RVC-CAL, has been included in the standardization of the RVC framework [10]. The language is a mixture between imperative and functional programming languages that introduces useful abstractions for dataflow programming. Comparing to the original CAL language [67], RVC-CAL provides a precise type-system as well as some practical features.

The language is used to describe the behavior of the components, called actors. The execution of an actor is composed of a sequence of ordered steps, applied repeatedly:

1. First, the actor consumes, or not, a given amount of data from its input ports.
2. Then, it may modify its internal state.
3. Finally, it produces, or not, a given amount of data to its output ports.

As a consequence, describing an actor execution involves the description of its interface such as the input and output ports, its internal state that is modeled by a set of state variables, as well as the procedural description of the computational steps and the internal scheduling strategy that ordered these steps.

HEADER In RVC-CAL, an actor can be decomposed in a header and a body such as presented in Listing 4. The header of the actor is composed of its signature along with its interfaces that are both declared at the top of the actor description:

- The **signature** that identifies precisely the actor in the VTL. The signature is composed of the name of the actor (`IT4x4_1d`), and the name of the package where the actor is located (`devel.org.sc29.wg11.mpegh.part2.xIT`), in a Java fashion.
- The **interfaces** of the actor that describe the structure used to interact with the *outside*. The interfaces is composed of the input ports (`int(size=16)Src`) and output ports (`int(size=16)Dst`) connected to the communication channels, and the parameters (`int shift`) that are variables initialized only when the actor is instantiated within a network.

```
1 package devel.org.sc29.wg11.mpegh.part2.xIT;
2
3 actor IT4x4_1d(int shift) int(size=16) Src ==> int(size=16) Dst :
4
5   // body
6
7 end
```

Listing 4: Header of an actor

PROCEDURAL CODE RVC-CAL describes the behavior of an actor by the way of imperative programming paradigm, among other specific structures that we detail below. To do so, the language supports the common concepts that are traditionally used by procedural language (Listing 5), such as variables, functions and procedures .

```
int MAX_RANGE = 15;
int BIT_DEPTH = 8;
int coeff := 32;

function abs(int(size=32) x) --> int(size=32) :
  if(x > 0) then x else -x end
end

procedure nextLcuAddressFilt()
begin
  xCuFilt := xCuFilt + 1;
  if(xCuFilt = picSizeInCu[0]) then
    xCuFilt := 0;
    yCuFilt := yCuFilt + 1;
  end
end
```

Listing 5: Procedural code

ACTION An action corresponds to a firing function, which describes, in a procedural manner, the behavior of the actor during a firing. For example, the action, presented in Listing 6, reads 16 tokens from its input port `Src` and copies them in a new order to its output port `Dst`.

```
action Src:[ src ] repeat 16  ==> Dst: [ dst ] repeat 16
var
  List(type:int(size=16), size=16) dst
do
  dst :=  [ src[ 4 * column + row ] :
              for int row in 0 .. 3, for int column in 0 .. 3 ];
end
```

Listing 6: An action that transposes 4x4 blocks

INTERNAL SCHEDULING CAL is a control-oriented language, several mechanisms are offered to describe explicitly the internal scheduling within an actor [67]:

- The **guards** and the **patterns** implement together the concept of the firing rules introduced by the DPN model (Section 3.3.2). A guard is a condition on the fireability of the action depending on the value of the state variables and/or the incoming tokens, while the pattern focuses on the amounts of tokens/rooms that have to be available on the communication channels. As an example, being able to fire the action `getPixValue`, presented in Listing 7, requires that the condition of the line 3 is valid, and that the channel connected to the input port `B` has at least 256 tokens available.

```
getPixValue: action B :[Bytes] repeat 256 ==>
guard
  nbBlockGot < pictureSizeInMb
do
  // Body
end
```

Listing 7: Guard and pattern

- The **priorities** define a partial-order relation on the firing rules (lines 9-12 of Listing 8). Since two firing rules are not necessarily exclusive, the description of priorities reduces the possible non-determinism.

  The **Finite State Machines (FSMs)** have also been introduced to describe the internal scheduling of an actor in a convenient way (lines 1-7 of Listing 8). In fact, the FSMs do not provide a larger expressive power to the language than the one already available with the guards, the patterns and the priorities.

  Additionally, all actions without names, that is to say **untagged actions**, are not constrained by any schedules [67]. Thus, untagged actions always have the highest priority and can be executed from any state of the FSM. Untagged actions are another practical feature that should be however used carefully since they can imply a large scheduling overhead.

```
1  schedule fsm initLength:
2    initLength ( computeNewLength ) --> copy;
3    copy       ( copyData         ) --> copy;
4    copy       ( endCopy          ) --> padding;
5    padding    ( zeroPadding      ) --> padding;
6    padding    ( endZeroPadding   ) --> initLength;
7  end
8
9  priority
10   endCopy > copyData;
11   endZeroPadding > zeroPadding;
12 end
```

Listing 8: FSM and priorities

TYPES The specification of the RVC-CAL language defines an accurate type system containing the following data types [10]:

- An integer data type that can be signed or unsigned, declared with the `int` and `uint` keywords respectively. An integer data type can also be bit-accurate, for instance the type `int(size=8)` considers a signed integer coded on 8 bits.
- Three floating-point types coded on 16, 32 and 64 bits, that are defined respectively using the `half`, `float` and `double` keywords.
- A logical data type, having two potential values `true` and `false`, unsurprisingly declared using the keyword `bool`.
- A type to describe a sequence of characters, `String`.
- A list type that is declared with a given type and size, such as `List( type:int, size=8)` that represents a list of 8 integers, so closely related to a simple array.

In fact, the type system is one of the major difference between the original CAL and the one standardized within the RVC framework. When CAL keeps an abstract type system authorizing untyped data [67], RVC-CAL defines a practical type system dedicated to the development of signal processing algorithms.

All these features make RVC-CAL fully analyzable, in the sense of actor-level analysis as opposed to network-level analysis, contrary to general-purpose programming languages such as C that are hardly analyzable because of complex mechanisms like pointers [97].

4.4 APPLICATIONS

The inclusion of a subset of CAL in the MPEG RVC framework has enabled the development of several video decoders, along other applications, using dynamic dataflow programming. Such a collection of applications offers a great opportunity to study all the problematics related to dynamic dataflow programs.

4.4.1 *Video Codecs*

Since the standardization of H.261, the first block-based digital video coding standard, in 1988 by the ITU, all existing ITU/MPEG video codecs have globally kept the same structure [150]. The difference between the standards comes mainly from the evolutions of the algorithmic part that offer an increasing compression rate. In fact, the decoding process can be divided in 3 distinct parts, that make the application graph of all RVC-based video codecs quite similar [126]:

1. The first part, called *parser*, extracts values needed by the next processing from the compressed data stream. The stream is decompressed with entropy decoding techniques, then the syntax elements composing the stream are extracted in order to be transmitted to the actors that they may concern.
2. Another one, known as *residual*, decodes the error resulting of the image predication using inverse transforms, such as the well-know IDCT. The transforms allow spatial redundancy reduction within the encoded residual image.
3. And, a last part, called *prediction*, performs the intra and inter prediction. Intra prediction is done with collocated blocks in the same picture whereas inter prediction is performed as a motion compensation with other pictures. The inter prediction also implies the use of a buffer containing decoding pictures to be able to perform the temporal prediction.

Since its creation, the RVC working group has standardized the implementation of 3 video decoders detailed below:

MPEG-4 PART 2 MPEG-4 Part 2 standard, also known as MPEG-4 visual, was released in 1999 by the joint ISO/ITU consortium. The popular DivX and Xvid codecs, that have largely contributed to the development of video sharing over the Internet, implement this standard. In fact, The Simple Profile of MPEG-4 Part 2 decoder was the first application standardized by the RVC working group. Given the novelty of the approach, the decoder was the source of dozens of experiments that have conducted to the development of several versions of the decoder with variable granularities.

Figure 19 presents the normative version of the description. As presented, the structure of the application graph matches well with the

structure of the video standard. The graph can be partitioned into three regions, each one corresponding to a dedicated processing: parsing, residual decoding and motion compensation. To increase the parallelism exposed within the decoder, the parser can separate the processing of each image components, luma and chroma, in three parallel paths (Y, U and V). The image components are then merged back at the end of the processing.

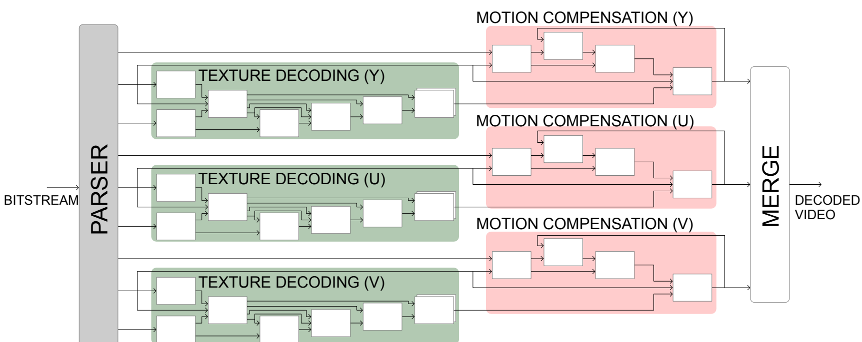


Figure 19: RVC-based description of the MPEG-4 Part 2 SP decoder

MPEG-4 PART 10 Introduced in 2003, MPEG-4 Part 10, also known as MPEG-4 AVC / H.264, is a widely-used video standard since the advent of High Definition in everyday usage [176]. In fact, AVC is currently one of the most exploited standards within commercial video services, going from web streaming to digital broadcasting including camera recording.

As presented in Table 4, two profiles of the AVC codec were standardized, Constrained Baseline Profile (CBP) [87, 23] and Progressive High Profile (PHP). The large number of actors and FIFO channels states of the complexity of the decoder, as well as the controlled fashion of the descriptions.

MPEG-H PART 2 MPEG-H Part 2, also known as MPEG HEVC / H.265, is the last born video coding standard, developed conjointly by ISO/ITU, as a successor to MPEG-4 AVC / H.264. HEVC is improving the data compression rate, as well as the image quality, in order to handle modern video constraints such as the high image resolutions 4K (3840 x 2160) and 8K (7680 x 4320) [164]. Another key feature of this new video coding standard is its capability for parallel processing that offers scalable performance on the trendy parallel architectures [164].

Such parallel capabilities offer a great opportunity to prove the interest of the RVC approach. Consequently, the RVC working group has developed, in parallel with the standardization process, an implementation of the HEVC decoder using the RVC framework, which is presented in Figure 20. This joint effort has permitted the demonstration of a functional version during the 103th MPEG meeting in January 2013. At the same time, the final draft of the HEVC standard was approved.

Table 4 summarizes the properties of each description of these well-known decoders: Respectively, the profile of the decoder, the parallelization of the decoding for each component, the number of actors and FIFO channels.

The RVC-based video decoders are described with an average granularity (at block level), contrary to the traditional coarse-grain dataflow (at

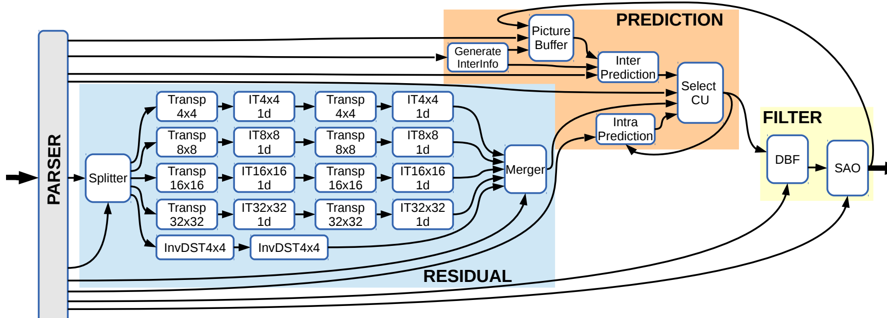


Figure 20: RVC-based description of the MPEG-H Part 2 SP decoder

Standard	Profile	Version	YUV	#Actors	#FIFOs
MPEG-4 Part 2	Simple Profile	RVC	yes	41	143
		Xilinx	no	34	86
		Ericsson	yes	42	105
		EPFL	no	13	29
		Irisa	yes	41	104
MPEG-4 AVC	CBP	RVC	yes	94	270
	PHP	RVC	yes	114	404
MPEG HEVC	Main	RVC	no	34	109
	Still Picture	RVC	no	31	74

Table 4: Statistics about the RVC-CAL description of several MPEG video decoders

frame level). This fine-grain streaming approach induces a high potential in pipeline parallelism and the use of small communication channels, usually between 512 and 8192 rooms.

4.4.2 *Other Applications*

While the RVC framework has been introduced in the video context, it is actually a more general-purpose framework that is usable to model any application, as long as it can be described in a stream-based fashion.

Beside the standardization context, the RVC framework has been used to develop a cryptographic library [15, 14], some image codecs such as JPEG, JPEG 2000 and LAR [104].

4.5 EXISTING TOOLS SUPPORTING RVC

Apart from the normative parts, which focus on the conceptual vision of RVC and the standardization of the FUs, both industrials and academic researchers have developed a set of tools supporting the RVC framework.

4.5.1 *OpenDF*

The Open Dataflow Environment (OpenDF) [28] is a dataflow toolset, initiated in 2007 under open-source licensing (BSD), dedicated to the CAL language. OpenDF is embedding an interpreter of CAL code, as well as a multi-target compiler and some analysis tools.

The compiler is composed of three backends able to generate code for different platforms, from an XML-based intermediate representation named XLIM [127]: The first one is an HDL backend based on OpenForge [103] that targets Xilinx FPGAs. The last two backends generate C code, one targeting the SystemC toolchain [151] and the other one dedicated to embedded platform based on ARM processor [172].

Given the technological limitations of OpenDF, the project has been progressively dropped in favor of Open RVC-CAL Compiler (Orcc) (Section 4.5.2) and is now no longer being maintained. Eker et al. have recently introduced a new tool, called Caltoopia, that is promoting itself as an development kit for CAL. Caltoopia is built on the top of a new software architecture but uses some of the work made in OpenDF.

4.5.2 *Orcc*

Started in 2009, Orcc [134, 177] is an open-source toolkit dedicated to the development of RVC applications. Orcc is a complete Eclipse-based Integrated Development Environment (IDE) that embeds two editors for both actor and network programming, a functional simulator and a dedicated multi-target compiler. In fact, this compiler has been the experimental laboratory of this thesis, as presented in Chapter 5.

The compiler is able to translate the RVC-based description of an application into an equivalent description in both hardware and software languages for various platforms (FPGA, GPP, DSP, etc). A specific compiler back-end has been written to tackle each configuration case:

- Two **software** back-ends that generate C/C++ programs usable on most of the programmable processors [180, 177]. Dataflow application produced with the software back-ends have multi-core abilities that benefit from the inherent parallelism of dataflow applications [4, 7], more details are provided in Chapter 7. A rapid prototyping can additionally be performed on static applications [3].

- Two **hardware** back-ends using well-known High-Level Synthesis tools to generate synthesizable HDL code for FPGA and ASIC implementations [12, 24].

- A back-end that targets **embedded multi-core platforms** [5]. The back-end is implemented as an entire co-design flow that generates the software code as well as the hardware design that executes it. This back-end is the main contribution of this thesis and is described along Chapters 5 and 6.

- A back-end that produces **libraries of components**, usually called VTLs, in dataflow-specific bitcode [88] to implement the ADM of the RVC framework [29]. The libraries of components are then used to perform adaptive execution based on virtual machine mechanism using Jade [2].

Additionally, some advanced analysis dedicated to dynamic dataflow applications can be performed during the compilation. A dynamic analysis, called actor classification [178, 179], can detect predictable behavior within a network that may allow compile-time optimization such as static actors scheduling. Another analysis, based on model-checking techniques [69], can prune all unreachable execution paths to remove the unnecessary tests and accelerate the execution.

The Orcc environment has also been the foundation of two external tools, known as the Just-in-time Adaptive Decoder Engine (Jade) and Turnus:

JADE can be considered as the software implementation of RVC concept, a generic decoder able to configure itself according to a configuration using a VTL and the virtual machine -based mechanisms [89, 86].

TURNUS is a proprietary tool based on the Orcc simulator engine, which is dedicated to the profiling and design space exploration of RVC-CAL application [39].

4.6 ADVANCES AND CHALLENGES OF THE RVC FRAMEWORK

Since its introduction in 2004, the RVC framework has been subject to many studies from academic and industrial researchers. While tools and applications have now reached a certain maturity, there still are some open challenges that prevent the wide-spreading of RVC, CAL and more generally dynamic dataflow programming. In fact, one of the fundamental challenges of the RVC framework is paradoxically the development of tools concurrently with the development of applications.

4.6.1 *Tools Development*

Since dynamic dataflow programming has not been heavily studied, a large part of the research work on RVC framework focuses on tools involved in the development of new applications, e.g. Orcc or OpenDF.

ASSISTED PROGRAMMING Unlike most research tools, our development environment is actually used to write applications, and not just small benchmarks but real-world applications that involve complex and error-prone tasks. As a consequence, our toolset has to provide a certain number of features that are usually present into most of the modern IDE to make the development of applications easier and help to use a *not-so-natural* programming paradigm:

- Graphical editing of application graph, since one of the main interest of dataflow programming is its ability for visual programming.
- Syntax coloring, code completion, code validation are all basic functions that are expected in a modern text programming editor.
- Programming implies necessarily the need for debugging capabilities, this need is increasing with the application complexity, but debugging dataflow programs is more challenging than debugging traditional programs [146].

While all these features truly simplify the development of new applications, they require a time-consuming development effort that is largely beyond our research interests.

RETARGETABILITY One of the main accomplishments of the RVC framework is the portability of applications over a large range of platforms (FPGA, GPP, DSP, etc). The idea of targeting multiple platforms with a single application description may be very attractive, but it raises a certain number of issues in term of software architecture such as:

- An increasing number of compiler back-ends, which requires the capitalization of algorithms.
- An extensive use of third-party tools to smoothly connect with all targeted platforms.

INNOVATION Most of the contributors to our development environment have a main background in signal processing. On the one hand, they can take advantage of their expertize and propose innovative techniques to improve dynamic dataflow programs. On the other hand, their weak knowledge in compilation and software engineering is quickly becoming an obstacle due to the complexity of a software such as a compiler.

To sum up, RVC-based development environments are complex pieces of software that have to provide the simplicity, reliability and flexibility asked by application developers, while keeping a maintainable, extensible and scalable architecture for boosting research innovations of tool developers. To do so, we need to setup a pragmatic software development process.

4.6.2 *Applications Development*

Dozens of applications for a variety of domains have already been produced within the RVC framework (see Section 4.4). In fact, much of the development time has served as manual design exploration to achieve the expected performance. Thus, Brunet et al. have proposed a set of automatic and semi-automatic techniques implemented in Turnus [39] to assist application developers in this exploration task [36, 38]. But, the development of applications still involves open challenges that are portability and re-usability.

PORTABILITY Some of these applications, especially video decoders which were heavily studied, present a large number of variations that have been proposed in response to particular needs whereas RVC-CAL aspires to be portable. As an example, there are at least four distinct descriptions of video decoders implementing the MPEG-4 Part 2 compression standard:

- The normative description considered as the reference;
- A single-rate description developed by Xilinx to specifically target hardware platforms;
- A description optimized by Ericsson for multi-core processors;
- A description, proposed by Mattavelli et al., resulting of application design exploration;

From our point of view and in respect to the RVC principles, the application developers should develop a single version of each description, preferably a high-level description to ease the development process, and let the compiler automatically optimize this description according to the targeted platforms, such as the multi-to-single rate transformation that is discussed below to target hardware platforms.

RE-USABILITY Another major interest of the RVC framework is the re-usability of the dataflow components, called FUs, over multiple applications. Actually, Palumbo et al. have proposed a methodology that takes advantage of this re-utilization over multiple applications to build multi-purpose hardware systems with a limited resource usage [135, 136, 129]. Gorin et al. have also benefited from the reutilization in their adaptive decoder to speed-up the reconfiguration [89, 86].

Unfortunately, the applications that are currently available do not make an extensive utilization of component re-usability. In fact, only AVS and AVC decoders really share a significant amount of common components [87, 184].

4.6.3 *Platform Implementation*

Another challenge that have to face dynamic dataflow programs is the demonstration of efficient implementations that can achieve performance constraints imposed by modern applications.

HARDWARE SYNTHESIS Design flows from RVC applications to hardware platforms, in the sense of FPGA and ASIC, have been implemented in OpenDF [103] and Orcc [158, 24, 12, 25]. The basic idea of these approaches is the direct transformation of RVC-CAL descriptions into Register Transfer Level (RTL) ones suitable for FPGA or ASIC synthesis.

The major difference between the methodologies comes from the abstraction level of the generated code: Janneck et al. generate low-level and optimized HDL code dedicated to a specific platform (*close-to-gate* RTL) [103, 24, 25], whereas Siret et al. generate high-level, portable and readable HDL code (*close-to-hand-written* RTL) and let the synthesizer perform the optimizations [158]. A novel approach capitalizes on HLS tools that are able to translate software code into RTL descriptions, such as Xilinx Vivado HLS, in order to focus on the generation of understandable C code [12].

All of these methodologies suffer however from a severe limitation as they are only applicable on single-rate RVC-CAL programs, i.e. actors can only read and write single tokens at once, that require the use of low-level actors and fine dataflow granularity. Jerbi et al. describe an automated transformation from multi-rate RVC-CAL programs to a single-rate programs to overcome this limitation [105, 104]. Nevertheless, the RTL descriptions present an explosion in the logical gate count and a significant reduction in throughput performance due to the complexity of the resulting code.

SOFTWARE SYNTHESIS To illustrate the difficulties to provide efficient implementations of dynamic dataflow descriptions, Figure 21 presents the impressive evolution of the performance of the most studied video decoder within the RVC framework on desktop mono-processors, namely the normative description of the decoder implementing the MPEG-4 Part 2 Simple-Profile standard. We present frame-rates of a small QCIF video because of its extensive utilization in the literature.

Most of the past work on software synthesis within the RVC framework has studied the implementation of applications on GPP [177, 86**?**]. But, in view of the current market of electronic products, we need to focus on embedded systems and especially MPSoC-based platforms. Thus, new constraints, like power consumption, have to be taken into account.

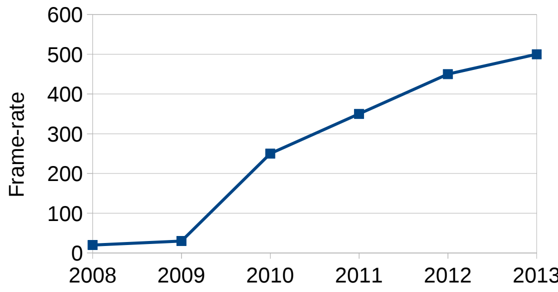


Figure 21: Performance evolution of an RVC-based video decoder. Frame-rates of the *foreman* sequence (QCIF) using the *normative description* of the MPEG-4 Part 2 Simple-Profile decoder executed on *mono-processor* desktop computers [151, 180, 93].

4.7 CONCLUSION

This chapter presents a development framework introduced by MPEG under the name of RVC. This framework was initially proposed to overcome the limitations of the standardization process of video compression format. Thanks to dataflow programming, organized on top of the RVC-CAL language, the RVC framework proposes a flexible development process that produces modular, scalable and portable applications. These advantages make RVC suitable for the development of any multimedia application that manipulates rich media contents.

The next chapters explore the programming of embedded multi-core platforms by the way of the RVC framework. Chapter 5 starts by introducing the development environment, in other words the design flow from the application description to the executing platform. Then, Chapter 6 proposes an implementation of dynamic dataflow programs optimized for embedded multi-core platforms. Finally, we describe a set of actor mapping and scheduling strategies that can handle the dynamism of our applications to provide efficient execution.

Part II

CONTRIBUTIONS

5 ADVANCED DEVELOPMENT ENVIRONMENT FOR DATAFLOW PROGRAMMING

Remember that all models are wrong;

the practical question is
how wrong do they have to be
to not be useful.

— George E. P. Box [35]

The development and the implementation of multimedia applications, such as video codecs, are time-consuming and error-prone tasks due to the increasing complexity of the algorithms as well as the increasing variety of the multimedia devices. In fact, the progression of parallel computing as the only alternative to meet the performance requirement has made the development and the implementation of the applications even harder. As we have seen in Chapter 2, parallel computing has not only introduced programming challenges but also architectural and executional challenges (synchronizing the different tasks, balancing their loads, etc). Consequently, the need for efficient development methods and tools is becoming increasingly important so as to meet the requirement of time-to-market.

This chapter describes an IDE dedicated to dataflow programming that aims to make the development of dataflow programs easier, especially for embedded multi-core platforms. Starting from the initial work of Wipliez who has created a compilation infrastructure for dataflow programs [177] targeting GPP, ASIC and FPGA, the main contributions of this chapter are:

1. Enhancement of the compilation infrastructure by the way of modern software engineering techniques such as meta-modeling. A compiler is a complex piece of software that requires reliability and robustness offered by Model-Driven Engineering (MDE).
2. Introduction of an architecture model for embedded multi-core platforms dedicated to dynamic dataflow programs.
3. Extension of the compilation infrastructure with an entire co-design flow that targets embedded multi-core platforms based on our dedicated architecture model.

This chapter is organized as follows. We start in Section 5.1 by describing how the compilation infrastructure proposed by Wipliez [177] can be enhanced by MDE. Then, we introduce in Section 5.2 our architecture model that aims to design embedded multi-core platforms dedicated to the execution of RVC-based dataflow programs. Finally, we present an extension of the compilation infrastructure for specifically targeting our dedicated architecture model in Section 5.3.

5.1 ENHANCED DATAFLOW-SPECIFIC COMPILATION INFRASTRUCTURE

Compilers and programming languages are the foundation of software engineering which is now present in our whole society. In the last 50 years,

the field of compiling was focused on the translation of high-level language programs into efficient machine code. However, the increasing complexity of software and machines has raised new challenges such as parallel programming or reliability of complex systems [96].

Starting from the initial work of Wipliez [177], we propose an enhanced compilation infrastructure for dataflow programs that takes advantage of meta-modeling and aspect-oriented programming. The contributions of this section are:

- Building of the whole compiler infrastructure upon meta-tools.
- Maximizing of code reutilization thanks to the implementation of a unified graph API.
- Separation of dataflow and procedural concerns within our compilation infrastructure.
- Formal specification of our enhanced dataflow-specific Intermediate representation (IR) by the way of meta-modeling.

Please notice that most of the implementation work associated with the contributions described in this section has been made jointly with Matthieu Wipliez.

5.1.1 *Multi-Target Compilation Infrastructure*

The compilation infrastructure for dataflow programs, included in Orcc toolset [134], on which we have worked during this thesis is introduced by Figure 22. Started by Wipliez [177], this compilation infrastructure is a *trans-compiler*, also called a *source-to-source compiler*, that basically translates dataflow descriptions into more traditional source codes, instead of generating directly machine codes like many compilers. But, the compilation flow stays very similar to traditional compilers [16] and may be summed up by the following three steps performed consecutively:

1. **Front-end** (*orcc-fe*): The input program, written in a textual language, is parsed and translated into an Abstract Syntax Tree (AST). The AST is then transformed into another IR that allows fast analysis and advanced optimizations using complex data-structure. During this step, the front-end performs semantic validation, type inference, and expression evaluation.
2. **Middle-end** (*orcc-core*): Target-independent optimizations are repeatedly performed as transformations on the IR. Since IRs aim to make easier optimizations, the same program can be successively described with several IRs during its optimization.
3. **Back-end** (*orcc-be*): Target-specific optimizations are finally performed before the code generation, which translates the optimized IR into the targeted source code. In fact, a DSL is often translated into general-purpose programming languages to benefit from the power of industrial-level compilers. The interaction with target compilers is simplified by the generation of build scripts along with the generation of the source code.

The IR is the key data structure of any compilation infrastructure in the sense that analysis and optimizations are applied on it. Actually, one of the

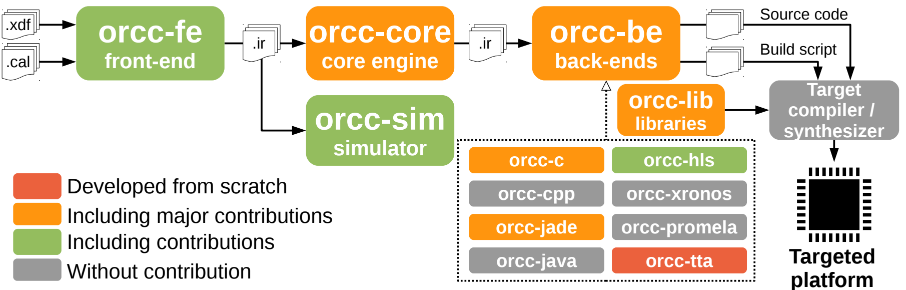


Figure 22: Multi-target trans-compilation infrastructure

shortcoming problems of compiler design is the increasing complexity of the IRs since more and more information is required to perform advanced optimizations [106]. Similarly to DSLs, domain-specific IRs address a part of this problem by focusing on the specificity of the application domain and, additionally, by breaking down complex structure into smaller pieces.

In his thesis [177], Wipliez describes the implementation of an IR dedicated to dataflow compilers and shows naturally that such a domain-specific IR is well suited for performing advanced analysis and optimizations on dataflow programs. Now, we show how modern software engineering techniques, such as meta-modeling and aspect-oriented programming, can improve the manipulation of this IR and the whole compilation flow.

5.1.2 *Model-driven Compilation Infrastructure*

The use of meta-models and MDE technologies speeds up the software development by automating time-consuming and error-prone tasks:

- **Maintainability:** The global homogeneity of the software is increased thanks to the generative approach of the meta-tools.
- **Documentation:** Models are inherent source of documentation, equivalent to UML.

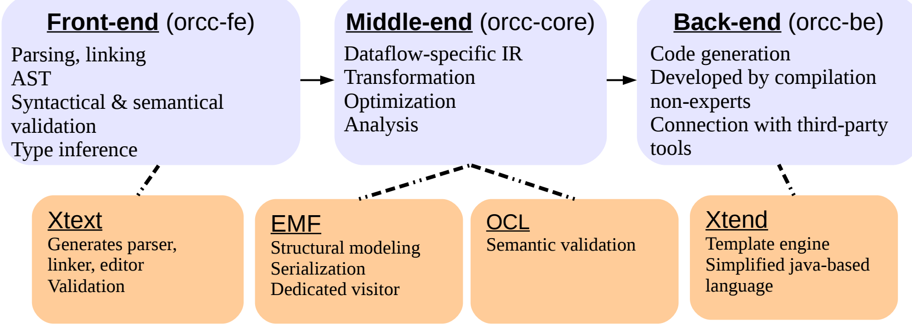


Figure 23: Compilation flow based on meta-tools

Meta-modeling also offers many advantages for a compiler infrastructure [76, 106]. As presented in Figure 23, the three consecutive steps of our compilation flow are built upon specific meta-tools that aim to solve their particular problematics:

ORCC-FE has been implemented on top of Xtext [66], a framework dedicated to the development of DSL that generates parser, linker and editor from the grammar of the language.

ORCC-CORE is build upon our dataflow-specific IR, which has been modeled with the Eclipse Modeling Framework (EMF) [160], an open-source framework implementing the Object Management Group (OMG) specifications. EMF offers many useful methods for manipulating a data structure thanks to the containment relationship. EMF also offers the automatic serialization of the IR allowing incremental compilation.

Additionally, the model describing the IR is annotated with a set of constraints , expressed in Object Constraint Language (OCL), that guarantees its semantic validity. An OCL constraint may be an invariant that must be valid between each transformation. An OCL constraint may also be a post/pre condition that must be verified before/after the realization of an operation. Theses constraints are transformed automatically into the equivalent Java code.

ORCC-BE realizes the code generation using Xtend [65], which provides a flexible template-based code generation approach that is accessible for non-expert in compilation, thanks to a simplified programming language based on Java and fully integrated within Eclipse, while providing an efficient code generation.

In fact, the definition of models allows the developer to skip most of the implementation details and to focus on the specification. Additionally, the modeling approach also forces the developer to deeply specify the model without leaving out any details.

5.1.3 *Unified Graph Library*

Most data structures in compilation are related to graph theory, this is especially true for the compilation of dataflow programs. Consequently, we choose to develop a graph library containing a unified model of graph (Figure 24) and the implementation of state-of-the-art algorithms to increase the code reutilization and speed-up the development of new features.

