Dataflow Programming for Systems Design Space Exploration for Multicore Platforms

THÈSE N^{O} 5069 (2011)

PRÉSENTÉE LE 28 JUIN 2011

À LA FACULTÉ SCIENCES ET TECHNIQUES DE L'INGÉNIEUR

GROUPE DU LSM

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Christophe LUCARZ

acceptée sur proposition du jury:

Prof. G. De Micheli, président du jury
Dr M. Mattavelli, directeur de thèse
Prof. E. Charbon, rapporteur
Dr J. Janneck, rapporteur
Dr M. Raulet, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2011

Version résumée

De nos jours, les systèmes de traitement de l'information deviennent de plus en plus complexes dû au nombre croissant d'applications exigeantes présentes sur ces systèmes, surtout dans le domaine du traitement du signal et de la vidéo. Progressivement, la conception de tels systèmes devient un vrai défi, et ce pour trois raisons:

L'avènement de l'ère du parallélisme La précédente ère durant laquelle les performances des processeurs séquentiels doublent tous les ans est définitivement terminée. Aucune amélioration significative de leurs performances est à noter dans les cinq dernières années. Tous les projets visant à passer la barrière des 4 Ghz ont été annulés par les fabricants. Afin de continuer cette amélioration des performances et du pouvoir de calcul des processeurs, il n'y a pas d'autre choix que d'implanter plusieurs processeurs tout en essayant de réduire la consommation énergétique globale.

Les programmes de références ne sont plus adaptés Jusqu'à maintenant, l'utilisation de programmes de références écrit d'une manière séquentielle était l'approche la plus viable pour exploiter au mieux le potentiel de chaque nouvelle génération de processeurs dit séquentiels. Mais l'avènement cette nouvelle ère du parallélisme bouleverse les méthodes de conception et d'implantation des systèmes, résultant en l'obsolescence de la représentation séquentielle des programmes de références qui deviennent de plus en plus dur à utiliser comme point de départ pour concevoir ces nouveaux systèmes.

La demande grandissante des fonctionnalités et des performances En plus de la miniaturisation des systèmes et de la nécessité de satisfaire des contraintes sévères liés aux ressources, les fonctionnalités et la complexité d'une large gamme de systèmes tend à augmenter, les rendant comparables aux ordinateurs personnels classiques. La multitude des contraintes de temps et de qualité rend la conception architecturale de ces systèmes très problématique, tant au niveau de l'allocation des ressources qu'à celui de l'ordonnancement des tâches. Le probléme devient très préoccupante et constitue un défi sans précédent.

Les systèmes devenant de plus en plus complexes due à l'implantation d'applications exigeantes, il devient très difficile de les concevoir au plus proche de leur implantation finale en utilisant les langages respectifs à chaque plateforme (VHDL/Verilog et C/C++). Une solution à ce problème serait de créer un nou-

veau niveau d'abstraction pour la conception de ces systèmes. Mais la création de ce nouveau niveau suppose qu'il soit capable de modéliser correctement les niveaux sous-jacents, ce qui implique que le parallélisme exposé à un niveau supérieur d'abstraction puisse être applicable et utilisable aux niveaux inférieurs. En conséquence, la portabilité du parallélisme envers les différentes plateformes cibles doit être un point central dans l'élaboration de cette nouvelle couche d'abstraction et des méthodologies associées. La création d'un nouveau niveau d'abstraction peut présenter le risque de s'éloigner des caractéristiques réelles de l'implantation et de n'avoir plus assez de précision à haut niveau pour permettre d'obtenir des implantations efficaces.

Ce travail examine la solution qui consiste à utiliser un langage flux de données, en l'occurence CAL, pour servir de base à ce nouveau niveau d'abstraction. La résolution du problème passe par l'élaboration d'un environnement logiciel permettant l'exploration de l'espace des solutions à haut niveau pour faciliter l'implantation d'applications complexes sur des plateformes hétérogènes. Cela consiste à extraire des métriques à travers des analyses statiques et dynamiques des programmes CAL dans le but de guider les algorithmes heuristiques de partitionnement et d'ordonnancement afin de trouver les meilleures configurations possibles. Cela implique aussi le développement des méthodes visant à guider les concepteurs pour l'optimisation des programmes à haut niveau. Au final, c'est une méthodologie holistique qui a été élaborée sous forme de procédure pour guider les concepteurs depuis la spécification des applications jusqu'à leur implantation.

Alors que dans les approches traditionnelles, les différents niveaux d'abstractions doivent être tous abordés pour tester une solution d'implantation, le langage CAL comme base d'un nouveau niveau d'abstraction, permet aux concepteurs de séparer clairement trois aspects orthogonaux : fonctionnalité, partitionnement et implantation. Cette séparation des ces trois aspects de conception facilite l'exploration de l'espace des solutions et permet l'accroissement de la productivité dans la conception des systèmes complexes.

Ce travail a permis la création de plusieurs outils aidant à la conception : (1) des outils de profilage pour l'extraction de métriques à partir de programmes flux de données CAL et (2) des outils d'exploration qui incluent des algorithmes heuristiques de partitionnement et d'ordonnancement mais aussi d'évaluation des performances. Ces outils ont été développés dans le but de guider les concepteurs dans leur exploration de l'espace des solutions afin d'arriver à des implantations efficaces. Une méthode systématique a également été élaborée et appliquée avec succès à des applications tirées du monde industriel. Grâce aux mesures extraites des programmes CAL et aux méthodologies d'optimisation, les facteurs limitant d'un décodeur vidéo MPEG-4 Simple Profile ont été identifiés et optimisés, résultant en une augmentation des performances d'un facteur 7.

Mots clés Flux de données, parallélisme, exploration des solutions, méthode de conception, abstraction haut niveau, plateformes hétérogènes, profilage, partitionnement, ordonnancement, évaluation des performances, génération de code.

Abstract

Nowadays processing systems are asked to support increasing complex and demanding high-performance applications, especially in the signal processing and video processing domains. The design of these systems are becoming extremely challenging because of several factors. Among them the most relevant are:

The advent of the parallel era The sequential general purpose processor era during which performance increased more than two-fold every year, is definitely over. There has been no further increase in leading edge processor performance for over five years now and all projects aimed at breaking the 4 GHz barrier have been canceled by the major processor manufacturers. So as to continue increasing the computing power of computer systems, manufacturers have to use several slower processing cores, while trying to minimize the power consumption.

Sequential reference softwares are no longer adapted Up to now, sequential software have proven to be the winning approaches for exploiting the potential of each new generation of sequential processors. But the advent of the new parallel era changes the way systems are implemented and sequential reference software used as specification base are no longer appropriate starting points for designing complex systems.

The increasing demanding functionality and performance In parallel with miniaturization and the necessity to handle severely resource-constrained implementation platforms, the functionality and complexity of large classes of systems continue to increase making them from many respects comparable to ordinary desktop personal computers. A mixture of different types of timing and quality constraints make overall system architecture design, resource allocation and scheduling more important and challenging topics than ever before. Additionally, the market asks companies to build efficient systems while using the minimum of resources as fast as possible.

Due to the level of complexity reached by high-demanding digital systems, it is no longer reasonable to handle such systems at low levels of abstraction (e.g. VHDL/Verilog or C/C++). A solution to the problem is to higher the level of abstraction at which the design of such systems is performed. But for raising the level of abstraction, the need for portable parallelism is primordial so as to be able to implement all kind of systems on a wide range of target platforms. Thus, enabling portable parallelism should be a primordial requirement for the

development of this level of abstraction and the associated methodology and tools. However, creating a new abstraction layer may incur in the risk of loosing contact with implementation details and accuracy necessary for achieving efficient implementations.

This work focuses on tackling such problem by using the CAL dataflow language as main abstraction at the basis of the design process. The attempt of solving such challenging problem consists in building an environment for enabling high level design space exploration of complex parallel application for seamless implementation on heterogeneous platforms using CAL. It includes the extraction of metrics (static analysis and profiling) with the intention to use them as a basis for partitioning and scheduling heuristics aiming at finding the most efficient partitions. It also implies developing methodologies for guiding designers in the optimization of the high level specifications. Finally, a holistic approach has been defined in form of a complete design flow starting from CAL specifications down to implementations.

Where traditional approaches are faced to traverse, at each iteration of the design flow, all the levels of abstraction for testing a design solution, the use of CAL as an abstraction layer enables designers to separate clearly three domains of concerns that are made orthogonal: functionality, partitioning and implementation. The separation of these domains of concerns eases the design space exploration and leads to an increase in productivity in the design of complex systems.

This work leads to the creation of several new design tools: (1) profiling tools which extracts metrics from CAL programs and (2) exploration tools which include partitioning and scheduling heuristics and performance evaluation. These tools have been implemented with the aim of guiding designers in the high level design space exploration of systems in order to achieve efficient implementation. A systematic methodology has been also elaborated and has been tested successfully on a real world applications. Thanks to the metrics extracted from the CAL program, the design bottlenecks of a MPEG-4 SP video decoder have been identified and removed at a high level of abstraction, leading to improvements up to 7 times faster implementation.

Keywords Dataflow, parallelism, design space exploration, design flow, high level of abstraction, heterogeneous platforms, profiling, partitioning, scheduling, performance evaluation, code generation.

Acknowledgements

I would sincerely like to thank Dr. Marco Mattavelli for having supervised me during all these years. Special thanks for having involved me in so many MPEG meetings and in the ACTORS European project, it was very rewarding.

Several parts of my research work would have not been possible without the collaboration with many co-workers. I would like to thank – in alphabetical order – Dr. Jani Boutellier, Dr. Johan Eker, Abdallah Elguigy, Pascal Faure, Dr. Jörn W. Janneck, Dr. Andreas Karrenbauer, Samuel Keller, Martin Niemeier, Carl von Platen, Dr. Mickaël Raulet and Dr. Ghislain Roquier. I would like to thank to the IETR team (INSA-Rennes, France) for their great work which enables me to work on my research topics. I would like also to thank all my co-authors of the numerous publications. We all know how publications are important in research.

I have spent a very good time discussing, joking and re-inventing the world with the *french-speaking team* of the lab, Romuald, Ghislain, Richard, Anthony, Endri and Simone. Thank you all!

Claire, je voudrais te remercier d'avoir pris autant sur toi pour supporter toutes ces longues et dures années de vie à distance. Il est maintenant temps de mettre fin à cette séparation géographique !

N'oublions pas de remercier également Goldorak, Blanche Neige, le lapin au chapeau, Hop Hop Hop, le *critical path*, Johnny Be Good, Kolmogorov, le *good* CAL model, les Phares à On, le fonctionnaire, Baghoune le chatCAL...

Abbreviations

ADM	Abstract Decoder Model
ASIC	Application-Specific Integrated Circuit
AVC	Advanced Video Coding
AVS	Audio Video Standard
BSDL	Bitstream Syntax Description Language
BSD	Bistream Syntax Description
BSD	Berkeley Software Distribution
CA-2D-VLC	Context-Adaptive Two Dimensional Variable Length Coding
CAL	Caltrop Actor Language
CL	Computational Load
CMOS	Complementary MetalOxideSemiconductor
CP	Critical Path
CPU	Central Processing Unit
CSDF	Cyclo-Static Dataflow
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DDF	Dynamic Dataflow
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First In, First Out
FIR	Finite Impulse Response
FU	Functional Unit
FND	Functional unit Network Description
FNL	Functional unit Network Language
FSM	Finite State Machine
FPGA	Field-Programmable Gate Array
GALS	Globally Asynchronous Locally Synchronous
GPP	General Purpose Processor
GUI	Graphical User Interface
HDL	Hardware Description Language
HVC	High-Efficiency Video Coding
HW	Hardware
IDCT	Inverse Discrete Cosine Transform
IICT	Inverse Cosine Transform

IEC	International Electrotechnical Commission
IPC	Inter-Processor Cost
IQ	Inverse Quantization
IS	Inverse Scan
IT	Inverse Transform
ISO	International Organization for Standardization
JPEG	Joint Photographic Experts Group
KPN	Kahn Process Networks
MoC	Model of Computation
MPEG	Motion Picture Expert Group
NP	Nondeterministic Polynomial
ORCC	Open RVC-CAL Compiler
PNSR	Peak Signal-to-Noise Ratio
PU	Processing Unit
QCIF	Quarter Common International Format (178×144)
SDF	Synchronous Dataflow
SVG	Scalable Vector Graphics
RTL	Register Transfer Level
RVC	Reconfigurable Video Coding
SIT	Software Instrumentation Tool
SNR	Signal-to-Noise Ratio
SoC	Systems on Chip
SP	Simple Profile
SVC	Scalable Video Coding
SW	Software
TLM	Transaction Level Modeling
VHDL	Very high speed integrated circuit Hardware Description Language
VLC	Variable Length Codes
VLD	Variable Length Decoding
VOL	Video Object Layer
VTL	Video Tool Library
UML	Unified Modeling Language
WCET	Worst Case Execution Time
XDF	XML Dataflow
XSLT	Extensible Stylesheet Language Transformations

Contents

Chapter 1

Introduction

The research challenges in the field of digital systems for signal processing have radically changed since the first digital signal processing pioneering works in the 60s and early 70s aimed at showing the potential of transforming analog signals into digital samples. At that time, the implementation of basic building blocks of digital processing systems, such as analog to digital converters, FIR filters and FFT for example, represented the main architectural challenges. From a technology point of view the challenges were represented by the development of new silicon technologies, capable of providing faster and smaller circuits. With the advent of silicon CMOS technology, scaling into higher density components emerged as the main motor driving architectures and processing innovation. Indeed, in the past two decades, the performance of digital systems have progressed at an astounding pace sustained by the successful scaling of silicon CMOS technology in the submicron range providing powerful platforms in the form of general purpose processors, DSP, dedicated SoC and FPGA satisfying new demanding applications such as multimedia processing, digital transmission and video compression.

The importance of digital systems in modern economies is growing. They are used in mass market products and services in many domains: security, medicine, automation, transportation, consumer electronics, telecommunications. Digital systems span all aspects of daily life. The added value is created by supplying either functionality or quality. Functionality is defined as the service rendered to the user. Quality characterizes extra-functional properties of the product or service, such as performance, or dependability. For instance, a cellular phone offers functionality for mobile media communication, while quality is characterized by audio fidelity, battery life, and durability.

Modern digital systems are characterized by severe resources limitations, high complexity, and stringent performance requirements. Nowadays, the design methodologies adopted to cope with these difficulties have not yet reached maturity and are based on empirical techniques derived from the experience of the designer. Most of the design process is performed by trial and error. Thus, several iterations are needed to fit correctly the applications onto the target

platform. This type of approach is time-consuming is far from being optimal if the complexity of the applications and architecture of the target platform are complex. What is required is a unified approach to the design problem which borrows from different disciplines (from computer science, control systems, electronic engineering) to build the foundations of a solid design practice. Such an approach should enable the designers to model the application performance as a function of the underlying hardware and estimate the change in performance and the cost for migrating an application to another platform.

1.1 Motivation of the work

The design of digital systems is becoming extremely challenging because of several factors.

The race for the fastest processor ended For most of the history of silicon-based computing, the relentless scaling of silicon technology has led to ever faster sequential computing machines, with higher clock rates and more sophisticated internal architectures exploiting the improvements in silicon manufacturing. Backwards compatible processor designs ensured that software remained portable to new machines, which meant that legacy software automatically benefited from any progress in the way processors were built. In recent years, however, this has ceased to be the case: in spite of continued scaling of silicon technology, individual sequential processors are not becoming faster. All projects aimed at breaking the 4 GHz barrier have been canceled by the major processor manufacturers. The two major factors driving this radical change are the need to reduce power dissipation and the limits of electrical design models in the microwave range. Consequently, rather than building more sophisticated and complex single processors, manufacturers have used the space gained from scaling the technology by building more processors onto a single chip, making multi-core machines a nearly ubiquitous commodity in a wide (and increasing) range of computing applications. As a result, the performance gains of modern computing machines are primarily due to an increase in the available parallelism.

The sequential general purpose processor era is definitely over. Platforms are now mainly composed of several processing units.

Reference softwares and methodologies are no longer adapted Sequential models for specifying algorithms, sequential software and processors architectures have proven to be the winning approaches for exploiting the potential of each new generation of silicon CMOS scaling. Software developers and hardware designers knew that at each scaling the boost in performance achievable would be higher and easier to obtain than what could have been expected by developing new parallel systems, tools and architectures. The only important exception to this rule is represented by a specific field, the graphic world where the intrinsic wide parallelism of tasks has lead to the development of a

family of parallel architectures only used for a very restricted set of basic operations (e.g. polygon rendering). So, while the success of silicon CMOS scaling has been the main driving force for performance progress, it has at the same time practically restricted the way system specification software development, hardware design and all related tools and formalisms have evolved in the last twenty-five years. The consequence is that parallelism has not been a primordial issue when designing digital systems.

The case of MPEG is a good illustration of the situation. The "Moving Picture Experts Group" (MPEG) is a working group of experts that was formed by ISO and IEC to set standards for audio and video compression and transmission. Different forms of specifications have been adopted since the very beginning of standard video coding. MPEG-1 and MPEG-2 were only described by textual specifications. For MPEG-4, the C reference software became the true standard specification. However, the specification of the standard by means of such monolithic C/C++ descriptions will present several limitations and drawbacks. During the sequential processor era, it was convenient to have a monolithic reference software, which is no longer the case with the advent of the new parallel era. Mapping the monolithic C/C++ reference software means refactoring the application completely in order to distribute the computations on the different processing units. This task is far from being easy and is very time-consuming. The problem is even harder when the application should be mapped on heterogeneous platforms, made of sequential processors and FPGA for instance.

No common formalism, language or methodology is available to design, analyze and map parallel applications onto software and/or hardware platforms from the specification down to the implementation through all the different levels of abstractions.

Increasing demanding functionality and performance In the meanwhile, digital algorithms, such as digital video compression and multimedia processing, have grown in functionality and performance, at the expense of an ever growing complexity. In the domain of video coding technology, MPEG has produced, in the last 20 years, many important and innovative video coding standards, such as MPEG-1, MPEG-2, MPEG-4 Part 2, Advanced Video Coding (AVC), Scalable Video Coding (SVC) and is currently working on High-Efficiency Video Coding (HVC). Video coding technology in these years has reached very high levels of complexity.

Digital systems are asked to support an increasing number of complex applications, while keeping a high level of performance and minimizing resources consumption.

The implementation gap is growing The increasing complexity of systems coupled with the fact parallel programming is more difficult than usual sequential programming enlarges the gap between the traditional sequential specifications and final parallel implementation. Indeed, complexity has reached levels for which mapping application specification onto parallel architectures

now need to be performed using new methodologies and tools capable of assisting and supporting the work of the designer. The size of current source codes has become too large for any efficient optimization stage driven by designer creativity and becomes frustrated by the quantity of resource he needs to spend to yield a satisfactory working design. Figure 1.1 (Source: [1]) illustrates the problem faced by designers. Filling this gap is and will remain the main challenge of digital system design during the next two decades.

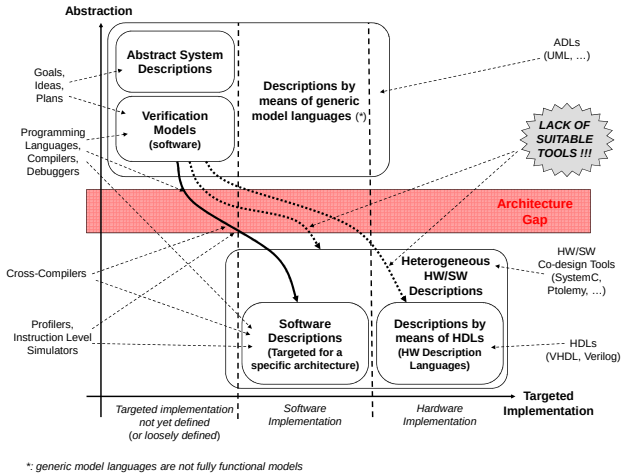


Figure 1.1: Methodologies and tools are missing for assisting designers in the implementation of complex sequential applications specifications on parallel platforms.

The gap between the application specification and its implementation was "filled" by the designer. This gap is enlarging and what is even more worrying is the fact that it is becoming much more difficult or even impossible to be covered by the designer's intuition and creativity alone.

Need for raising the level of abstraction Due to the level of complexity reached by high-demanding digital systems, it is no longer reasonable to handle such systems at low levels of abstraction (e.g. VHDL/Verilog or C/C++). A solution to the problem is to higher the level of abstraction at which the design of such systems are performed. The abstraction enables designers to focus on a small set of criteria during the design of the system and not to be slowed down by low-level implementation details. Abstraction is a well-tried technique. In integrated circuit design, several level of abstractions have been created over the years: logic gates, Register Transfer Modeling (RTL), VHDL. The same phenomena is visible in software: C/C++, object-oriented languages and high level

language in general. Abstraction through high-level languages make complex programming simpler and allows achieving quicker larger efficient designs. The architecture of high-demanding systems may include heterogeneous processing units (mix of FPGA/ASIC and software processors). The model of computation of these two devices are different, making even harder the design of such systems. Thus, abstracting them supposes a language that is capable of handling the two models of computations of these two processing units.

Several issues need to be solved: which formalism or language is capable of supporting such an abstraction and how to generate efficient software and hardware implementation code from these high level specifications?

Need for exposing portable parallelism An approach aimed at dealing with parallelism only by means of specific tools during the final mapping stage into a (parallel) platform is clearly inadequate when considering the complexity of today's specifications and algorithms. **Specifications and high level models should explicitly expose parallelism from the very beginning of any design flow and should be enough portable to be mapped easily of different types of platforms.** It will then be up to designer's creativity and to the specific design tools to specify the level of parallelism suitable for the architecture of the final platform.

1.2 Organization of the document

The state of the art of the existing methodologies and tools for dealing with the design of complex digital systems is presented in Chapter 2. It shows that the existing approaches do not gather all the necessary ingredients for supporting a robust high level methodology for seamless implementation of complex applications onto parallel platforms. It argues for a shift of paradigm for the specification, design and implementation of parallel systems.

Based on this new formalism - dataflow programming and CAL language - methodologies need to be developed in order to have methodologies and heuristics for designing, mapping and exploring the design space in a systematic way which guarantees efficient results with less efforts. Chapter 3 proposes a strategy for optimizing dataflow programs at the algorithmic level. Chapter 4 tackles the problem of mapping these optimized programs onto parallel platforms. Chapter 5 presents a systematic methodology aiming at guiding designers in the design space exploration. It includes profiling tools, partitioning heuristics and performance evaluation.

Chapter 6 presents how the tools, metrics and heuristics described in the previous chapters have been used and applied to a real world application, the MPEG-4 Simple Profile decoder. It details the different results at each step of the optimization and shows how the metrics and heuristics helped in guiding the designer during the optimization process.

The work for introducing this shift of the paradigm at the specification level has been concretized by the creation of a new ISO/IEC standard, Reconfigurable Video Coding (RVC), presented in Chapter 7 . It introduces dataflow programming and the CAL language for the specification of the video coding technology.