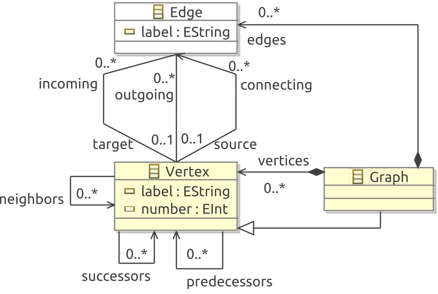


Figure 24: Class diagram of Graph

Our model of graph, presented in Figure 24, is composed of three classes:

GRAPH is the top-level class of the model. A Graph object contains a list of vertices and a list of edges. Since dataflow graphs are naturally hierarchical, the Graph class inherits from Vertex.

VERTEX is a class that describes one kind of element of a Graph object. A *vertex* references a list of *incoming* and *outgoing* edges from which it deduces a list of *successors* and *predecessors*.

EDGE is a class that implements the directed edges of the graph. An edge references its *source* and *target* vertices.

The library implements a set of algorithms for searching in a graph following well-known strategies, such as Breadth-First Search and Depth-First Search [175], for finding strongly connected components, or additionally for computing the dominator or the post-dominator of a given vertex.

5.1.4 *Separation of Concerns*

We have chosen to divide the IR into dataflow aspect and procedural aspect. Whereas dataflow modeling naturally separates networks from actors; the analysis and transformations performed on dataflow programs by the compilers do not usually respect such a separation. On the one hand, dataflow compilers can perform procedural analysis and transformation just as general-purpose compilers [177, 16]. On the other hand, compilers can really benefit from the formalism upon dataflow MoCs by performing dataflow-level analysis and transformation [179, 32, 105]. Thus, our dataflow-specific IR is likewise divided in two distinct models:

- The low-level model that contains the classical procedural description including the instructions, expressions, blocks or even the Control-Flow Graph (CFG), and
- The high-level model that is related to the dataflow information, such as the interconnection between the components or their production/-consumption rates.

Moreover, such a separation of concerns within our IR makes easier its manipulation by the meta-tools, knowing that one of the major limitation of MDE is the scalability of meta-tools [106], i.e. their efficiency on large models.

5.1.5 *Procedural Aspect of the Intermediate Representation*

The procedural aspect of our dataflow-specific IR describes the computation, in the sense of the computational step of the imperative programming paradigm. This aspect is composed of the following classes:

PROCEDURE is the top-level class of the procedural level of our IR that corresponds to a piece of program (Figure 25). A procedure has a *name* and is composed of an ordered set of *blocks*, and a set of local *variables* without the notion of the scope.

VAR is a class that implements the concept of variable (Figures 25 and 26), i.e. the association of a storage location and an identifier. A variable has a name, a type and may be assignable. A variable refers to its definitions as well as its utilizations. A variable may be local or global according to its containment.

PARAM/ARG are two classes (Figure 25) used to implement the parametrization of an object (procedures, actors, networks, etc). In fact, the parametrization is a useful mechanism to increase the re-usability within the code.

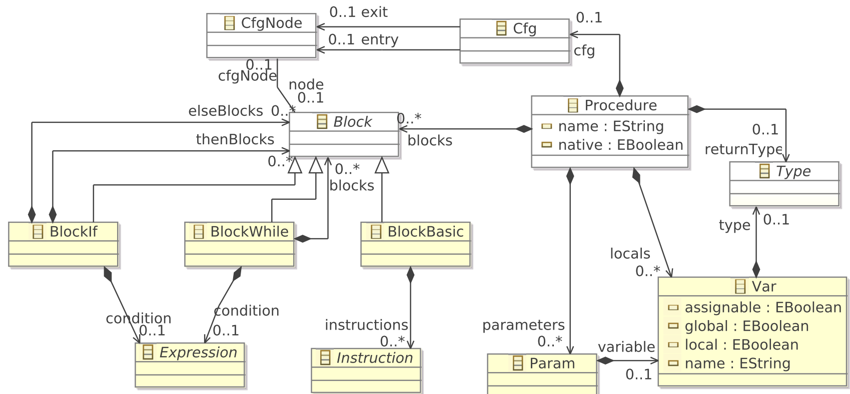


Figure 25: Class diagram related to Procedure

For instance, a parametrized actor may be more easily reused in several descriptions.

USE/DEF are two classes that model respectively the utilizations and the definitions of the variables (Figures 26 and 27). To do so, the two classes refer to a unique variable. On the one hand, a *def* is created every time an instruction sets a variable. On the other hand, a *use* is created when the value of a variable is required by an instruction. This kind of variable management is commonly used to make easier the static analysis and transformations of procedural code. Def/use are, for example, heavily used to convert a code into its Static Single Assignment (SSA) form [16].

BLOCK is an abstract class that describes a common program structure (Figure 25). The blocks organize the sequences of instructions in respect to the semantic of the program. There are three subclasses of Block:

- BlockBasic is the simplest structure that contains a list of ordered *instructions* without branching.
- BlockIf describes conditional structures. Such a block contains an expression, the *condition*, and two ordered lists of Block objects (*thenBlocks* and *elseBlocks*) as well as a specific basic block, called *joinBlock*, used by the SSA form.
- BlockWhile describes similarly loop structures.

CFG is a direct subclass of Graph that describes the control flow within a procedure (Figure 25). The vertices of the graph refer to the basic blocks, and the edges of the graph describe the branching between the blocks. Such a representation is useful to perform many static analysis and code optimizations.

We have chosen to not directly use the graph as the structural representation of the control-flow of a procedure in our IR considering the difficulty to maintain the graph all along the compilation flow. Consequently, we have chosen to model the control flow of a procedure using the control blocks as well as the block hierarchy from which the *cfg* may be directly deduced.

INSTRUCTION is an abstract class that describes a set of statements that can be performed within a procedure (Figure 26). The subclasses of Instruction are:

- InstAssign that simply describes the assignment of an expression to a local variable.
- InstLoad and InstStore that describe the accesses to variables that require memory (i.e. state variables and arrays), and may refer to an indexing expression in case of an array.
- InstCall and InstReturn that model function-specific instructions.
- InstPhi that is a special instruction required by the SSA form, closely related to a conditional assignment.

Actually, the distinction between memory accesses (load/store) and temporary assignments is very important in a trans-compiler due to the variety of targets. For instance in hardware languages like VHDL, the accesses to registers and Random Access Memory (RAM) are totally different.

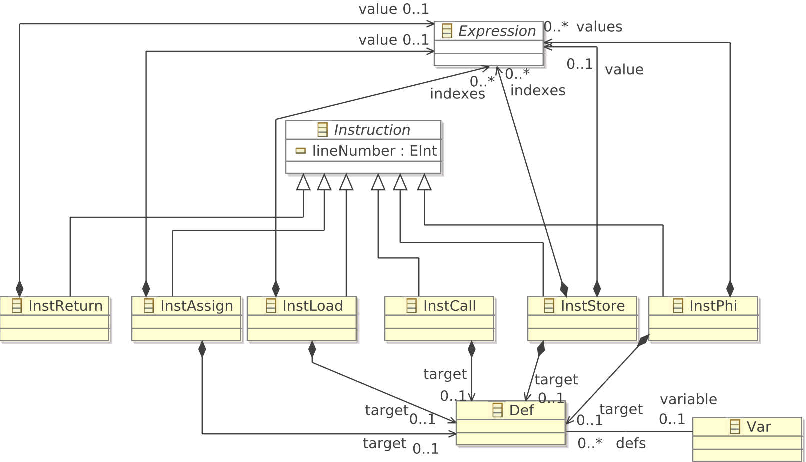


Figure 26: Class diagram related to Instruction

EXPRESSION is an abstract class that models the evaluation of a combination of constants, variables, values, operators and functions (Figure 27). The subclasses of Expression are:

- ExprVar that models the evaluation of a variable or a constant. Such an expression contains a *use* that refers to the associated variable/constant following the Use/Def mechanism.
- ExprInt, ExprString and so on that contain directly a value of the corresponding type.
- ExprCall that models the evaluation of a function. Such an expression contains a list of *arguments* that are required to evaluate the referenced *procedure*.
- ExprUnary and ExprBinary that model the evaluation of a contained expression (respectively a combination of contained expressions) according to a contained operator.

TYPE is the upper abstract class of our type system, which is limited to compile-time type inference in order not to reduce the portability of the applications [178]. Our type system is composed of 7 subclasses:

- TypeInt and TypeUint, both for the type of the integers.
- TypeBool for the booleans.

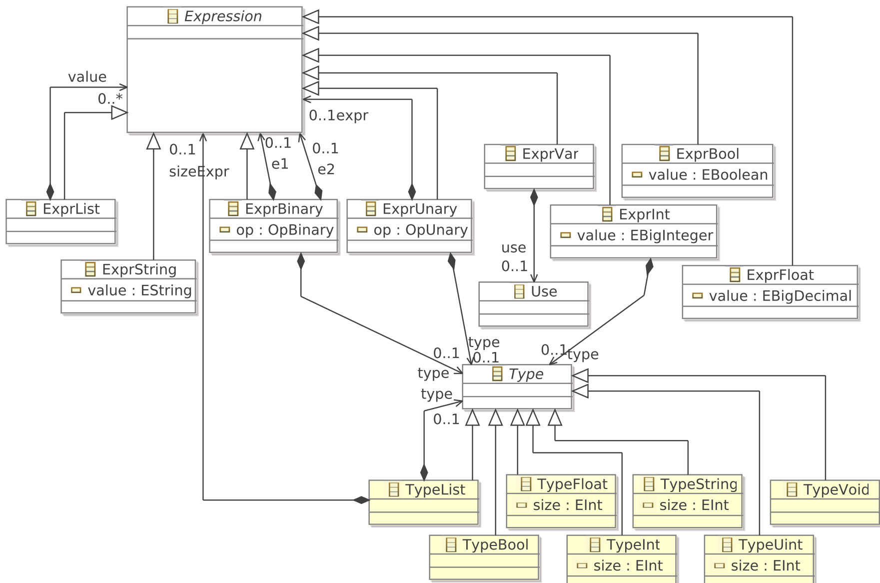


Figure 27: Class diagram related to Expression and Type

- TypeFloat for the floating-points.
- TypeString for the sequence of characters.
- TypeList for the lists.
- TypeVoid for the procedures that do not return any value.

As opposed to general purpose IR, our dataflow-specific IR does not provide a pointer type. In fact, we argue that the use of pointers is an answer to the lack of functionality of the type system in the application domain, which should not occur with well-designed domain-specific IRs. In addition of being a well-known source of bug, the use of pointer requires complex pointer analysis that can ultimately lead to the inefficiency of the optimizations [97].

The procedural aspect of our IR is generic, which means that nothing related to dataflow programming is included. But, the procedural aspect of our IR is also restricted to our application domain, especially the type system. All of this makes that this aspect of our IR can be considered as a subset of general-purpose IRs used in industrial compilers. While general-purpose IRs, such as the one used in Low Level Virtual Machine (LLVM) [116], mainly inherit from compilation experiences, our IR benefits from both compilation and MDE [106].

5.1.6 *Dataflow Aspect of the Intermediate Representation*

Applications that are developed using dataflow programming are composed of additional pieces of information in comparison with classical programming:

NETWORK is the top-level class of the dataflow level of our IR, which inherits directly from the Graph class (Figure 28). A *network* has a *name* and contains two sets of ports, *inputs* and *outputs*. A network also contains a set of *connections* and a set of vertices called *children*, that may be constituted indifferently by Entity or Instance.

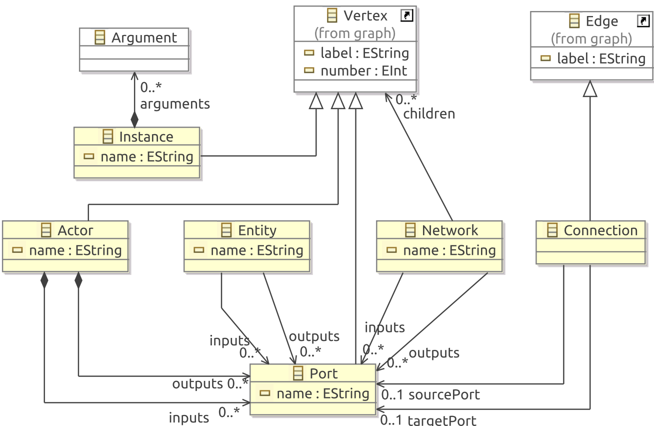


Figure 28: Class diagram related to Network

ENTITY is a class that simply contains a set of input and output ports that may be view as a superclass between Actor, Network and Instance. In fact, an *entity* is used to make easier the instantiation of Actor and Network.

CONNECTION is a class that inherits directly from Edge. A *connection* models the communication channel between two *entities* of the network. Additionally to its *source* and *target*, a connection refers to its *sourcePort* and *targetPort*, as well as its *size*.

ACTOR is a class that represents the basic component of a dataflow program (Figure 29). An *actor* contains two sets of ports, *inputs* and *outputs*, that defines its interfaces. An *actor* also contains a set of *procedures*, a set of variables called *stateVars*, and a set of *actions* ordered according to their priorities. An actor may also contains an *fsm*, used to schedule its actions, and a set of *parameters* to increase its re-usability.

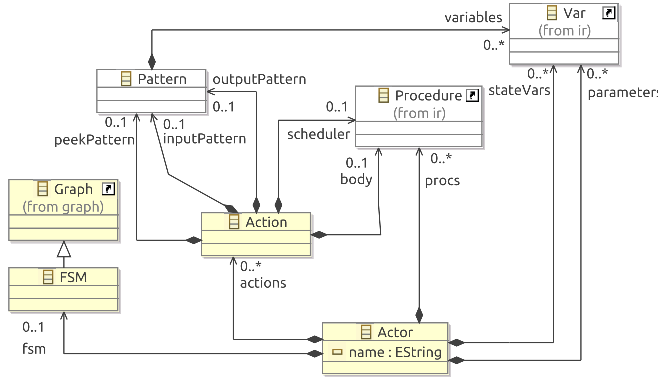


Figure 29: Class diagram related to Actor

INSTANCE is a subclass of Vertex. Instance is useful to reference a single entity, network or actor, several times in the same description without duplicating it. An instance may refer a set of arguments if the referenced entity is parametrized.

PORT is a class that implements the external interface of Entity, Actor or Network. A *port* has a *name* and a specific *type*.

FSM is a direct subclass of Graph that implements a FSM by representing states by vertices and transitions by edges. Here, an *fsm* describes, in a practical manner, a partial order between the action executions. Using an FSM to describe the action scheduling is not only a practical structure of the programming language, it allows a factorization of the generated code as well.

ACTION is a class that implements the firing function. An action is composed of two procedures: Its *body* that models the processing of the action, i.e. the firing function, and its *scheduler* that describes its guard, i.e. the firing rule. An action also contains three patterns: An *inputPattern* and an *outputPattern* that correspond to the token production/-consumption of the action, as well as a *peekPattern*[1] that corresponds to the amount of tokens that has to be validated by the *scheduler*, i.e. the guard.

PATTERN is a useful class to abstract the use of communication channel inside the procedural code. In fact, a *pattern* describes a mapping between the input/output *ports* of an actor and the procedural *variables* that will contain the consumed/produced/accessed tokens by an action. A *pattern* also describes the amount of tokens that are concerned.

As a result, communication-specific functions are no longer required in the procedural code, neither access functions (*read*, *write* and *peek*[1]) nor conditional function (*hasRooms* and *hasTokens*), so that the compiler may perform procedural analysis and transformation without taking into consideration the dataflow aspect.

Actually, the genericity of our IR makes it usable to describe most of the dataflow MoCs (DPN, SDF, etc). This is put in evidenced by the fact that our IR is the central data structure of two dataflow-based IDEs, Orcc (Section 4.5.2) and Synflow studio (Section 3.7), using different dataflow MoCs to target different usages.

5.2 ARCHITECTURE MODEL FOR DEDICATED EMBEDDED MULTI-CORE PLATFORMS

After describing the global structure of our dataflow-based compiler, we focus on a particular target, the main subject of this thesis, that is the embedded multi-core platforms. The development of a design flow targeting such platforms requires the definition of an architecture model that matches the behavior of the targeted platform, while keeping a high-level of abstraction and enough configuration options to allow Design-Space Exploration (DSE). Alternatively, architecture models can be presented as customizable multi-core processor templates [100] that setup the main architectural aspects.

Considering the complexity of multi-core architectures, together with the efficiency and the reliability required by embedded systems, we propose to specialize our architecture model for the execution of dynamic dataflow programs in order to take advantage of the knowledge inherent to our application domain, similarly to DSL. This section makes the following contributions:

- The introduction of Transport-Trigger Architecture (TTA) as the inner architecture of the processors used to execute dataflow actors.

1 *Peeking* refers the reading of a token without consumed it. *Peeking* is an operation required by the DPN model to describe the firing rules.

- A set of predefined configurations of the processors to simplify the DSE.
- A dataflow-specific memory architecture in order to overcome the *memory wall* often reached by dataflow programs.

5.2.1 *Processor Architecture*

The processor cores underlying our abstract platform is based on a VLIW-style architecture known as TTA [55]. TTA was chosen for the following reasons:

- **Instruction-Level Parallelism:** TTA processors are able to take advantage of the only type of parallelism which is not inherent in dataflow model. TTA processors resemble VLIW processors in the sense that they fetch and execute multiple instructions each clock cycle. A major difference, however, is that TTA processors have only one instruction: *move*, which simply transfers data from a processor internal place to another one.
- **Embedded processors:** TTA processors are ideal for targeting embedded systems. Corporal states that direct programming of the data transports reduces the register file traffic when compared to VLIW [55], but however makes the compiler design quite challenging, as it is the compiler that schedules the data transports and makes sure conflicts are avoided. Since the compiler makes these decisions at design time, the run-time system is simplified and hence there are savings on the processor gate count and energy consumption.
- **Flexible architecture:** TTA processors are extremely configurable. The designer can make the processor tiny and energy-efficient or, if needed, increase the instruction-level parallelism of the processor. We present 4 predefined configurations in Table 5 that have been used during the experiments.

As an example, Figure 30 presents a simple TTA-based processor composed of two buses, two Arithmetic and Logic Units (ALUs), one Register File (RF), one Load/Store Unit (LSU) (to manage RAM accesses) and one control unit connected to the Read-Only Memory (ROM) containing the instructions. Like most modern processors, TTA processors are based on the *Harvard* architecture that physically separates storage and pathway for instructions and data.

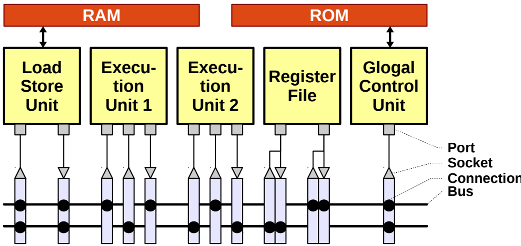


Figure 30: A simple processor based on Transport-Trigger Architecture

5.2.2 *Predefined Configurations of Processors*

Table 5 presents 4 predefined configurations of TTA-based processors used during our experiments (respectively *Standard*, *Custom*, *Fast* and *Huge*). The configurations characterize internal aspects of the processors such as the number of FUs, ALUs, multipliers and LSUs, the number of integer and boolean RFs as well as the number of registers they contain, and the number of buses that interconnect all together FUs and RFs. The connectivity of the interconnection network is also characterized as *Full* or *Custom*. While a *Full* connectivity does not limit the data movement between FUs and RFs, a *Custom* connectivity avoids the decrease of the clock frequency when the complexity of the interconnection network increases.

The first one, called *Standard*, is almost equivalent to a RISC processor: inside the TTA processor the interconnection network is composed of 3 buses that can provide two operands to the FU at each clock cycle and move the result when it is available. The 3 last configurations, *Custom*, *Fast* and *Huge*, define larger processors composed of several FUs and buses able to take advantage of the instruction-level parallelism of the application (like a VLIW processor). Concerning the *Huge* configuration, its characteristics are deliberately over-sized to acquire the maximal performance, so this configuration is only used in simulation purposes.

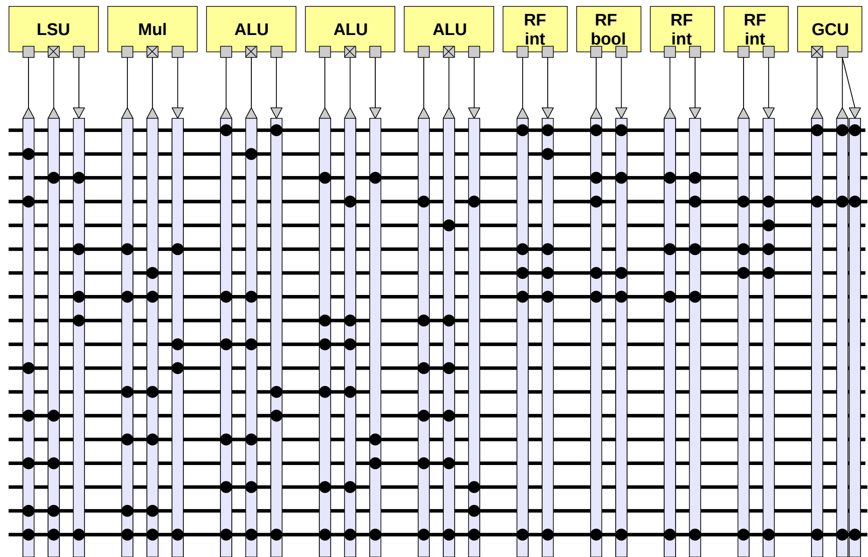
Processor	*Standard*	*Custom*	*Fast*	*Huge*
ALUs	1	2	3	12
Multipliers	1	1	1	8
LSUs	1+	1+	1+	2+
Integer RFs (32 bits)	2x12	3x12	3x14	8x32
Boolean RFs (1 bit)	1x2	1x2	1x6	1x6
Buses	3	6	18	32
Connectivity	Full	Full	Custom	Full

Table 5: Comparison of 4 predefined processor configurations

The *Fast* configuration, introduced in [71] and presented in Figure 31a, provides clustered TTA-based processors that can reach high-frequency on FPGA with large potential of parallel computing. Table 31b presents a comparison of a *Fast* TTA-based processor with the well-known softcore architectures Xilinx Micro-Blaze and Altera NIOS II.

5.2.3 *Dataflow-specific Memory Architecture*

Now, we introduce an hybrid memory architecture specially designed for dataflow programs. To limit the traditional memory bottleneck, our architecture model contains both shared and private memories, as shown in Figure 32, making the memory architecture a mixture of UMA and NORMA organization (see Chapter 2 for the definitions). Thus, the processors ($P_1, ..., P_k$) have their own private memories ($M_1, ..., M_k$) used for executing their actors, but the processors are also connected, through an interconnection network, to a set of shared memories ($S_1, ..., S_n$) devoted to inter-processors communications.



(a) Predefined processor configuration built on top of a clustered interconnection network

FPGA	Softcore	FMax	LUTs	Reg
Xilinx Virtex 5	TTA	192	5218	2785
	MB (3)	169	1537	1318
	MB (5)	195	1889	1841
Altera Stratix II	TTA	148	5024	3485
	Nios	175	2322	1896

(b) Comparison with well-known softcore architectures (from [71])

Figure 31: *Fast* TTA-based processors target high clock-frequency implementation [71].

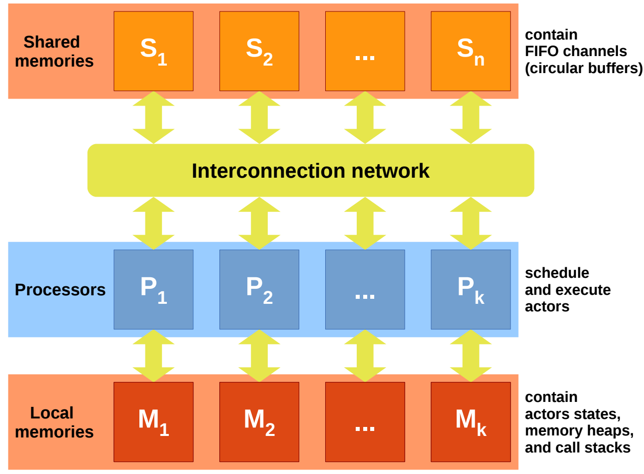


Figure 32: An hybrid memory architecture dedicated to DPN-based programs

Modeling multi-core platforms dedicated to the execution of DPN-based programs [121] allows us to make the following assumptions:

- Actors can only communicate through communication channels. Thus, shared memories do not need to store data apart from the content of FIFO-based communication channels, implemented as circular buffers that are detailed later in Section 6.2.
- The DPN model allows stateful actors. Thus, private memories may have to store the current states of the actors that are assigned to the processor to which they are related. Additionally, private memories have to store the heap and the call stack used during the execution of the actions just as traditional programs.

Furthermore, storing communication channels in shared memory increases the flexibility of the design flow. Knowing that a single memory component can contain multiple channels, the compiler has to assign not only actors to processors but also FIFO channels to memory components. Actually, FIFO channels can be freely mapped to memory components since they are not dependent from each other. But, some architectural constraints may have to be considered, such as the topology of the interconnection network or the size of the memory components.

5.3 DATAFLOW COMPILER FOR EMBEDDED MULTI-CORE PLATFORMS

The difficulty of efficiently programming embedded multi-core platforms, as presented in Chapter 2, still makes the design process an open challenge. This section presents an automated co-design flow, designed from scratch during this thesis, that intends to implement DPN-based programs onto dedicated embedded multi-core platforms. This co-design flow has been used to perform most of the experiments presented in the next chapters, making it a key component of this thesis [5]. To summarize, the contributions of this section are:

- The extension of our compilation infrastructure for embedded multi-core platform by the interfacing with a co-design toolkit dedicated to ASIP.
- The setup of an advanced simulation process that benefits from dataflow modeling to facilitate the debugging and analysis of applications onto embedded multi-core platforms.

This section is organized as follows: First, the global co-design flow is introduced; Then, the hardware design flow that generates the HDL description of the platform is detailed; Next, the software compilation flow that generates the binaries for each processor is described; And, finally, a description of the simulation infrastructure is given.

5.3.1 *Multi-stage Co-design Flow*

Like most of design approaches of embedded multi-core platforms, our design flow follows the Y-chart [113] (Figure 33) that separates the specification of the application, the platform and the mapping. Such a separation of concerns facilitates the design space exploration by varying some of the aspects while fixing the other aspects. For example, the application can be parallelized to partition its components more equitably without modifying the platform configuration.

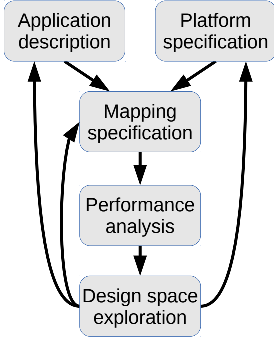


Figure 33: Designing embedded multi-core platforms following the flexible Y-chart approach [113]

The co-design flow is implemented around two open-source projects known as Orcc [134] and TTA-based Co-design Environment (TCE) [166]. In fact, Orcc can be considered as a dataflow front-end for TCE, and inversely TCE can be considered as a processor-specific back-end for Orcc. Orcc performs the high-level stage of the design flow and provides a functional simulator, and both are entirely independent from the architecture of the processors. For its part, TCE performs the low-level stage of the design flow and provides an instruction-set simulator.

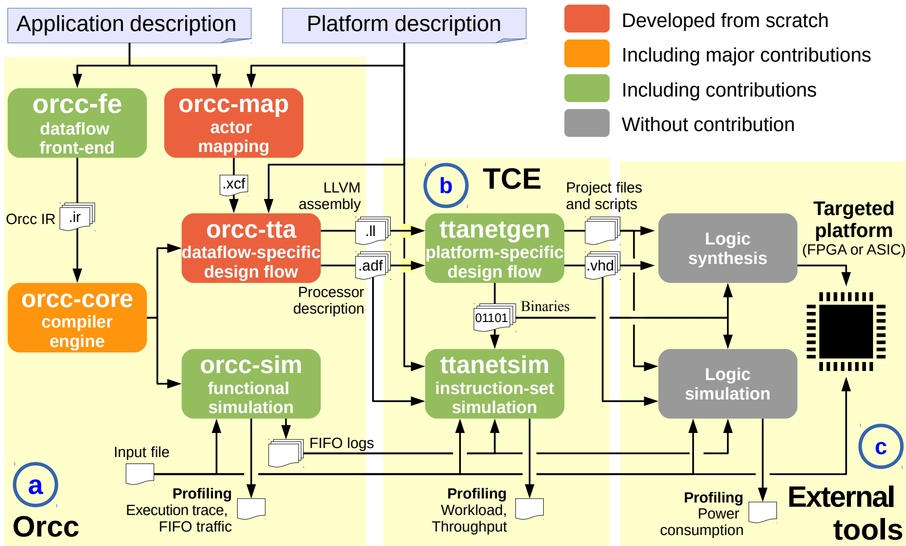


Figure 34: Multi-stage co-design flow

As shown in Figure 34, our co-design flow is multi-stage. First, the dataflow-specific stage (a) is implemented in Orcc, our dataflow compiler, and composed of 5 different parts:

ORCC-FE & ORCC-CORE are the initial steps in our trans-compiler, performed before all back-ends. The application description (provided in RVC-CAL and XDF) is translated into our dataflow-specific IR, which is then analyzed and transformed to be finally ready for the next steps, depending on the targeted platform.

ORCC-TTA is the high-level step of the design flow. This step transforms our dataflow-specific IR into a general-purpose IR (LLVM IR) that can be understood by the target compiler. This step also generates XML-based processor description files, known as Architecture Definition File (ADF) [50], according to the initial specification of the platform.

ORCC-MAP is the process that assigns a processor to each actor. The actor mapping has been implemented independently from the compilation step to make it usable at any stage of the design flow (even at run-time). The actor mapping is based on the heuristics described in Chapter 7.

ORCC-SIM is a functional simulator that interprets our dataflow-specific IR. Such a functional simulator enables a quick validation of the application, without requiring any code generation or third-party tools. The simulator can also produce useful profiling data such as the logs of the FIFO channels or the execution trace.

Then, the processor-specific stage (b) is implemented in the TCE toolkit, and composed of two parts:

TTANETGEN is the low-level stage of the design flow. This stage both transforms the general-purpose IR into instructions that can be executed by the targeted platform, and generates the HDL description of the whole platform from its high-level description.

TTANETSIM is an instruction-set simulator for TTA-based multi-core platforms. *ttanetsim* can either simulate the execution of the whole platform according to an input file, or simulate a single processor in a standalone fashion using the logs of FIFO channels previously recorded with *orcc-sim*.

And finally, the hardware-specific stage (c) is performed by third-party tools but automatized by a set of scripts and project files generated by the previous stage:

LOGIC SYNTHESIS is the last step of the design flow. In fact, the logic synthesis aims either the generation of an FPGA bitstream or the creation of an ASIC.

LOGIC SIMULATION is the lower level of simulation: It can be performed at varying degrees of physical abstraction (transistor level, gate level, or register-transfer level). Similarly to cycle-accurate simulation, the logic simulation can be performed either on the whole platform using an input file, or on a standalone processor using the FIFO channel logs.

The development of this dataflow-based co-design flow has involved some contributions in both tools (Figure 34).

5.3.2 *Hardware Synthesis*

The hardware synthesis simply transforms the specification of the platform into a synthesizable HDL description of multi-core systems. To this end, the sizes of the memories have to be estimated in order to correctly instantiate the corresponding RAM blocks. Since our architecture model is built on memory-based interconnection, there are two types of memory:

- The shared memories that interconnect the processors. A shared memory simply contains the communication channels that connect two actors mapped onto two different processors, i.e. the buffers and the read/write pointers.
- The local memories that are connected to their own processor. A local memory may contain the state of the actors that are mapped on, the buffer of the internal communication channels, as well as the stack and the heap.

Then, the HDL description of each processor is generated from its high-level description by the TCE using a pre-existing database of standard hardware components.

5.3.3 *Software Synthesis*

The software synthesis compiles dataflow-based programs into instruction codes that are executable on the associated embedded multi-core platforms. In contrast with the hardware synthesis, the flow can really be decomposed in two successive steps (Figure 35):

1. A first step that translates the whole dataflow description into a procedural IR, low-level but still target independent, which has been developed for the LLVM project [116]. This step is performed by our dataflow compiler, described in Section 5.1. In fact, the compiler partitions the dataflow application over the platform according to the mapping specification, generating a separated program for each processor.
2. Then, a second step that successively compiles the program of each processor from LLVM IR into processor instructions thanks to the processor description: This makes the whole application executable on the embedded multi-core platform. This step is performed by the compiler of the targeted processor, thus preventing it from being platform agnostic.

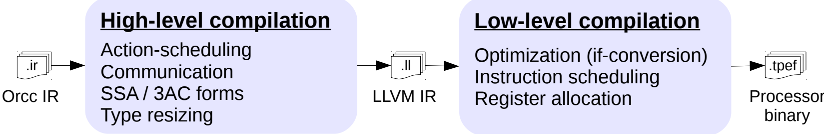


Figure 35: Two-step compilation flow

The LLVM IR provides flexibility and type safety, low-level operations, and also permits the proper representation of most of the high-level programming languages. Additionally, the IR respects the SSA and Three-Address Code (3AC) properties to improve the results of code analysis and optimizations by simplifying the properties of variables.