Chapter 2

State of the art

This chapter provides an overview of the existing languages, methodologies and tools used for high level design space exploration of parallel embedded systems. Research in embedded system design consists in finding the most efficient way to execute an **application** (e.g. an MPEG decoder, a JPEG encoder, a control of a robot, etc) onto a given **target platform with a given architecture** (composed of a set of processing units connected together through communication channels and memories) with the best performance while minimizing the resources. Another aspect to take into account in the equation is the time needed to reach such a design. The time required for achieving efficient implementation of parallel systems is higher than with sequential programs.

All the difficulty in system design resides in *how* to fit the application onto the target platform so that the system has the best performance while minimizing the resources. When dealing with parallel systems, i.e. a parallel program executed on several processing units, this task is even more complex because there are many ways to distribute the computations of the application onto the different processing units of the platform. This task of exploring and finding the best mapping of the application onto the target platform is called the **design space exploration**. The application and the architecture of the target platform are described with different languages and representations, which are not always suited for performing efficient design space exploration.

Furthermore, the language used to specify the application is not necessary the same as the one used for its implementation onto the platform. And even if it is the same, the code needs to be revisited in order to fit to the constraints of the platform. In any case, there is some burdensome and time consuming work to convert the specification of the application into the final implementation on the target platform. **The application can also be described in different languages between its specification and its implementation on the parallel platform.**

Section 2.1 reviews the different languages and representations to describe applications. Section 2.2 reviews the different languages for describing the architecture of the target platform and the tools capable of simulating the execution

of an application onto a target platform. Section 2.3 presents the different means for characterizing the application and the architecture and extracting the metrics guiding the design space exploration. Section 2.4 reviews the existing approaches and frameworks to bridge the implementation gap between specification and implementation. Finally, section 2.5 discusses the state of the art and shows how this work is providing a solution to a real need in the design of complex parallel embedded systems.

2.1 Application: languages and modeling formalisms

There are lots of **languages** that are capable of specifying application. There is no golden representation, it really depends on the domain of the application, the know-how of a company and the experiences of the designers. But the choice is constrained by the use of target platforms which accept most of the time C/C++ for software and VHDL/Verilog for hardware. Thus, the specification of the applications are often written in these languages. The problem with traditional programming languages (C, C++, VHDL, Verilog) is that they lie at a low level of abstraction, providing all the necessary implementations details. The problem is that these low-level details are an obstacle to the rapid exploration of the different implementation solutions because for each tested solution, every detail must be taken into account. This process is time-consuming and error-prone. The abstraction of these layers is a solution for dealing with this problem.

"*A **model of computation** (MoC) is an abstraction of a real computing device*" [2]. Because, working at the implementation level is not very convenient for exploring seamlessly the design space, a MoC aims at abstracting these lower levels by suppressing some properties and details that are irrelevant for design space exploration and they focus on other properties that are essential. The plethora of existing MoC makes the choice the "right" one a very difficult task. The choice of a MoC really depends on the field of application. Some are focusing on the reactivity of the system while other described more data paths. Being based on the classification of the MoC reported in [3], the following paragraphs report briefly the different models of computation used in embedded system design.

Finite State Machines Finite State Machines (FSM) have been used for decades because it is a convenient way to represent the different states and computations (transitions) of a program. A FSM can be represented as a graph containing a finite set of states and transitions. Each firing of a transition leads to a given state of the system. Two types of FSM exist: Moore type (or state-based), in which the output depends on the activation of a state [4] and the Mealy type (or transition-based) in which the output depends of the activation of a transition. A lot of extensions of the usual FSM exist in order to cope with precise aspects needed by the different application domains.

Finite State Machine with DataPath (FSMD) [5] is an extension of the common FSM formalism in order to cope with the lack of support of data memory and data processing capabilities. Hierarchical/Concurrent Finite State Machine (HCFSM) extends the common FSM formalism in order to cope with the lack of support of the concurrency and hierarchical structures of the model. Statecharts [6, 7] are the most well known formalism. Co-Design Finite State Machines (CFSM) [8,9] connect individual sequential elements in a global asynchronous network. Communications between elements are asynchronous and the elements (described by a FSM) are executed sequentially and synchronously, based on events. It is based on a Global Asynchronous Locally Synchronous (GALS) scheme and can well express the parallelism of an application. Program-State Machines (PSM) allows the use of programs to define a transition of usual FSM. VHDL and C code can be used to specify these actions allowing code generation. PSM formalism was used by *SpecCharts* [10] and *SpecC* [11] languages.

Petri nets Petri nets [12], invented by Carl Adam Petri in 1962, are directed graphs with two types of nodes: places and transitions. Places and transitions are connected alternatively, i.e. two places or two transitions cannot be connected. Places correspond to the states of the program and transitions are the computational entities. A firing of the transition implies the consumption of a token in each of the input places and output a token in each of the output places. The consumption and production rates of the transitions can be also weighted. Petri net have many declinations: Colored Petri nets, Timed Petri nets, Free Choice Petri Nets, etc. The reader is referred to the Petri net community to have further details.

Discrete Event In a discrete event MoC, events are sorted by a time stamp stating when they occur and analyzed in chronological order [3]. Transaction Level Modeling (TLM) is a discrete-event model of computation built on top of SystemC, where modules interchange events with time stamps. Interactions between software and hardware are modeled using shared buses. Modules can be specified at several levels of abstraction, making possible to specify functionals, untimed state machine models for the application and specify an instruction-accurate performance model for the architecture. TLM raises the level of abstraction one step above SystemC but remains low, making it hard to use for design space exploration.

Synchronous models In the synchronous MoC, all events are synchronous: signals have events with equal tags to other events in other signals. This type of MoC is more suited for programming control and real-time systems. Esterel [13] is one of these synchronous languages. A compiler exists for translating Esterel programs into Finite State Machines and in software and hardware implementation. Other synchronous languages are Lustre [14] and Signal [15].

Petri net community: `http://www.informatik.uni-hamburg.de/TGI/PetriNets/`

Dataflow models Dataflow models are graphs where nodes represent operations (actors) and edges represent data paths on which tokens are flowing. A token is an atomic piece of data. Dataflow graphs are often used to represent data-dominated systems, like signal processing applications. Several dataflow models are referred in the literature: Kahn Process Networks (KPN), Dataflow Process Networks and Synchronous Dataflow graph (SDF).

Kahn Process Networks (KPN) [16] have been introduced by Gilles Kahn in 1974 and consists in concurrent processes which communicate through unbounded and unidirectional FIFO. Each process is sequential but the global execution at the processes level is parallel. KPN cannot be scheduled at compile-time because its firing rules do not allow to build *a priori* a schedule such that the system does not block in any circumstances. The YAPI model [17] extends KPN by associating a data type with each channel and by introducing non-determinism. Dataflow Process Networks [18] is a special case of KPN.

Synchronous Dataflow graphs (SDF) [19] are similar to KPN but with additional restrictions: each firing of a node consumes and produces a given number of tokens, and for each firing the number of available tokens at its input must be larger that its consumption rate. The edges of the SDF graph are annotated with the number of data tokens produced and consumed by the computation at the node. These extra information are necessary for computing schedules at compile-time and memory requirements for buffering data between processing units.

2.1.1 CAL language

CAL [20] is a domain-specific language for describing actors in dataflow programs. It has been developed as part of the Ptolemy project at the University of California at Berkeley and was released in December 2003. More generally, it allows defining complex applications by interconnecting CAL actors. It provides useful abstractions for writing dataflow programs using several models of computations, for handling parallelism and for dealing with large complexity. Indeed, the CAL language has been used for specifying a wide variety of applications, and is well-adapted for describing complex signal processing algorithms with a set of encapsulated actors communicating with each others in a dataflow manner [143, 144]. The following sections present briefly the language. The reader is referred to [20] [145] for an in-depth description of the language.

2.1.1.1 Principle

The interconnection of CAL actors forms a dataflow program (or CAL application). Figure 2.1 illustrates a simple CAL application.

Actors are independent and can run concurrently. As a consequence, (1) the state of an actor is not shared, i.e. an actor cannot modify the state of another actor and not even access it and (2) actors can exchange data only through the exchange of tokens (atomic piece of data) using FIFOs between the ports (i.e. inputs/outputs) of the actors. Each actor is composed of one or several

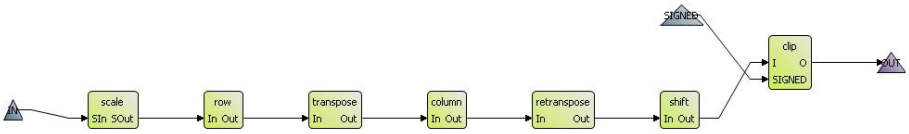


Figure 2.1: A CAL application results from the interconnection of actors specified in CAL language.

actions which are executed sequentially. Inside actors, firing rules determine which action must be fired next.

The CAL language allows defining applications in several models of computations seamlessly within the same environment: Synchronous Dataflow (SDF), Dynamic Dataflow (DDF), Cyclo-Static Dataflow (CSDF). CSDF extends SDF in the sense that production and consumption rates are constants for fixed sequence of firing of an actor. DDF extends SDF in the sense that actors may have multiple production and consumption rates and may be fired conditionally on input tokens values and actor state. The support of multiple MoC allows using it for describing a wide range of applications.

2.1.1.2 Basics

Structure of an actor The structure of a CAL actor is shown in Figure 2.2. The actor `splitter_BTYPE` has one input port `BTYPE` and three output ports `Y, U, V`. This actor contains three actions (`cmd.newVop, skip_BTYPE` and `cmd.split`). Action *cmd.newVop* consumes one token on the input port and produces one token on each output ports. Action *skip_BTYPE* consumes two tokens (keyword `repeat`) on the input port and produces two tokens on each output port. The Finite State Machine (keyword *fsm*) manages the firing of the actions, according to the values of the input tokens (keyword *guards*) and priorities between actions (keyword *priority*). Internal variables can be defined, such as `B_SZ` and `NEWVOP`.

Actions An action is an atomic piece of code. They can be labeled or not. Each action firing implies:

1. reading and consuming input tokens,
2. *and/or* modifying the internal state of the actor,
3. *and/or* producing output tokens.

Firing rules An actor may contain several actions that are executed sequentially. Firing rules and the corresponding language constructs exist for determining which action should be fired next:

Token availability An actions is fired only if the input tokens are available.

```
actor splitter_BTYPE () int(size=B_SZ) BTYPE ⇒
int(size=B_SZ) Y, int(size=B_SZ) U, int(size=B_SZ) V:

 int B_SZ = 12;
 int NEWVOP = 2048;

 cmd.newVop: action BTYPE:[cmd] ⇒ Y:[cmd], U:[cmd], V:[cmd]
 guard
  bitand(cmd, NEWVOP) != 0
 end

 skip_BTYPE: action BTYPE:[cmd] repeat 2 ⇒
Y:[cmd[0],cmd[1]], U:[cmd[0],cmd[1]], V:[cmd[0],cmd[1]]
 end

 cmd.split: action BTYPE:[list] repeat 6 ⇒
Y:[list[0],list[1],list[2],list[3]], U:[list[4]], V:[list[5]]
 end

 schedule fsm cmd :
   cmd ( cmd.newVop ) --> skip;
   cmd ( cmd.split ) --> cmd;
   skip ( skip_BTYPE ) --> cmd;
 end

 priority
   cmd.newVop > cmd.split;
 end

end
```

Figure 2.2: Example of a CAL actor.

Guards An action is fired only if the values of the input tokens satisfies the conditions.

Finite State Machine An action is fired only if it is enabled given the state of the actor. In Figure 2.2, the actor has two states: `cmd` and `skip`. When the actor is in the `skip` state, only the action `skip_BTYPE` is enabled and can be fired.

Priorities In case several actions are enabled, the priority statement indicates which action should be fired next.