Our dataflow-specific IR is translated into an equivalent procedural IR in order to be compiled for the processor. To do so, we perform several sophisticated transformations that are explained below:

1. **Communication**: The operations related to FIFO channels (*read*, *write*, *peek*) are instantiated to the procedural IR. More details about our implementation of communications are given in Chapter 6.

2. **Low-level form**: A total transformation procedure enables us to make LLVM IR representations out of our dataflow-specific IR by respecting properties such as SSA and 3AC. This procedure consists of variable indexing, ϕ-function addition and splitting of complex expressions to multiple primitive instructions.

3. **Correct handling of word-lengths**: Our dataflow-specific IR allows the designer to express bit-accurately the word-length of each variable and communication channel. The respective property is also found in the LLVM IR. However, when a computation has to be performed with two variables of different word-lengths, the correct result must be ensured by the use of an explicit *cast* instruction.

4. **Action scheduler**: In our dataflow-specific IR, the scheduling of actions is expressed by the use of an FSM and priorities between the actions. We need to express the action scheduler in a procedural way to make it understandable by the compiler. More details about our implementation of action scheduling are given in Chapter 6.

After applying these fundamental transformations, the resulting LLVM IR representation is suitable first for the target-independent powerful optimizations of the LLVM compiler, and then for the specific optimizations of the TTA compiler (included in *ttanetgen* in Figure 34).

5.3.4 *Simulation Infrastructure*

Much of the difficulties of adopting embedded multi-core platforms is due to the following reasons:

- **Debugging** of parallel hardware is very difficult when compared to debugging of software. Execution tracing of hardware blocks is very limited when compared to the tracing of software executions.
- **Performance analysis** at platform level is very difficult. Based on the performance of individual blocks, it is impossible to tell anything about the performance of the whole platform.
- **System integration** for embedded multi-core platforms is a slow and error-prone process.

Our co-design flow tackles these difficulties by offering an advanced simulation process that eases the debugging and the profiling of the application during its integration on the platform. In fact, we take advantage of dataflow properties, such as the strong encapsulation of components and the exposed communications, to simulate *by pieces* the application, i.e. simulate each actor in a standalone fashion.

FUNCTIONAL SIMULATION The functional validation can be performed directly from the development environment (i.e. Eclipse) using *orcc-sim* that simply interprets our dataflow-specific IR generated by the front-end. Knowing that the interpretation of complex application can be time-consuming, the functional validation can also be performed by compiling the application for the host platform (i.e. developer's computer) using *orcc-c* in order to allow fast executions. Actually, the developer can easily validate his application with both methods by displaying texts, images or videos.

Apart from the validation, this early simulation phase can be used to record the logs of the FIFO channels, i.e. the value of the tokens that go through the FIFO channels. Thus, the actors composing the application can be executed independently from each other in latter simulation phases. The simulator can also build execution traces [37], that are used by tools like Turnus to explore the application design, or to profile the communication rates within the application.

PLATFORM SIMULATION In a latter phase of development, the developer can simulate the application execution on the targeted platform to get precise profiling information. Our co-design flow offers a two-level simulation process that can be used with the instruction-set simulator included in TCE, known as *ttanetsim* [99], and with any logic simulator (e.g. Mentor Graphics ModelSim):

- **Platform level:** The whole design is simulated to check the functionality of the application including the communications between the processors. This enables us to evaluate the global performance of the system including the synchronization between the processors.
- **Processor level:** Each processor can be tested independently from the others, as presented in Figure 36, using the logs of the FIFO channel that have been previously produced. To do so, the FIFO simulators ($F_1, ...F_m$) read the log files and write the shared memories ($S_1, ...S_n$), supplying the simulated processor (P_i) just as incoming communications from other processors. And, inversely, the outgoing communication data produced by the simulated processor (P_i) are automatically consumed by the FIFO simulators ($F_1, ...F_m$) and compared to reference output data.

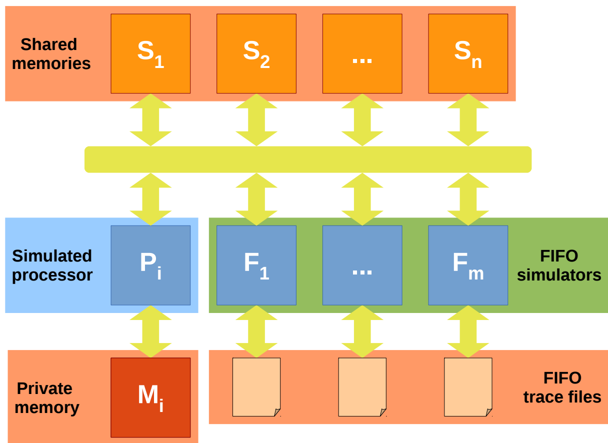


Figure 36: Processor simulation in standalone fashion by way of FIFO simulators

As a matter of fact, the simulation speed is directly related to the simulation precision: The more precisely a simulator describes the platform, the slower it is. For instance, the instruction-set simulator is about two hundreds time faster than an RTL simulator.

5.4 CONCLUSION

This chapter has presented a development environment built upon the dataflow programming paradigm. This development environment, initially developed by Wipliez [177], has served as an experimental area for studying the compilation of dataflow programs over multiple platforms. We have shown how modern software engineering techniques such as meta-modeling can be used to improve our compilation infrastructure in terms of robustness, efficiency and architecture. Starting from the global structure of the trans-compiler, we have also detailed an entire process [5] to design embedded multi-core platforms dedicated to dataflow programs from a flexible architecture model.

We are now going to focus on the execution of dynamic dataflow programs on multi-core platform, starting by the description of an optimized software implementation (Chapter 6) and following by the description of actor mapping/scheduling techniques that can handle the unpredictable behavior of dynamic dataflow programs (Chapter 7). We are also going to study the effectiveness of our contributions on both modern desktop platforms and embedded platforms based on our architecture model.

6 OPTIMIZED SOFTWARE IMPLEMENTATION OF DYNAMIC DATAFLOW PROGRAMS

Now you're coming back to Earth,
and things are getting more and more dynamic.

— Duane G. Carey, NASA astronaut, 2002

The main challenge that dynamic dataflow programs have to face is the demonstration of efficient implementations that can achieve performance constraints imposed by modern applications. For instance, video decoders have to provide real-time frame-rates for high-definition video sequences, from 25 FPS for 720p format on mobile terminals to 50 FPS at 8K format on cinema screens. While the efficiency of traditional language programs is the result of 50 years of work on compilers to exploit memory locality, abandoning memory-oriented programming in favor of dataflow programming requires the development of new compilation techniques to fully benefit from the processor architecture. Moreover, the attractiveness of more restricted dataflow models has often deflected attention from DPN-based programming.

This chapter describes an optimized software implementation of dynamic dataflow programs following the DPN model. Our implementation targets especially the efficient execution of video decoders onto embedded multi-core platforms, but most of the principles can be applied to all DPN-based programs and multi-core platforms. The main contributions of this chapter are:

1. Software implementation of dynamic dataflow programs including optimized communications and scheduling. Please notice our implementation has been integrated in the co-design flow we have introduced in Chapter 5, so all the experiments presented below are supported by our compilation flow.

2. Analysis of our software implementation onto dataflow-based video decoders that have been developed within the RVC framework. Let us point out that real-time decoding frame-rates of high definition video sequences are achieved on desktop processors using the RVC descriptions of several video decoders, including a description of the emerging HEVC standard which is still being developed.

This chapter is organized as follows. We start by introducing the implementation of dynamic dataflow programs. Then, we present our optimized implementation of DPN-based programs, starting by the communications in Section 6.2, then the scheduling in Section 6.3. Finally, we evaluate our implementation onto RVC-based video decoders in Section 6.4

6.1 IMPLEMENTATION OF DATAFLOW PROCESS NETWORKS

One of the fundamental interests of dynamic dataflow programming for designing embedded software is the *formalism* provided by the underlying MoC.

In fact, such a formalism provides a basis for analysis of system properties, like *reliability* and *efficiency*, which are central in embedded systems design. Thus, implementing DPN-based programs onto programmable processors requires the translation of the semantic rules of the DPN MoC [121] into imperative constructions, which can be executed by our processors, respectful of these rules.

In general, the implementation of dynamic dataflow programs faces two problematics to achieve performance requirements:

- **Communication** is the major bottleneck of dataflow programs. Since the actors can only communicate through the FIFO channels, the execution requires a massive amount of data movements that can ultimately lead to poor performance.

- **Scheduling** is a well-known bottleneck of dynamic dataflow programs. In fact, the expressive power offered by the DPN models requires a large number of control structures. This is supported by the variety of syntactic constructions available in CAL [67] that control the inner execution of an actor (FSM, priority, pattern, guard, untagged actions) as presented in Chapter 4.

These factors are directly impacted by the application granularity, defined in Chapter 3 as the ratio of computation to the amount of communication [156]. In the literature, the applications are informally classified in fine-grain, average-grain and coarse-grain families. On the basis of this classification, we define video decoders as fine-grain when they process each pixel at a time, as average grain when they process blocks, and as coarse-grain when they process frames.

6.2 OPTIMIZED COMMUNICATIONS

In theory the DPN model defines FIFO-based channels with unbounded capacity [121], in practice our FIFO-based channels are bounded to limit memory usage and avoid the overhead of dynamic memory allocation. We assume here that the size of the channel is provided by the application developer, knowing that some works target their optimization by way of critical path analysis [38] (See Chapter 3). Actually, bounded FIFO channels have been studied extensively since the emergence of the first wait-free algorithm presented by Lamport in the late 70s, but communicational channels of DPN-based programs have specificities that make their implementation quite challenging.

6.2.1 *To Be or Not To Be FIFO Channels*

As presented in Chapter 3, the DPN model defines action firing as an indivisible quantum of execution. Therefore, an action is fired if and only if enough tokens are available in the input channels. Thus, the implementation of FIFO channels for DPN-based programs requires the ability to check their state, i.e. the number of tokens available, during the execution.

Additionally, the DPN model introduces the concept of *firing rules* to order the execution of the *firing functions*, namely the actions. An action can be fired if and only if its firing rule is valid, and this validity depends on the internal state of the actor and the value of incoming data. Therefore, DPN-based actors *peek* tokens from input channels, i.e. they check values of

incoming tokens without consuming them, to evaluate action fireability and thus break FIFO principle.

Since the FIFO principle imposes that the tokens are accessed in order, a respectful implementation would be a buffering mechanism that conserves the tokens until they are truly consumed by the actor. In this regards, Jerbi et al. have proposed an automatic transformation using buffering to remove *peek* operations while respecting the DPN model [105, 104]. Their transformation aims to facilitate the portability over hardware platforms, but ultimately increases the memory usage as well as the scheduling overhead. Hardware FIFO being traditionally implemented using acknowledgments, hardware implementations [103, 157, 158] have extensively performed the *peek* by reading the token value without acknowledgment. While this approach is very simple and efficient, the *peek* stays however limited to the first token of the FIFO channel and thus reduces the support of dynamic dataflow programs.

6.2.2 *Software Circular Buffer*

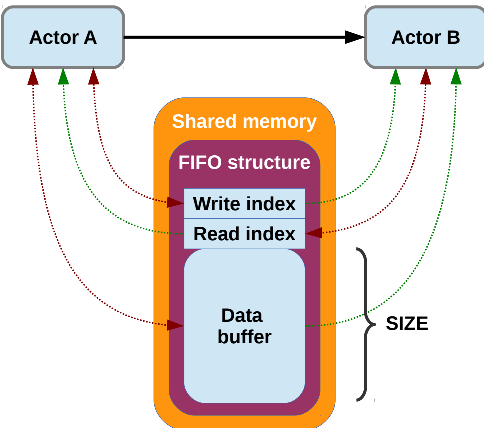


Figure 37: Concurrency-safe implementation of FIFO channels in shared-memory

```
1 struct fifo_s {
2   const int SIZE;       /* FIFO size    */
3   unsigned int rdInd;   /* Read index   */
4   unsigned int wrInd;   /* Write index  */
5   tokenType *content;   /* Data buffer  */
6 };
```

Listing 9: Software data structure of FIFO channels

In software, FIFO channels are traditionally implemented by a circular buffer allocated in shared memory (Figure 37 and Listing 9). *Read* and *write* are then achieved by accessing the buffer according to read and write indexes that are updated afterwards (Listing 10). Moreover, the comparison of the indexes is sufficient to know the state of the FIFO channel. Similarly to hardware implementations, a *peek* is a *read* without the update of the read index, but any token can be peeked thanks to the full accessibility of the shared memory.

```
void write(Fifo *fifo, tokenType *buff, int n) {
  for(int i=0; i<n, i++) {
    fifo->content[fifo->wrInd] = buff[i];
    fifo->wrInd++
    if(fifo->wrInd == fifo->SIZE) {
      fifo->wrInd = 0;
    }
  }
}

void read(Fifo *fifo, tokenType *buff, int n) {
  for(int i=0; i<n, i++) {
    buff[i] = fifo->content[fifo->rdInd];
    fifo->rdInd++
    if(fifo->rdInd == fifo->SIZE) {
      fifo->rdInd = 0;
    }
  }
}
```

Listing 10: FIFO accesses based on circular buffer

6.2.3 *Control-Free Communications*

Using circular buffer to implement FIFO channels avoids side shuffles of data after each reading, but implies an advanced management of memory indexes that can ultimately lead to poor performance. For instance, the update of the indexes may require checking if the end of the buffer is reached to go back to the beginning.

Avoiding checks on the position of the indexes is however possible using absolute indexes, as proposed by Wipliez, with the cost of additional modulo operations. Thus, performing *read* and *write* increases the indexes infinitely until the overflow of the variables.

```
void write(Fifo *fifo, tokenType *buff, int n) {
  for(int i=0; i<n, i++) {
    fifo->content[fifo->wrInd % fifo->SIZE] = buff[i];
    fifo->wrInd++
  }
}

void read(Fifo *fifo, tokenType *buff, int n) {
  for(int i=0; i<n, i++) {
    buff[i] = fifo->content[fifo->rdInd % fifo->SIZE];
    fifo->rdInd++
  }
}
```

Listing 11: Control-free FIFO accesses

Since computing the modulo is costly on most processor architectures, it is translated to a simple right shift by forcing the size of the buffer to a power of two:

$$\forall\ fifo_i \in \text{FIFO},\ |chan_i| = 2^n \ \textit{with}\ n \in \mathbb{N} \tag{10}$$

Paradoxically, such a constraint on the size of the communication channels does not have a large impact on the memory usage, especially compared to the large needs of video decoders. Indeed, the initial sizes of our FIFO channels being reasonable, the round-up to the next power of two is relatively small.

6.2.4 *Multi-rate Communications*

One of the high-level features of CAL is its ability to describe *multi-tokens* actions [67], i.e. actions reading and writing pools of data at each firing, such as the transposition of 4x4 block presented in Listing 12 that reads and writes 16 tokens by firing.

```
1 transp: action Src:[ src ] repeat 16  ==> Dst: [ dst ] repeat 16
2 var
3   int(size=16) dst[16] =
4     [ src[ 4 * column + row ] :
5       for int row in 0 .. 3, for int column in 0 .. 3
6     ]
7 end
```

Listing 12: Transposition of a 4x4 block in RVC-CAL

Following this semantic, Wipliez has proposed an implementation of DPN-based programs for GPP [177]. Considering an action, such as the one described in Listing 12, the dataflow description is translated into the C code presented in Listing 13, which is compilable with most of C compilers. In fact, the function implementing the action body is decomposed into 3 steps as follows:

1. **Reading:** Incoming tokens are read *in order* from the input FIFO channels and stored into the local variables referenced by the input *pattern*. E.g., in Listing 12, 16 tokens are read from the port `Src` and stored in the local array `src`.

2. **Processing:** The action is processed, as defined in its CAL description, using the local variables referenced into the input and output *patterns* as interfaces. As a consequence, the processing of data is not necessarily described *in order*.

3. **Writing:** Outgoing tokens are written *in order* from local variables referenced by the output *pattern* into the output FIFO channels. E.g., in Listing 12, 16 tokens are written successively from the local array `dst` to the port `Dst`.

While this implementation stays respectful of the FIFO principle, with the exception of the *peeking*, it also involves two additional copies between the circular buffers and the local variables.

```
static void transp() {
  i16 local_Src[16], local_Dst[16];
  i32 row, col;

  // Read the input tokens in order
  read(fifo_Src, local_Src, 16)

  // Transpose the tokens
  row = 0;
  while (row <= 3) {
    col = 0;
    while (col <= 3) {
      local_Dst[row * 4 + col] = local_Src[4 * col + row];
      col = col + 1;
    }
    row = row + 1;
  }

  // Write the output tokens in order
  write(fifo_Dst, local_Dst, 16);
}
```

Listing 13: Transposition of a 4x4 block generated in C

6.2.5 *Copy-Free Communications*

Since our FIFO channels are implemented in shared memory without access restriction, we can remove the additional copies to local buffers by accessing directly to the content of the FIFO channels within the processing of the action. So, accesses to input and output variables, such as `src` and `dst`, are replaced by direct accesses to FIFO channels, such as `Src` and `Dst` respectively. Unfortunately, *race conditions*, i.e. synchronization issues, can occur when the action processing does not ensure that the FIFO accesses are performed in order (such as the accesses to `src`).

But, the DPN model defines an action firing as a quantum of execution [121], in other words an action firing is an atomic step that cannot be interrupted. Thus, the FIFO indexes can be updated just once at the end of the action without changing the semantic of the application, such as presented in Listing 14. Then, the implementation stays respectful of the FIFO principle. Indeed, other processors cannot access the FIFO rooms involved by this processing since the FIFO indexes are not updated until the action is entirely processed.

To summarize, the three first steps of action firing (Reading, processing, and writing) can be merged together, reducing the memory footprint and the number of instructions to implement the action, as long as the FIFO indexes are updated after the action processing, and thus let the other actors using newly produced data and newly released rooms.

```
1  static void transp() {
2    i32 ind_Src, ind_Dst;
3    i32 row, i32 col;
4
5    // Transpose the tokens directly from/to the FIFO channels
6    row = 0;
7    while (row <= 3) {
8      col = 0;
9      while (col <= 3) {
10       ind_Src = (fifo_Src->rdInd + (4 * col + row)) % fifo_Src->
             SIZE;
11       ind_Dst = (fifo_Dst->wrInd + (row * 4 + col)) % fifo_Dst->
             SIZE;
12       fifo_Dst->content[ind_Dst] = fifo_Src->content[ind_Src];
13       col = col + 1;
14     }
15     row = row + 1;
16   }
17
18   // Update indexes
19   fifo_Src->rdInd += 16;
20   fifo_Dst->wrInd += 16;
21 }
```

Listing 14: Copy-free execution

Furthermore, this optimization is highly simplified by the `Pattern` class composing our dataflow IR which links together variables and ports (Section 5.1.6 of Chapter 5). Therefore, when the compiler meets a variable access in an action body (load or store), it just has to check if the given variable is contained in one of the patterns associated to the action in order to generate the access accordingly. For instance in Listing 12, the variable `src` is associated to the input port `Src` by the input pattern of the action, so when the compiler meets an access to this variable `src`, it can translate it to a direct access to the FIFO channel `fifo_Src->content`.

6.2.6 *Efficient Broadcasting of Communications*

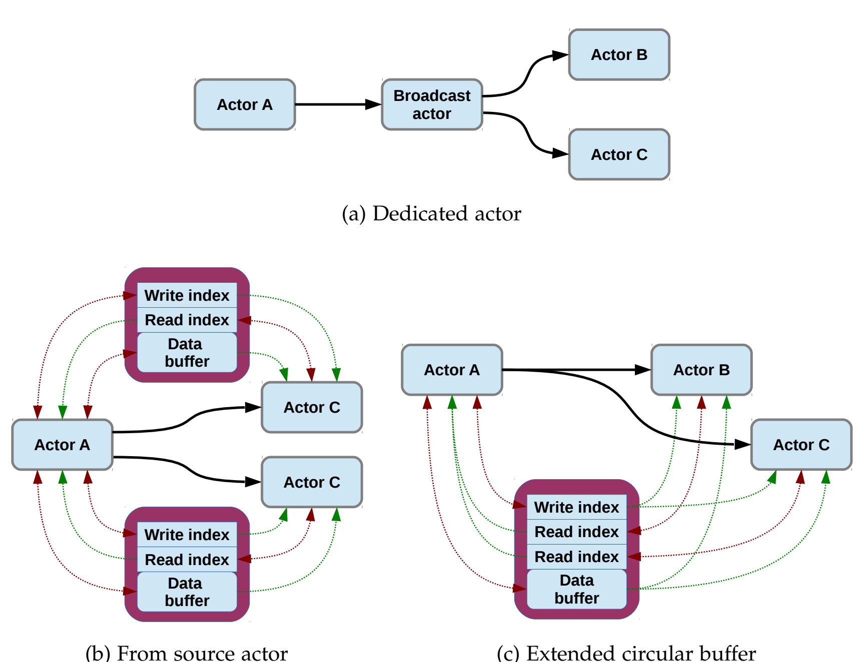
Now, our dataflow applications also support broadcasting communication following the 1-*producer*/N-*consumers* scheme. Thus, actors can produce data that are transmitted simultaneously to multiple target actors through a single port. In fact, the implementation of the broadcasting is another critical point of communication in dynamic dataflow programs, especially for our video decoding applications that have an extensive use of broadcasting.

As a result, the implementation of our communication channels has to be able to efficiently broadcast the data over several actors, and, to our knowledge, broadcasting tokens can be implemented in three ways that are illustrated by Figure 38:

1. Adding a specific actor in charge of copying the data produced to all targeted actors (Figure 38a), as described by Wipliez in his thesis [177]: While this implementation is flexible, adding actors complicates the actor scheduling and involves extra data movement.

2. Asking the source actor to broadcast itself the tokens into multiple communication channels (Figure 38b): While the implementation is natural, the data are copied for each target.

3. Using circular buffers with multiple read indexes (Figure 38c), the smallest one being the global index: While this implementation reduces the data movements to maximum, the managing of the FIFO channels is complicated and all the FIFO channels need to be mapped on the same address space.



(a) Dedicated actor

(b) From source actor

(c) Extended circular buffer

Figure 38: Three way of broadcasting communications

Knowing the memory architecture of our architecture model (Section 5.2 on Chapter 5), the implementation of the broadcast depends on the repartition of the actor over the platform. If all target actors are mapped together on the same processor, the broadcasting can be performed using multiple read indexes. On the contrary, when the target actors are mapped to different processors, then the broadcast is performed directly by the source in order to duplicate the data over several address spaces.

6.3 OPTIMIZED SCHEDULING

In Chapter 3, we have stated that one essential benefit of the DPN model lies in its strong expressive power, so as to simplify algorithm implementation for programmers. This expressive power includes: the ability to describe data-dependent computations through token production/consumption, where production/consumption may vary according to values of tokens; the ability to produce time-dependent behaviors that rely on the time at which tokens are available on the input of an actor; and, the ability to express non-determinism, which can be used to construct actors that respond to unpredictable sequences of tokens.

However, when dealing with the scalability of this model, we have stated that this strong expressive power has a bad influence on the efficiency of its

implementation, as several operations may be scheduled at run-time on a single processing unit. The overhead caused by a scheduling strategy, along with its variable chance of success between test/validation of a firing rule for each operation, can lead to inefficient implementation of dataflow programs or to unsteady performance on their executions.

6.3.1 *Scheduling Scheduling*

As defined by Lee and Parks [121], the execution of a DPN-based actor is modeled by the repeated evaluation of the firing rules that are, in case of a success, followed by the firing of the associated action. This process is usually defined as the *action scheduling*.

Apart from this internal scheduling, the execution of a DPN program in a concurrent environment requires actor scheduling. Section 3.6 of Chapter 3 has presented three models which can execute several actors on a single processing unit, and especially two scheduling strategies, known as *round-robin* and *data-driven / demand-driven*, dedicated to DPN-based actors. Both scheduling strategies assume that an actor should not be fired indefinitely without external contribution (other actors that consume/produce the tokens). So, the actor currently scheduled will be blocked at some point, with no chance to be fired anymore, and will exit from the action scheduler to let the actor scheduler decide the next actor to schedule.

To conclude, the execution of DPN-based programs involves both *actor scheduling* and *action scheduling* (Figure 39). While they are two distinct levels of scheduling, they are intimately related since the success of the action scheduling within an actor is directly dependent on the production/consumption performed by its predecessors/successors.

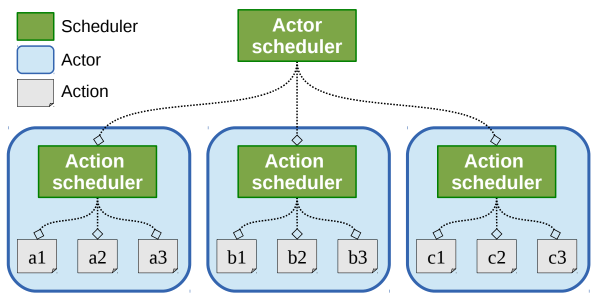


Figure 39: Hierarchical scheduling

6.3.2 *Action Scheduling*

As we have seen before, the action scheduler evaluates the firing rules so as to determine the next action to fire. In fact, the firing rules are evaluated successively according to the *partial order* defined within the actor (priorities and FSM). Thus, the action scheduler can be implemented by a simple function that evaluates the firing rules *in order* [177] such as presented in Listing 15.

In theory, the scheduler evaluates only two conditions to determine the fireability of an action: the *input pattern*, the amount of tokens required in the input channel, and the *guard*, the potential condition on the values of tokens and/or state variables.

In practice, the scheduler has also to evaluate the *output pattern* so as to ensure that enough rooms are available in the output channels to allow the firing of the action without blocking. While the validation of the output pattern is not required by the DPN model, it is necessary when several actors are executed concurrently on the same processor. Indeed, waiting for the availability of an output channel, using blocking writes for instance, inevitably leads to a deadlock if the target of the channel, the *consumer*, is mapped to the same processing unit.

```
void Transpose4x4_0_scheduler() {

  while (1) {
    if (hasTokens(fifo_Src, 16) && isSchedulable_untagged_0) {
      if (hasRooms(fifo_Dst, 16)) {
        // Return back to the actor scheduler
        goto finished;
      }
      transp(); // Fire the action
    } else {
      // Try to fire the next action, but no others
      // Return back to the actor scheduler
      goto finished;
    }
  }

finished:
  return;
}
```

Listing 15: Action scheduler

6.3.3 *Actor Machine*

The large number of tests involved in actor execution so as to evaluate the firing rules, along with their unpredictable chance of success, can ultimately lead to inefficient implementation of DPN-based actors. Thus, a different approach, introduced by Janneck and Cedersjö, tries to reduce the number of tests performed during the evaluation of the firing rules using a new execution model, called *actor machine* [101, 46], that also considers the evaluation results of previous firing rules.

Actor machine deals with the memorization of the test results involved in the validation of previous firing rules to limit their reproduction. For instance, let two firing rules R_i and R_j tested successively such as $R_i = [P_{i,1}, P_{i,2}]$ and $R_j = [P_{j,1}, P_{j,2}]$ with $P_{i,1} = P_{j,1} = [*, *]$; if R_i is evaluated false such as $R_i = [true, false]$ then $P_{j,1}$ could be already known valid during the evaluation of R_j and the evaluation of $P_{j,2}$ should be sufficient. To do so, the evaluations of previous patterns are preserved by the use of an automaton mechanism. Several connected actor machines can also be composed in order to increase the potential reduction [101].

On the one hand, the scheduling of an actor machine could be more efficient compared to the traditional firing model thanks to the reduction of the number of tests performed. On the other hand, the translation to the actor machine execution model induces an explosion of the number of states in the scheduling algorithm due to the need of memorization. Moreover, a circular buffer implementation of the communication channel allows a

similar test reduction by means of compiler optimization. Indeed, common sub-expression elimination can search for identical patterns in firing rules evaluated successively, and can replace them with a single variable holding the result of their evaluation.

6.3.4 *Quasi-Static Scheduling*

The challenge when optimizing the execution of a dataflow description is to conserve the strong expressive power of DPN while reducing the overhead caused by its required run-time scheduling. Quasi-static scheduling intends to make scheduling decisions as much as possible at compile-time by determining all static behaviors and by keeping only the necessary decision for run-time. The literature has introduced a large panel of methodologies to perform quasi-static scheduling of dynamic dataflow programs in different manners [93, 90, 69, 70, 32, 33, 34].

Some of them try to prune all unreachable execution paths to remove all unnecessary tests using code instrumentation [32, 33, 34] or model checking [69, 70] to determine the possible executions. However, both of them are limited by their need of input data to perform their analysis. Such a requirement prevents the full support of all possible execution paths.

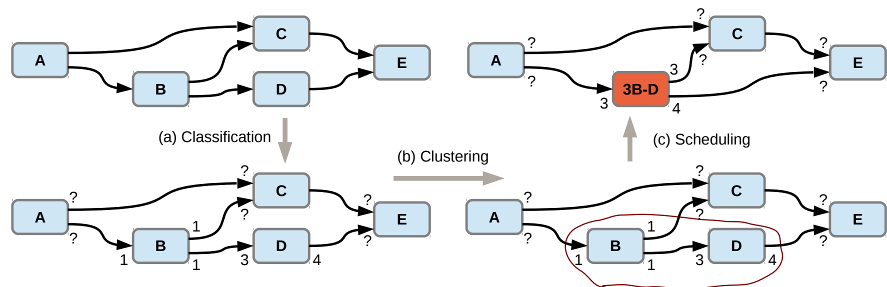


Figure 40: Quasi-static scheduling using actor clustering

Another approach, based on the classification results, tries to reduce the number of actors that are required to be scheduled at run-time, by clustering network regions that have a locally static behavior [93, 90]. We mean by one locally static region a set of connected actors in the description that have a firing order we can determine statically, regardless the data stored in the FIFO channels of the description. The actor clustering approach is based on three existing algorithms that are applied sequentially as follows:

1. The actors with predictable behaviors presented in the dataflow description are detected using actor classification as described in Section 3.5.1 of Chapter 3.

2. Predictable actors connected to one another are clustered into a single node, called *composite node*, to obtain a valid sequence of firing in it that can be determined at compile-time. As such, an essential condition to set a composite node is to determine whether such a sequence of firing is possible, the *composition theorem* described by Pino [144]. The resulting cluster becomes a *composite node* in the graph of the dataflow description.

3. Actors grouped in a composite node are scheduled at compile-time using the Single-Appearance Scheduling (SAS) strategy [131], the opti-

mum static scheduling strategy for code minimization where all repetitions of a same actor can be found side by side. The other remaining actors, along with the resulting composite nodes, are scheduled at runtime.

The methodology is illustrated by the Figure 40 on a dataflow example containing 5 actors. Each actor is firstly classified to determine, if possible, its production/consumption rates in order to detect the existing static region that can finally be scheduled.

6.4 STUDY OF RVC-BASED VIDEO DECODERS

Chapter 3 has introduced the performance of the implementation as a key point of the widespread adoption of dynamic dataflow programming. We have also claimed that the two main problematics of software implementation to achieve the expected performance is the communication and the scheduling.

Now, this section focuses on the implementation of our RVC-based video decoders, because understanding the application is the key to getting the most out of an embedded system [182]. Actually, the identification of the application characteristics enables the specialization of the system to perform powerful optimizations, this is obviously the main interest of application-specific and domain-specific platforms.

6.4.1 *Experimental setup*

In order to study the implementation on both desktop and embedded processors, we define two different configurations of our experimental setup as follows:

A. **Desktop implementation:** In this configuration, the tested software implementations are generated by use of the C back-end of Orcc (previously called *orcc-c*), and the generated C code is compiled with GCC and executed on an Intel Xeon W3670 clocked at 3.2GHz on top of Ubuntu GNU/Linux.

B. **Embedded implementation:** In this configuration, the tested software implementations are generated by use of the TTA back-end of Orcc (previously called *orcc-tta*), then the generated code is compiled by the TTA compiler for the processor, usually configured as *Fast* considering this is our best trade-off between performance and power consumption. Apart from the processor configuration, the multi-core platform is based on the architecture model defined in Section 5.2. The evaluation is made thanks to the instruction-set simulator including in the TCE (previously called *ttanetsim*).

The experiments have been conducted for some of the RVC descriptions of video decoders that have been introduced in Chapter 4, and using 720P sequences containing I/P/B frames. Here is the detailed list:

- **MPEG-4 Part 2 SP** - *Old town cross* (25fps, 6Mbps) : A custom description (known as *Irisa*) that has been optimized by hand according to the execution analysis extracted from our embedded implementation.
- **MPEG-4 AVC / H.264 PHP** - *A Place at the Table* (25fps, 6Mbps) : The normative version that decomposes the processing of each component (Luma and Chroma).

- **MPEG hevc / H.265 Main** - *Kristen And Sara* (60fps, 1Mbps) : The normative version that is still being developed but already compliant with most of HM10.0 bitstreams.

During all our experiments, all the fifo channels in our applications are bounded to 8192 elements in order not to impact on the results.

6.4.2 *Analysis of Global Performance*

First of all, we analyze the global performance of each implementation of our rvc-based video decoders.

desktop implementation Table 6 presents the frame-rates observed during the decoding of the 720p video sequences on our desktop processor. For this experiment, all the actors are mapped to the same processor core and scheduled using the round-robin strategy.

Decoder	Video sequence	Frame-rate
MPEG-4 Part 2 SP	*Old town cross* (720P)	33,7 FPS
MPEG-4 avc php	*A Place at the Table* (720P)	4,5 FPS
MPEG hevc Main	*Kristen And Sara* (720P)	12,7 FPS

Table 6: Maximal frame-rates achieved by our desktop implementation using round-robin scheduling strategy

The results show a large difference between the performances of the 3 decoders. MPEG-4 Part 2 SP clearly is the most efficient and easily achieves real-time decoding for 720p sequences: This can be explained by its lower complexity and by the fact that this specific description is the result of several DSEs to improve the performance of the decoder.

But, our implementation of the hevc decoder surprisingly is more efficient than the one of the MPEG avc decoder even though the hevc standard is much more complex than its predecessor. This can be explained by the wrong implementation choices that have been taken during the development of the rvc-based description of the avc decoder. As presented in Table 4 of Chapter 4, the description is composed of many more actors than the other decoders: This causes a larger scheduling overhead and requires much more communications to disseminate all the information within the decoder. In fact, these differences clearly demonstrate the high importance of the decomposition granularity within dpn-based programs.

As a result, we have deliberately chosen to focus, in the next sections of this chapter, on the analysis of MPEG-4 Part 2 and MPEG hevc / H.265 without detailing the analysis of MPEG-4 avc / H.264.

embedded implementation Now, we evaluate the global performance of our embedded implementation. First, let us point out that a functional embedded implementation is much more difficult to obtain than a desktop implementation. Indeed, debugging dataflow programs within embedded multi-core platforms is a hard and time-consuming task that requires an expertize from hardware and software aspects. Moreover, the simulation speed is rapidly becoming one of the main limitations compared to the execution speed on desktop processors.

Table 7 summarizes the maximal frame-rates achieved with our embedded implementation on both the MPEG-4 Part 2 decoder and the MPEG HEVC decoder (*Still Picture* profile that does not contain the inter-prediction). The evaluated embedded platforms are composed of *Fast* TTA-based processors clocked at 100MHz, wherein each actor is mapped to its own processor. Thus, there is no need for an actor scheduling strategy: The global scheduling is achieved by the action scheduler that checks repetitively the validity of the firing rules.

Decoder	Video sequence	Frequency	Frame-rate
MPEG-4 Part 2 SP	*Foreman* (QCIF)	100MHz	175 FPS
	N/A (720P)	1GHz	40 FPS[1]
MPEG HEVC	*BasketBallPass* (240p)	100MHz	4 FPS
	N/A (720P)	1GHz	5 FPS[1]

Table 7: Maximal frame-rates achieved by our embedded implementation using the *Fast* configuration clocked at 100MHz. These frame-rates have been evaluated during an execution of the entire multi-core platform within the instruction-set simulator (*ttanetsim*)

The results clearly demonstrate the functioning of our co-design flow (presented in Section 5.3). Actually, these results are obtained from a simulated execution, but let us point out that successful implementations of the MPEG-4 Part 2 SP decoder [5] has already been synthesized on two different FPGA boards: Altera Stratix III and Xilinx Virtex 6.

Besides the functional demonstration, the results also show a large difference of performance between the two decoders, i.e. the frame-rate observed on MPEG-4 Part 2 is about 40 times better while the tested video sequence is only 4 times smaller. This can be explained by the performance tuning that we have already made on the description of MPEG-4 Part 2, along with the development status of our description of HEVC.

Considering the current performance, our embedded implementation cannot achieve real-time decoding of high definition sequences. But, these results open promising perspectives about a more optimized implementation, and about a generic video decoding platform that could reach higher clock frequency thanks to integrated circuit technology. To this end, we have extrapolated the results for higher clock frequency (1GHz) and video definition (720P).

6.4.3 *Analysis of Internal Communications*

A major interest of dataflow programs is the explicit communication between the components of the application that makes them easier to analyze. In DPN-based video decoders, communication rates are usually irregular and very sensitive to multiple factors (size of the FIFO channels, actor scheduling, etc). But, communication rates become globally stable when the observed time-slice is sufficient. Thus, Figure 41 presents the communication rate observed at each output port within the MPEG-4 Part 2 SP and MPEG HEVC decoders during the decoding of few frames of the tested video sequences. Figure 41 additionally presents the degree of broadcasting of the ports, i.e.

1 Extrapolated frame-rates

the number of actors to which the ports are connected, in order to highlight the duplication of data.

We can clearly identify two categories of communications from the results presented in Figure 41:

VIDEO STREAM This stream is characterized by a large amount of data that usually goes through the decoder by a single path (For instance `parser_blkexp.QFS` in Figure 41a). The stream may be however divided to increase the data parallelism, by separating the decoding of either the image components (Luma and Chroma) [124] or the image regions (like the tiles in HEVC / H.265 [164]).

Besides, broadcasting the video stream involves a large amount of data duplication but is only performed one or two times (For instance `motion_add.Vid` in Figure 41a), when the decoded frames are transmitted to both the display and the image buffer used by the inter prediction. This stream being clearly the largest of the application, this specific broadcast can be the cause of a data congestion.

CONTROL INFORMATION These communications are characterized by a small amount of data disseminated through multiple channels within the video decoder. A typical example is the transmission of the type of the current block, `parseheader.BTYPE` in Figure 41a. A major part of these communications is produced by the *parser* which extracts the syntax elements from the input stream to parametrize the actors. As a result, their behavior of video decoders should match well with the semantic of quasi-static dataflow MoCs.

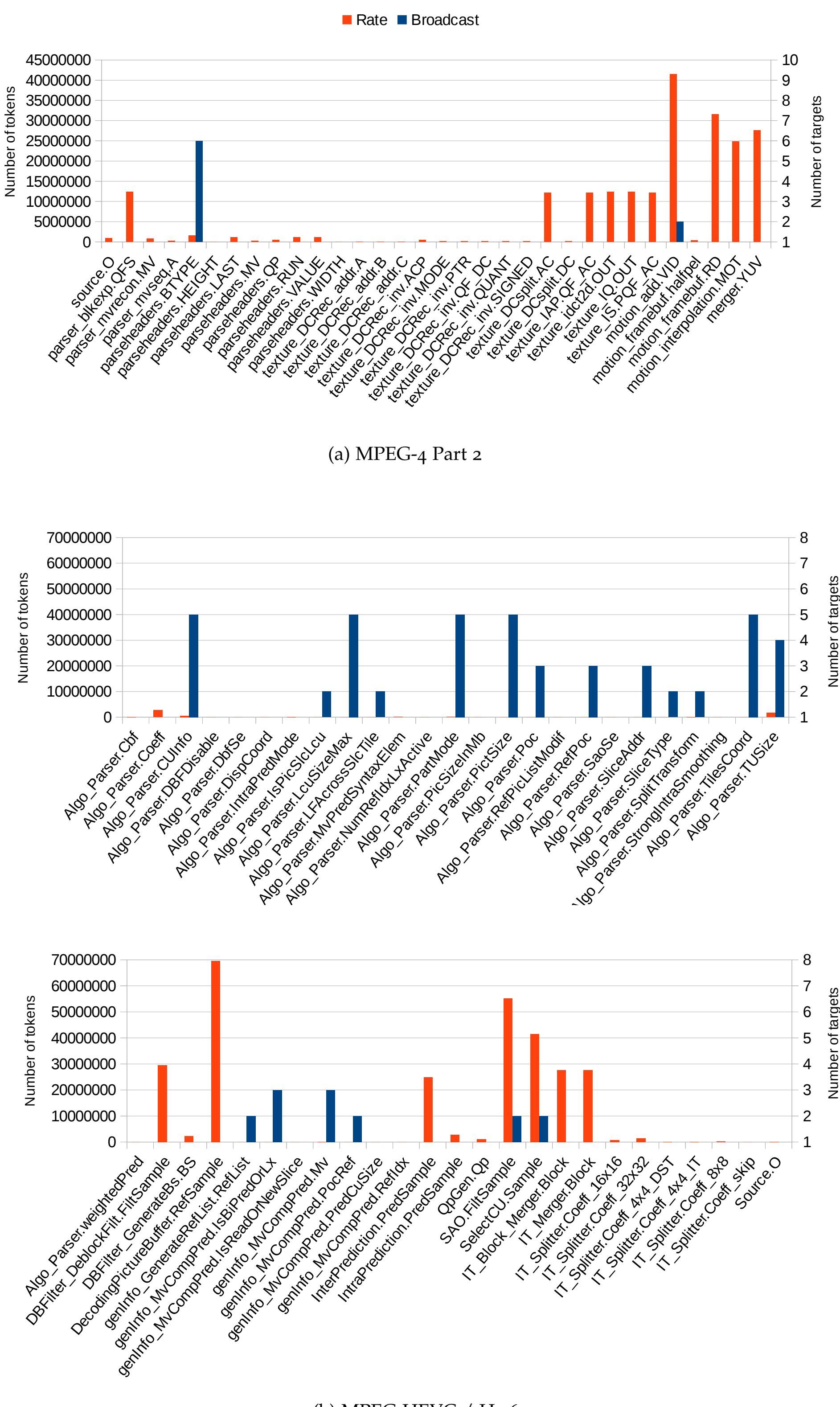
As opposed to the video stream, broadcasting the control information implies a smaller amount of data but more consumers. For example, control tokens generated by the *parser* may be transmitted to most of the next actors, like `Algo_Parser.CUInfo` in Figure 41b, so even a small amount of data can introduce a lot of checks to control the state of the communication channels.

To sum up, the video stream is processed block after block through the actors which behave according to control data. Moreover, the broadcasting may be an additional source of bottlenecks, causing either data congestions or management overheads.

6.4.4 *Analysis of the Application Decomposition*

Now, let us take a look at the application decomposition which is fundamental for targeting multi-core platforms.

WORKLOAD DISTRIBUTION We start by analyzing the distribution of the computational workload within the video decoders, i.e the computational workload of the actors, for both embedded and desktop implementations. The results for two video decoders, MPEG-4 Part 2 and HEVC, are presented in Figure 42. On the one hand, the workloads of the embedded implementation are evaluated for each actor independently in a standalone simulation. In other words, each actor is simulated on its own processor with all incoming data available, in order to hide the impact of the stream dependences within the network. On the other hand, the workloads of the desktop implementation are evaluated when all actors are mapped to the same processor

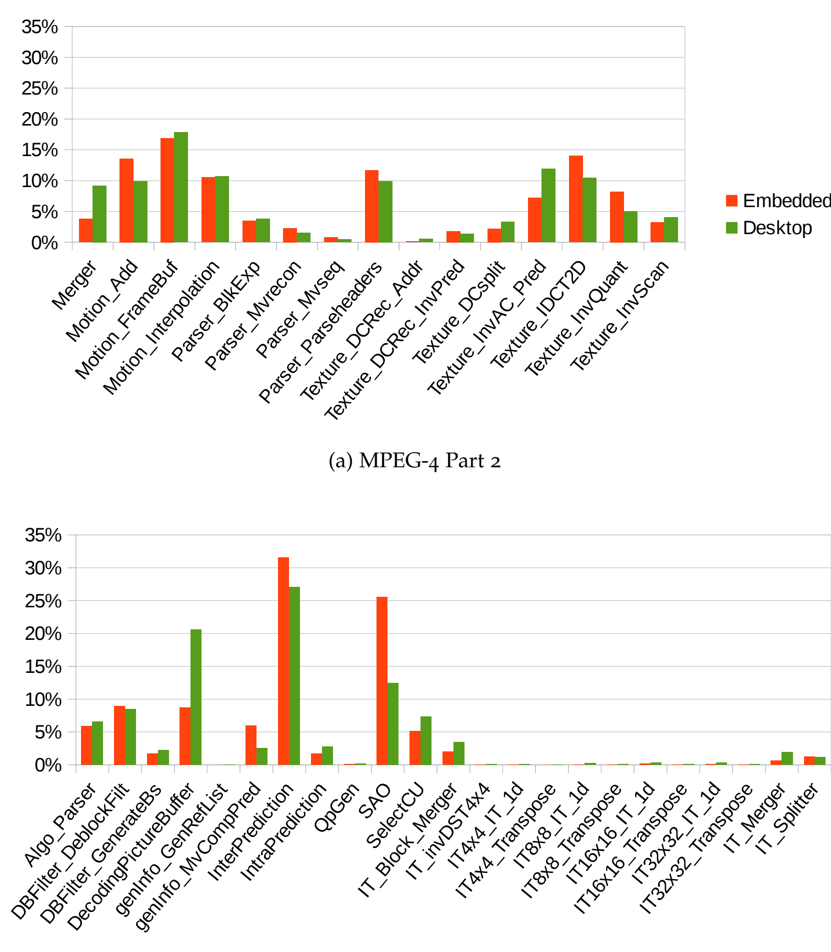


(a) MPEG-4 Part 2

(b) MPEG HEVC / H.265

Figure 41: Communication analysis (rates and broadcasting) within RVC-based video decoders

which may cause an overhead due to the variable chance of success between test/validation of the firing rules.



(a) MPEG-4 Part 2

(b) MPEG HEVC / H.265

Figure 42: Repartition of the workload within RVC-based video decoders using both desktop and embedded implementations

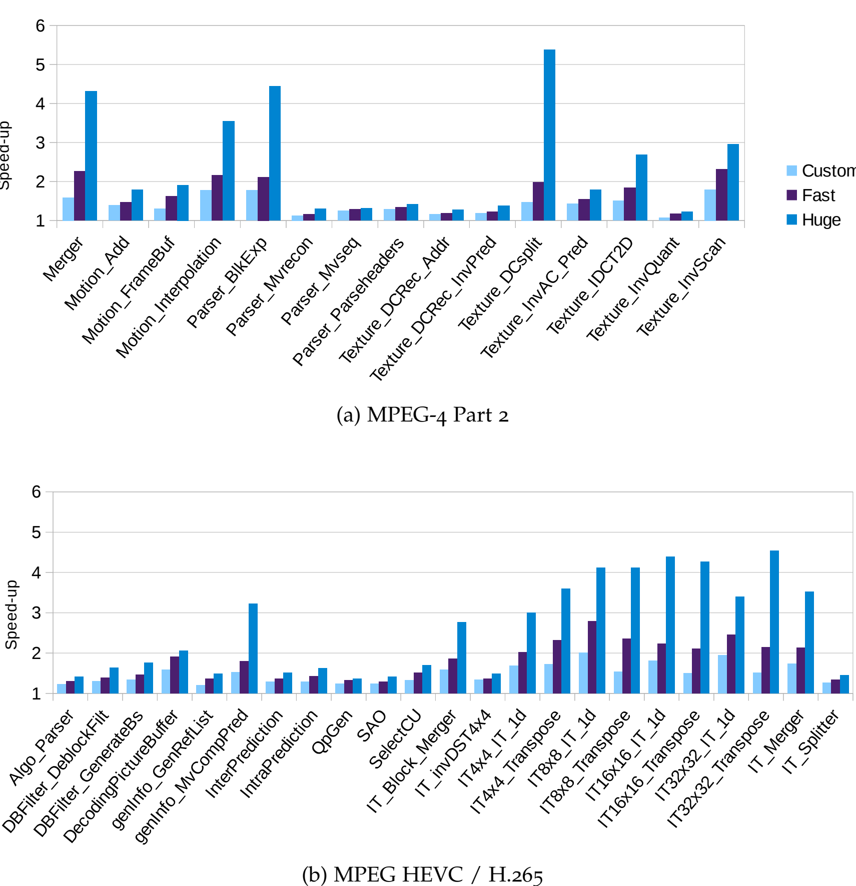
The results clearly show that our description of MPEG-4 Part 2 SP is more equitably balanced than our description of HEVC. This difference can be partially explained by the fact that the HEVC decoder is still being developed, especially concerning the complexity of the *inter-prediction*. Another explanation is the difference between the implementations of the *inverse transforms*, designed with 1 actor (the *IDCT2D*) in MPEG-4 Part 2 and 12 actors (the *ITs*) in HEVC.

The results show as well that the repartitions of the workload within both implementations are very similar, except for the *DecodingPictureBuffer* of HEVC that is subject to cache effects with the desktop implementation, and the *SAO* filter that is badly compiled by the TTA compiler.

Moreover, it should be noted that the computational workload could be balanced more equitably by increasing the coarse-grain parallelism in the decoder. In video decoding, increasing the parallelism is usually achieved by separating the decoding of the image component or by splitting the image. On the one hand, the separation of the processing of the components is bounded by the luma processing which is four times the complexity of each chroma processing. On the other hand, the decomposition of the image

itself is restrained by the spatial and temporal dependences resulting of the prediction. Actually, parallel processing is one of the main achievement of the emerging HEVC / H.265 standard [164] that introduces several advanced decomposition (wavefront, tiles, etc).

INTERNAL PARALLELISM Thanks to the flexibility of TTA-based processors in our embedded implementation, we can also study the potential parallelism within the actors. In fact, the predefined processor configurations, presented in Section 5.2, have all their own parallel processing capability, which let us study the Instruction-Level Parallelism (ILP) potential within actors. Therefore, Figure 43 presents the execution speedup of actors of two video decoders on *Custom*, *Fast* and *Huge* processors according to their execution time on a *Standard* processor. In fact, the *Standard* processor is equivalent to a RISC processor that can only perform one operation at a time because of its 3 buses. The actors are again executed in a standalone fashion.



(a) MPEG-4 Part 2

(b) MPEG HEVC / H.265

Figure 43: Exploring the parallelism potential of actors composing video decoders thanks to their execution speedup on TTA-based processors using *Custom*, *Fast* and *Huge* configurations from a sequential execution with *Standard* configuration

The results clearly show two types of actors. On the one hand, actors that benefit well from the parallel capabilities of TTA-based processors by presenting impressive speedups that reach factors up to 5.5, such as those involved in the processing of inverse transforms. We define them as the *compute-intensive* actors. On the other hand, actors that do not take advantage from the parallel capabilities of TTA-based processors by presenting

speedups that hardly reach factors of 1.5, such as the ones involved in entropy decoding. We define them as *control-intensive* actors.

From all these results, we can identify the traditional *bottleneck* actors of our RVC-based video decoders: The *parser* that is controlled by a complex FSM (e.g. the parser of our HEVC decoder contains about 200 actions), the *buffer* which is usually strangled by the number of hardly predictable memory accesses, and finally the *predictions* as well as the *loop filters* that all involve complex processing requiring careful implementations. In conclusion, video decoders are now complex applications containing heterogeneous algorithms which make their implementation so challenging.

6.4.5 *Comparison of the Scheduling Strategies*

Finally, we experiment the different scheduling strategies in order to show their impact on the efficiency of our video decoders.

CLUSTERING Quasi-static scheduling (See Section 6.3.4) based on local regions clustering has been evaluated on the tested video decoders. Table 8 presents the number of clusters found in each description, as well as the number of actors and FIFO channels affected by the clustering.

Results show that MPEG-4 AVC and HEVC seem much more static than MPEG-4 Part 2 SP. But, most of static regions come from the *inverse transforms*, and as seen before the *inverse transform* of the MPEG-4 Part 2 decoder is only composed of one actor, the *IDCT2D*, that results from an early handmade clustering [123]. Another important consideration is the current decomposition of the decoders. While the tested descriptions of the MPEG-4 AVC and MPEG-4 Part 2 decoders process the image components in parallel, the description of the MPEG HEVC decoder still processes them sequentially. As a result, HEVC contains a lot less actors since the *residual* and *prediction* parts are not duplicated.

	Clusters	Actors		FIFOs	
	#	#	%	#	%
MPEG-4 Part 2	3	6/41	15	3/104	3
MPEG-4 AVC / H.264	8	30/114	26	22/404	5
MPEG HEVC / H.265	5	18/34	53	13/109	12

Table 8: Clustering results

To take a more detailed example, we focus on the *inverse transform* composing the description of the HEVC decoder, see Figure 20 in Chapter 4. This *inverse transform* is decomposed in five parallel paths. The first paths are dedicated to the prediction residual decoding of each existing Transform Unit (TU) sizes, respectively 4x4, 8x8, 16x16 and 32x32. The last one is used to compute the inverse Discrete Sine Transform (DST) for 4x4 luma transform blocks that belong to an intra-coded region.

All actors of the inverse transform are classified SDF except the splitter and the merger. In fact, the splitter and the merger belong to the PSDF model (parametrized by the size of the block) but the classifier is not able to detect

them due to the complexity of their firing rules. Therefore, a clustering node is built for each paths with the schedule presented in Figure 44.

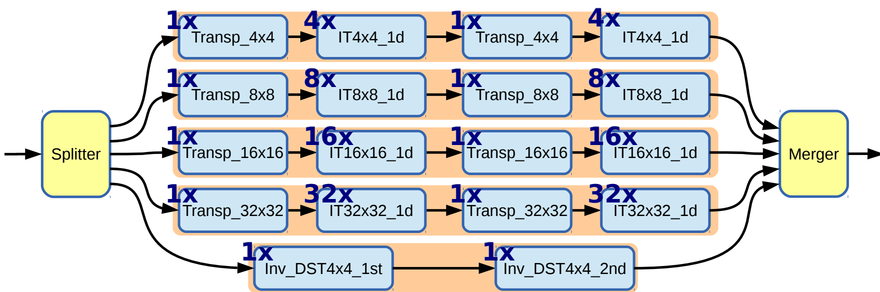


Figure 44: Quasi-static scheduling of the *inverse transform* of MPEG HEVC / H.265 based on graph clustering

DESKTOP IMPLEMENTATION First of all, the scheduling strategies are evaluated with the desktop implementation. The Table 9 summarizes the results of the sequential execution of the tested video decoders with different scheduling approaches, that are the *round-robin* and *data-driven/demand-driven* strategies (called respectively *Robin* and *Driven*) with or without clustering of locally static regions. The number of *switches*, *firings* and *misses* is detailed, additionally to the percentage of misses according to the total number of switches, as well as the resulting frame-rates. A *switch* corresponds to the execution of the next schedulable actor that occurs when the current actor is not fireable anymore due to the empty/full state of its communication channels; A *firing* corresponds to the execution of an action when all its firing rules are valid; And a *miss* is a switch that does not result in a firing.

The results show that clustering techniques can permit an impressive reduction of *misses* during the scheduling when the number of static actors is sufficient, such as in the AVC and HEVC decoders. But, this reduction does not necessarily induce a large improvement of the global performance of the application. As we have seen before, the computational workload of the *inverse transforms* within our decoders is minimal. Knowing that most of the detected static regions are localized in these *inverse transforms*, the impact on the global performance stays insignificant, expect for the AVC decoder due to its smaller granularity. Past experimentations [178, 90, 93] have however presented interesting improvements of the performance by using the same video decoders which are optimized with static scheduling of locally predictable regions. This difference can be explained by the high efficiency of our optimized implementation that masks the improvement induced by the clustering.

Moreover, the results show that the more advanced strategy, known as *data-driven / demand-driven*, reduces drastically the number of *misses* during the scheduling, but slightly improves the performance of MPEG-4 Part 2 SP and HEVC decoders. This can be explained by the fact that the improvement observed from the test/validation of the firing rules is masked by the execution overhead implied by the complexity of the strategy. However, we have shown in previous works [4] that DPN-based programs can benefit from this more advanced strategy when the number of actors increases.

EMBEDDED IMPLEMENTATION Now, let us compare the scheduling strategies more precisely thanks to the flexibility of our embedded implementa-

Strategy	Switch	Firing	Miss	%	FPS
Robin	4306	2812677	801	19	33,7
Robin+Cluster	4224	2809683	771	18	34,2
Driven	2420	2812654	252	10	34,6
Driven+Cluster	2324	2809632	232	10	34,8

(a) MPEG-4 Part 2 SP

Strategy	Switch	Firing	Miss	%	FPS
Robin	522205	12526470	447065	86	4,5
Robin+Cluster	403348	11825273	237835	59	5,4
Driven	52096	12539650	25633	49	7,3
Driven+Cluster	50348	11824252	26369	52	7,6

(b) MPEG-4 AVC PHP

Strategy	Switch	Firing	Miss	%	FPS
Robin	16389	232752	12855	78	12,7
Robin+Cluster	9929	226807	147	1	12,8
Driven	2808	233057	147	5	13,1
Driven+Cluster	2647	227228	91	3	13,2

(c) MPEG HEVC Main

Table 9: Comparison of static and dynamic scheduling strategies within the desktop implementation using the tested video decoders. The number of switches, firings and misses are expressed in 10^3

tion. Run-time and quasi-static scheduling strategies are again both experimented, but this time we consider them locally on static regions so as to really compare their efficiency. We consider once more the *inverse transform* of the HEVC decoder, described in Figure 44, and especially the first 4 paths.

To do so, we define 2 configurations. On the one hand, the 4 actors composing the path are mapped to the same processor and scheduled dynamically with our dynamic scheduler using *round-robin* strategy. On the other hand, the 4 actors are clustered in a composite node that is scheduled using SAS, then mapped to its own processor. Table 10 summarizes the results for each TU sizes, respectively the number of incoming tokens, the number of cycles to process them for both scheduling strategies, and the acceleration rate resulting from the clustering.

The results clearly show that TTA processors can benefit from static scheduling by presenting interesting acceleration rates depending on the sizes of the TU. In fact, the compile-time predictability allows much more potential ILP by removing the test and validation of the firing rules.

6.5 CONCLUSION

As discussed throughout this thesis, although dynamic dataflow programming can be considered as a flexible approach for the development of scalable applications, there are still some concerns about its efficiency. Therefore,

Region	Tokens	Round-robin	Clustering	Acc
IT4x4	13696	385876	272982	1.41
IT8x8	38848	986408	664708	1.48
IT16x16	42752	2917961	1557331	1.87
IT32x32	19424	1747047	1567697	1.11

Table 10: Comparison of static and dynamic scheduling strategies on the *inverse transform* of HEVC using our embedded implementation

we have presented in this chapter an optimized software implementation of dynamic dataflow programs, including details about the communication and the scheduling. In fact, we have partially solved one of the main challenges of dynamic dataflow models by demonstrating for the first time, to our knowledge, that DPN-based video decoders achieves real-time framerate of high-definition sequences on desktop processors.

This demonstration has also given us the chance to deeply analyze the video decoders that have been developed within the RVC framework. As a result, we claim that the granularity of the description is the key to achieve high performance. On the one hand, the application has to be decomposed in a sufficient number of actors to be balanced equitably on a multi-core platform. On the other hand, the actors have to contain a sufficient amount of computations to take advantage of the parallel capabilities of each processor core.

Now, the next Chapter focuses on multi-core scheduling of dynamic dataflow programs starting from our optimized implementation in order to demonstrate the scalability of our approach.

7 SCALABLE MULTI-CORE SCHEDULING OF DYNAMIC DATAFLOW PROGRAMS

Everyone knows Amdahl's Law,
but quickly forgets it.

— Thomas Puzak, IBM, 2007

The emergence of massively parallel architectures has revived the interest on dataflow programming. In fact, programming multi-core platforms involves complex and error-prone tasks (Chapter 2), but describing an application by means of dataflow programming naturally guarantees its decomposition in multiple tasks while exposing data dependences (Chapter 3). On the one hand, the practicality and the expressiveness of dynamic dataflow languages enable a pragmatic development of complex applications. On the other hand, dynamic dataflow languages come up with the need of runtime scheduling decisions in order to achieve performance requirements, especially on multi-core platforms given the number of parameters involved. Actually, the mapping of applications has been identified as one of the most urgent problems to be solved for implementing embedded systems [155].

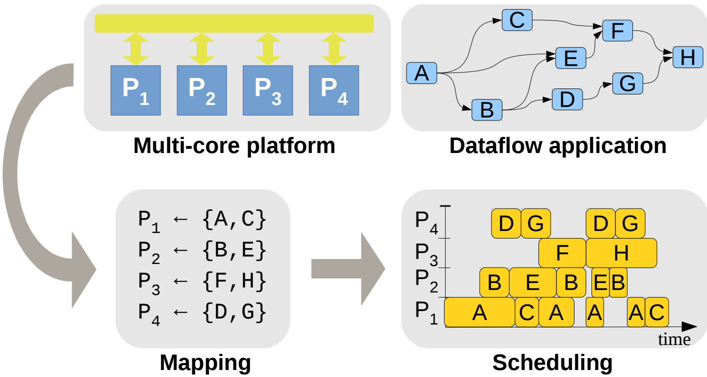


Figure 45: Multi-core scheduling of dynamic dataflow programs

Chapters 5 and 6 have presented our development environment and a methodology to translate dynamic dataflow descriptions into optimized software codes that can be executed on processors. Now, this chapter describes a set of run-time multi-core scheduling systems (Figure 45) that can handle the dynamism of our DPN-based programs and achieve scalable performance. In other words, management systems dynamically *assign* actors to processors, *order* actor execution, and *run* actors, without causing *starvation* or *deadlock*. All of our algorithms aim at maximizing data throughput for applications so as to achieve real-time constraint. Similarly to Chapter 6, we assume that the communication channels are previously sized by the application developer.

This chapter makes the following contributions:

1. A runtime system based on genetic algorithms that automatically searches efficient actor mappings for DPN-based programs.

2. A low-cost actor mapping system for dataflow programs based on graph partitioning. The low-cost of the approach makes it doable repeatedly at runtime to maintain a good load balancing over the targeted platform even with dynamic applications.

3. A runtime actor scheduling system for DPN-based programs that supports multi-core architecture thanks to a distributed architecture.

This chapter is organized as follows. We start by introducing our actor mapping systems in Section 7.1. Then, we describe our actor scheduling systems in Section 7.2. Finally, we present an analysis of the scalability of the video decoders developed within the RVC framework, which enables the evaluation of our whole multi-core scheduling system (i.e. actor mapping and scheduling).

7.1 ACTORS MAPPING

Executing efficiently programs onto multi-core platforms requires to balance the computational load over the processor cores so as to limit the consequences of sharing resources on the global performance of the system.

Actually, the load balancing can be achieved either by statically determining the computational load or by migrating the actors during the execution. But, in general, the determination of the computational load at compile-time is made impossible by the strong expressive power of dynamic dataflow programs. Thus, finding efficient mapping of dynamic dataflow programs requires run-time analysis. The challenge is then to define low-cost analyses to limit their execution overhead.

7.1.1 *Definition of the metrics*

Let us introduce the different parameters that are involved in the actor mapping. First, we consider the constraints that restrain the mapping:

- The number of the computational **resources** is constrained by the platform itself. When the number of available processors on the platform is inferior to the number of actors composing the description of the application, several actors have to share the same processor core.

- The **interconnections** between the processor cores and the memories are also constrained by the platform itself. When two connected actors are mapped onto two distinct processors, these two processors must share a common memory to let the actors communicate.

Then, we define some metrics about the given application and the targeted platform that help us to determine an efficient actor mapping:

- Knowing that the **communications** are the common bottleneck of dataflow applications, the *connectivity* between actors, as well as the communication *rates*, are key factors of the actor mapping. For instance, mapping two actors which frequently exchange information onto the same processor reduces the communications between the processors so as to limit the pressure on the shared memory and the interconnection network.

- The **performance** of a dataflow program is usually characterized by its *throughput*, which is limited by the critical path of the program.

Thereby, we need to reduce the impact of the actor mapping on the critical path and consequently on the global performance. To this end, we define the *workload* of an actor, in a given time interval, as the ratio of the computation time to the total execution time on the targeted platform. As a result, the workload of a processor is the sum of the workloads of all its actors plus a small overhead introduced by their scheduling. When the workload of a processor overcomes its computation capacity, the critical path may be increased because of the global data dependence of a dataflow program.

Additional metrics could also be investigated such as the behavior of the actor along with the processor affinity, either *compute-intensive* or *control-intensive* (Section 6.4), or such as the power consumption of a processor resulting from the execution of an actor, i.e. the effect on dynamic power consumption, etc.

Now, starting from this analysis of the constraints and the metrics involved in the partitioning of our dataflow application, we present two different run-time actor mapping systems: The first one based on genetic algorithms and the other one on graph partitioning.