2.1.1.3 Networks of actors

The connections of the actors form the CAL application. The network of actors is specified using an XML dialect called XML DataFlow (XDF). The language is defined in MPEG-B (ISO/IEC 23001-4 [21]). The corresponding description of the network of Figure 2.1 is illustrated in Figure 2.3. The network itself has input and output ports and the instantiated entities may be either actors or other networks, which allows hierarchical design.

```
<XDF name="IDCT2D">
 <Port kind="Input" name="IN">
  <Type name="int">
   <Entry kind="Expr" name="size">
    <Expr kind="Literal" literal-kind="Integer" value="13"/>
   </Entry>
  </Type>
 </Port>
 <Port kind="Input" name="SIGNED">
  <Type name="bool"/>
 </Port>
 <Port kind="Output" name="OUT">
  <Type name="int">
   <Entry kind="Expr" name="size">
    <Expr kind="Literal" literal-kind="Integer" value="9"/>
   </Entry>
  </Type>
 </Port>
 <Instance id="scale">
  <Class name="Proprietary.IETR.Scale"/>
 </Instance>
 <Instance id="row">
  <Class name="Proprietary.IETR.scaled_1d_idct"/>
 </Instance>
 <Instance id="col">
  <Class name="Proprietary.IETR.scaled_1d_idct"/>
 </Instance>
 <Instance id="transp">
  <Class name="Proprietary.IETR.Transpose"/>
 </Instance>
 <Instance id="retr">
  <Class name="Proprietary.IETR.Transpose"/>
 </Instance>
 <Instance id="shift">
  <Class name="Proprietary.IETR.rightshift"/>
 </Instance>
 <Instance id="clip">
  <Class name="Proprietary.IETR.Clip"/>
 </Instance>

 <Connection dst="clip" dst-port="SIGNED" src="" src-port="SIGNED"/>
 <Connection dst="clip" dst-port="I" src="shift" src-port="Out"/>
 <Connection dst="" dst-port="OUT" src="clip" src-port="O"/>
 <Connection dst="scale" dst-port="SIn" src="" src-port="IN"/>
 <Connection dst="row" dst-port="In" src="scale" src-port="SOut"/>
 <Connection dst="transp" dst-port="In" src="row" src-port="Out"/>
 <Connection dst="col" dst-port="In" src="transp" src-port="Out"/>
 <Connection dst="retr" dst-port="In" src="col" src-port="Out"/>
 <Connection dst="shift" dst-port="In" src="retr" src-port="Out"/>
</XDF>
```

Figure 2.3: A CAL application is defined by the interconnection of CAL actors. The network of actors is specified in XML DataFlow (XDF).

2.2 Architecture: specification and simulators

Simulating the execution of an application onto a given architecture is a very convenient feature by avoiding the designer to implement on the real platform the complete system, which is very time-consuming. The simulation allows evaluating the performance and the necessary resources for the current design, in terms of speed, power consumption, memory or area consumption.

Architecture Description Language (ADL) is a standard way for representing system architectures, aiming at promoting mutual communication, the embodiment of early design decisions, and the creation of a transferable abstraction of a system. Sophisticated ADL allow for early analysis and feasibility testing of architectural design decisions. There is a large variety in ADLs developed by either academic or industrial groups. In principle, ADLs differ from requirements languages, because ADLs are rooted in the solution space, whereas requirements describe problem spaces. They differ from programming languages, because ADLs do not bind architectural abstractions to specific point solutions. Modeling languages represent behaviors, where ADLs focus on representation of components. ADLs allow designers to create, refine and validate architectures. ADLs provide a basis for further implementation, so it must be able to add information to the ADLs specification to enable the final system specification to be derived from the ADLs. ADLs differ in their ability to handle real-time constructs, such as deadlines and task priorities, at the architectural level, to support the specification of different architectural styles. The positive element of ADLs is that it constitutes a formal way of representing architecture, are intended to be both human and machine readable, support descriptions of systems at a high level, permit analysis of architectures completeness, consistency, ambiguity, and performance. The drawback is that there is not a common agreement on what ADLs should represent, particularly as regards the behavior of the architecture. Representations currently in use are relatively difficult to parse and are not supported by commercial tools. The most known are Aesop [22], C2 [23], Darwin [24], LILEANNA [25], MetaH [26], Rapide [27], SADL [28], UniCon [29], Weaves [30] and Wright [31]. Medvidovic et al. [32] present a classification and comparison framework for these architecture description languages.

IP-XACT is an XML format that defines and describes electronic components composing the architectures. It was created by the SPIRIT Consortium as a standard to enable automated configuration and integration through tools. It has been recently standardized by IEEE [33]. The goals of the standard are to ensure delivery of compatible component descriptions from multiple component vendors, to enable exchanging complex component libraries between electronic design automation (EDA) tools, to describe configurable components using metadata and to enable the provision of EDA vendor-neutral scripts for component creation and configuration.

Simics is a full-system simulator used to run unchanged binaries of the target hardware at high-performance speeds. Simics can simulate systems such as Alpha, x86-64, IA-64, ARM, MIPS, MSP430, PowerPC, POWER, SPARC-V8 and V9, and x86 CPUs. Many operating systems have been run on various varieties of the simulated hardware, including MS-DOS, Windows, VxWorks, OSE, Solaris, FreeBSD, Linux, QNX, and RTEMS. The purpose of simulation in Simics is often to develop software for a particular type of embedded hardware, using Simics as a virtual platform. The Device Modeling Language (DML) enables to create and configure non-standard devices such as ASICs and FPGAs. DML largely automates the routine task of creating code to manage the hundreds and often thousands of registers in a modern system. A compiler translates DML into high-performance device models that enable Simics to simulate complete electronic systems at a performance measured in speeds of up to billions of simulated instructions per second. DML enables developers to start programming earlier, saving time and capital early in the product life-cycle. Simics [34] is now a product of WindRiver company.

Simulators at different levels of abstraction At the system level, **service curves** [35] describe a non-linear worst-case envelope for the computation or communication capabilities of system-level components for all possible time intervals. Service curves are measured in units of cycles, instructions, bytes, or service time per second. This model is used in EXPO [36] to model building blocks of SoC designs. At the task level, in **task-accurate models** the timing behavior of a resource is described by a list of supported tasks and their worst-case or average (estimated) execution times on this resource. This abstraction level is suited for SoC designs in which the level of granularity of interest is on the level of computation cores, memories, and buses. This model is used in the work of Ascia et al. [37]. At the lowest level of abstraction, in **instruction-accurate models**, the timing behavior of a resource is described by a list of symbolic instructions and their associated latencies. Traces of symbolic instructions are generated by annotated application models during execution and handed to the architecture models to determine the overall execution time of an application. This type of model is used in the SPADE framework [38].

2.3 Extracting metrics

When dealing with large software specifications, it is difficult for designers to implement them by relying on their intuition. Designers need tools to analyze finely the application and to detect quickly the bottlenecks in order to take the right decisions in the early steps of the design. The two main approaches for analyzing a program (i.e. the application) are static analyzes (Section 2.3.1) and dynamic analysis (Section 2.3.2).

In order to achieve the implementation that satisfies the requirements of the systems, developers need to know how fast and how resource-consuming their designs are. Program profiling can record all the operations performed during

the execution of the program, providing an exhaustive basis for the design space exploration. These extracted metrics can be classified into three categories: memory, computations, communication.

When dealing with parallel systems, another aspect must be taken into account: how the parallelism of the application is exploited by the platform? How the computations are distributed onto the different processing units?

These analyzes can be done at different levels of abstraction. At a low level of abstraction, it implies that the system is fully implemented, profiling tools exist and is not intrusive in the execution. It would results in very precise profiling but it is very time-consuming. At a high level of abstraction, it is less time-consuming but also less precise. The aim of the profiling is to extract the meaningful metrics which will lead to the right design choices during the design space exploration. Even if the profiling does not reflect exactly the behavior of the application on the target platform, it supports the designer to improve their designs without implementing the whole system, saving much time.

2.3.1 Static analysis techniques

Static analyzes are the analysis of computer software that is performed without actually executing programs built from that software. The methods based on a static analysis of the source code range from the simple counting of the number of operations appearing in a program up to sophisticated approaches determining lower and upper running time of a given program on a given processor [39, 40]. While the simple counting technique provides a very accurate evaluation of the operations, it cannot handle loops, recursion and conditional statements except for some particular cases. Explicit or implicit enumeration of program paths can handle loops and conditional statements and can yield bounds on run time best and worst case [39, 40].

The main drawback of these techniques is that the real computations of many real-life algorithms heavily depends on the input data while static analysis depends only on the application. For video coding algorithms, for instance, strict worst-case analysis can lead to results one or two orders of magnitude higher than the typical complexity values [41]. Consequently the range [*worst case, best case*] is so wide that results are meaningless.

2.3.2 Profiling tools

Profiling consists of the investigation of a program's behavior using information gathered as the program executes. There are several approaches to gather these information at runtime. There are two main approaches: either the source code is instrumented and compiled with extra code gathering runtime information (Section 2.3.2.1) or new instructions are inserted at run-time in the executable in order to gather these informations (Section 2.3.2.2).

2.3.2.1 Instrumentation during source code compilation

During the compilation of the source code, the program structure is known and including extra code at the source level is the way profiling metrics are obtained. The extra code is inserted in every function to the program so that when the program executes, the profiler gathers the measures. However these methods require the availability of the source code of the files, which is not possible with proprietary applications.

Software Instrumentation Tool (SIT) [42] has been developed with the goal of measuring the computations of a specific application independently from the underlying platform on which the program is run. Former methodologies are always platform- or compilation-dependent. The approach of the SIT is possible by means of a breakthrough in the instrumentation / overloading technology enabling a complete detection of all C operators without any limitation in the way pointers and data structures are used. There are no limitations for ANSI C and K&R compliant C code. It is based on GCC [43]. The SIT can be seen as a virtual-machine for running C source code. Furthermore, customizable virtual memory architectures can be plugged into the virtual-machine extending the analysis capabilities to the data-transfer and memory domain. The whole instrumentation process, from the source files to the instrumented executable, is automatic.

The SIT is performing the following analyzes:

- the **complexity analysis** provides information on the type and the number of operations executed. The type of the operands associated to each operation is also available. It helps the designer to partition the system into software and hardware, because hardware is better for intensive computing and software for control.
- the **dataflow analysis** gives the data transfers between the different functions of the program. It becomes very easy to visualize at a glance what are the large data transfers among the functions. This is a useful information when designing the architecture of the system.
- the **critical path analysis** shows the functions which belong to the critical path. Understanding which parts of the algorithm are critical is very important. For instance, it can help the designer to improve the throughput of the system by shortening as most as possible the critical path. It gives also an interesting indication useful in the HW/SW partitioning step.
- the **memory analysis** shows what type of memory operations are done.
- the **memory architecture analysis** helps the designer to find an optimal memory architecture for a given algorithm. The designer runs the C/C++ code with a virtual memory architecture and the tool provides the number of read or write misses and the read or write hits. Memory is a very

important issue in multimedia systems. Thus, it is very important to find the optimal solution at the first stage of the design flow.

- the **execution tree** indicates the execution order of the functions. It provides also the hierarchy of the function calls. How many times a part of the algorithm is run is also a valuable information to detect computer intensive blocks.

Other tools like Abstract Execution [44] and iprof tool [45] operate at the source code level. ATOM [46] is inserting instrumentation code after its compilation into the object files during linking.

2.3.2.2 Binary instrumentation

Binary instrumentation techniques insert dynamically (i.e. at runtime) additional instructions at precise locations in programs, collecting the necessary information for the profiling. The advantage is that source code is not needed, there is no need to recompile the source code with certain profiling options and the compiler-link process remains unchanged. The main drawback of these approaches is that the profiling is based on the binary and thus depends on the processor architecture, the operating system and the executable representation.