7.1.2 *Evolutionary-based Actor Mapping*

In general, genetic algorithms are used to determine a set of good solutions to a given problem in a faster way than exhaustive researches [84], which are often inapplicable when the number of solutions is too important. In fact, genetic algorithms are one of the several existing ways to solve an optimization problem such as integer linear programming.

Genetic algorithms are an analogy of Darwin's evolution theory and follows the same evolution steps. To explain them, let us first introduce the main vocabulary (Figure 46):

GENE A part of the solution to solve. Here, a gene is the assignment of an actor to a specific core of the platform.

INDIVIDUAL Any possible solution to the problem. Here, an individual is the partitioning of all the actors composing the application over the multi-core platform.

POPULATION A group of individuals, in other words a set of distinct partitioning solutions. The population regularly evolves in a new generation that intends to contain better solutions to the problem.

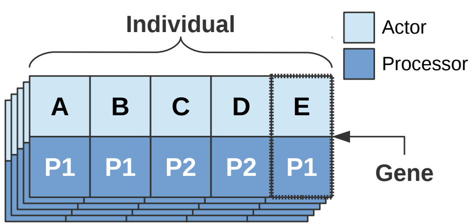


Figure 46: A population considering the mapping of the actors A, B, C, D and E onto the processors P1 and P2

Now, let us introduce the proceeding of our evolutionary-based actor mapping system. Our genetic algorithm is composed of the 4 consecutive

Figure 47: Proceeding of our evolutionary-based actor mapping system

steps that are performed repeatedly, such as presented in Figure 47. This process continues until a suitable solution has been found or until a predefined number of generations have passed:

1. **Initialization:** First, the individual contained in the initial population are produced randomly, except for a set of predefined ones used to speed-up the resolution. For instance, a specific individual that maps all actors to a single core is added to serve as the lower bound during the selection. Besides, the size of the population is an important parameter of generic algorithm: It has to be large enough to find interesting solutions, but small enough not to slow down the research.

2. **Evaluation:** Then, the individuals composing the population are successively evaluated. In other words, the application is partitioned over the platform according to the individual specification, and is executed during a given time-slice to evaluate its efficiency. In our case, the evaluation considers the throughput, i.e. the frame-rate after decoding few frames of the video sequence. This process is applied repeatedly until the whole population has been evaluated, that is why the choice of the size of the population is important.

3. **Selection:** Next, a set of interesting individuals is selected. In our implementation the selection is made by a one-turn tournament, i.e. two individuals are randomly chosen and the one with the better evaluation is kept.

4. **Evolution:** Finally, a new generation of population is created thanks to evolutionary mechanisms. Selected individuals are *crossed* and *mutated* so as to find better solutions (Figure 48). The cross-over (a) is done by cutting two individuals at a randomly chosen position so that the left side of one gene is joined with the right side of the other one and vice versa, producing two new individuals. The mutation (b) is a random modification of a randomly-chosen gene of a given individual.

Genetic algorithms are well-known for their applicability to multi-objective optimizations [78]. Thus, our evolutionary-based actor mapping could be extended with additional objectives like power consumption.

7.1.3 *Graph Partitioning problem*

The mapping of an actor-based application onto a multi-core platform is equivalent to the partitioning of the dataflow graph describing the application.

Given a platform composed of k processors and an application graph $G = (V, E)$ with V a set of vertices modeling the actors and E a set of edges representing the communication channel between the actors. We define $|V|$ the number of vertices and w a function assigning weights to each vertex

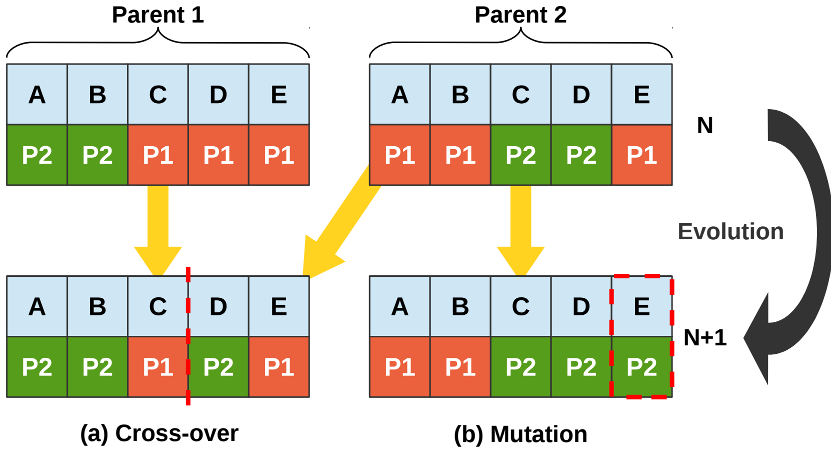


Figure 48: 2-ways evolution of the population from generation N to $N+1$: New individuals are created either by (a) cross-over or by (b) mutation.

$v \in V$ such that the weight of a vertex corresponds to the workload of the actor represented by the vertex.

The k-way graph partitioning problem is to partition V into k subsets $V_1, V_2, ..., V_k$, with $V_i \cap V_j = \emptyset$ for $i \neq j$ and $\bigcup_i V_i = V$, such that the sum of the vertex-weights in each subset is balanced and the number of edges whose incident vertices belong to different subsets is minimized. Consequently, the actors will be balanced onto the processors according to their workload in order to minimize the critical path of the whole system. Furthermore, since the algorithm minimizes the edge-cut, the communications between the processors are also minimized.

7.1.4 *Graph partition methodology*

The graph partitioning problem is NP-complete but some algorithms find high quality partitions in short time using multilevel scheme. A k-way partition is solved by recursive bisection, i.e. the graph is successively split into two balanced partitions. Basically, the bisection is performed using a multilevel algorithm based on the three following phases:

1. **Coarsening phase:** First, the graph G is successively transformed into a series of smaller graphs G_0, .., G_m such that $|V| > |V_0| > ... > |V_m|$. Each of them is the result of the contraction of some edges from the previous graph. For instance, contracting an edge $(a, b) \in E$ is performed by creating a new vertex $c \in N$ with a weight $w(c)$ equal to the sum of the weight of the edge $w(a, b)$ plus the one of each vertex $w(a)$ and $w(b)$. And, in case the vertice a and b are both connected to the same vertex d, a new edge (c, d) is created such as $w(c, d) = w(a, d) + w(b, d)$.

2. **Partitioning phase:** Then, a bisection of the latest graph G_m is performed, i.e. the coarser graph is partitioned in two balanced partitions. Since the reasonable size of the coarser graph, well-known bisection methods such as the Kerninghan-Lin heuristic [112] can find satisfying solutions in a reasonable time.

3. **Uncoarsening phase:** Finally, the resulting partitions are projected back to the original graph by following each transformation made during the coarsening phase, i.e. building successively the equivalent partitions in each intermediate graphs G_0, .., G_m. Between each step, a refinement of the partitioning is performed using the Kerninghan-Lin heuristic [112].

Such an algorithm has been implemented in a tool called Metis [111]. This tools is able to partition a graph with millions vertices into hundreds parts in a few seconds, but does not support constraints on the partitioning (interconnection network, core affinity, etc).

7.1.5 *Mapping Flow*

Our metrics-based mapping flow maps an application based on a dynamic dataflow model onto a multi-core platform. Since the dynamic behavior of our application makes it unpredictable in most cases, our approach is based on a low-cost profiling analysis of the execution. Consequently, we assume that minimal profiling mechanisms are available on the targeted platform.

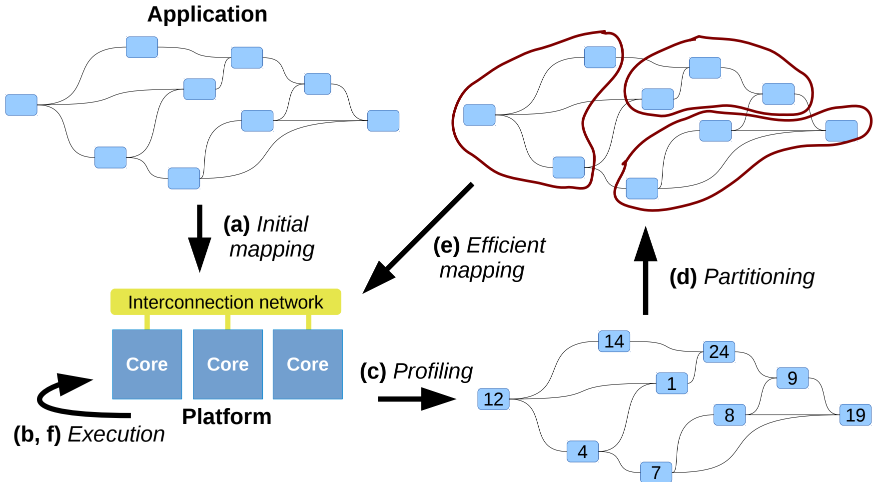


Figure 49: Actor mapping flow based on graph partitioning

Figure 49 shows the main steps of our mapping flow. The flow starts from the compilation of the application for the targeted platform by using an initial mapping (a), e.g. executing all actors on the same processor core. Then, the application is run on the platform (b) during a predefined time-slice that has to be long enough to match the full behavior of the application. After that, the execution profile is analyzed (c) to find out the *workload* of each actor, and thus to get a weighted application graph. For now, the *communication rates* are not considered, only the *connectivity* of the graph, i.e. all the edges are weighted by 1. Next, the graph partitioning is performed (d) to determine an efficient mapping of the actor onto the available cores of the platform. Finally, the application is recompiled, or reconfigured, according to the computed mapping (e) to enable its efficient execution (f) on the targeted platform.

Finding an optimal mapping of an unpredictable application over a multi-core platform is an NP complete problem. This is why we solve an equivalent graph partitioning problem by using reputed heuristics that are able

to find an interesting solution in few milliseconds. Moreover, this mapping flow is doable at regular time to keep a good balance of the application over the processor cores, and thus to get an efficient processing during the execution of the application.

7.2 ACTOR SCHEDULING

Being able to partition the actor network over the multi-core platform is not sufficient, we need to schedule the actor execution on each processor core. Thus, this section describes a *distributed* and *lock-free* scheduling system that can use *round-robin* and *data-driven / demand-driven* strategies to execute dynamic dataflow programs onto multi-core platforms.

7.2.1 *Distributed Scheduler*

We consider a *distributed* scheduler in charge of the execution of our DPN-based programs over multi-core platform. Thus, our scheduler is composed of multiple local schedulers that are executed in parallel, each one being assigned to its own processor core. As a result, we avoid the design of a *centralized* scheduling scheme that might be performed by a dedicated core. Indeed, the centralization of the scheduling decision would not scale well with the increasing number of cores, especially with the fine-granularity of the scheduling within DPN-based programs.

Beside the architectural aspect, we assume here that all the actors have previously been mapped over the platform, either by hand or by using an automated actor mapping system such as the ones presented in Section 7.1. We also assume that the actors remain attached to the processor core to which they are assigned, during the execution of our distributed scheduler. In fact, migrating stateful actors among the cores might imply a large overhead to the execution. That is why, we claim that the migration should be allowed only at given reconfiguration points which could occur at regular intervals of time in order to maintain the load balancing.

7.2.2 *Multi-core Scheduling Strategies*

Now, let us take a look at the way to apply dynamic scheduling strategies within our distributed scheduler. We have presented, in Section 3.6 of Chapter 3, two dynamic scheduling strategies dedicated for DPN-based programs, known as *round-robin* and *data-driven / demand-driven*, both executing an actor until it cannot fire anymore.

On the one hand, the round-robin strategy simply goes over a list of all the actors that are statically ordered. Thus, realizing the round-robin strategy within our distributed scheduling just requires splitting this list among the local schedulers in accordance with the actor mapping, such as presented in Figure 50.

On the other hand, the data-driven / demand-driven strategy dynamically orders the execution of the actors, that requires communications between the local schedulers. In fact, this strategy uses a dynamic list that contains the next schedulable actors. When the current actor cannot fire anymore, its predecessors/successors are added to the list depending on the reason of the blocking. But, these predecessors and successors can possibly be assigned to a different core than the one of this actor. Thus, this strategy requires communications among the local schedulers, that are in-

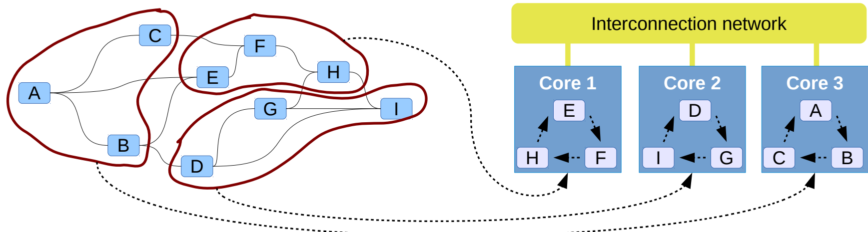


Figure 50: Distributed scheduler onto a multi-core platform using round-robin

vestigated in Section 7.2.3. For example in Figure 50, if C is blocked during its execution because its common FIFO channel with F is full, then the scheduler has to add C to the list of schedulable actors. However C is managed by another scheduler so an inter-scheduler communication is needed.

Moreover, we propose a new scheduling strategy that both combines round-robin and data-driven / demand-driven to avoid starvation of our local schedulers. Indeed, distributed *data-driven / demand-driven* strategy cannot guarantee the non-emptiness of the schedulable list. Thus, our combined strategy initially applies data-driven and demand-driven principles, but switches to the static list when the schedulable list is empty, as presented in Listing 16.

```
1  void CombinedScheduling(Scheduler *sched) {
2    Actor *actor;
3    while(1) {
4      if(sched->ddd_next_schedulable == sched->ddd_next_entry) {
5        // Round-robin when the schedulable list is empty
6        actor = sched->actors[sched->rr_next_schedulable];
7        sched->rr_next_schedulable++;
8        if (sched->rr_next_schedulable == sched->num_actors) {
9          sched->rr_next_schedulable = 0;
10       }
11     } else {
12       // Data-driven / demand-driven otherwise
13       actor = sched->schedulable[sched->ddd_next_schedulable %
             MAX_ACTORS];
14       actor->in_list = 0; // not a member of the list anymore
15       sched->ddd_next_schedulable++;
16     }
17     actor->sched_func(); // Execute the next actor
18   }
19 }
```

Listing 16: Combined scheduling algorithm

7.2.3 *Lock-Free Scheduling Communications*

Lock-free communications between distributed schedulers are used to avoid the synchronization of local schedulers. Indeed, the fine granularity of the actors makes the actor scheduler a critical part of the execution so that the smallest overhead can have disastrous consequences on the performance. Moreover, it is essential to order remote schedulers to add new schedulable actors to their schedulable lists: Otherwise, a deadlock could occur during the execution. In other words, if this scheduling information is not com-

municated, a self-contained cycle can appear and cause a deadlock of the application.

Another kind of FIFO channels called *scheduling-fifos* is used to communicate between schedulers without synchronization. Lamport has proved that locks are not necessary in the case of single producer, single consumer FIFO [115]. However it is important not to confuse the two types of FIFO which work with the same mechanism but have two distinct uses: the FIFO channels are used to carry on the application stream and the *scheduling-fifo* channels are used to share scheduling informations (in our case a set of next schedulable actors).

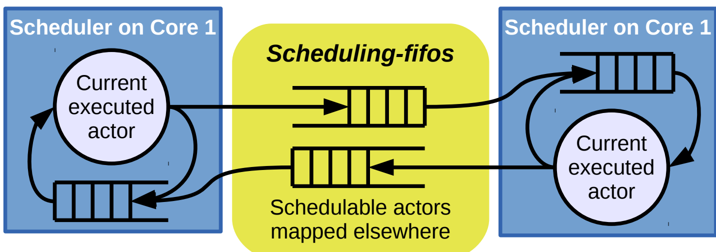


Figure 51: Lock-free scheduling communication for data-driven / demand-driven strategy on multi-core platforms

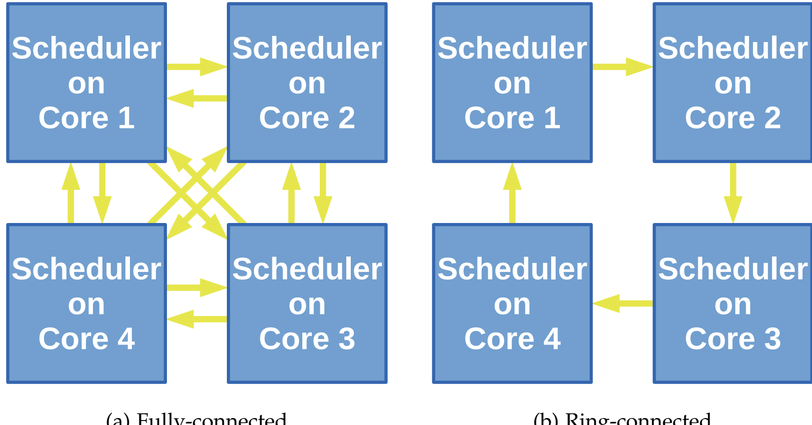
Figure 51 shows the inter-core communication mechanism. When an actor execution is blocked, the scheduler adds the predecessor or the successors of the blocking FIFO channels to its schedulable list. In some cases this actor is not executed by the current scheduler so this actor is sent to its associated scheduler by a *scheduling-fifo* channel.

We propose two kinds of communication network topology (Figure 52):

- A **fully-connected topology** that interconnects the schedulers using a bidirectional communication channel between each couple of schedulers. As a consequence, the communication overhead is limited since the schedulers communicate directly, but the number of *scheduling-fifos* increases exponentially according to the number of cores.
- A **ring-connected topology** that interconnects the schedulers with a limited number of *scheduling-fifos*: On a N-core processor, N *scheduling-fifos* are needed. However, with this topology the communication could cross $N - 2$ schedulers in the worst case before the targeted scheduler receives it. For example in Figure 52b, when the scheduler on core 1 wants to communicate with the other one mapped on core 4, the schedulers on cores 2 and 3 are used as intermediaries.

7.3 SCALABILITY ANALYSIS OF RVC-BASED VIDEO DECODERS

In Chapter 6, we have analyzed the implementation of RVC-based video decoder onto desktop and embedded platforms. Now, this section focuses on the scalability of these implementations. Thus, we have realized a set of experiments to demonstrate this scalability using our mapping/scheduling algorithms.



(a) Fully-connected (b) Ring-connected

Figure 52: Topologies of the interconnection network of the distributed scheduling management

7.3.1 *Experimental setup*

Similarly to Chapter 6, we study our multi-core scheduling strategies on video decoders using both the desktop and embedded implementations that have been specified in Section 6.4. However, we need to introduce some details concerning our parallel implementations:

A. **Desktop implementation:** This implementation is still generated with the C back-end of Orcc (*orcc-c*). The multi-core ability of this implementation is achieved thanks to multi-threading upon the operating system: One thread is created for each local scheduler and forced to be run only on its associated core processor. Besides, we have both implemented *round-robin* and *combined* scheduling strategies in the runtime library (previously called *orcc-lib*).

   Concerning the platform that has been used during these experiments, our Intel Xeon W3670 is composed of 6 homogeneous cores sharing 12MB of L3 cache.

B. **Embedded implementation:** This implementation is still generated with the TTA back-end of Orcc (*orcc-tta*). As opposed to the desktop one, our embedded implementation does not require multi-threading. Indeed, the processor cores execute their own binaries and communicate thanks to common address-spaces withing the shared memories. As a result, the execution cannot be disturbed by processes external to the tested application.

   However, for now, our embedded implementation supports only the *round-robin* scheduling strategy because of its simplicity. Additionally, our embedded implementation does not support arbitration between concurrent accesses to the shared memories. In other words, the processor cores are directly connected to each others through dual-port block RAMs of the FPGA board on which the platform is synthesized (See Section 5.3 to get more details about the co-design flow).

A global view of the experimental flow integrating our automated actor mapping system is presented in Figure 53. After compiling the application

for one of the multi-core platforms, its execution is profiled by using profiling tools, e.g. Valgrind, or by computing the decoding frame-rate. This profiling is made, depending on the targeted platform, either during an instrumented execution for the desktop implementation or during an instrumented simulation for the embedded implementation. Then, our actor mapping system, known as *orcc-map*, interacts either with our genetic algorithm or with *Metis*, the external tool used for partitioning the graphs [111], in order to determine a new mapping. Finally, the application is recompiled or reconfigured according to the computed mapping.

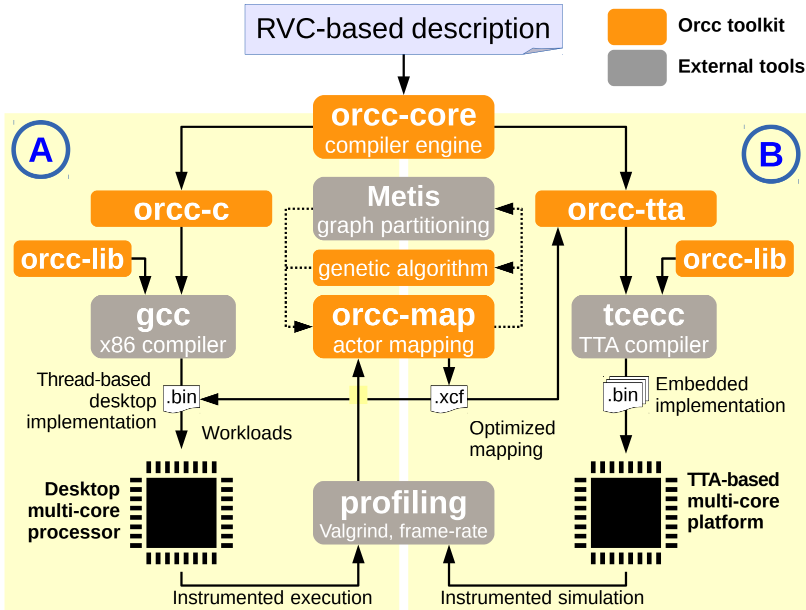


Figure 53: Experimental flow to analyze the scalability of RVC-based applications on both desktop (A) and embedded (B) multi-core platforms

Our implementations have been experimented on the same descriptions of video decoders, as well as the same video sequences, as those we have previously used.

7.3.2 *Desktop Multi-core Implementation*

We start by analyzing the scalability of our implementation of RVC-based video decoders onto a desktop multi-core processor.

ACTOR MAPPING First, let us compare the different approaches to map the actors composing our video decoders onto the processor cores. To do so, we have evaluated the execution of the MPEG-4 Part 2 decoder over 2 processors after applying either *hand-made* or *automated* mapping methodologies, both by using the *round-robin* scheduling strategy. On the one hand, our hand-made mappings try to balance the application over the multi-core platform in natural ways, either by separating the decoding of the image components (chroma/luma) or by pipelining the decoding steps (parsing, residual, prediction). One the other hand, we have applied our automated mapping system using our genetic algorithm and using the graph partitioning. Table 11 summarizes the observed frame-rates, as well as the acceler-

ation factor compared to the execution with one processor core, for each configuration.

Table 11: Comparison of our approach with handmade mappings of MPEG-4 Part 2 executed on 2 processor cores

Actors mapping	Frame-rate	Speed-up
Handmade Luma/Chroma	39 fps	1.15x
Handmade Pipelining	45 fps	1.33x
Genetic Algorithm	56 fps	1.66x
Graph partitioning	58 fps	1.72x

The results show the difficulty to find pertinent mapping solutions by hand, even with a simple problem (2 cores and 41 actors). Thus, this difficulty demonstrates the interest of an automated approach.

Additionally, the results highlight the difference between our two actor mapping systems, based either on the genetic algorithm or on the graph partitioning. On the one hand, the performance achieved using both methodologies are very similar. On the other hand, our genetic algorithm requires much more time to find a good solution because of its iterative methodology, and its resolution time increases exponentially with the complexity of the problem. Moreover, the system simply evaluates the global performance, by means of the throughput, without considering the communications. But, these communications quickly become the main limitation when the number of actors and processors grows. All of these makes our genetic algorithm hardly usable at run-time. As a result, the next experiments consider only our actor mapping system based on graph partitioning.

ACTOR SCHEDULING Now, Table 12 summarizes the decoding frame-rates obtained for each application using our different scheduling strategies, i.e. the round-robin strategy and the combined strategy, the latter using both the ring-connected topology and the fully-connected topology.

First, the results clearly show the bounded scalability of the tested RVC-based descriptions of video decoders onto a desktop multi-core processor. Indeed, these descriptions present acceleration factors from $1,37$ to $3,16$ with 6 processor cores available. This can be explained by the limited decomposition and the data dependences within our decoders, and by the overhead caused by the inter-core communications due to cache effects. As presented in Figure 42 of Chapter 6, the highest actor workloads go up to around 20% and 30% of the total workload of the decoder. In theory, the maximum speed-up would be up to around 3.3x and 5x. In practice, the parallelism is limited by the cost of the communications between the cores, and by the data dependences between the actors, such as the loop reaction of the data stream between the image buffer and the inter prediction which is a known bottleneck of dataflow-based video decoders.

Moreover, at first glance the results surprisingly show that MPEG-4 Part 2 SP and MPEG-4 AVC are more scalable than MPEG HEVC. On further examination, this can be explained by the development status of our HEVC description that does not yet parallelize the video decoding beyond the common decomposition (i.e. parsing, residual decoding, prediction and filtering), and by the current high complexity of the inter-prediction (about 30% of the global workload). However, the HEVC / H.265 standard has been

Strategy	RR		C/R		C/F	
Nb cores	FPS	Acc.	FPS	Acc.	FPS	Acc.
1	33,7	-	34,6	-	-	-
2	58,1	1,72	58,4	1,68	-	-
3	80,5	2,39	80,8	2,34	80,3	2,32
4	94,6	2,81	92,2	2,66	**93,2**	**2,69**
5	**95,4**	**2,83**	**92,5**	**2,67**	90,7	2,62
6	93,1	2,76	90,1	2,60	90,3	2,61

(a) MPEG-4 Part 2 SP

Strategy	RR		C/R		C/F	
Nb cores	FPS	Acc.	FPS	Acc.	FPS	Acc.
1	4,5	-	7,3	-	-	-
2	6,7	1,49	12,4	1,69	-	-
3	10,6	2,36	16,3	2,23	16,0	2,19
4	12,0	2,44	16,9	2,32	17,0	2,32
5	13,1	2,91	19,0	2,60	18,8	2,58
6	**14,2**	**3,16**	**19,2**	**2,63**	**19,3**	**2,64**

(b) MPEG-4 AVC PHP

Strategy	RR		C/R		C/F	
Nb cores	FPS	Acc.	FPS	Acc.	FPS	Acc.
1	12,7	-	13,1	-	-	-
2	16,4	1,29	**18,0**	**1,37**	-	-
3	**17,4**	**1,37**	17,3	1,32	17,2	1,31
4	15,3	1,20	15,6	1,19	15,8	1,21
5	15,2	1,19	14,9	1,14	15,1	1,15
6	10,4	0,82	9,3	0,71	10,4	0,79

(c) MPEG HEVC Main

Table 12: Bounded scalability of the RVC-based video decoders according to decoding frame-rates (FPS) obtained on a desktop multi-core processor. The decoders are first mapped using our dedicated system based on graph partitioning, then scheduled using different strategies, respectively round-robin strategy (RR) and combined strategy with either ring topology (C/R) or fully-connected topology (C/F).

designed to take advantage of the increasing parallelism potential of the decoding platforms [164]. Thus, future versions should be much more scalable thanks to the new advanced decomposition (wavefront, tiles, etc).

Finally, the results show that the combined strategy is more interesting when the application contains a large numbers of actors. For instance, the MPEG-4 AVC decoder, which contains many more actors (114) than the others (41 and 34), always obtains its best frame-rates using the combined strategy. However, the results also show that the round-robin strategy becomes

more interesting when the number of cores increases. This can be explained by the fact that the improvement of the performance brought from a more advanced strategy is hidden by the larger scheduling overhead. Indeed, the potential of improvement decreases because the mapping reduces in general the number of actors on each core.

Furthermore, even if the fully-connected schedulers show better results than ring-connected one, the slight difference demonstrates the interest of the ring-based topology.

7.3.3 *Embedded Multi-core Implementation*

Now, let us take a look at our embedded implementation. As said previously, the actor are mapped by our automated system based on graph partitioning and scheduled by the round-robin strategy. Figure 54 presents the influence of the number of processors on the frame-rate of the MPEG-4 Part 2 SP decoder. In this case, we consider the decoding of a video sequence with a smaller definition, i.e. *foreman* at QCIF resolution, to reduce the simulation time and to reduce the size of the *framebuffer*. Actually, the results can be extrapolated for higher definition sequences by dividing the frame-rate according to the size of the video. The decoding is simulated onto our TTA-based multi-core platform clocked at 100MHz, using the *Fast* configuration for the processors.

We do not present the results with the other decoders for the following reasons: On the one hand, the MPEG HEVC / H.265 decoder is still being developed, and the *inter-prediction* is currently a major bottleneck to any parallelization. One the other hand, the development of the embedded implementation of the MPEG-4 AVC / H.264 decoder would require too much work (test, debugging, tuning, etc) in view of the perspectives.

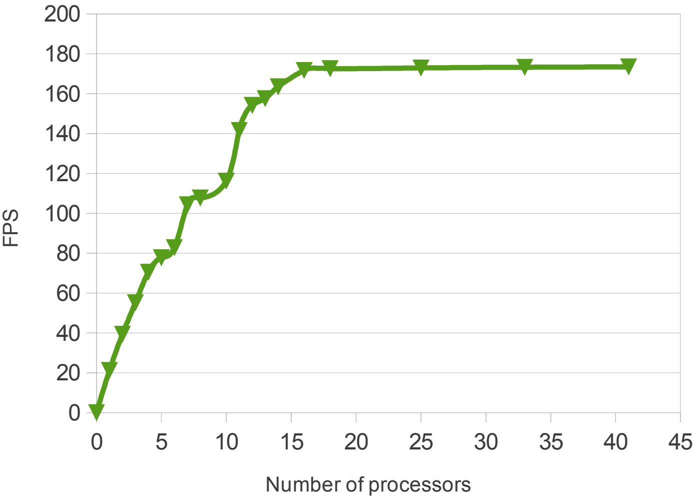


Figure 54: Influence of the number of processors on the performance of MPEG-4 Part 2 Simple Profile decoder

The form of the curve clearly shows the limit of the coarse-grain parallelism (task-level) of the application. Actually, the maximum decoding frame-rate of our MPEG-4 Part 2 SP decoder is reached with 16 processors. Increasing further the number of processors does not provide higher decoding frame-rate.

Thus, the maximum speedup in comparison with the single processor execution is 8.2x, and achieved with 16 processors. Therefore, the maximum speedup achieved with our embedded implementation is much bigger than the maximum speedup achieved with our desktop implementation (which seems to be around 3x). This can be mainly explained by the fact that the communications between the cores within our embedded implementation do not induce any overhead. To conclude, these results demonstrate the interest of the dedicated memory organization that we have designed specifically for our embedded multi-core platforms (see Section 5.2).

7.4 CONCLUSION

In this chapter, we have presented a set of algorithms so as to schedule dynamic dataflow programs over multi-core platforms. We have also proposed an automated and low-cost metric-based actor mapping system, which is doable at run-time in order to overcome the unpredictability of our applications. The mapping is described as an equivalent graph partitioning problem both considering the architectural constraints and the profiling information. Then, the graph partitioning problem can be solved using dedicated tools in order to balance the workload of the whole application.

Starting from our work on multi-core scheduling, we have then analyzed the scalability of RVC video decoders, including the one based on the new HEVC standard. As a result, this scalability seems mainly limited by the cost of the communication between the cores, which is not really surprising.

The next and final chapter concludes this document by summarizing the work we have presented all along this thesis. Finally, we detail a set of promising perspectives for future work that could take advantage of our work to improve the development of dynamic dataflow programs, and especially video codecs.

8 CONCLUSIONS AND OUTLOOK

If you want a happy ending,
that depends, of course,
on where you stop your story.
— Orson Welles

To achieve both the high constraints of today embedded systems and the time-to-market pressure, the electronic market is clearly turned to the design of heterogeneous platforms, known as MPSoCs, integrating more and more specialized processors onto a single chip in order ononto bridge the gap between hardware efficiency and software flexibility. But, surprisingly, few applications, like video codecs, are still performed by dedicated hardware components in most of current MPSoCs. The work presented in this thesis takes place in a context of increasing demand for multimedia applications along with the lack of flexibility of embedded devices with video codecs.

8.1 SUMMARY

As presented in Chapter 2, expressing an algorithm so as to exploit all its parallelism is already a complex and error-prone task, and the variety of parallel platforms makes the task incredibly harder. As regards to the complexity of parallel programming, embedded computing has to move towards programming methodologies based on higher level of abstraction to achieve their high constraints of efficiency and reliability.

Chapter 3 focuses on a particular programming solution, known as dataflow programming, which relies on MoC formalism defining semantic rules to compose and connect concurrent components, the *actors*, with a dataflow graph, the *network*. Tonhen, we demonstrate that the dynamic dataflow model, also known as DPN, is the best trade-off considering the practical point of view of programmers. But, the DPN model also raises the challenge of efficient implementations.

MPEG attempts to improve the standardization process of video coding standards by following the same methodology, as described in Chapter 4. Thus, MPEG has introduced an innovative development framework, called RVC, which exploits dynamic dataflow programming to provide modularity, scalability and portability to video coding tools. While the efficient implementation of DPN-based programs is still challenging, tools and applications included in the RVC framework have now reached enough maturity to demonstrate the viability of this development approach for real-world applications.

Starting from this state-of-the-art, we have built an entire co-design flow (presented in Chapter 5), integrating an advanced simulation infrastructure, which relies upon an architecture model dedicated to dataflow programs. Building this co-design flow has also given the opportunity to enhance the reliability and the flexibility of the underlying compilation infrastructure by way of modern software engineering techniques.

Furthermore, we have partially solved one of the main challenges of dynamic dataflow models by presenting in Chapter 6 our optimized software

implementation of DPN-based programs. Indeed, our implementation is the first one, to our knowledge, that demonstrates video decoders achieving real-time frame-rate of high-definition sequences on desktop processors. Our implementation has also showed encouraging results on embedded multi-core platforms. Moreover, we have demonstrated the scalability of our implementation of DPN-based programs in Chapter 7 with a set of efficient actor mapping/scheduling algorithms that can handle the dynamism of our applications.

In our point of view, it appears important to point out that all contributions made during this thesis including our co-design flow, as well as the experimented applications, are available under open-source licenses and documented [134, 166].

8.2 PERSPECTIVES

All the work made during this thesis can serve as basis for future researches. We detail here some perspectives based on our development environment dedicated to dataflow programming, on our software implementation of dynamic dataflow programs, and on embedded platforms dedicated to RVC-based programs.

8.2.1 *An Even More Advanced Development Environment*

The development of our complete IDE for dataflow programming is a laborious task that is clearly open to new perspectives that could promote the development of new applications within the RVC framework.

ASSISTED DEVELOPMENT Our development environment includes an editor for hierarchical graphs that takes advantage of the visual programming ability of the dataflow paradigm. But, our editor suffers from its underlying technology, such as the automatic layout feature that offers poor results on complex graphs. Actually, we could benefit from dedicated meta-tools, such as Eclipse's Graphiti [63] and Spray [64], in order to develop a state-of-the-art graph editor. Moreover, the generative approach of these meta-tools could greatly simplify the development and the maintenance of our user interfaces.

Debugging RVC applications from our IDE is usually achieved by generating the C code for the host platform in order to use traditional debuggers, e.g. GDB. But, applications already rely on many implementation choices that can complicate the debugging. For example, the inspection of FIFO channels based on circular buffers is quite challenging during the debugging. As a result, integrating a debugger directly in our IDE could greatly facilitate the development of complex applications.

DESIGN FLOW The availability of new SoCs that embeds programmable logic (for example the Xilinx Zynq SoC), along with the ability of our compiler to target multiple platforms, enables the development of a design flow targeting complex heterogeneous systems (MPSoCs) including ARM processors, TTA-based processors and hardware accelerators.

Such an heterogeneity introduces however new challenges to our design flow. First, the simulation infrastructure should interconnect multiple simulators to handle the co-simulation of the whole system. Then, targeting heterogeneous platforms would imply the development of an advanced ac-

tor mapping process considering both hardware and software components. Finally, a multi-target build-system would be required to simplify the interaction with an increasing number of third-party tools.

8.2.2 *An Even More Optimized Software Implementation*

The efficiency of the implementation is clearly the central problematic of dynamic dataflow programming. Even if a big progress has been made since the introduction of the RVC framework, there is still a long way to go before the adoption of dynamic dataflow programming by industrials. But, we already have several perspectives that require future investigations to improve the performance of our *state-of-the-art* implementation.

COMMUNICATIONS The optimized implementation of communications, described in Chapter 6, prevents potential word-level parallelism due to absolute indexes. In fact, the accesses to the circular buffer could be vectorized since the width of FIFO channels within our applications are often inferior to 32 bits (usually 8 or 16 bits), but the compiler cannot know if the access are aligned in the memory or if the end of the circular buffer is reached. Thus, we are currently investigating a more advanced implementation of multi-rate communications. Multi-tokens actions are generated in two versions, standard and vectorized, that are executed according to the current position in circular buffer.

SCHEDULING The results presented in Chapter 6 have clearly shown that our application does not really benefit from quasi-static scheduling. The main reason comes from the limits of our current classifier. Our classifier does not interpret the communication between actors, so it cannot perform network-level analyses. Moreover, our classifier is unable to extract complex behavior within a single actor, such as indirect dependences between guards and tokens. A solution would be the utilization of dedicated third-party tools to perform a more reliable abstract interpretation.

```
1  package devel.org.sc29.wg11.mpegh.part2.main.IT;
2
3  actor IT_Splitter()
4    int(size=16)  Coeff,
5    @Classif=param  // Annotation
6    int(size=7)   Size
7      ==>
8    int(size=16)  Coeff_4x4_DST,
9    int(size=16)  Coeff_4x4_IT,
10   int(size=16)  Coeff_8x8,
11   int(size=16)  Coeff_16x16,
12   int(size=16)  Coeff_32x32,
13   int(size=16)  Coeff_skip:
14
15   // Body
16
17 end
```

Listing 17: Directive-based actor classification

Another solution would be the introduction of dedicated directives, similarly to the ones used for the assisted parallelization scheme described in Chapter 2. The developer could help the compiler to find quasi-static be-

havior within its application by adding specific directives to the description. In fact, a directive simply contains additional information that may not be determined automatically by the compiler. For instance, Listing 17 presents an actor in which the input port `Size` has been annotated as a parameter.

To summarize, we provide a quasi-static semantic to our dataflow programming language that is initially dynamic. The challenge would then be to identify what information would be useful to help the actor classification. While this approach would be unsatisfying from a practical point of view, it would represent another step to bridge the gap between the expressive power of dynamic models and the efficiency of static/quasi-static models.

8.2.3 *Towards a Platform Dedicated to RVC-based Video Decoders*

This thesis targeted initially the investigation of an MPSoC-based platform, combining both hardware and software components, dedicated to video coding applications that benefit from the RVC framework. While the design of such a platform is not achieved yet, our design flow already enables the DSE of dedicated embedded multi-core platforms based on our architecture model. So, we are now investigating a generic platform dedicated to all RVC-based codecs that opens many interesting perspectives for future works. In fact, several studies [98, 20, 31] have already investigated RVC-specific embedded platforms, but none has really demonstrated the viability of its proposal.

HETEROGENEITY As shown during this thesis, video decoders are complex applications composed of components with very different types of behavior. On the one hand, the results of our experiments have clearly demonstrated the interest of VLIW-like processors to take advantage of the internal parallelism of actors. On the other hand, the results have highlighted the weakness of TTA-based processors for handling the high dynamism of certain actors within our applications. A solution would be the addition to our architecture model of another type of processor including a branch predictor, the LatticeMico32 soft-core for instance. Thus, we would design dedicated platforms that could handle efficiently both compute-intensive and control-intensive actors composing our video decoders.

The heterogeneity of the platform would however introduce additional challenges to our co-design flow such as seen earlier. The heterogeneity would also introduce new constraints to the actor mapping. On the one hand, compute-intensive actors (e.g. the *inverse-transform*) should be mapped to TTA processors. On the other hand, control-intensive actors (e.g. the *parser*) should be mapped to the other processors. As a result, a dedicated analysis would be required to identify the behavior of actors.

CLUSTERING As described in Chapter 4, all video decoders share a common structure that globally matches with the application graph of their RVC description. Thus, they are usually decomposed in 4 subnetworks: *parser*, *residual*, *prediction* and *filters*. In fact, the interconnections between these subnetworks are also quite similar in all decoders. Starting from this observation, we are investigating a dedicated memory architecture which clusters processors to fit with the global structure of video decoders. An early specification based on a two-level shared memory hierarchy is presented in Figure 55.

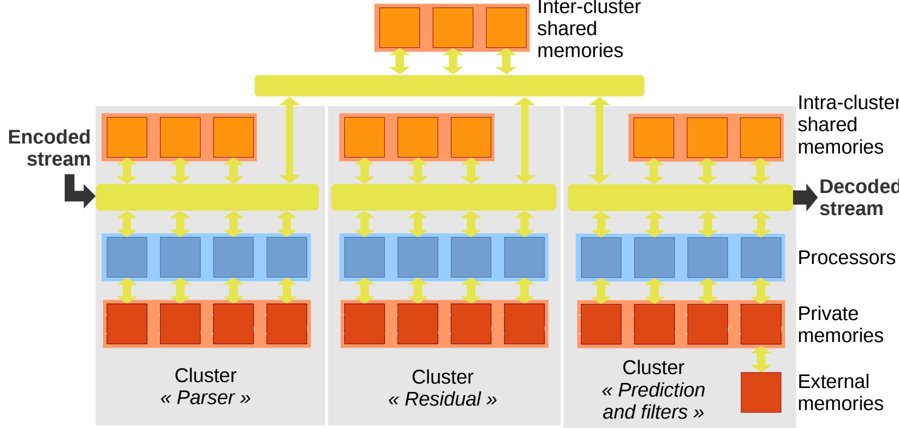


Figure 55: Dedicated clustered architecture based on video decoders structure

Moreover, we can also identify the actors that require large amounts of memory, such as the *frame-buffer*, and map them to dedicated processors that have access to the external memory via caches or scratch-pads.

ADAPTIVITY Previous works in our research group have already demonstrated the functioning of an adaptive video codec called Jade using the RVC framework. As described in Chapter 4, Jade exploits mechanisms of the virtual machine available in LLVM, such as just-in-time compilation, to provide a universal decoder. In fact, the decoder is configured *on-the-fly* according to a configuration encapsulated with the video stream and thanks to a library of components (i.e. the VTL). Thus, the incoming stream configures the decoder according to the encoding format of the video stream.

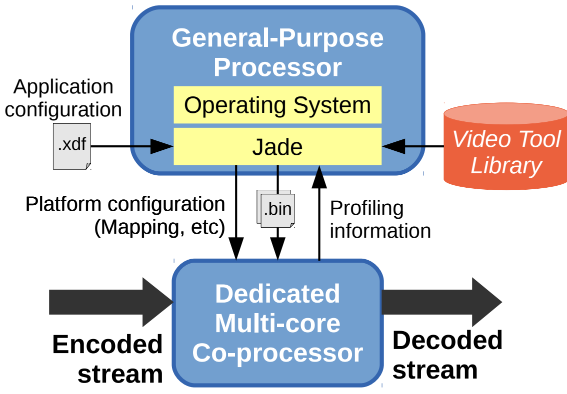


Figure 56: Hardware Adaptive Decoder Engine

So far, Jade is executable exclusively on GPP (x86 and ARM) which is not energy efficient. Thus, we could couple the GPP with our dedicated multi-core platform as a co-processor in charge of the video decoding, such as presented in Figure 56. To do so, Jade would be extended with a host/-client feature wherein the GPP would configure the multi-core co-processor which would perform the decoding. And, at the reception of an incoming stream, Jade would cross-compile the actors for the multi-core co-processor and would determine an initial mapping. Latter, the multi-core co-processor

would send back to Jade some profiling information to allow the determination of an optimized mapping.

Part III

APPENDIX

RÉSUMÉ EN FRANÇAIS

Qani lezu gidès, ainkhan él mart ès.

"Autant tu connais de langues,
autant de fois tu es un homme."

— Proverbe arménien

Cette thèse étudie les approches pragmatiques de programmation d'applications du monde réel pour les systèmes embarqués actuels et à venir. En fait, l'expérience de programmation est devenue un problème central pour l'informatique embarquée. D'une part, les systèmes embarqués sont devenus des systèmes matériels complexes, connus sous le nom *Multi-Processor System-on-Chip* (MPSoC), qui contiennent de plus en plus d'éléments assez hétérogènes sur une seule puce afin d'augmenter les fonctionnalités du produit et pour répondre aux attentes du marché. D'autre part, la complexité du logiciel déployé sur ces appareils ne cesse de croître de manière exponentielle, car ces derniers sont utilisés pour résoudre des problèmes techniques de plus en plus difficiles. En conséquence, les programmeurs doivent développer des applications de plus en plus complexes et les implémenter sur des appareils eux aussi de plus en plus complexes, tout en respectant des exigences de coûts et de temps de production.

Cette thèse vise à fournir un ensemble d'outils pour faciliter, d'un point de vue pragmatique, le développement d'applications réelles pour des plates-formes embarqués de type MPSoC. Ainsi, nous mettons en œuvre et évaluons un ensemble de méthodologies pour la conception de systèmes embarqués, de la spécification d'application à la mise en œuvre de la plate-forme. Afin de bénéficier de tout le parallélisme présent dans les algorithmes, les applications sont décrites par avance sous une forme décomposée, appelé *Dataflow Process Network* (DPN), au moyen d'un langage de programmation flux de données héritant de CAL Actor Language. Les applications sont ensuite exécuté sur des processeurs de type Very Long Instruction Word qui sont capables d'exécuter plusieurs opérations en même temps. Nous évaluons notre suite d'outils à l'aide de décodeurs vidéo provenant de l'état de l'art, dont l'un basé sur la nouvelle norme HEVC.

Maintenant, jetons un coup d'œil sur le paysage de l'informatique embarquée afin de comprendre la complexité de la problématique auquel cette thèse est confrontée.

A.1 SYSTÈMES EMBARQUÉS

Les systèmes embarqués sont aujourd'hui largement utilisés, bien plus que d'autres systèmes informatiques avec des milliards vendus chaque année, inondant le marché des ordinateurs à usage général. De récentes analyses ont montré une baisse des ventes d'ordinateurs de bureau en faveur de smartphones, tablettes et autres appareils embarqués. Par opposition aux ordinateurs à usage général, les systèmes embarqués doivent répondre à des objectifs quantifiables : la performance en temps réel, la consommation

d'énergie et le coût du marché. Ainsi, la conception des systèmes embarqués est entièrement guidée par ces objectifs quantifiables qui la rendent beaucoup plus difficile que la conception des ordinateurs à usage général.

A.1.1 *Matériels embarqués*

Jusqu'à ces dernières années, les systèmes embarqués étaient conçus autour d'un processeur unique associé à un ensemble de périphériques et d'accélérateurs matériels. Cependant, la demande croissante de flexibilité du marché de l'embarqué a entraîné une migration du matériel vers le logiciel. En d'autres termes, les fonctionnalités qui étaient directement implémentées par le matériel sont désormais effectuées par des processeurs programmables.

Pour gérer des applications de plus en plus exigeantes, la conception de processeurs plus performants a été réalisé, jusqu'à ces dernières années, grâce à l'augmentation de la fréquence de calcul. Mais, comme pour les ordinateurs à usage général, les systèmes embarqués ont atteints les limites de la technologie des semi-conducteurs, ce qui oblige les fabricants de puces à se tourner vers des architectures multi-cœurs pour améliorer la performance globale du système. En conséquence, les systèmes embarqués intègrent des processeurs de plus en plus programmable, mais contrairement aux ordinateurs à usage général, la plupart de ces processeurs sont adaptés à des tâches spécifiques afin de combler l'écart entre l'efficacité du matériel et la flexibilité du logiciel.

Les systèmes embarqués sont désormais des plates-formes multi-cœurs hétérogènes complexes avec un nombre croissant de cœurs de processeurs afin de répondre à l'exigence de performance. Par exemple, les plates-formes commerciales comme l'Intel SCC, le TILE de Tilera ou la SThorm de STMicroelectronics [22] contiennent déjà des centaines de cœurs programmables. L'augmentation du nombre de cœurs de processeur a cependant soulevé de nouvelles questions sur la conception de composants matériels, tels que l'organisation de la mémoire et le réseau d'interconnexion, et sur la façon de programmer de telles architectures.

A.1.2 *Logiciels embarqués*

Les premières formes de logiciels embarqués étaient de petits programmes habituellement écrites en assembleur afin d'obtenir des performances maximales. Ils peuvent maintenant être de complexes applications contenant de multiples algorithmes [183]. En outre, la nature des calculs effectués dans les différentes parties de l'application peut varier considérablement (types d'opérations, l'exigence de mémoire, parallélisme, etc.) En fait, cette variabilité correspond bien aux architectures hétérogènes. Par exemple, compte tenu de la structure de décodeurs vidéo modernes [150] , la compensation de mouvement est clairement la plus grande exigence dans l'espace de mémoire et de bande passante, tandis que le décodage résiduel et la prédiction intra sont plutôt calculatoire.

Le marché de l'embarqué est actuellement entraînée par les exigences des utilisateurs qui augmentent la complexité des logiciels embarqués. Par exemple, d'une part, le nouveau standard de compression vidéo à savoir HEVC réduit ses exigence de débit de 50 % avec la même qualité d'image que son prédécesseur, et permet ainsi de vidéo à haut définition. D'autre part, la norme HEVC augmente la complexité de calcul de 1,6 x par rapport à

son prédécesseur [171]. Les applications complexes sont souvent limitées à certains domaines d'application, comme le multimédia et la communication. Par exemple dans un téléphone 3G, plus de 60 % de la puissance de calcul et plus de 90 % de la performance disponibles sont consommés par des applications de radio et multimédia [170].

Au-delà de l'hétérogénéité et la complexité des applications, les plates-formes multi-cœur soulèvent de nouvelles questions concernant les logiciels embarqués, tels que la décomposition de l'application en tâches parallèles ainsi que le placement et l'ordonnancement de ces tâches sur la plate-forme en question.

A.1.3 *Conception de systèmes embarqués*

Aujourd'hui, l'informatique embarquée est confronté à une évolution technologique rapide et une grande diversité de systèmes informatiques. Par conséquent, des processus de conception très souples sont nécessaires. En fait, la conception de systèmes embarqués peut être décomposée en trois aspects (architecture, l'application et la méthodologie).

Puisque le logiciel et le matériel sont étroitement couplés dans la conception de systèmes embarqués, les concepteurs de systèmes embarqués doivent tenir compte de tous les aspects architecturaux, y compris l'organisation des composants matériels (processeurs, mémoires, interconnexions), la décomposition du logiciel en tâches afin de bénéficier autant que possible parallélisme, et la correspondance entre le matériel et le logiciel pour obtenir les meilleures performances. En outre, les concepteurs doivent comprendre en profondeur leurs applications pour tirer parti de toutes les optimisations possibles. Enfin, les méthodes sont essentielles pour la conception de systèmes embarqués efficaces. La modélisation permet une haute abstraction, nécessaire pour gérer la complexité croissante des systèmes embarqués. En ce qui concerne la difficulté d'analyser et déboguer les plates-formes matérielles, la simulation et l'analyse sont nécessaires pour déterminer l'efficacité et le coût de la conception. La conception assistée par les modèles nécessite des outils de synthèse traduisant les spécifications de haut niveau dans des implémentations optimisées. En outre, les processus de vérification automatique sont également essentielles pour atteindre le niveau de fiabilité requis à un coût minime.

En conclusion, les outils sont particulièrement importants dans la conception de systèmes embarqués. Les outils permettent de concevoir rapidement des systèmes embarqués pour faire face à la pression du temps d'accès au marché tout en réalisant leurs contraintes élevées d'efficacité et de fiabilité.

A.2 APPROCHE ET CONTRIBUTIONS

L'émergence des architectures massivement parallèles, ainsi que le besoin de modularité dans la conception de logiciels, a ravivé l'intérêt dans la programmation de flux de données. En effet, la programmation flux de données offre une approche de développement flexible qui permet de construire des applications complexes tout en exprimant concurrence et parallélisme explicitement. Paradoxalement, la plupart des études reste concentrée sur la programmation par flux de données statique, même si un processus de développement pragmatique nécessite l'expressivité et la praticité offerte par programmation par flux de données dynamique.

MPEG a toutefois mis en place un cadre innovant, appelé *Reconfigurable Video Coding* (RVC), qui peut être considérée, à notre connaissance, comme la première expérimentation à grande échelle de la programmation par flux de données dynamique. RVC a été initialement mis en place pour pallier le manque d'interopérabilité entre les différents codecs vidéo déployés sur le marché. Le cadre permet le développement d'outils de codage vidéo, entre autres applications, de façon modulaire et réutilisable grâce à l'inclusion d'un sous-ensemble du langage de programmation CAL.

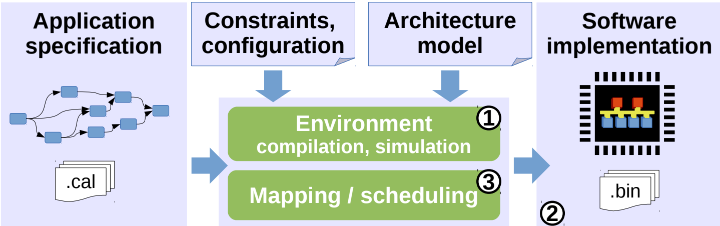


FIGURE 57 – Contributions of this thesis on dataflow-based embedded system design

Tout au long de cette thèse, nous étudions toutes les étapes du développement de décodeurs vidéo basée sur RVC (Figure 3), de leur spécification basée sur le paradigme de flux de données à leur mise en œuvre sur les plates-formes multi-core embarqués. Cette thèse apporte les contributions suivantes :

- **Contribution 1** [5, 7**?** , 3] : Un flot de conception pour développer des applications basées RVC sur des plates-formes multi-core embarqués. Le flot de co-conception repose sur un processus de simulation de pointe et un modèle d'architecture dédié. En outre, l'infrastructure de compilation sous-jacent notre flot de co-conception a été amélioré grâce à des techniques de génie logiciel modernes tels que l'ingénieurie dirigée par les modèles et la programmation orientée aspect.

- **Contribution 2** [8, 1] : Une implémentation logicielle optimisée des programmes flux de données dynamiques basée sur des techniques de communication efficaces qui limitent les accès à la mémoire, et sur des stratégies d'ordonnancement avancées qui réduise le coût de l'expressivité.

- **Contribution 3** [4, 6] : Un ensemble d'algorithmes d'ordonnancement et de projection d'acteur exécutable lors de l'exécution afin de gérer le comportement imprévisible des programmes de flux de données dynamiques, et d'atteindre des performances passant à l'échelle sur des plates-formes multi-cœur, que ce soit des processeurs multi-cœur de bureau ou embarqués.

En plus de la spécification de notre processus de développement basé sur le paradigme de programmation flux de données, nous évaluons l'efficacité de cette contribution en utilisant un ensemble de décodeurs vidéo qui ont été mis en œuvre dans le cadre de RVC, y compris un décodeur basé sur la nouvelle norme HEVC.

Tout ce travail a été mis en place dans les deux logiciels open-source suivant : un environnement de développement basé sur le paradigme flux de données connu sous le nom de Orcc [134], et un ensemble d'outils de co-design appelé TCE [166] utilisant TTA comme architecture modèle.

A.3 ÉTAT DE L'ART

Comme présenté dans le chapitre 2, exprimer un algorithme afin d'exploiter l'ensemble du parallélisme est déjà une tâche complexe et sujette aux erreurs, mais la diversité des plates-formes parallèles rend la tâche incroyablement plus difficile. Par conséquent, l'informatique embarquée doit se tourner vers des méthodologies de programmation basée sur des niveaux d'abstraction plus élevés afin d'atteindre les contraintes d'efficacité et de fiabilité inhérentes au domaine.

A.3.1 *Programmation flux de données*

Le chapitre 3 se concentre sur une solution de programmation particulière, connue sous le nom de programmation flux de données, qui s'appuie sur le formalisme des modèles de calcul pour définir des règles sémantiques permettant de décrire des composants, les *acteurs*, et de les connecter avec un graphe de flux de données, le *réseau*.

Un modèle de calcul est une spécification abstraite de la façon dont un calcul peut progresser. Un modèle de calcul est utile pour définir la sémantique d'un modèle de programmation, à savoir le type d'éléments qu'il peut contenir et la manière dont ils interagissent [152]. Les exemples classiques de modèles de calcul sont la machine de Turing et le lambda-calcul. Au cours des vingt dernières années, des dizaines de modèles de calcul flux de données ont été étudiés en raison de l'utilisation intéressante de programmation de flux de données pour le développement d'applications de traitement du signal.

Les modèles de calcul flux de données existant peuvent être divisés en deux catégories principales : Les modèles *statiques* qui ne peuvent décrire que des comportements prévisibles, permettant de faire l'ordonnancement au moment de la compilation. Les modèles *dynamiques* qui ont un comportement dépendant des données. La plupart des études sur l'implémentation des programmes de flux de données ce concentre sur les modèles statiques en raison de l'efficacité des techniques de synthèse sur ces modèles en raison de leur analysabilité. Malheureusement, ils ne prennent pas en considération la flexibilité et l'expressivité offerte aux programmeurs par les modèles flux de données dynamiques.

Le modèle DPN [121], également connu sous le nom de DDF, est étroitement liée aux réseaux de Kahn [110]. Le modèle DPN est Turing-complet, ce qui signifie qu'il peut modéliser n'importe quel algorithme, même non déterministe. Dans ce modèle, une application est représentée par un graphe dont les sommets / processus sont appelés *acteurs*. En plus des réseaux de Kahn, il introduit la notion de *tir*. Un *tir* de l'acteur, ou de l'action, est une quantum indivisible d'exécution qui correspond à une fonction de correspondance entre des jetons d'entrée et des jetons de sortie qui est appliquées de manière répétée et séquentiellement sur un ou plusieurs flux de données. Cette fonction est composée de trois étapes ordonnées et indivisible : lecture des données, calculs, et enfin écriture des données. Ces fonctions sont gardées par un ensemble de règles de tir qui spécifie quand un acteur peut être tiré, c'est à dire le nombre et la valeur des jetons qui doivent être disponibles sur les ports d'entrée pour déclencher l'acteur.

Ensuite, nous démontrons que le modèle de flux de données dynamique, aussi connu sous le nom DPN, est le meilleur compromis compte tenu du

point de vue pratique des programmeurs, mais le modèle DPN soulève également le défi d'implémentations efficaces (Figure 58).

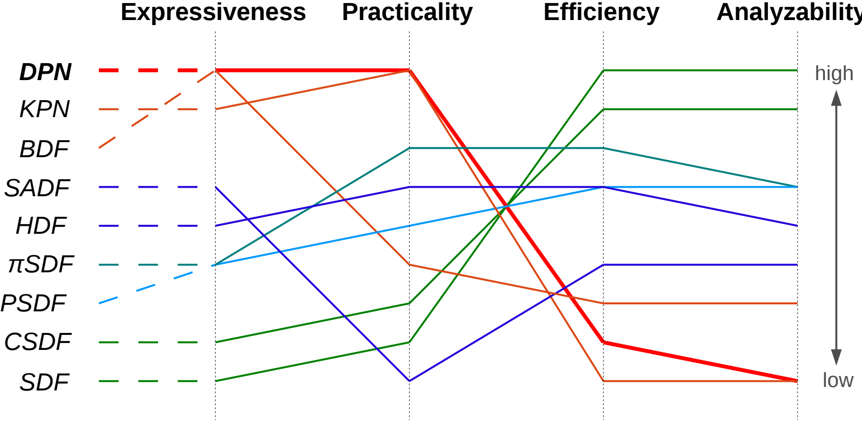


FIGURE 58 – Comparaison de modèles de calcul flux de données, étendant la classification de Stuijk et al. [163], qui met en avant le côté pratique du modèle DPN

A.3.2 *Reconfigurable Video Coding*

MPEG tente d'améliorer le processus de normalisation des normes de codage vidéo en suivant la méthodologie flux de données, telle que décrite dans le chapitre 4. Pour pallier le manque d'interopérabilité entre les normes de compression vidéo déployées sur le marché, MPEG a mis en place un cadre innovant, appelé RVC [125, 29, 126], qui exploite la programmation flux de données dynamique pour fournir la modularité, l'évolutivité et la portabilité nécessaire aux outils de codage vidéo.

MPEG RVC définit deux normes qui ont été produites par le groupe de travail RVC :

- La représentation de configuration du codec (ISO / IEC 23001-4 ou MPEG-B pt. 4) [10] décrit le format avec lequel un décodeur RVC peut être défini comme un réseau de blocs de calcul, ainsi que d'un langage textuel pour la définition des blocs de codage vidéo (Section 4.3).
- Une bibliothèque d'outils vidéo (ISO / IEC 23002-4 ou MPEG-C pt. 4) [11] qui standardise les acteurs nécessaires pour décrire les normes de codage vidéo existants (Section **??**), actuellement MPEG-4 part 2 et MPEG-4 part 10.

En fait, RVC ne fournit pas seulement un nouveau processus de normalisation qui dépasse les limites du processus de normalisation en cours, mais introduit également un cadre qui favorise le développement multimédia, en offrant tous les avantages de la programmation par flux de données avec le pragmatisme requis par le développement d'applications complexes. Alors que l'implémentation efficace des programmes basés DPN reste difficile, les outils et les applications incluses dans le dossier RVC cadre ont maintenant atteint une maturité suffisante pour démontrer la viabilité de cette approche de développement pour des applications réelles.

Depuis sa création, le groupe de travail RVC a standardisé l'implémentation de 3 décodeurs vidéo détaillés ci-dessous :

MPEG -4 PART 2 La norme MPEG-4 part 2, également connu sous le nom de MPEG-4 visual, a été publié en 1999 par la consortium commun ISO/ITU. Les codecs DivX et Xvid populaires, qui ont largement contribué au développement du partage de vidéo sur Internet, sont des implémentations de cette norme. En fait, le simple profile du décodeur MPEG-4 Part 2 a été la première application standardisée par le groupe de travail RVC. Compte tenu de la nouveauté de l'approche, le décodeur a été la source de plusieurs dizaines d'expériences qui ont mené au développement de plusieurs versions d'un décodeur avec granularité variable.

MPEG -4 PART 10 Introduit en 2003, MPEG-4 Part 10, également connu sous le nom MPEG-4 AVC / H.264 est une norme vidéo largement utilisé depuis l'avènement de la haute définition dans l'usage quotidien [176]. En fait, AVC est actuellement l'une des normes les plus exploitées au sein des services vidéo commerciaux, allant de web de streaming à la radiodiffusion numérique, y compris l'enregistrement de la caméra.

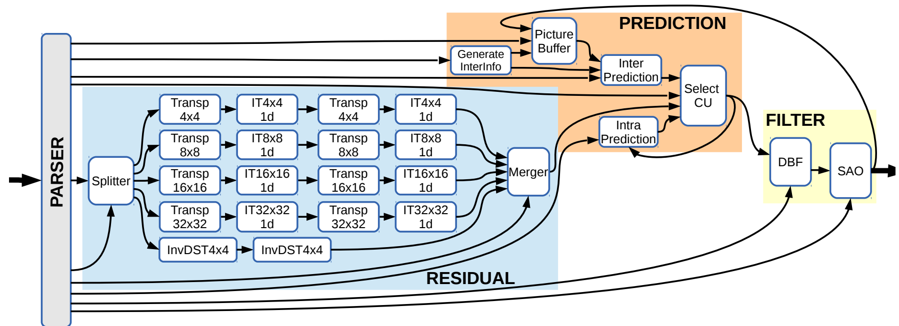


FIGURE 59 – Description basé sur RVC d'un décodeur vidéo implémentant le standard HEVC

MPEG -H PART 2 MPEG-H Part 2 , également connu sous le nom MPEG HEVC / H.265 , est la dernière norme de codage vidéo, développé conjointement par l'ISO et l'ITU, en tant que successeur MPEG-4 AVC / H.264 . HEVC améliore le taux de compression des données, ainsi que la qualité d'image, afin de gérer les contraintes de vidéo modernes tels que les hautes résolutions d'image 4K (3840x2160) et 8K (7680x4320) [164]. Une autre caractéristique clé de cette nouvelle norme de codage vidéo est sa capacité de traitement parallèle qui offre des performances évolutives sur les architectures parallèles à la mode [164] .

Ces capacités parallèles offrent une grande opportunité de prouver l'intérêt de l' approche RVC. Par conséquent, le groupe de travail RVC a développé , en parallèle avec le processus de normalisation , une implémentation d'un décodeur HEVC en utilisant le cadre RVC, qui est présentée dans la figure 59. Cet effort commun a permis la démonstration d'une version fonctionnelle lors de la 103ème réunion MPEG en Janvier 2013. Au même moment, la version finale du standard HEVC a été approuvé.