Pin [47] is a dynamic binary instrumentation framework (for Intel architectures) that enables the creation of dynamic program analysis tools. Pin is used in tools such as Intel Parallel Inspector, Intel Parallel Amplifier and Intel Parallel Advisor. The tools created using Pin, called Pintool, can be used to perform program analysis on user space applications in Linux and Windows. As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code.

Valgrind [48], QPT [49, 50] and Pixie [51] are other tools for binary instrumentation.

Libraries (such as Executable Editing Library [52] and Libbfd) have been also developed in order to read and modify the executable of a compiled program. Tools using these libraries are able to add foreign code before or after almost any instruction. The well-known Gprof [53] is using the Libbfd library for collecting the number of calls of each function and the amount of time spent in each, as well as the call graph.

2.3.3 Metrics guiding the design space exploration

Nowadays, it is no longer possible for designers dealing with complex parallel systems to find in a reasonable time the optimal partitions of the system by relying only on their intuitions. They need metrics to guide their choices.

During the design space exploration step, which consists in finding the best partitioning of the parallel application onto the target platform, developers should refer to these metrics to decide which part of the application and/or architecture to optimize. In embedded system design, the most three important

aspects are : computations, communication and memory. The following sections present the different metrics for characterizing the application, in these three domains.

2.3.3.1 Usual metrics

A system can be characterized by its **speed**, in different ways: the global throughput for computations and communications, the total amount of data processed or transferred, the reactivity of the system in the case of real-time systems, the global latency of the system, the time necessary for a given amount of computations or the clock speed supported by the design.

Resources consumption is an important issue in the current context of energy reduction. It shows how much resources (how many processors, how much memory, what is the data traffic between processing units?) are needed by the application. Energy is closely related to the use of these components. On the one hand, systems optimized for speed have to cope with the heat of components which could reduce the system lifetime. On the other hand, embedded systems have to deal with power leakage during idle periods. For hardware systems (FPGA, ASIC), designers talk about **area consumption**. It corresponds to the amount of silicon used for the implementation of the application. Reducing this area allows companies to use smaller chips, reducing resource consumption.

2.3.3.2 Computations-related metrics

Some parts of the application are more suited for a given type of target than others. For example, Computation-intensive tasks are more suited for hardware and control tasks for software. This is basically the aim of these metrics, to detect what kind of target is better suited for the different parts of the application.

An **affinity metric** is defined in the work of Sciuto et al. [54] [55]. The considered targets are GPP, DSP and ASIC. The authors define a set of constructs which are more adapted for different targets. The affinity metrics are the result of the analysis and highlight instruction sequences which are DSP-oriented (buffer circularity, MAC operations inside loops, etc.), ASIC-oriented (bit level instructions) or GPP-oriented (conditional or I/O instructions ratio).

A method for **selecting processors** is presented in [56]. Three metrics give the orientation of functions in terms of control, data transformation and data accesses by counting specific instructions from a processor independent code. A distance is computed, with specific characteristics of processors regarding their control, bandwidth and processing capabilities. It does not take into account the instruction dependencies and there is no detail about the different types of memory accesses regarding the abstract processor model used. This approach is at a very low level of abstraction, dealing with specific processors characteristics to choose what is the best processor for a given application.

A **regularity metric** is defined in [57]. If a given application exhibits a high degree of regularity, i.e., requires the repeated computation of certain patterns

of operations, it is possible to take advantage of such regularity for minimizing implementation overhead. It is interesting to detect these different types of regularity in order to optimize the design for one special target.

The **Control Orientation Metric (COM)** has been introduced in [58]. This metric indicates the frequency of control operations which cannot be eliminated at compile time. This metric is useful for evaluating the need for complex control structures to implement a function. Functions with high COM values will be more efficiently implemented in a global processing processor than in hardware (in which large FSM would be needed).

2.3.3.3 Communications-related metrics

Communications are very important in embedded system design because they can constitute a serious bottleneck if they are not well handled. Some methodologies help designers to analyze these communications at a high level of abstraction. Among them, clustering techniques aim at gathering functions / tasks / operators which are closely related and at separating functions / tasks / operators which are independent. These techniques can reduce the communication overhead. They can be applied at different levels of abstraction.

The metrics defined in [59, 60] are computed at the functional level to highlight resource, data and communication channel sharing capabilities in order to perform a pre-partitioning resulting in function clustering to guide hardware/-software partitioning. The aim is to optimize communications and resource sharing.

Locality of computations is defined in the work of Peixoto et al. [57]. Computations are said to have a high degree of locality if the computations, represented as nodes in a execution graph, constitutes an isolated, strongly connected (sub)graph. If an application exhibits several clusters with a good degree of locality of computations, such clusters define a clear way of organizing the functional units into modules so as minimize the number of global buses, control and steering logic.

At a lower level of abstraction, other communications-related metrics are presented in [61]. The goal is to quantify the communications between arithmetic operators (through memory or registers). These metrics focus on a fine grain analysis and are mainly used to guide the design of data paths, especially to optimize local connections and resource reuse.

2.3.3.4 Memory-related metrics

The metrics related to the memory are important in order to determine the best memory architecture of the system. A well-designed cache architecture in multiprocessor systems can lead to a significant speedup of the system. The number of conflict and capacity misses, cache hit and miss ratios, as well as the locality of accesses extracted from the application are useful in the early stage of the design flow to adapt either the application and/or the architecture to achieve good implementation.

The **Memory Orientation Metric (MOM)** [58] indicates the frequency of global memory accesses, i.e. accesses to input/output data. By referring to this metric, the designer can see which functions require special focus for implementation: those with large MOM values are most likely to require a good data bandwidth. The MOM metric also indicates the potential need for a memory hierarchy since, this metric is computed for all the hierarchy levels of a function graph.

The Software Instrumentation Tool (SIT) [42] have capabilities to test different cache architectures. As seen in Section 2.3.2.1, this tool is capable to provide the number of **read/writes misses** and the **read/write hits** for different memory architectures defined by the designer.

2.3.3.5 System-level metrics

Parallelism is a fundamental metric nowadays, as the target platform contains several processing units. Being capable to characterize a application in terms of parallelism is now becoming compulsory. Being able to detect which part of the application is inherently sequential or can be written in a more parallel way is a very precious information. In [58], they define a metric which indicates the average parallelism of a function. Those having the largest values offer more optimization opportunities since, they are likely to present a number of implementation alternatives offered by their inherent parallelism.

The notion of **flexibility** has been firstly described in [62]. The flexibility of a design is difficult to express quantitatively. However, many fundamental design decisions are based on the need of programmability or dynamic reconfigurability in order to extend the life-time of a design, to be able to incorporate late fixes due to, for instance, changes in communication standards, or to ease the remote maintenance of an embedded system. An application is described as a hierarchical directed acyclic graph (DAG) where hierarchy levels represent different options (algorithms) to implement the same functionality denoted by the parent node in the graph. An architecture is supposed to be more flexible the more options described in the application DAG it can implement, given timing and cost constraints.

2.4 Existing approaches for bridging the implementation gap

This section details the existing approaches aiming at bridging the implementation gap, i.e. mapping an application onto an architecture in the most efficient way. The reviews [8,63–67] present a large overview of the different methodologies.

2.4.1 Model-driven techniques

Daedalus [68] provides a unified environment for rapid system-level exploration, high-level synthesis, programming and prototyping of multimedia MP-SoC architectures. In this framework, the implementation is based on a library of pre-determined and pre-verified IP components. Daedalus design flow is strongly related to SPADE [38], Artemis [69] and Compaan framework [70]. This latter enables the automatic transformation of nested loop programs written in Matlab to Kahn-like process networks. This framework is based on Kahn Process Networks [16]. The design flow is fully automatic and comprises several tools. The KPNgen tool converts automatically sequential applications (written in C/C++) into a parallel Kahn Process Network (KPN). These latter are inputs to the Sesame modeling and simulation environment to perform system-level architectural design space exploration. The output of the Sesame tool is a set of candidate system designs with their corresponding specifications including system level platform description, application-architecture mapping description, and application description. Then, the ESPAM tool inputs these latter specifications with RTL description of the corresponding components from the IP library to perform VHDL synthesis. It implements the candidate MP-SoC platform architecture. C/C++ code is also generated for application processes that are mapped onto programmable cores. This implementation can be mapped onto FPGA using commercial synthesis tools and compilers. The Daedalus framework is very interesting, making the design flow fully automatic from Kahn Process Networks or directly from C/C++ specifications. KPN are well suited for signal processing systems. Nevertheless, some modifications in the C/C++ specification are sometimes needed in case the input specification did not entirely meet the requirements of the KPNgen tool, but these modifications seem not to be very time-consuming. The modeling of interrupts is difficult because of the nature of KPN models. Thus, it makes the study of time-dependent systems limited.

Metropolis [71] [72] [73] is a framework allowing the description and refinement of a design at different levels of abstraction and integrates modeling, simulation, synthesis, and verification tools. Metropolis is based on the Polis design environment [74]. The application is modeled as a set of processes that communicate through media. Architecture components are represented by performance models where events are annotated with the costs of interest. A mapping between functional and architecture models is determined by a third network that correlates the two models by synchronizing events (using constraints) between them. The framework uses an internal representation called the Metropolis Meta-Model (MMM). It uses different classes of objects to represent a design in which processes, communication media, and netlists describe the functional requirements, in terms of input / output relations.

Mescal [75] aims at designing heterogeneous, application-specific, programmable (multi) processors. The goal is to allow programmers to describe

an application in any combination of models of computation that is natural for the application domain. The goal is also to find a disciplined and correct by construction abstraction path from the underlying micro-architecture to an efficient mapping between application and architecture. The architecture development system is based on an architecture description language. The Mescal architecture development framework (called Teepee) implements a design space exploration methodology based on the Y-chart.

Ptolemy framework [76] focuses on component-based heterogeneous modeling and allows to combine hierarchically different models of computations at high level of abstraction in a Simulink [77] fashion. It uses tokens as the underlying communication mechanism. Controllers regulate how actors fire and how tokens are sent between each actors. This mechanism allows different models of computation to be combined within the Ptolemy framework.

PeaCE framework [78] specifies the system behavior with a heterogeneous composition of several models of computation. The PeaCE environment provides seamless co-design flow from functional simulation to system synthesis, utilizing the features of the formal models maximally during the whole design process. This framework is based on the Ptolemy project [76] [79]. When dealing with C/C++ specifications, the PeaCE approach does not propose an automatic procedure to transform this specification into dataflow graphs. This step is manual and an example of this transformation on a MPEG-4 decoder is given in [80].

2.4.2 C-based methodologies

The starting point of this type of tools is the specification written in an imperative language as C/C++.

C-to-Gates tools as ImpulseC [81], Handel-C [82] [83] and Spark [84] generate VHDL code from C-like specifications. They do not aim at exploring the design space but only translating a C-like representation of the algorithm into hardware code. Handel-C and ImpulseC tools allow some partitioning but the well-known C-to-Gate tool from Mentor Graphics (CatapultC [85]) does not allow any partitioning. Commercial offerings such as Mentor CatapultC, Celoxica Handel-C [86], C2Verilog, and Bach [87] defined a subset of ANSI C to do either synthesis or verification.

De Micheli and his students discussed in [88, 89] the main problems of using C as a HDL. The lack of concurrency and the concept of time are missing in C. The way communication mechanisms are written in imperative languages are not suited for hardware representation.