A.4 ENVIRONNEMENT DE DÉVELOPPEMENT DÉDIÉ

Partant de cet état de l'art, nous avons construit un flot de co-conception complet ciblant les plates-formes multi-cœur embarquées (présenté dans le

chapitre 5). Notre flot de conception intégre une infrastructure de simulation avancée et repose sur un modèle d'architecture dédié aux programmes flux de données. Construire ce flot de conception a également donné l'occasion d'améliorer la fiabilité et la flexibilité de l'infrastructure de compilation sous-jacente au moyen de techniques modernes de génie logiciel.

A.4.1 *Infrastructure de programmation flux de données*

À partir du travail initial de Wipliez [177] , nous proposons une infrastructure de compilation améliorée des programmes de flux de données qui tire profit de la méta-modélisation et de la programmation orientée aspect.

L'infrastructure de compilation pour les programmes de flux de données, inclus dans Orcc [134] , sur lequel nous avons travaillé au cours de cette thèse est présentée par la figure 60. Créée par Wipliez [177], cette infrastructure de compilation est un *trans-compilateur*, aussi appelé un *compilateur source-à-source*, qui traduit des descriptions de programmes flux de données dans des codes sources plus traditionnelles, au lieu de générer directement du code machine comme beaucoup compilateurs.

Dans sa thèse [177], Wipliez décrit la mise en œuvre d'une représentation intermédiaire (IR) dédiée aux compilateurs flux de données et montre qu'une telle IR, spécifique à un domaine précis, est bien adaptée pour effectuer analyses et optimisations avancées sur les programmes flux de données. Maintenant, nous montrons comment les techniques modernes de génie logiciel, tels que le méta-modélisation et la programmation orientée aspect, peut améliorer la manipulation de cette IR et l'ensemble du flot de compilation.

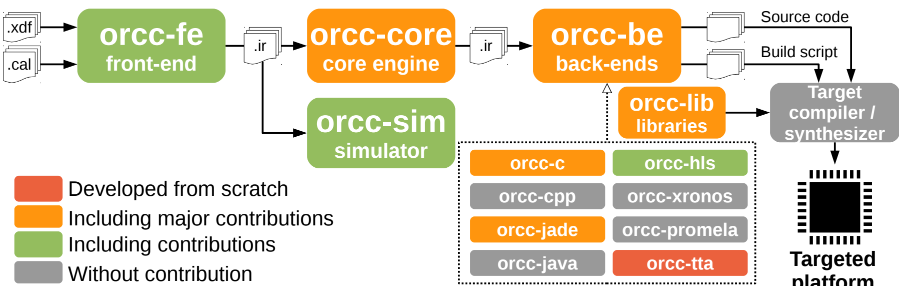


FIGURE 60 – Infrastructure de compilation multi-cibles

A.4.2 *Modèle d'architecture dédié*

Le développement d'un flot de conception ciblant ces plateformes nécessite la définition d'un modèle d'architecture qui correspond au comportement de la plate-forme ciblée, tout en gardant un haut niveau d'abstraction et suffisamment de liberté de configuration pour permettre l'exploration de l'espace de conception. Compte tenu de la complexité des architectures multi-core, ainsi que de l'efficacité et de la fiabilité requise par les systèmes embarqués, nous proposons de spécialiser notre modèle d'architecture pour la réalisation des programmes de flux de données dynamiques afin de tirer parti de la connaissance inhérente à notre domaine d'application, de la même manière que les DSLs.

Les cœurs de processeur sous-jacent à notre plate-forme abstraite est basé sur une architecture de type VLIW connue sous le nom de TTA [55]. Cette architecture particulière a été choisi pour les raisons suivantes :

- Les processeurs TTA sont en mesure de tirer profit de la seule forme de parallélisme qui n'est pas inhérente au modèle flux de données. Les processeurs TTA ressemblent VLIW processeurs dans le sens où ils vont pouvoir exécuter plusieurs instructions par cycle d'horloge. Une différence importante, cependant, est que les processeurs TTA n'ont qu'une seule instruction : *Move*, qui transfère simplement les données d'un endroit interne du processeur à l'autre.

- Les processeurs TTA sont idéaux pour cibler les systèmes embarqués. En fait, la programmation directe du transport des données permet de réduire le trafic au niveau de la file de registre par rapport aux processeurs VLIW [55], mais cependant la conception du compilateur assez difficile, car c'est le compilateur qui planifie le transport des données et fait en sorte que les conflits soient évités. Comme le compilateur prend ces décisions lors de la conception, le système d'exécution est simplifiée et donc il ya des économies sur le nombre de portes logiques nécessaire par le processeur et donc de la consommation d'énergie.

- Les processeurs TTA sont extrêmement configurables. Le concepteur peut faire le processeur minuscule et économes en énergie ou, le cas échéant, d'augmenter le parallélisme d'instructions du processeur. Nous présentons 4 configurations prédéfinies qui ont été utilisés lors des expériences.

Maintenant, nous introduisons une architecture de mémoire hybride spécialement conçue pour les programmes flux de données. Pour limiter le traditionnel goulot d'étranglement lié aux accès mémoire , notre modèle d'architecture contient des mémoires privées et partagées, comme le montre la figure 61, ce qui rend l'architecture de mémoire, un mélange d'organisation UMA et NORMA (définies dans le chapitre 2). Ainsi, les processeurs sont associés à leur mémoires privées utilisés pour l'exécution de leurs acteurs, mais également reliées, à travers d'un réseau d'interconnexion, à un ensemble de mémoires partagées consacrés aux communications inter-processeurs.

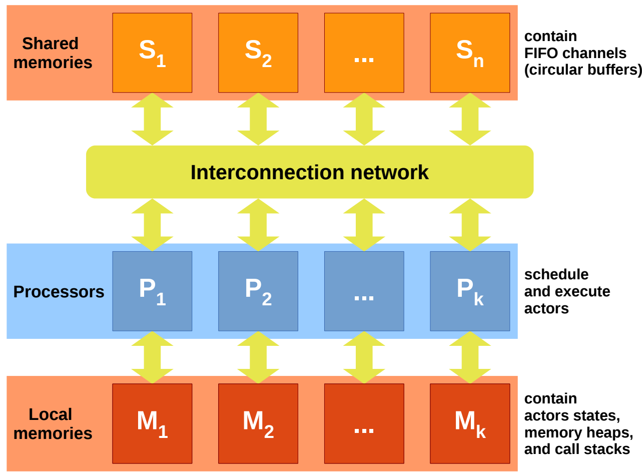


FIGURE 61 – An hybrid memory architecture dedicated to DPN-based programs

A.4.3 *Co-conception de systèmes embarqués*

La difficulté de programmer efficacement les plates-formes embarquées multi-cœur, telle que présenté dans le chapitre 2, fait que le processus de conception est toujours un défi ouvert. C'est pourquoi nous présentons un flot de co-conception automatisée, conçue à partir de zéro au cours de cette thèse, qui se destine à la mise en œuvre de programmes basés DPN sur des plates-formes multi-cœr dédiées. Ce flot de co-conception a été utilisé pour effectuer la plupart des expériences présentées ici, ce qui en fait un élément clé de cette thèse [5] .

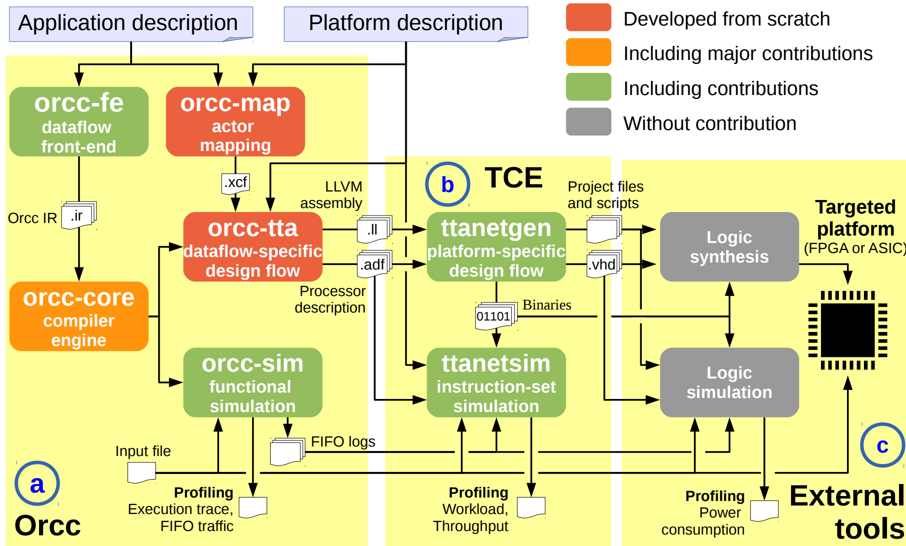


FIGURE 62 – Infrastructure de co-conception de plates-formes multi-cœur embarquées

Le flot de co-conception est mis-en-œuvre autour de deux projets open-source appelé respectivement Orcc [134] et TCE [166]. En fait, Orcc peut être considéré comme un front-end flux de données pour TCE , et inversement TCE peut être considéré comme un back-end spécifique au processeur pour Orcc. En fait, Orcc effectue la partie haut niveau du flot de conception et fournit un simulateur fonctionnel, et les deux sont totalement indépendants de l'architecture des processeurs. Pour sa part, TCE effectue la partie bas niveau du flot de conception et fournit un simulateur de jeu d'instructions. Comme le montre la figure 62, notre flux de co-conception s'effectue en plusieurs étapes : D'abord, l'étape spécifique à la modélisation flux de données (a) est mise en œuvre dans Orcc. Ensuite, l'étape spécifique au processeur (b) est mise en œuvre dans la suite d'outils TCE. Et enfin, le stade spécifique au matériel (c) est réalisée par des outils tiers, mais automatisée par un ensemble de scripts et fichiers de projets générés par l'étape précédente.

A.5 IMPLÉMENTATION LOGICIELLE DES PROGRAMMES FLUX DE DONNÉE

Le principal défi auquel les programmes flux de données dynamiques doivent faire face est la démonstration d'implémentations efficaces qui peuvent atteindre les contraintes de performance imposées par les applications modernes. Par exemple, les décodeurs vidéo doivent permettre l'affichage en

temps réel de séquences vidéo haute définition, à partir de 25 FPS pour le format 720p sur des terminaux mobiles à 50 FPS dans le format 8K sur les écrans de cinéma. Alors que l'efficacité des programmes décrits à l'aide des langages de programmation traditionnels est le résultat de 50 années de travail sur les compilateurs afin d'exploiter la localité mémoire, l'abandon du paradigme de localité des données en faveur du paradigme flux de données nécessite le développement de nouvelles techniques de compilation pour profiter pleinement de l'architecture du processeur. En outre, l'attrait des modèles de flux de données plus restreintes a souvent détourné l'attention de la programmation basée DPN.

A.5.1 *Implémentation optimisée*

En outre, nous avons partiellement résolu l'un des principaux défis des modèles flux de données dynamiques en présentant notre implémentation logicielle optimisée pour les programmes basés sur le modèle DPN. Notre implémentation vise notamment l'exécution efficace des décodeurs vidéo sur plates-formes multi-cœur embarquées, mais la plupart des principes peuvent être appliqués à tous les programmes basés DPN et toutes les plates-formes multi-cœur. Notre implémentation est purement logicielle mais comprend autant l'optimisation des communications que celui de l'ordonnancement. Veuillez noter que notre implémentation a été intégré dans notre flot de co-conception, que nous avons introduit dans le chapitre 5, de sorte que toutes les expériences présentées ci-dessous sont pris en charge par notre flot de compilation.

De plus, notre implémentation est la première, à notre connaissance, ayant démontré sa capacité à réaliser le décodage temps réel de séquences vidéos haute définition sur des processeurs de bureau (Table 13). Notre implémentation a également montré des résultats encourageants pour des plates-formes multi-core embarquées (Table 14).

Decoder	Video sequence	Frame-rate
MPEG-4 Part 2 SP	*OldTownCross* (720P)	33,7 FPS
MPEG-4 AVC PHP	*PlaceAtTheTable* (720P)	4,5 FPS
MPEG HEVC Main	*KristenAndSara* (720P)	12,7 FPS

Table 13 – Fréquence d'image maximale atteinte sur notre processeur généraliste en utilisant la stratégie d'ordonnancement round-robin

Decoder	Video sequence	Frame-rate
MPEG-4 Part 2 SP	*Foreman* (QCIF)	175 FPS
MPEG HEVC Still Picture	*BasketBallPass* (240p)	4 FPS

Table 14 – Fréquence d'image maximale atteinte sur notre plate-forme embarquée en utilisant le configuration de processeur *Fast* cadencé à 100MHz

A.5.2 *Implémentation extensible*

De plus, nous avons décrit dans le chapitre 7 plusieurs systèmes de projection pour exécution multi-cœur qui peuvent gérer le dynamisme de nos programmes basés sur le modèle DPN et d'atteindre des performances évolutives. En d'autres termes, nos systèmes d'ordonnancement *assigne* les acteurs aux processeurs, *ordonne* l'exécution des acteurs, et *exécute* les acteurs, sans causer de *famine* ou *blocage*. Tous nos algorithmes visent à maximiser le débit des données pour les applications de manière à atteindre la contrainte temps réel. Ce chapitre décrit :

1. Un système d'exécution basé sur un algorithme génétique qui recherche automatiquement les projections efficaces d'acteurs sur n'importe quelle architecture multi-cœur pour les programmes basés sur le modèle DPN.

2. Un système de projection d'acteur à bas coût pour les programmes de flux de données basé sur un système de partitionnement de graphe. Le faible coût de l'approche le rend faisable à plusieurs reprises lors de l'exécution de maintenir un bon équilibrage de la charge sur la plate-forme ciblée, même avec des applications dynamiques.

3. Un système de d'ordonnancement de l'exécution des acteurs pour programmes flux de données qui prend en charge l'architecture multi-cœurs grâce à une architecture distribuée.

Enfin, nous avons démontré l'extensibilité de notre implémentation des programmes flux de données basés sur le modèle DPN en utilisant les algorithmes d'ordonnancement décrits dans le chapitre 7. En effet, nous avons évaluer l'évolution des performances en fonction du nombre de cœurs de processeur utilisés pour notre processeur de bureau (Table 15) et notre plateforme embarquée (Figure 63).

Décodeurs	2 cœurs	4 cœurs	6 cœurs
MPEG-4 Part 2 SP	58,1 FPS	94,6 FPS	93,1 FPS
MPEG-4 AVC PHP	12,4 FPS	17,0 FPS	19,0 FPS
MPEG HEVC Main	18,0 FPS	15,8 FPS	10,4 FPS

Table 15 – Influence du nombre de processeurs sur la performance du decodeur MPEG-4 Part 2 Simple Profile sur notre processeur généraliste

A.6 CONCLUSION ET PERSPECTIVES

Pour supporter à la fois les contraintes élevées de systèmes embarqués d'aujourd'hui et la pression du marché sur les délai de production, le marché électronique est clairement tourné vers la conception de plates-formes très hétérogènes, connu sous le nom MPSoCs, intégrant des processeurs de plus en plus spécialisés sur une seule puce afin de combler le fossé entre l'efficacité du matériel et de la flexibilité du logiciel. Mais, étonnamment, certaines applications, comme les codecs vidéo, sont toujours effectuées par des composants matériels dédiés dans la plupart des MPSoCs. Le travail présenté dans cette thèse s'inscrit dans un contexte de demande croissante pour les

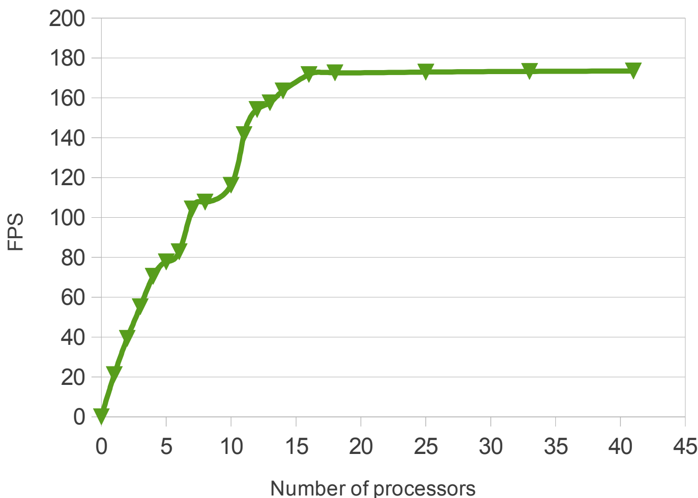


Figure 63 – Influence du nombre de processeurs sur la performance du decodeur MPEG-4 Part 2 Simple Profile sur notre plate-forme embarquée

applications multimédia ainsi qu'un manque de flexibilité des systèmes embarqués pour le décodage vidéo.

Les travaux réalisés au cours de cette thèse peut servir de base pour de futures recherches. Nous détaillons ici quelques perspectives sur la base de notre environnement de développement dédié à la programmation flux de données, sur notre implémentation logicielle de programmes flux de données dynamiques, et sur une plateforme embarquée dédiée aux programmes RVC.

A.6.1 *Environnement de développement avancé*

Le développement d'un IDE dédié à la programmation flux de données est une tâche laborieuse qui est clairement ouverte à de nouvelles perspectives afin de promouvoir le développement de nouvelles applications dans le cadre de RVC.

développement assisté Notre environnement de développement inclut un éditeur de graphes qui tire parti de la capacité de programmation visuelle du paradigme de flux de données . Mais, notre éditeur souffre de sa technologie sous-jacente, comme la fonction de mise en page automatique qui offre des résultats médiocres sur des graphiques complexes. En fait, nous pourrions bénéficier de méta-outils dédiés, tels que Graphiti d'Eclipse [63] et Spray [64], afin de développer un éditeur graphique à l'état de l'art. En outre, l'approche générative de ces méta-outils pourrait grandement simplifier le développement et le maintien de nos interfaces utilisateur.

Le débogage des applications RVC au sein de notre IDE est généralement obtenue par la génération du code C pour la plate-forme d'accueil afin de pouvoir utiliser les débogueurs traditionnels, par exemple GDB. Mais, les applications générées sont déjà le résultat de nombreux choix d'implémentation compliquant le débogage. Par exemple, l'inspection des canaux de communication basés sur des tampons circulaires est assez difficile pendant le débogage. En conséquence, l'intégration d'un débogueur directe-

ment dans notre IDE pourrait grandement faciliter le développement d'applications complexes.

FLOT DE CONCEPTION La disponibilité de nouveaux systèmes sur puce intégrant de la logique programmable (par exemple le Xilinx Zynq SoC), associé à la capacité de notre compilateur pour cibler de multiples plates-formes, permettrait le développement d'un flot de conception visant les systèmes complexes et hétérogènes (MPSoCs), comprenant des processeurs ARM, des processeurs basés TTA et des accélérateurs matériels .

Une telle hétérogénéité introduit cependant de nouveaux défis à notre flot de conception. Tout d'abord, l'infrastructure de simulation devrait interconnecter plusieurs simulateurs afin de gérer la co-simulation de l'ensemble du système. Puis, cibler des plates-formes hétérogènes impliquerait le développement d'un processus avancé de projection des acteurs considérant à la fois les composants matériels et logiciels. Enfin, un système de scripts de compilation automatique et multi-cible serait nécessaire pour simplifier l'interaction avec un nombre croissant d'outils externes.

A.6.2 *Implémentation logicielle optimisée*

L'efficacité de l'implémentation est clairement la problématique centrale de programmation basée sur des modèles de données flux de données dynamiques. Même si un grand progrès ont été réalisés depuis l'introduction du framework RVC, il y a encore un long chemin à parcourir avant l'adoption de la programmation flux de données dynamique par les industriels. Mais, nous avons déjà plusieurs perspectives qui nécessitent des travaux futures pour améliorer la performance de notre implémentation à l'*état-de-l'art*.

COMMUNICATIONS La mise en œuvre optimisée des communications, décrit dans le chapitre **??**, empêche le potentiel parallélisme de mots à cause des indices absolus. En fait, les accès à la mémoire tampon circulaire pourraient être vectorisés puisque la largeur de canaux au sein de nos applications sont souvent inférieurs à 32 bits (généralement 8 ou 16 bits), mais le compilateur ne peut pas savoir si l'accès sont alignés dans la mémoire ou si la fin de la mémoire tampon circulaire est atteinte. Ainsi, nous étudions actuellement une implémentation plus avancée de communications. Les actions dîtes multi-jetons sont générés en deux versions, l'une standard et l'autre vectorisée, qui sont exécutées en fonction de la position actuelle dans le tampon circulaire.

ORDONNANCEMENT Les résultats présentés dans le chapitre **??** ont clairement montré que nos applications n'ont pas vraiment bénéficier de l'ordonnancement quasi-statique. La raison principale vient des limites de notre classificateur à l'heure actuelle. Notre classificateur n'interprète pas la communication entre les acteurs, de sorte qu'il ne peut pas effectuer des analyses au niveau du réseau. En outre, notre classificateur est incapable d'extraire des comportements complexes au sein d'un seul acteur, comme des dépendances indirectes entre les gardes et les jetons. Une solution serait l'utilisation d'outils tiers dédiés pour effectuer une interprétation abstraite plus fiable.

Une autre solution serait la mise en place de directives spécifiques, à l'instar de celles utilisées pour la parallélisation assistée décrit dans le chapitre 2. Le développeur pourrait aider le compilateur à trouver les comportements

quasi-statiques dans son application en ajoutant des directives spécifiques à la description. En effet, une directive contient simplement des informations supplémentaires qui ne peuvent être déterminées automatiquement par le compilateur.

Pour résumer, nous rajoutons une sémantique quasi-statique à notre langage de programmation de flux de données qui est initialement dynamique. Le défi serait alors d'identifier quelle information serait utile pour aider à la classification de l'acteur. Bien que cette approche serait peu satisfaisant d'un point de vue pratique, cela représenterait une nouvelle étape pour combler l'écart entre la puissance expressive des modèles dynamiques et l'efficacité des modèles statiques et quasi-statiques.

A.6.3 *Plate-forme dédiée aux codecs vidéo RVC*

Cette thèse visait initialement l'étude d'une plate-forme basée MPSoC , combinant à la fois des composants matériels et logiciels, dédié aux applications de codage vidéo qui héritent du cadre RVC . Alors que la conception d'une telle plate-forme n'est pas encore atteint, notre flot de conception permet déjà l'exploration architecturale de plates-formes dédiées multi-cœr embarquées basées sur notre modèle d'architecture. Ainsi, nous étudions actuellement une plate-forme générique dédié à tous les codecs basés sur RVC ouvrant de nombreuses perspectives intéressantes pour des travaux futurs. En fait, plusieurs études [98, 20, 31] ont déjà étudié des plateformes embarquées spécifiques à RVC, mais aucune d'entre elles n'a réellement démontré la viabilité de sa proposition.

HÉTÉROGÉNÉITÉ Comme indiqué au cours de cette thèse, les décodeurs vidéo sont des applications complexes composées d'éléments ayant des comportements très différents. D'une part, les résultats de nos expériences ont clairement démontré l'intérêt des processeurs de type VLIW pour profiter du parallélisme interne des acteurs. D'autre part, les résultats ont mis en évidence la faiblesse de processeurs basés TTA pour la maîtrise du dynamisme de certains acteurs au sein de nos applications. Une solution serait l'addition à notre modèle d'architecture d'un autre type de processeur comprenant un prédicteur de branchement, le processeur LatticeMico32 par exemple. Ainsi, nous pourrions concevoir des plates-formes dédiées capable de gérer efficacement les deux types d'acteurs qui composent nos décodeurs vidéo, calcul intensif et contrôle intensif.

L'hétérogénéité de la plate-forme amène toutefois des défis supplémentaires à notre flot de co-conception comme vu précédemment. L'hétérogénéité introduit également de nouvelles contraintes au niveau de la projection des acteurs. D'une part, les acteurs réalisant des calculs intensifs (par exemple, le *transformation inverse*) doit être projeté sur des processeurs TTA. D'autre part, les acteurs contenant beaucoup de contrôle (par exemple, le *parser*) doivent être projeté sur les autres processeurs. En conséquence, une analyse spécifique serait nécessaire pour identifier le comportement des acteurs.

PARTITIONNEMENT Comme décrit dans le chapitre 4, tous les décodeurs vidéo partagent une structure commune qui correspond globalement avec le graphe de l'application de leur description RVC. Ainsi, ils sont généralement décomposés en 4 sous-réseaux : *parser*, *residual*, *prediction* et *filter* . En fait, les interconnexions entre ces sous-réseaux sont également très similaire dans

tous les décodeurs. Partant de ce constat, nous étudions une architecture de mémoire dédiée qui partitionne les processeurs en groupe (ou *cluster*) pour s'adapter à la structure globale de décodeurs vidéo. Une spécification précoce, basée sur une hiérarchie de mémoire partagée à deux niveaux est présentée dans la figure 64.

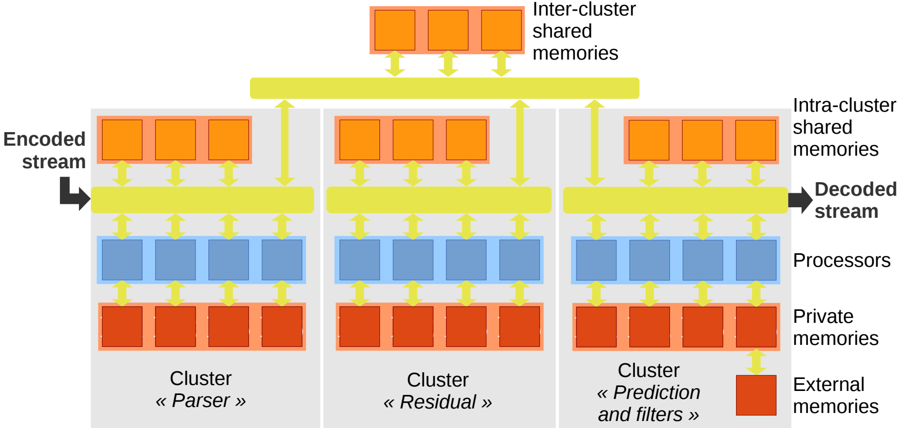


FIGURE 64 – Architecture mémoire dédiée aux décodeurs videos

En outre, nous pouvons également identifier les acteurs qui nécessitent de grandes quantités de mémoire, comme le *frame-buffer*, et les projeter sur des processeurs dédiés qui ont accès à la mémoire externe.

ADAPTIVITÉ Des travaux antérieurs de notre groupe de recherche ont déjà démontré le fonctionnement d'un codec vidéo adaptatif appelé Jade utilisant le framework RVC. Comme décrit dans le chapitre 4, Jade exploite les mécanismes de la machine virtuelle disponible dans LLVM, comme la compilation *just-in-time*, afin de fournir un décodeur universel. En effet, le décodeur est configuré *à-la-volée* selon une configuration encapsulé avec le flux vidéo et grâce à une bibliothèque de composants (i.e. la VTL). Ainsi, le flux entrant peut configurer le décodeur conformément au format de codage du flux vidéo.

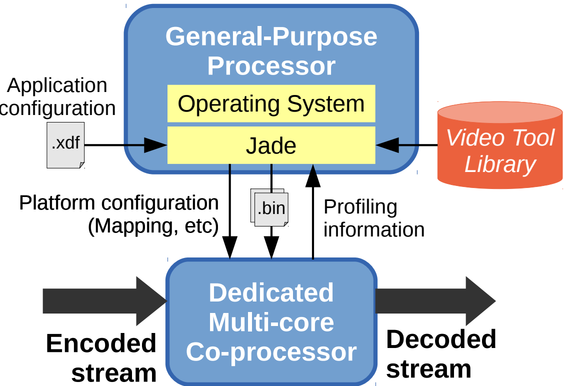


FIGURE 65 – Hardware Adaptive Decoder Engine

Jusqu'à présent, Jade est exécutable exclusivement sur processeurs généralistes (x86 et ARM) qui ne sont pas économe en énergie. Ainsi, nous

pourrions coupler un processeur généraliste avec notre plate-forme multi-cœr dédiée sous la forme d'un co-processeur en charge du décodage vidéo, telle que présentée dans la figure 65. Pour ce faire, Jade serait étendu avec une fonction hôte / client où le processeur généraliste configurerait le co-processeur multi-core qui effectuer le décodage. Et, à la réception d'un flux entrant, Jade pourrait cross-compiler les acteurs pour le co-processeur multi-core et permettrait de déterminer une projection initiale. Finalement, le co-processeur multi-core renverrait à Jade les informations de profilage pour permettre la détermination d'une projection plus optimisée.

TABLE DES FIGURES

Figure 1	Generic MPSoC-based platform	2
Figure 2	Aspects of embedded system design (adapted from Wolf's analysis [182])	3
Figure 3	Contributions of this thesis on dataflow-based embedded system design	4
Figure 4	Processor architectures characterizing multi-core platforms	12
Figure 5	Memory architectures characterizing multi-core platforms	13
Figure 6	Hierarchical memory organization for multi-core platforms	14
Figure 7	Hardware versus software implementation of hierarchical memories	15
Figure 8	The first dataflow representation, the graphical representation of an arithmetic computation, that was introduced by Sutherland in 1966 [165]	23
Figure 9	A dataflow network of five processes, the vertices named from A to E, that communicate through a set of communication channels, represented by the directed edges	24
Figure 10	The dataflow representation is modular by offering hierarchical ability, re-usability and reconfigurability.	25
Figure 11	Parallelizing the dataflow program presented in Figure 9 from a sequential execution (11a) to parallel execution using different strategies (11b, 11d, 11c). . .	26
Figure 12	A self-contained actor with its own state, actions and firing rules	28
Figure 13	Comparison of dataflow MoCs, extending the classification system introduced by Stuijk et al. [163], which shows that DPN is the most suitable model for a practical programming language	31
Figure 14	Round-robin scheduling of the actor of the dataflow network presented in Figure 9	35
Figure 15	Behavior of the dynamic list of next schedulable actor used by data-driven / demand-driven scheduling	36
Figure 16	Multiplication of the video compression standards	42
Figure 17	RVC vision	46
Figure 18	Visual representation of dataflow network	47
Figure 19	RVC-based description of the MPEG-4 Part 2 SP decoder	52
Figure 20	RVC-based description of the MPEG-H Part 2 SP decoder	53
Figure 21	Performance evolution of an RVC-based video decoder. Frame-rates of the *foreman* sequence (QCIF) using the *normative description* of the MPEG-4 Part 2 Simple-Profile decoder executed on *mono-processor* desktop computers [151, 180, 93].	58
Figure 22	Multi-target trans-compilation infrastructure	63
Figure 23	Compilation flow based on meta-tools	63
Figure 24	Class diagram of Graph	64

Figure 25 Class diagram related to Procedure 66
Figure 26 Class diagram related to Instruction 67
Figure 27 Class diagram related to Expression and Type 68
Figure 28 Class diagram related to Network 69
Figure 29 Class diagram related to Actor 69
Figure 30 A simple processor based on Transport-Trigger Architecture 71
Figure 31 *Fast* TTA-based processors target high clock-frequency implementation [71]. 73
Figure 32 An hybrid memory architecture dedicated to DPN-based programs 73
Figure 33 Designing embedded multi-core platforms following the flexible Y-chart approach [113] 75
Figure 34 Multi-stage co-design flow 75
Figure 35 Two-step compilation flow 77
Figure 36 Processor simulation in standalone fashion by way of FIFO simulators 79
Figure 37 Concurrency-safe implementation of FIFO channels in shared-memory 83
Figure 38 Three way of broadcasting communications 88
Figure 39 Hierarchical scheduling 89
Figure 40 Quasi-static scheduling using actor clustering 91
Figure 41 Communication analysis (rates and broadcasting) within RVC-based video decoders 96
Figure 42 Repartition of the workload within RVC-based video decoders using both desktop and embedded implementations 97
Figure 43 Exploring the parallelism potential of actors composing video decoders thanks to their execution speedup on TTA-based processors using *Custom*, *Fast* and *Huge* configurations from a sequential execution with *Standard* configuration 98
Figure 44 Quasi-static scheduling of the *inverse transform* of MPEG HEVC / H.265 based on graph clustering 100
Figure 45 Multi-core scheduling of dynamic dataflow programs 103
Figure 46 A population considering the mapping of the actors A, B, C, D and E onto the processors P1 and P2 . . . 105
Figure 47 Proceeding of our evolutionary-based actor mapping system 106
Figure 48 2-ways evolution of the population from generation N to $N+1$: New individuals are created either by (a) cross-over or by (b) mutation. 107
Figure 49 Actor mapping flow based on graph partitioning . . 108
Figure 50 Distributed scheduler onto a multi-core platform using round-robin 110
Figure 51 Lock-free scheduling communication for data-driven / demand-driven strategy on multi-core platforms . 111
Figure 52 Topologies of the interconnection network of the distributed scheduling management 112
Figure 53 Experimental flow to analyze the scalability of RVC-based applications on both desktop (A) and embedded (B) multi-core platforms 113

Figure 54 Influence of the number of processors on the performance of MPEG-4 Part 2 Simple Profile decoder . . . 116
Figure 55 Dedicated clustered architecture based on video decoders structure 123
Figure 56 Hardware Adaptive Decoder Engine 123
Figure 57 Contributions of this thesis on dataflow-based embedded system design 130
Figure 58 Comparaison de modèles de calcul flux de données, étendant la classification de Stuijk et al. [163], qui met en avant le côté pratique du modèle DPN 132
Figure 59 Description basé sur RVC d'un décodeur vidéo implémentant le standard HEVC 133
Figure 60 Infrastructure de compilation multi-cibles 134
Figure 61 An hybrid memory architecture dedicated to DPN-based programs 135
Figure 62 Infrastructure de co-conception de plates-formes multi-cœur embarquées 136
Figure 63 Influence du nombre de processeurs sur la performance du decodeur MPEG-4 Part 2 Simple Profile sur notre plate-forme embarquée 139
Figure 64 Architecture mémoire dédiée aux décodeurs videos . 142
Figure 65 Hardware Adaptive Decoder Engine 142

LISTE DES TABLEAUX

Table 1 Parallelism is multi-form and multi-level 10
Table 2 Comparison of interconnection networks 16
Table 3 Dynamism-based taxonomy of mapping and scheduling approaches, run-time or compile-time, from Lee and Ha's work [118] 21
Table 4 Statistics about the RVC-CAL description of several MPEG video decoders 53
Table 5 Comparison of 4 predefined processor configurations 72
Table 6 Maximal frame-rates achieved by our desktop implementation using round-robin scheduling strategy . . 93
Table 7 Maximal frame-rates achieved by our embedded implementation using the *Fast* configuration clocked at 100MHz. These frame-rates have been evaluated during an execution of the entire multi-core platform within the instruction-set simulator (*ttanetsim*) 94
Table 8 Clustering results 99
Table 9 Comparison of static and dynamic scheduling strategies within the desktop implementation using the tested video decoders. The number of switches, firings and misses are expressed in 10^3 101
Table 10 Comparison of static and dynamic scheduling strategies on the *inverse transform* of HEVC using our embedded implementation 102
Table 11 Comparison of our approach with handmade mappings of MPEG-4 Part 2 executed on 2 processor cores 114
Table 12 Bounded scalability of the RVC-based video decoders according to decoding frame-rates (FPS) obtained on a desktop multi-core processor. The decoders are first mapped using our dedicated system based on graph partitioning, then scheduled using different strategies, respectively round-robin strategy (RR) and combined strategy with either ring topology (C/R) or fully-connected topology (C/F). 115
Table 13 Fréquence d'image maximale atteinte sur notre processeur généraliste en utilisant la stratégie d'ordonnancement round-robin 137
Table 14 Fréquence d'image maximale atteinte sur notre plateforme embarquée en utilisant le configuration de processeur *Fast* cadencé à 100MHz 137
Table 15 Influence du nombre de processeurs sur la performance du decodeur MPEG-4 Part 2 Simple Profile sur notre processeur généraliste 138

LISTINGS

Listing 1 One simple directive to parallelize a loop 18
Listing 2 Textual representation of dataflow network 47
Listing 3 XML-based intermediate representation of dataflow network 47
Listing 4 Header of an actor 48
Listing 5 Procedural code 49
Listing 6 An action that transposes 4x4 blocks 49
Listing 7 Guard and pattern 49
Listing 8 FSM and priorities 50
Listing 9 Software data structure of FIFO channels 83
Listing 10 FIFO accesses based on circular buffer 84
Listing 11 Control-free FIFO accesses 84
Listing 12 Transposition of a 4x4 block in RVC-CAL 85
Listing 13 Transposition of a 4x4 block generated in C 86
Listing 14 Copy-free execution 87
Listing 15 Action scheduler 90
Listing 16 Combined scheduling algorithm 110
Listing 17 Directive-based actor classification 121

ACRONYMS

3AC Three-Address Code

ADF Architecture Definition File

ADM Abstract Decoder Model

ALU Arithmetic and Logic Unit

API Application Programming Interface

ASIC Application-Specific Integrated Circuits

ASIP Application-Specific Instruction-Set Processor

AST Abstract Syntax Tree

AVC Advanced Video Coding

AVS Audio Video Standard

BDF Boolean Data-Flow

BPDF Boolean Parametric Data-Flow

BSDL Bitstream Syntax Description Language

CAL CAL Actor Language

CBP Constrained Baseline Profile

CFG Control-Flow Graph

CISC Complex Instruction Set Computer

CSDF Cyclo-Static Data-Flow

DDF Dynamic Data-Flow

DPN Dataflow Process Network

DSE Design-Space Exploration

DSL Domain-Specific Language

DSP Digital Signal Processor

DST Discrete Sine Transform

EMF Eclipse Modeling Framework

FIFO First-In-First-Out

FNL Functional unit Network Language

FPGA Field-Programmable Gate Array

FSM Finite State Machine

FU Functional Unit

GPGPU General-Purpose Processing on GPUs

GPP General-Purpose Processor

GPU Graphics Processing Unit

HDF Heterochronous Dataflow

HDL Hardware Description Language

HEVC High Efficiency Video Coding

HLS High-Level Synthesis

HPC High Performance Computing

IBSDF Interface-Based Synchronous Dataflow

IDE Integrated Development Environment

IEC International Electrotechnical Commission

ILP Instruction-Level Parallelism

IR Intermediate representation

ISA Instruction Set Architecture

ISO International Organization for Standardization

ITU International Telecommunication Union

JADE Just-in-time Adaptive Decoder Engine

JPEG Joint Photographic Experts Group

KPN Kahn Process Network

LLVM Low Level Virtual Machine

LSU Load/Store Unit

MDE Model-Driven Engineering

MIMD Multiple Instruction Multiple Data streams

MISD Multiple Instruction Single Data streams

MOC Model of Computation

MPEG Moving Picture Experts Group

MPI Message Passing Interface

MPSOC Multi-Processor System-on-Chip

NOC Network-on-Chip

NORMA NO Remote Memory Access

NUMA Non-Uniform Memory Access

OCL Object Constraint Language

OMG Object Management Group

ORCC Open RVC-CAL Compiler

PHP Progressive High Profile

PIMM Parameterized and Interfaced Dataflow Meta-Model

πSDF Parameterized and Interfaced Synchronous Dataflow

PPN Polyhedral Process Network

PSDF Parameterized synchronous dataflow

RAM Random Access Memory

RF Register File

RISC Reduced Instruction Set Computer

ROM Read-Only Memory

RTL Register Transfer Level

RVC Reconfigurable Video Coding

SADF Scenario Aware Dataflow

SAS Single-Appearance Scheduling

SDF Synchronous Data-Flow

SIMD Single Instruction Multiple Data streams

SISD Single Instruction Single Data streams

SOC System-on-Chip

SPDF Schedulable Parametric Dataflow

SSA Static Single Assignment

TCE TTA-based Co-design Environment

TTA Transport-Trigger Architecture

TU Transform Unit

UMA Uniform Memory Access

VCEG Video Coding Experts Group

VLIW Very Long Instruction Word

VTL Video Tool Library

XDF XML Dataflow Format

XML eXtensible Markup Language

PUBLICATIONS

[1] D. de Saint Jorre, J. Gorin, J.-F. Nezan, M. Raulet, N. Siret, M. Wipliez, and H. Yviquel. MPEG / M20074 : Report on Performance of Generated Code (C, LLVM, and VHDL) from RVC Descriptions, 2010.

[2] J. Gorin, H. Yviquel, F. Prêteux, and M. Raulet. Just-in-time adaptive decoder engine. *Proceedings of the 19th ACM international conference on Multimedia - MM '11*, page 711, 2011.

[3] J. Heulot, K. Desnos, M. Pelcat, H. Yviquel, J.-F. Nezan, M. Raulet, P.-L. Lagalaye, and J.-C. Le Lann. An experimental toolchain based on high-level dataflow models of computation for heterogeneous MPSoC. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, 2012.

[4] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet. Efficient multicore scheduling of dataflow process networks. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 198–203, 2011.

[5] 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*, 28(10) :1295–1302, 2013.

[6] H. Yviquel, E. Casseau, M. Raulet, P. Jääskeläinen, and J. Takala. Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms. In *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, 2013.

[7] H. Yviquel, A. Lorence, K. Jerbi, A. Sanchez, G. Cocherel, and M. Raulet. Orcc : Multimedia development made easy. In *Proceedings of the 21st ACM international conference on Multimedia*, 2013.

[8] H. Yviquel, M. Wipliez, J. Gorin, M. Raulet, and E. Casseau. Classification-based optimization of dynamic dataflow programs. In *Advancing Embedded Systems and Real-Time Communications with Emerging Technologies*. IGI Global, 2014.

[9] International Standard ISO/IEC FDIS 23001-5 : MPEG systems technologies - Part 5 : Bit-stream Syntax Description Language (BSDL), .

[10] International Standard ISO/IEC FDIS 23001-4 : MPEG systems technologies - Part 4 : Codec Configuration Representation, .

[11] International Standard ISO/IEC FDIS 23002-4 : MPEG video technologies - Part 4 : Video tool library., .

[12] M. Abid, K. Jerbi, M. Raulet, O. Déforges, and M. Abid. System Level Synthesis Of Dataflow Programs : HEVC Decoder Case Study. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, 2013.

[13] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo. Cache-conscious scheduling of streaming applications. *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12*, page 236, 2012.

[14] J. J. Ahmad. *Secure Computing with the MPEG RVC Framework*. PhD thesis, Universität Konstanz, 2012.

[15] J. J. Ahmad, S. Li, A. Sadeghi, and T. Schneider. CTL : A Platform-Independent Crypto Tools Library Based on Dataflow Programming Paradigm. In *16th International Conference on Financial Cryptography and Data Security (FC 2012)*, volume 3, 2011.

[16] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers : principles, techniques, and tools*. 2007.

[17] M. A. Arslan, J. W. 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*, pages 1452–1456, 2012.

[18] O. Avissar and R. Barua. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 1(1) :6–26, Nov. 2002.

[19] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory : a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002*, pages 73–78, 2002.

[20] C. Beaumin, O. Sentieys, E. Casseau, and A. Carer. A coarse-grain reconfigurable hardware architecture for RVC-CAL-based design. In *Design and Architectures for Signal and Image Processing (DASIP)*, 2010.

[21] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur. BPDF : A Statically Analyzable DataFlow Model with Integer and Boolean Parameters. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, 2013.

[22] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012 : Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 983–987, 2012.

[23] E. Bezati, M. Mattavelli, and M. Raulet. RVC-CAL dataflow implementations of MPEG AVC/H. 264 CABAC decoding. In *Design and Architectures for Signal and Image Processing (DASIP)*, pages 207–213, 2010.

[24] E. Bezati, S. C. Brunet, M. Mattavelli, and J. W. Janneck. Synthesis and Optimization of High-Level Stream Programs. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, 2013.

[25] E. Bezati, M. Mattavelli, and J. W. Janneck. High-level synthesis of dataflow programs for signal processing systems. In *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, pages 750–754, 2013.

[26] B. Bhattacharya and S. S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. *IEEE Transactions on Signal Processing*, 49(10) : 2408–2421, 2001.

[27] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC : Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations. *Design Automation for Embedded Systems Journal*, 2(1) :1–33, 1997.

[28] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet. OpenDF : a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Computer Architure News*, 36(5) :29–35, 2009.

[29] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. Overview of the MPEG Reconfigurable Video Coding Framework. *Journal of Signal Processing Systems*, 63(2) :251–263, July 2009.

[30] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cvclo-Static Dataflow. *IEEE Transactions on Signal Processing*, 44(2) :397 – 408, 1996.

[31] J. Boutellier and O. Silvén. Towards Generic Embedded Multiprocessing for RVC-CAL Dataflow Programs. *Journal of Signal Processing Systems*, 73(2) :137—-142, 2013.

[32] J. Boutellier, C. Lucarz, S. Lafond, V. M. Gomez, and M. Mattavelli. Quasi-static scheduling of CAL actor networks for reconfigurable video coding. *Journal of Signal Processing Systems*, 63(2) :191–202, 2009.

[33] J. Boutellier, O. Silvén, and M. Raulet. Scheduling of CAL actor networks based on dynamic code analysis. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1609–1612, 2011.

[34] J. Boutellier, M. Raulet, and O. Silvén. Automatic Hierarchical Discovery of Quasi-Static Schedules of RVC- CAL Dataflow Programs. *Journal of Signal Processing Systems*, 71(1) :35–40, 2013.

[35] G. E. P. Box and N. R. Draper. *Empirical model-building and response surface*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[36] S. C. Brunet, M. Mattavelli, and J. W. Janneck. Profiling of Dataflow Programs using Post Mortem Causation Traces. In *IEEE Workshop on Signal Processing Systems (SiPS)*, pages 220–225, 2012.

[37] S. C. Brunet, M. Mattavelli, C. Alberti, and J. W. 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.

[38] S. C. 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, 2013.

[39] S. C. Brunet, M. Mattavelli, and J. W. Janneck. TURNUS : a design exploration framework for dataflow system design. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 654–654, 2013.

[40] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs : stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23(3) :777–786, 2004.

[41] J. T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, University of California, Berkeley, 1993.

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

[43] A. Carlsson, J. Eker, T. Olsson, and C. Von Platen. Scalable parallelism using dataflow programming. *Ericsson Review*, 2(1) :16–21, 2010.

[44] J. Castrillon. *Programming Heterogeneous MPSoCs : Tool Flows to Close the Software Productivity Gap*. PhD thesis, RWTH Aachen university, 2013.

[45] J. Castrillon, R. Leupers, and G. Ascheid. MAPS : Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics*, X(X) :1–19, 2011.

[46] G. Cedersjö and J. W. Janneck. Toward Efficient Execution of Dataflow Actors. In *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, pages 1465–1469, 2012.

[47] J. Ceng, J. Castrillon, W. Sheng, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS : An Integrated Framework for MPSoC Application Parallelization. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 754–759, 2008.

[48] P. Chandraiah and R. Domer. Code and Data Structure Partitioning for Parallel and Flexible MPSoC Specification Using Designer-Controlled Recoding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6) :1078–1090, June 2008.

[49] I. Chukhman, W. Plishker, and S. S. Bhattacharyya. Instrumentation-driven model detection for dataflow graphs. In *2012 International Symposium on System on Chip (SoC)*, pages 1–8, 2012.

[50] A. Cilio, H. J. M. Schot, J. A. A. J. Janssen, P. Jääskeläinen, and L. Laasonen. Architecture Definition File : Processor Architecture Definition File Format for a New TTA Design Framework. Technical report, Tampere University of Technology, Tampere, 2007.

[51] M. I. Cole. *Algorithmic skeletons : structured management of parallel computation*. 1989.

[52] D. Cordes and P. Marwedel. Multi-objective aware extraction of task-level parallelism using genetic algorithms. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 394–399, 2012.

[53] D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES/ISSS '10*, page 267, New York, New York, USA, 2010. ACM Press.

[54] D. Cordes, M. Engel, P. Marwedel, and O. Neugebauer. Automatic extraction of multi-objective aware pipeline parallelism using genetic algorithms. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '12*, page 73. ACM Press, 2012.

[55] H. Corporaal. *Microprocessor Architectures : from VLIW to TTA*. John Wiley & Sons, Chichester, UK, 1997.

[56] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[57] L. Dagum and R. Menon. OpenMP : an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1) :46–55, 1998.

[58] J. B. Dennis. First Version of a Data Flow Procedure Language. In B. Robinet, editor, *Programming Symposium*, pages 362–376. Springer Berlin Heidelberg, 1974.

[59] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *In Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132, 1975.

[60] K. Desnos, M. Pelcat, S. S. Bhattacharyya, and S. Aridhi. PiMM : Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration. In *Embedded Computer Systems (SAMOS), 2013 International Conference on*, 2013.

[61] A. V. Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *ACM Sigplan Notices*, 2000.

[62] R. Dolbeau, S. Bihan, and F. Bodin. HMPP : A Hybrid Multi-core Parallel Programming Environment. In *First Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–5, 2007.

[63] Eclipse. Graphiti : A Graphical Tooling Infrastructure, . URL `http://www.eclipse.org/graphiti/`.

[64] Eclipse. Spray : A Quick Way of Creating Graphiti, . URL `https://code.google.com/a/eclipselabs.org/p/spray/`.

[65] S. Efftinge and M. Völter. oAW xText : A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse*, volume 32 of *Abruf am 07.07.2012*, 2006.

[66] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, W. Hasselbring, R. von Massow, and M. Hanus. Xbase : implementing domain-specific languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, pages 112–121, 2012.

[67] J. Eker and J. W. Janneck. CAL language report : Specification of the CAL actor language. Technical report, University of California, Berkeley, Berkeley, 2003.

[68] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1) :127–144, 2003.

[69] J. Ersfolk, G. Roquier, F. Jokhio, J. Lilius, and M. Mattavelli. Scheduling of dynamic dataflow programs with model checking. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 37–42, 2011.

[70] J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli. Scheduling of dynamic dataflow programs based on state space analysis. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2012. ICASSP-12.*, pages 1661–1664, 2012. ISBN 9781467300469.

[71] O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez. Customized Exposed Datapath Soft-Core Design Flow with Compiler Support. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, pages 217–222, 2010.

[72] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Profile-guided deployment of stream programs on multicores. *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems - LCTES '12*, pages 79–88, 2012.

[73] W. M. Farmer. Chiron : A Multi-Paradigm Logic. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof : Festschrift in Honour of Andrzej Trybulec*, pages 1–19. University of Bialystok, 2007.

[74] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL) : An introduction. Technical report, Carnegie Mellon University, 2006.

[75] E. Fernandez-Alonso, D. Castells-Rufas, J. Joven, and J. Carrabina. Survey of NoC and Programming Models Proposals for MPSoC. *International Journal of Computer Science Issues*, 9(2) :22–32, 2012.

[76] A. Floch, T. Yuki, C. Guy, S. Derrien, B. Combemale, S. Rajopadhye, and R. B. France. Model-driven engineering and optimizing compilers : a bridge too far ? In *International Conference on Model Driven Engineering Languages and Systems*, 2011.

[77] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9) :948–960, Sept. 1972.

[78] C. M. Fonseca and P. J. Fleming. Genetic Algorithms for Multiobjective Optimization : FormulationDiscussion and Generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416—-423, 1993.

[79] M. P. I. Forum. MPI : A Message-Passing Interface Standard. Technical report, 2012.

[80] P. Fradet, A. Girault, and P. Poplavkoy. SPDF : A schedulable parametric data-flow MoC. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 769–774, 2012.

[81] M. R. Gary and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, San Francisco, Californiania, USA, 1979.

[82] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In *Proc. of the 12th European Symposium on Programming, ESOP 2003*, 2000.

[83] A. Girault, L. Bilung, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6) :742–760, June 1999.

[84] D. E. Goldberg and J. H. Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2-3) :95–99, 1988.

[85] H. Gonzalez-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks : high-level structured parallel programming enablers. *Software : Practice and Experience*, 40(12) :1135–1160, 2010.

[86] J. Gorin. *Machine Virtuelle Universelle pour Codage Vidéo Reconfigurable*. PhD thesis, Telecom Sud Paris, 2011.

[87] J. Gorin, M. Raulet, Y.-L. Cheng, H.-Y. Lin, N. Siret, K. Sugimoto, and G. G. Lee. An RVC dataflow description of the AVC Constrained Baseline Profile decoder. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 753–756. IEEE, 2009.

[88] J. Gorin, M. Wipliez, F. Prêteux, and M. Raulet. A portable video tool library for MPEG reconfigurable video coding using LLVM representation. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, 2010.

[89] J. Gorin, M. Wipliez, F. Prêteux, and M. Raulet. LLVM-based and scalable MPEG-RVC decoder. *Journal of Real Time Image Processing*, 6 (1) :59—-70, 2011.

[90] J. Gorin, M. Raulet, and F. Prêteux. Optimized dynamic compilation of dataflow representations for multimedia applications. *Annals of telecommunications - Annales des télécommunications*, 68(3-4) :133–151, 2012.

[91] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : a

seamless flow of graphs transformations. In *Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings. First ACM and IEEE International Conference on*, 2003.

[92] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. Technical Report September, ARM, 2011.

[93] R. Gu, J. W. Janneck, M. Raulet, and S. S. Bhattacharyya. Exploiting Statically Schedulable Regions in Dataflow Programs. *Journal of Signal Processing Systems*, 63(1) :129–142, Jan. 2010.

[94] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele. Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs. In *2009 IEEE/ACM/IFIP 7th Workshop on Embedded Systems for Real-Time Multimedia*, pages 35–44, 2009.

[95] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79 (9) :1305–1320, 1991.

[96] M. Hall, D. Padua, and K. Pingali. Compiler research : the next 50 years. *Communications of the ACM*, 2009.

[97] M. Hind. Pointer analysis : haven't we solved this problem yet ? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, 2001.

[98] J.-M. Hsiao and C.-J. Tsai. Analysis of an SOC Architecture for MPEG Reconfigurable Video Coding Framework. In *2007 IEEE International Symposium on Circuits and Systems*, pages 761–764, 2007.

[99] P. Jääskeläinen. Instruction Set Simulator For Transport Triggered Architectures. Technical report, Tampere University of Technology, 2005.

[100] P. Jääskeläinen, E. Salminen, C. S. de La Lama, J. Takala, and J. I. Martinez. TCEMC : A co-design flow for application-specific multicores. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 85–92, 2011.

[101] J. W. Janneck. A machine model for dataflow actors and its applications. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 756–760, Nov. 2011.

[102] J. W. Janneck, I. D. Miller, and D. B. Parlour. Profiling dataflow programs. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 1065–1068, June 2008.

[103] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing Hardware from Dataflow Programs. *Journal of Signal Processing Systems*, 63(2) :241–249, July 2009.

[104] K. Jerbi. *High Level Hardware Synthesis of RVC Dataflow Programs*. PhD thesis, INSA of Rennes, 2012.

[105] K. Jerbi, M. Abid, M. Raulet, and O. Déforges. Automatic Generation of Synthesizable Hardware Implementation from High Level RVC-CAL Description. In *International Conference on Acoustics, Speech, and Signal Processing*, volume 2012, pages 1–20, 2012.

[106] J.-M. Jézéquel, B. Combemale, S. Derrien, C. Guy, and S. Rajopadhye. Bridging the chasm between MDE and the world of compilation. *Software & Systems Modeling*, 11(4) :581–597, Aug. 2012.

[107] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1) :1–34, Mar. 2004.

[108] J. Joven, F. Angiolini, and D. Castells-Rufas. QoS-ocMPI : QoS-aware on-chip Message Passing Library for NoC-based Many-Core MPSoCs. In *2nd Workshop on Programming Models for Emerging Architectures (PMEA'10)*, 2010.

[109] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5) :589–604, July 2005.

[110] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74 :471–475, 1974.

[111] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1) :359–392, Jan. 1998.

[112] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 1970.

[113] B. Kienhuis, E. F. Deprettere, K. Vissers, and P. Van Der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 338–349, 1997.

[114] D. E. Knuth. Computer programming as an art. *Communications of the ACM*, 17(12) :667–673, 1974.

[115] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2) :190–222, Apr. 1983.

[116] C. Lattner and V. Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, number c, pages 75–86, 2004. ISBN 0-7695-2102-9.

[117] P. Le Guernic, M. Le Borgne, T. Gautier, and C. Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79 (9) :1321–1336, 1991.

[118] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond' (GLOBECOM), 1989. IEEE*, pages 1279–1283, 1989.

[119] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9) :1235–1245, 1987.

[120] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1) :24–35, 1987.

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

[122] F. Li, A. Pop, and A. Cohen. Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs. *IEEE Micro*, 32(4) :19–31, July 2012.

[123] C. Lucarz. *Dataflow programming for systems design space exploration targeting heterogeneous platforms*. PhD thesis, 2011.

[124] C. Lucarz, M. Mattavelli, and J. W. Janneck. Optimization of portable parallel signal processing applications by design space exploration of dataflow programs. In *2011 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 43–48. Ieee, Oct. 2011.

[125] M. Mattavelli, I. Amer, and M. Raulet. The Reconfigurable Video Coding Standard [Standards in a Nutshell]. *Signal Processing Magazine, IEEE*, 27(3) :159–167, 2010.

[126] M. Mattavelli, M. Raulet, and J. W. Janneck. MPEG reconfigurable video coding. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*, pages 281–314. Springer, New York, NY, USA, 2013.

[127] I. D. Miller. XLIM : An XML Language-Independent Model. Technical report, Xilinx DSP Division, 2007.

[128] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton. Erbium : A Deterministic, Concurrent Intermediate Representation to Map Data-Flow Tasks to Scalable, Persistent Streaming Processes. In *International Conference on Compilers Architectures and Synthesis for Embedded Systems (CASES'10)*, 2010.

[129] J.-F. Nezan, N. Siret, M. Wipliez, F. Palumbo, and L. Raffo. Multipurpose systems : A novel dataflow-based generation and mapping strategy. In *2012 IEEE International Symposium on Circuits and Systems*, pages 3073–3076, May 2012.

[130] H. Nikolov, M. Thompson, T. Stefanov, A. D. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. F. Deprettere. Daedalus : toward composable multimedia MP-SoC design. In *Proceedings of the 45th annual Design Automation Conference*, pages 574–579, 2008.

[131] H. Oh, N. Dutt, and S. Ha. Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems - CASES '05*, pages 157–165, 2005.

[132] H. Oh, N. Dutt, and S. Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Asia and South Pacific Conference on Design Automation, 2006.*, volume 3, pages 497–502, 2006.

[133] OpenACC. The OpenACC Application Programming Interface. Technical report, 2013.

[134] Orcc. The Open RVC-CAL Compiler : A development framework for dataflow programs. URL `http://orcc.sourceforge.net`.

[135] F. Palumbo, D. Pani, E. Manca, L. Raffo, M. Mattavelli, and G. Roquier. RVC : A multi-decoder CAL Composer tool. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, number Mdcc, pages 144–151, 2010.

[136] F. Palumbo, N. Carta, and L. Raffo. The Multi-Dataflow Composer tool : A runtime reconfigurable HDL platform composer. In *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, 2011.

[137] H.-w. Park. *Library Support in Actor-based Software Design for Multiprocessor Embedded Systems*. PhD thesis, Seoul National University, 2011.

[138] H.-w. Park, H. Oh, and S. Ha. Multiprocessor SoC Design Methods and Tools. *IEEE Signal Processing Magazine*, (November) :72–79, 2009.

[139] T. Parks. *Bounded scheduling of process networks*. PhD thesis, University of California, Berkeley, 1995.

[140] M. Pelcat, J.-F. Nezan, J. Piat, J. Croizer, and S. Aridhi. A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems. In *Design and Architectures for Signal and Image Processing (DASIP), 2009 Conference on*, 2009.

[141] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.-F. J.-F. Nezan. An Open Framework for Rapid Prototyping of Signal Processing Applications. *EURASIP Journal on Embedded Systems*, 2009(1), 2009.

[142] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan. *Physical Layer Multi-Core Prototyping : A Dataflow-Based Approach for LTE eNodeB*. Springer, 2012.

[143] J. Piat, S. S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, 2009.

[144] J. L. Pino and E. A. Lee. Hierarchical static scheduling of dataflow graphs onto multiple processors. In *1995 International Conference on Acoustics, Speech, and Signal Processing, 1995. ICASSP-95.*, pages 2643–2646, 1995.

[145] A. Pop and A. Cohen. A stream-computing extension to OpenMP. *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers - HiPEAC '11*, page 5, 2011.

[146] K. Pouget, M. Santana, P. L. Cueva, and J.-F. Mehaut. A novel approach for interactive debugging of dynamic dataflow embedded applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1547–1549, 2013.

[147] J. Protit, M. Tomasevit, and V. Milutinovit. Distributed shared memory : concepts and systems. *Parallel Distributed Technology : Systems Applications, IEEE*, 4(2) :63–71, 1996.

[148] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide : a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 519–530, 2013.

[149] M. Raulet, J. Piat, C. Lucarz, and M. Mattavelli. Validation of bitstream syntax and synthesis of parsers in the MPEG Reconfigurable Video Coding framework. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, 2008.

[150] I. E. G. Richardson. *H.264 and MPEG-4 Video Compression : Video Coding for Next-generation Multimedia*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[151] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour. Automatic software synthesis of dataflow program : An MPEG-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 281–286, 2008.

[152] J. E. Savage. *Models of computation : exploring the power of computing*. Addison-Wesley Pub, 1998.

[153] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 71–80, 2012.

[154] A. K. Singh, A. Kumar, and T. Srikanthan. A hybrid strategy for mapping multiple throughput-constrained applications on MPSoCs. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 175–184, 2011.

[155] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems : survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1 :1—-1 :10, 2013.

[156] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., Hoboken, NJ, USA, Apr. 2007.

[157] N. Siret. *Étude de l'implémentation automatisée sur plateforme mixte matérielle/logicielle d'applications de traitement du signal*. PhD thesis, INSA of Rennes, 2011.

[158] N. Siret, M. Wipliez, J.-F. Nezan, and F. Palumbo. Generation of Efficient High-Level Hardware Code from Dataflow Programs. In *Proceedings of Design, Automation and Test in Europe (DATE)*, 2012.

[159] T. B. Sousa. Dataflow Programming Concept, Languages and Applications. In *Doctoral Symposium on Informatics Engineering*, 2012.

[160] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF : Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley Professional, 2008.

[161] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6) :12–24, 1990.

[162] S. Stuijk, M. Geilen, and T. Basten. A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour. In *Proceedings of the 13th Euromicro Conference on Digital System Design : Architectures, Methods and Tools*, pages 548–555, Sept. 2010.

[163] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow : Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation*, pages 404–411, July 2011.

[164] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12) :1649–1668, Dec. 2012.

[165] W. R. Sutherland. *The on-line graphical specification of computer procedures*. PhD thesis, Massachusetts Institute of Technology, 1966.

[166] TCE. The TTA-based Co-design Environment. URL `http://tce.cs.tut.fi/`.

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

[168] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt : A language for streaming applications. In *Compiler Construction*, pages 179—-196, 2002.

[169] A. Vajda. *Programming Many-Core Chips*. Springer US, Boston, MA, 2011.

[170] C. H. van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1260–1265. Ieee, Apr. 2009.

[171] M. Viitanen, J. Vanne, T. D. Hämäläinen, M. Gabbouj, and J. Lainema. Complexity Analysis of Next-Generation HEVC Decoder. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 20–23, 2013.

[172] C. von Platen and J. Eker. Efficient realization of a cal video decoder on a mobile terminal. In *2008 IEEE Workshop on Signal Processing Systems*, pages 176–181, Oct. 2008.

[173] C. von Platen, J. Eker, A. Nilsson, and K.-E. Arzén. Static Analysis and Transformation of Dataflow Multimedia Applications. Technical report, Lund University, 2012.

[174] I. Watson and J. Gurd. A prototype data flow computer with token labelling. In *Proceedings of the National Computer Conference*, pages 623–628, 1979.

[175] D. B. West. *Introduction to Graph Theory*. Pearson, 2001.

[176] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H. 264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7) :560–576, 2003.

[177] M. Wipliez. *Compilation infrastructure for dataflow programs*. PhD thesis, INSA of Rennes, Dec. 2010.

[178] 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*, 2010.

[179] M. Wipliez and M. Raulet. Classification of Dataflow Actors with Satisfiability and Abstract Interpretation. *International Journal of Embedded and Real-Time Communication Systems*, 3(March) :49–69, 2012.

[180] M. Wipliez, G. Roquier, and J.-F. Nezan. Software Code Generation for the RVC-CAL Language. *Journal of Signal Processing Systems*, 63(2) : 203–213, June 2009.

[181] M. Wipliez, N. Siret, N. Carta, F. Palumbo, and L. Raffo. Design IP Faster : Introducing the C~ High-Level Language. *Design & Reuse*, 2013.

[182] W. Wolf. *High-performance embedded computing : architectures, applications, and methodologies*. 2010.

[183] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10) :1701–1713, Oct. 2008.

[184] G. Yang. Hybrid Decoder Reconfiguration of AVS-P7 and MPEG-4 /AVC in the Reconfigurable Video Coding Framework. *International Journal of Image, Graphics and Signal Processing*, 4(8) :57–65, Aug. 2012.

[185] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles - SOSP '87*, pages 63–76, 1987.

[186] C. Zebelein, J. Falk, C. Haubelt, and J. Teich. Classification of General Data Flow Actors into Known Models of Computation. In *2008 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 119–128, June 2008.