.NET framework based tool The .NET framework based tool [90] unifies and automates the hardware and the software design flows. It is capable of

refining automatically a high-level system specification, given in a programming language supported by the .NET framework, to a hardware/software description. The .NET framework [91] is a set of tools and mechanisms for the development of multi-lingual software components. The application is described with a high-level programming language supported by the .NET framework and is compiled into Common Intermediate Language (CIL) [92]. The hardware is described in CASM (Channel-based Algorithmic State Machine), an intermediate-level hardware description language [93].

This framework is interesting because it lets designers specify the system at a much higher level of abstraction than the traditional frameworks, with a large number of different programming languages, making the design flow more flexible. But design space exploration tools are missing. It is one-way design flow and there is back annotation capabilities in order to guide the design space exploration step aiming at leading to a satisfactory implementation.

2.4.3 Transaction Modeling and SystemC

SystemC, based on C/C++, is a solution for representing functionality, communication, software and hardware at various system levels of abstraction. It is mainly used for simulation because it is not totally synthesizable. In practice, it must be reduced for ensuring the use of a synthesizable subset which is at a low level of abstraction. Because of that, the simulation capabilities at a high level of abstraction are compromised. One can use the non-synthetizable subset to perform the high level simulation, but the code must be modified for the synthesis, which is a burdensome and error-prone task.

StepNP [94] is a system-level exploration framework based on SystemC targeted at network processors and developed by ST Microelectronics in collaboration with universities. It enables rapid prototyping and architectural exploration. It provides well-defined interfaces between processing cores, coprocessors, and communication channels to enable the usage of component models at different levels of abstraction. It enables the creation of multi-processor architectures with models of interconnects (functional channels, NoCs), processors (simple RISC), memories and coprocessors.

BlueSpec [95] takes as input a SystemVerilog or a SystemC subset and manipulates it with technology derived from term rewriting systems (TRS) [96] initially developed at MIT by Arvind et al. It offers an environment to capture successive refinements to an initial high-level design that are guaranteed correct by the system.

An other approach [97], more focused on SW, consists in making high level refactoring of the source code in order to take into account the target platform. This work is useful when mapping a program on processors but does not apply to the hardware domain.

Interface design Interfaces are often a serious bottleneck in embedded system design. Thus, the work of Jerraya and al. [98] focusses on interface design for multiprocessors SoC.

2.4.4 From graph specifications

In a more abstract way, the algorithm specifications can be described using different type of graphs. The three frameworks presented here use respectively Petri nets, control data flow graph or synchronous data flow graphs.

CodeSign framework [99] uses Object Oriented Time Petri Nets (OOTPN) as the modeling language. This language is quite powerful to model real-time systems and has a mathematical formalization that makes it easily analyzable. The notion of time allows performance evaluation at the early stages of the design. In order to facilitate the hardware/software partitioning task according to system constraints, CodeSign supports generic models which are refined with implementation details. Interfaces between hardware and software are automatically inserted according to the protocol specifications. Since the configuration of the interfaces can have a large impact on the system performance, CodeSign allows exploring different implementations derived from a master specification. CodeSign supports C and VHDL code generators for software and hardware implementation. Codesign project produced the Moses Tool Suite, a tool for modeling and simulating and evaluating heterogeneous systems using Petri nets. This tool supports the CAL language [20], a dataflow- and actor-oriented language, especially suited for modeling and simulating signal processing systems.

Trotter design flow [100] enables rapid prototyping and design space exploration by using an internal graph representation of the application. The framework provides metrics, useful for evaluating the impact of the design choices (of the application) on resource requirements in terms of processing, control, memory bandwidth and potential parallelism at different levels of granularity. The application must be written in C and is immediately converted into Hierarchical and Control Data Flow Graphs (HCDFG). The required information and the results are stored as attributes in this graph, composed of elementary nodes (processing, memory, control), dependence edges (control, data) or subgraphs. This framework performs automatically the application design space exploration at the event-based level. Tools are available for simulation, formal proof and code generation at the event-based level but they do not consider any path to hardware.

Syndex [101] [102] is a Computer-Aided-Design software aiming at mapping an application onto an architecture. It implements the Algorithm Architecture Adequation methodology (AAA) [103]. Within this environment, the designer defines an algorithm graph, an architecture graph and system constraints.

The methodology is based on graphs models to exhibit both the potential parallelism of the algorithm and the available parallelism of a given multi-component architecture. The methodology consists in distributing and scheduling the algorithm graph on the multi-component architecture graph while satisfying real-time constraints, maximizing a unique criterium, throughput. The heuristics take into account the execution time of the computations and inter-component communications. The results of the graphs transformations is an optimized Synchronized Distributed Executive (SynDEx), automatically built from a library of architecture dependent executive primitives composing the executive kernel.

2.4.5 UML-based design flow

Unified Modeling Language (UML) is largely used in software engineering for designing large software programs. The most complete design flow using UML for system modeling is achieved by Kukkala et al. and is called Koski.

The target of the Koski design flow [104] is multiprocessor System-on-Chip (SoC). It is a library-based method that hides unnecessary details from high-level design phases, but does not require a plethora of model abstractions. The design flow provides an automated path from UML design entry to FPGA prototyping, including functional verification, automated architecture exploration and back annotation. The design of the architecture is based on the application model: it results in a application-specific implementation. The flow has been successfully applied to a WLAN Video Terminal [105].

The targeted platform is a multiprocessor SoC platform implemented on FPGA. UML models (i.e. application, platform and mapping models) are written thanks to the experience of the designers and there is no help for their elaboration. Furthermore, one can wonder if UML is a nice way to express parallelism? When dealing with C/C++ specifications, UML models must be written, which can be a burdensome task for the designer. The design space is restricted to multiprocessor SoCs. An other aspect is highlighted in [106]: graphical languages are not well accepted because it is slower to use than writing code. Coding an algorithm in a textual manner is more productive than drawing the equivalent flow chart.

2.4.6 Commercial tools

There are several commercial tools helping in the design of complex systems, at different levels of abstraction.

Matlab-Simulink and LabVIEW Simulink from Mathworks [77] and LabVIEW from National Instruments [107] are the most known tools for modeling control and signal processing application using a nice graphical environment. These tools focus on the functional aspects of the algorithm and not really on the implementation aspect even if some progresses have been made in this domain. For example, the Xilinx System Generator for DSP tries to

fill the gap between the high-level abstract version of a design and its actual implementation in a FPGA.

Electronic System Level (ESL) tools The Signal Processing Worksystem (SPW) [108] is a tool using data flow formalism even earlier than Ptolemy. SPW has been acquired by CoWare in 2003. Coware proposes a system-level tool performing design, analysis and simulation of system specified in SystemC and using TLM.

Synopsys System Studio [109] is a model-based design and analysis tool helping the designer to build systems at a high level of abstraction. It supports all the models of computation supported by SystemC. It is possible to run co-simulation of HDL descriptions with SystemC models or Matlab algorithms. This tool supports also hardware synthesis from SystemC. C/C++ and SystemC are used to describe a high level model or to integrate existing Intellectual Property (IP) blocks. Because it is hard to make reusable models in C/C++, SystemC is often used as an encapsulation mechanism. The problem of these tools is that there is no explicit methodology which helps the designer in choosing the right mapping decisions.

2.5 Discussion

The state of the art covered briefly the different aspects of design space exploration applied to parallel digital systems: how the application is specified, how the architecture is handled, how to extract the metrics to characterize the metrics that guide the design space exploration heuristics and what are the existing tools and methodologies which support designers to reach efficient implementation of parallel applications.

There is still something missing

Many tools exist but most of them address the problem only partially and focus only on well-bounded problems such as efficient hardware or software mapping, interfaces, partitioning, scheduling, etc. C-to-Gates tools [81–87] address the problem of translating C/C++ code into hardware. The .NET framework based tool [90] unifies the hardware and software design flow to have an automatic mapping of the application on the different processing units. But it lacks design space exploration tools. Kangas et al. [104] present the Koski design flow that only covers the design phases from system-level modeling to FPGA prototyping.

Few of them consider an holistic approach to the problem of bridging the implementation gap, from the specification down to the implementation [78] [76] [68].

The more evolved approaches that are closed to consider the problem as a whole are the Ptolemy/Peace project [78], the Daedalus framework [68], Metropolis [71] and Mescal [75].

Up to the author's best knowledge, there is no tool that provides a complete design flow for the design of complex digital systems from the specification down to the implementation onto the target platform comprising:

High level specification and design that guarantees the seamless writing of complex applications with features such as conciseness, reuse and portability. The well-defined language is necessary for enabling the compatibility between the developed tools.

Systematic design space exploration that guides the designer in the exploration of the large design space. It should include profiling tools, partitioning / scheduling heuristics and an optimization strategy for helping designer in making the right design choice in the early steps of the design flow.

Efficient code generation that translates the high level description into efficient native implementation code according to the target platform.

Proposal: raise the level of abstraction...

Abstraction is the solution adopted by many approaches for bridging the implementation gap because it allows to focus on a small set of aspects and to hide low-level implementation details. Working at a higher level of abstraction is a non-negligible gain of time and resources for the design and the debug of applications. Memory management, time dependencies and synchronization processes are such aspects that can be hidden.

Raising the level of abstraction is an interesting mean for shortening the design cycle of digital systems. Performing the design space exploration at high level of abstractions allows designers to make the right choices at the early steps of the design flow, avoiding loosing time in implementing non-efficient solutions at low-level of abstractions.

The design of digital systems with competitive performance is speed-up thanks to the deeper exploration of the design space, made possible through the raise of the level of abstraction.

...using dataflow programming...

The increasing complexity of digital systems coupled with the advent of parallel platforms put back into question the use of sequential specification and ask for a shift of paradigm. Why continuing specifying complex applications in a way that does not help for the implementation? Sequential specifications were extremely useful while using traditional sequential processors to implement them, they belong to the same paradigm. It is not the case anymore: target platforms are now parallel and specifications remain in a sequential form.

Sequential programming hides parallelism Imperative programs are modeled as a sequence of operations and the flow of data between these operations is secondary. The over specification of imperative programming in terms

of control hides the potential parallelism of the application, limiting the way operations can be executed and thus the way these operations are mapped on threads and processing units. A direct consequence of this over specification is that it does not give any freedom in the scheduling of the different operations of the application onto processing units [110].

A strong lack of flexibility *"Threads are a seemingly straightforward adaptation of the dominant sequential model of computation to concurrent systems"* [111] argued Edward A. Lee. He explained clearly why threads are not a good idea for dealing with parallelism.

Applications are structured explicitly using threads and processes that communicates using shared memory or other means of communications (e.g. messages, pipes, semaphores). *"Threads are often not a practical abstraction because creating the illusion of shared memory is often too costly"* [111]. The structure of the application is hardly modifiable to fit the target platform because of the too expensive management of the locks and synchronization processes. The cost of refactoring the applications to fit the different target platforms is very costly in terms of rewriting effort and is risky because bugs can be introduced. The direct consequence of this lack of flexibility is that the degree of exposed parallelism is rarely adapted to the available parallelism of the platform: either too many threads are mapped on processing units, incurring large scheduling overhead or processors are under-utilized. An application with a given level of exposed parallelism is optimal only if implemented on a platform with the same level of parallelism [110].

The other way round? As long as processors were sequential, reference software were written in imperative language and were the most appropriate way for specifying and implementing systems. With the advent of the multicore era, programmers need to parallelize applications to fit parallel platforms. It is only a deep knowledge of the application algorithm that allows restructuring the program so that it exposes more parallelism. One can think of the other way round: why not exposing the potential parallelism of the application and *sequentialize* (if necessary) the operations of the application so that it fit to the available parallelism of the target platform [110]?

Unlike traditional approaches, dataflow programming is a natural way to expose the parallelism of signal processing applications because the structure itself of the program (set of interconnected actors) exposes explicitly this parallelism. Actors are independent from each other, providing a great flexibility for the design and implementation of parallel systems. This flexibility allows exposing seamlessly different levels of parallelism of the applications, which is a very interesting feature for exploring the design space.

...and Cal language

Each tool described in Section 2.4 has its own language with their own specificities. The choice of the language capable of supporting a high level design space exploration is primordial and is specific to the application domain. The review of the existing languages used in digital systems design highlights the primordial properties of languages capable of supporting a high level design space exploration.

The tools based on various graphs [99] [100] [102] for the representation of the application are limited either by the fact that the representation is not well formalized or limited to a very small domain of application, limiting the simulation and portability of the application, or by the fact that it is not convenient to generate implementation code from it. Tools [94] [95] based on SystemC offer a too low level of abstraction to support high level design space exploration because too many implementation details are exposed.

The Cal language (Section 2.1.1 and [20]) has some interesting feature for describing the computational entities of a dataflow representation, namely the actors.

Conciseness It provides a concise high-level description of an actor. Cal is a textual language for writing down the functionality of actors: input and output ports, parameters, states, finite state machines, typing constraints and firing rules. The advantage is obviously a reduction of the lines of code to be written.

Formal specification Having a rigorous specification of the language is very important for its simulation and for generating implementation code from it. Otherwise, compatibility issues are a bottleneck for the development of such tools. It ensures also the portability of the programs and is a larger guarantee for its adoption.

Analyzability Through its representation in Cal, an actor is easily analyzable. One can extract the tokens rates of actors, the sequence of actions firing inside an actor, and can be used for dataflow analysis and code generation. This is a very important feature for being able to support high level design space exploration.

Several models of computations The Cal language can describe actors in different Models of Computation. Thus, in a single dataflow program, actors may be of different natures: Synchronous Dataflow (SDF), Cyclo-Static Dataflow (CSDF) and Dynamic Dataflow (DDF).

Modularity and reuse Cal language insulates the behavior of each actor, making them independent from each others during the execution of the program. This encapsulation property enables the seamless composability of

dataflow programs. This ability to create new computational entities by using existing entities is a valuable feature for easing the coding of applications. It makes the writing of actors more accessible and makes actors portable and reusable. The possibility to define an actor as a sub-network of actors enables hierarchical design of programs, helping managing the high degree of parallelism.

Required features

It is nice to raise the level of abstraction, but in order to be usable, it should support simulation, be portable for code generation and be able to support high level design space explorations methodologies.

Simulation capabilities The level of abstraction should have the necessary constructs and semantics to support simulation, which is a essential step for designers for the design and the debug of parallel applications. Too abstract languages do not often support simulation and architectural representations are at a too low level of abstraction, making the design harder. The CAL language, through its formal definition [20], has these necessary simulation capabilities.

Code generation and portability Having an appropriate representation at high level of the application is a nice feature but is useless if there is no way to make profit of it by generating implementation code from it.

The architectures of target platforms are various and present a wide range of levels of available parallelism. The traditional thread-based approach lacks portability because application threads cannot scale properly to the available parallelism of the platform and cannot be fully exploited. The variety of the target processing units causes the problem of portability of the code. CUDA (Compute Unified Device Architecture) is a parallel computing architecture specially developed by NVIDIA and compatible for a small set of cards. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. In October 2009, Tilera Corporation, a semiconductor company focusing on scalable multicore embedded processor design, announced a new chip TILE-Gx100 that features up to 100 cores at 1.5 GHz. Tilera provides software development tools - Multicore Development Environment (MDE) - and a line of boards, both built around the Tile processor.

The loss of time and resources may be considerable if changing the target platform means rewriting completely the application. As a consequence, the design space exploration is so expensive that it is inexistent. Flexibility and cross-platform portability are major issues that guarantee efficient system design.

Because CAL language makes no assumptions about the underlying physical architecture, native implementation code for a wide range of programming devices (sequential processors, multicore, manycore, processors arrays, FPGA, GPU, etc.) can be generated from CAL programs.

High level design space exploration Representing a program as a network of connected encapsulated actors is a strong guarantee for providing the necessary flexibility for efficient design space exploration. Dataflow programming is a natural way to express the inherent parallelism through the interconnection of actors. One can expose different levels of parallelism of any application according to the level of granularity chosen to specify the actors. The encapsulation property of CAL actors allows the easy interchange of actors with different levels of granularity. A monolithic actor can be replaced by a sub-network of actors to increase the degree of parallelism and vice-versa to reduce it. The encapsulation property prevents any actor to modify the state of another actor, allowing the safe and seamless transfer of any actor from a processing unit to another because there is no need to take care of sharing the states of actors among the different processing units. Unlike in imperative programs in which the sequence of computations is specified, dataflow programming with CAL provides a valuable degree of freedom that allows designers to apply different scheduling techniques in order to best fit target platforms.

The advantageous features of dataflow programming with CAL (scalable parallelism, partitioning and scheduling capabilities, portability) are mandatory for supporting heuristics and methodology for the exploration of the design space at a high level of abstraction.

Summary

Traditional approaches lack flexibility for scaling reference softwares on massively parallel platforms and there is no existing approach that gathers all the required features for efficient design of complex systems, namely:

- High level specification of complex applications based on a formalized language,
- High level design space exploration, including profiling tools, partitioning and scheduling heuristics and optimization strategy,
- Efficient code generation.

Dataflow programming with CAL language for describing actors is an interesting paradigm for building an high level design space exploration environment in order to ease the specification, the design and the implementation of complex digital systems.

Chapter 3

A strategy for the algorithmic optimization of dataflow programs

Most of the applications are traditionally specified as sequential programs that do not reveal the available algorithmic parallelism, thereby hindering the efficient mapping of this application onto a parallel multi-processing units platform. CAL language provides the necessary constructs to describe concisely parallel application. Actors are sequential processes that are executed concurrently, on the condition that data dependencies are respected. Thanks to this duality, an application can be written in different ways, exposing more or less parallelism, letting infinite manners to write CAL programs.

On one side, writing sequential CAL programs which do not expose enough parallelism are not suitable for multicore platforms because the inherent parallelism of the application cannot be exploited by the platform, resulting in large actors that cannot be reused.

On the other side, exposing too much parallelism is not the solution either. CAL programs have to be translated into implementation code (C/C++ for software and VHDL/Verilog for hardware) for the target platforms. CAL language allows to define applications at a higher level of abstraction, but the price to pay for this new feature is the introduction of some overhead in the management of the firing of the actions in a parallel way. Fine-grain parallelism implies too much communication and management overhead compared to the computations of the application. However, it is not always the case for all types of platforms. In hardware, too much parallelism is not really a problem because all the actors can run in parallel. In software, it causes a problem if the number of processors is less below the number of actors, the cost for managing the actions firing on the different processors may become important.

Having in mind all these aspects, the aim of the developer is to write a CAL program that *scales* properly to the target platform in terms of parallelism in

order to fully exploit its available resources and to obtain the best performance. But the problem resides in how to proceed to determine the appropriate level of parallelism that needs to be exposed.

This chapter proposes a strategy which consists in guiding the designer in the refactoring of the program by highlighting the actors that impact the most the completion date of the program and by proposing appropriate refactoring techniques to remove the potential bottlenecks. A first section introduces preliminary notions on parallel programming. Section 3.2 describes different metrics that can be extracted from CAL programs and their associated tools. Section 3.3 exposed the optimization strategy and the last section presents the tools that implement this strategy.

3.1 Notions on parallel programming

Section 3.1.1 introduces some notions about the speed-up of programs and Section 3.1.2 lists the different types of parallelism.

3.1.1 Amdahl's law

In case of sequential programs, the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion γ of that computation where the improvement has a speedup of S. For example, if an improvement can speed up 30% of the computation, γ will be 0.3; if the improvement makes the portion affected twice as fast, S will be 2. Amdahl's law states that the overall speedup of applying the improvement will be:

$$Speedup = \frac{1}{(1-\gamma) + \frac{\gamma}{S}} \tag{3.1}$$

In case of parallel programs, this law states that if γ is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and (1 - γ) is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is

$$Speedup = \frac{1}{(1-\gamma) + \frac{\gamma}{N}} \tag{3.2}$$

In light of these laws, CAL programs can be optimized on two main axes:

1. Optimize the unbreakable sequential portions of the program.
2. Remove the unnecessary dependencies between actors, responsible for the sequentiality, identify the portion of the programs that can be made parallel and split actors to expose this new available parallelism.

3.1.2 Parallelism taxonomy

There are different types of parallelism (task, data, pipeline) at different levels of granularity (coarse- and fine-grain). In parallel computing, granularity

means the amount of computation in relation to communication, i.e. the ratio of computation over communication.

Task parallelism *"refers to pairs of actors that do not have precedence relations in a dataflow program. Precedence relations are inferred from the tokens dependencies between actors in a dataflow program (*A precedes B *means B cannot execute until completion of A)"* [112].

Whereas the splitting of actors may increase task parallelism (in case the splitting does not imply internal dependencies), the merging of actors reduces it. One can exploit this type of parallelism by mapping the actors on distinct processing units. Exposing task parallelism by distributing the computations into actors cannot be automated. It is the knowledge of the application that guides this decomposition.

Data parallelism *"refers to actors that have no (state or token) dependencies between successive firing. A set of input data can be processed concurrently"* [112]. In other words, N sets of tokens can be processed by N different actor instance.

Exposing data parallelism consists in splitting an actor into several identical instances, each one processing a subset of the input data. One can exploit data parallelism by mapping the multiple instances of the same actor on distinct processing elements. There exists no tool capable of automatically splitting such actors.

Pipeline parallelism *"refers to chain-structured region of the dataflow program. An actor does partial processing and then forwards the result to another. [...] This kind of parallelism is useful when dealing with a stream of data on which a certain number of processing should be applied. The gain will be observed on a set of data and a small cost induced by the initialization of the pipeline (that implies a latency on the first processing)."* [112]

Exposing pipeline parallelism consists in structuring actors into a chain such that each actor depends only on its predecessor. One can exploit pipeline parallelism by mapping those actors on distinct processing elements. There is no tool for organizing the actors such a way.

Coarse- and fine-grain parallelism Relatively small actors - in terms of code size and execution time - that are exchanging tokens frequently refers to fine-grain parallelism. On the contrary, relatively large actors that are exchanging tokens infrequently refers to coarse-grain parallelism. *"The finer the granularity, the greater the potential for parallelism and hence speed-up, but the greater the overheads of synchronization and communication* .

Fine-grain parallelism is hardy exploitable by software platforms because the communication costs tends to hide the gain obtained from the parallelization

Source: `http://foldoc.org/granularity`

and sometimes results in worse performance. Fine-grain parallelism is more appropriate to hardware platforms that can exploit this parallelism with lower communication costs. On the contrary, coarse-grain parallelism is more appropriate to software platforms because communications are expensive. Whereas splitting actors lowers the level of granularity, merging actors contributes to raise it. The splitting of actors cannot be automated but the merging of actors can.

Figure 3.2 (Source: [112]) depicts the different kinds of parallelism, applied to a simple dataflow program (Figure 3.1 ; Source: [112]). Let $C_A = C_B = \frac{C_C}{3} = C_D = C_E = C_F$ be the computational load of each actor. Communication costs are not considered.

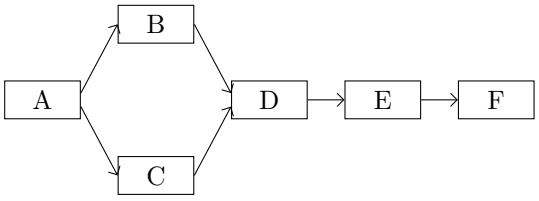


Figure 3.1: A simple dataflow program.

3.2 Extracting metrics from CAL programs

Efficient design space exploration heuristics need realistic metrics, characterizing the lower levels of abstraction in order to compute high level solutions that lead to efficient implementations. Having a high level representation of an application is useless if it cannot reflect the real underlying implementation. The profiling of the implementation code generated directly from high level descriptions is a viable way to get realistic metrics and to attach them to the CAL program.

Ideally, the profiling should occur at the algorithmic level and should not be altered by specific architectures on which the application is executed. A reliable algorithmic complexity analysis must be performed at the abstraction level of the source code, i.e. at the same abstraction level of verification models, which are conceived and developed for no specific target architecture but for algorithmic specification and validation purposes only. Any complexity analysis based on the modification of the compiled code inevitably takes into account the code transformations due to the compilation process (e.g., source-code to intermediate-format transformations, optimization transformations, target-architecture specific transformations) and strictly depends on the instruction-set of the target architecture. When a pure algorithmic complexity analysis at high-abstraction level is of concern, the results of a platform- or compilation-dependent analysis can yield misleading complexity evaluations because of the aforementioned transformations.

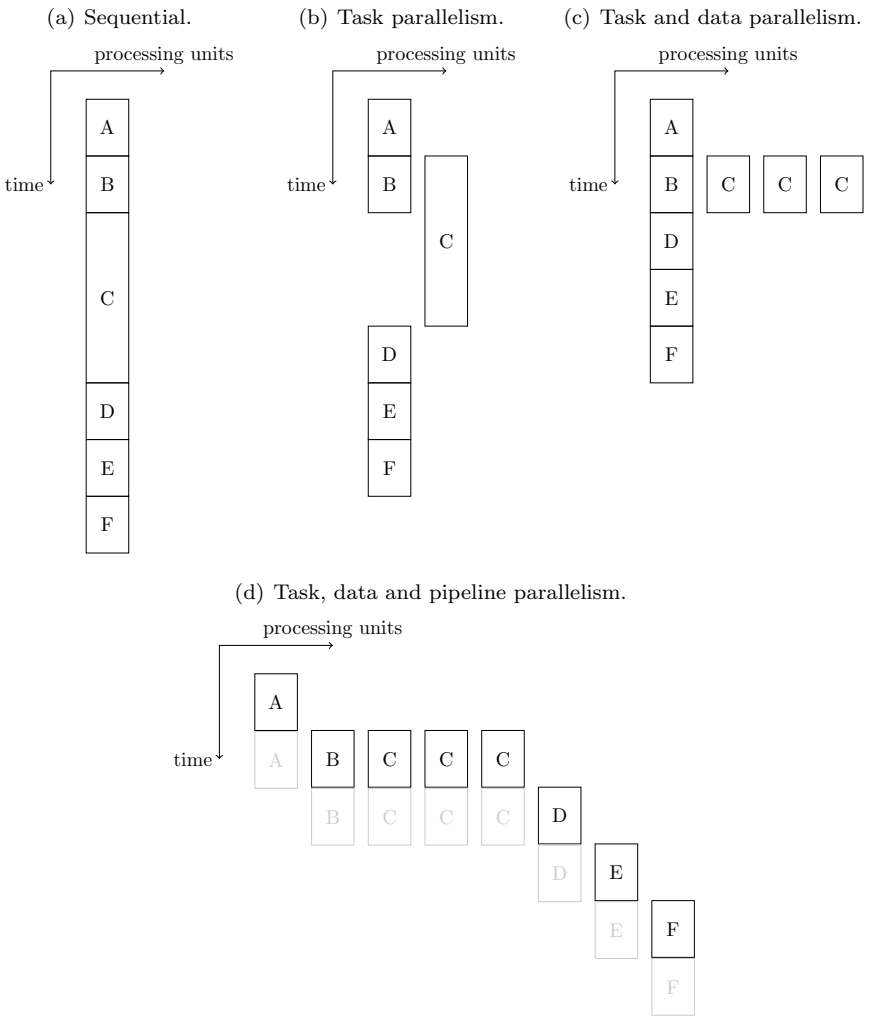


Figure 3.2: Parallelization of a simple dataflow program.

CAL programs can be easily analyzed in different ways: static code analysis (Section 3.2.1) and profiling (Section 3.2.2).

3.2.1 Static code analysis

Static code analysis consists in analyzing the source code without executing the program.

3.2.1.1 Nature of actors

As mentioned in Section 2.1.1, CAL is capable of describing applications in different models of computations (SDF,CSDF,DDF). Thus, according to the way actors fire their actions, several types of actors can be distinguished:

Static actors retain the ability to be scheduled at compile-time. They can be part of two subsets:

- SDF actors [113] include one or several actions with the same token rates are SDF.
- CSDF actors [114] include one or several actions that may have different token consumption and production. Theses actions are fired following a fixed cycle. For each cycle, the numbers of consumed and produced tokens are identical. They extend SDF actors in the sense that they are still static at the cycle level.

Dynamic actors [115] refer to the most general type of deterministic actors, i.e. a given input sequence of tokens always produces a unique output sequence. They can contain several actions with different token consumption and production. They can test the availability of the input tokens and can check their values to fire actions. Dynamic actors need to be scheduled at run-time because the value of the input tokens cannot be known *a priori* at the compile-time.

Time-dependent actors fire actions depending on the arrival time of the tokens. Thus, time-dependent firing sequences may produce non-deterministic output, since their effect might depend on the arrival time of inputs.

Several tools aim at classifying actors according to their MoC. A first tool has been implemented by Ericsson as a contribution to OpenDF [116]. This work has been integrated into the CAL STATIC ANALYZER (Section 3.5.1). One can also mention the classifier implemented within the ORCC framework (Section 4.4.1).

3.2.2 Profiling

Profiling is a kind of dynamic program analysis (as opposed to static code analysis). It is the investigation of the behavior of a program using information gathered as the program executes. Thus, the profiling of any program implies that the results are closely linked to the input data being used during the analysis. In order these results to be valuable, the input data must be a representative

set that correspond to a normal behavior of the program, otherwise the profiling results may not be meaningful.

The following sections present a non-exhaustive list of metrics that can be extracted from CAL programs.

3.2.2.1 Actions profiling

Having an estimation of the execution time of the actions is compulsory for computing efficient mapping of the CAL program on a given platform. This profiling can be performed at different levels:

CAL level, platform-agnostic At a high level of abstraction, the values extracted by this profiling are far from the real implementation and are not suitable for efficient optimization on a given platform. It can be used for pure algorithmic optimization.

Implementation level, platform-agnostic At a lower level of abstraction, one can profile the implementation code (C/C++, for instance) by not taking into account the architecture of the underlying platform. This profiling measures the computational load of each action in terms of C/C++ operators. PROFICAL (Section 3.5.3) provides this profiling by instrumenting the source code generated from the CAL program by the C++ backend of the ORCC code generator (Section 4.4.1).

Implementation level, platform-aware At the lowest level of abstraction, the profiling on the given platform is the most precise, because the real implementation code is analyzed. Thus, this profiling is platform-dependent. Two solutions have been envisaged:

- Record the total time spent by each of the action of the program, using the well-known Gprof [117].
- Measure the computational load of each action in terms of executed instructions. This can be done by instrumenting the binaries at runtime. This metric is extracted by the CAL DYNAMIC ANALYZER (Section 3.5.2).

3.2.2.2 Causation trace

CAL language allows writing dynamic programs, i.e. the sequence of actions firings depends on the input token values. In order to analyze this type of programs, one needs to record the sequence of actions firings during an execution of the program given input data. Janneck et al. [118] firstly introduces the causation trace to represent such a recording and defines it as follows:

Definition 1. *The **causation trace** of a dataflow program is a directed acyclic graph such that:*

- *every node is a firing of an action of an actor in the program,*

- *every edge from the node v_1 to the node v_2 is a dependency (either through a token, state or port) from v_2 on v_1, implying that therefore action v_1 has to be executed before action v_2.*

The causation trace is extracted by the CAL DYNAMIC ANALYZER (Section 3.5.2) or by PROFICAL (Section 3.5.3).

3.2.2.3 Trace critical path

By assigning to each node of the causation trace the corresponding execution time of the action that has been extracted (Section 3.2.2.1), one can obtain a weighted causation trace. Identifying the longest weighted path of this causation trace reveals very valuable information. The problem of identifying the longest path in a program is called a critical path problem [119]. The critical path profiling is a metric explicitly developed for parallel programs [120] that proved to be useful.

Indeed, extracting the critical path allows identifying the components in a parallel program that limits its performance. It is an effective metric for tuning parallel programs and is especially useful during the early stages of the application design. As long as the functions belonging to the critical path are optimized, the critical path may change completely, highlighting the new functions to optimize. Any delay in the execution of the actions belonging to the critical path impacts directly the completion date. It also helps to find out which components should be prioritized to complete the program execution in time. When a task has to be completed in a given time, the critical path analysis helps to focus on the essential activities to which attention and resources should be devoted. The extraction of the critical path at the algorithmic level of the execution of C/C++ programs has been already studied by Christophe Clerc [1]. In this research work, this notion of critical path has been applied to dataflow programs and thus can be defined as follows:

Definition 2. *The critical path of a dataflow program execution is simply defined as the longest weighted sequence of events from the start of the program to its termination. The parallel computations being described by the causation trace, the* ***trace critical path*** *is the longest* ***weighted*** *path from the source to the sink node of the causation trace.*

The trace critical path (illustrated in Figure 3.3) is the longest **weighted** path because considering only the longest path - i.e. considering unit weight for all the actions of the causation trace - is not relevant because it does not allow for real impact of actions onto the performance. By considering unit weights for all actions, it means that complex and simple actions have the same impact and thus same execution time. This is obviously not the case. By weighting the actions with their respective running time (extracted thanks to profiling tools), it guarantees that the trace critical path will include the actions that impact the most the completion date of the execution.

The CROSSCAL tool (Section 3.5.4) extracts the trace critical path of the execution of a program from its causation trace. As nodes in the causation trace

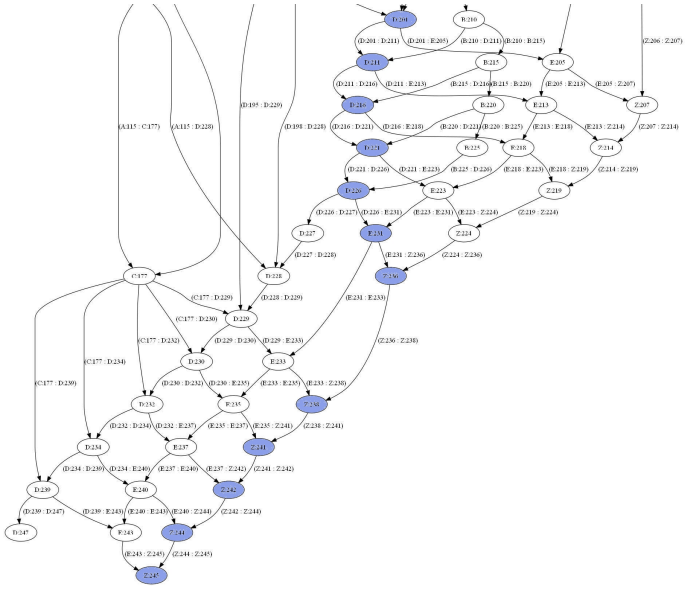


Figure 3.3: Illustration of the trace critical path of the execution of a program represented as a Directed Acyclic Graph (DAG).

are actions firing, the trace critical path is at the action level. This is coherent with the atomic nature of the actions in CAL. The principle of Algorithm 1 which extracts the trace critical path from the causation trace is:

1. Label each node with its respective weight, given by the profiling.
2. Traverse the causation trace from the source to the sink and computing of the distance of each node from the source using the weights. Record of the precedent node used to compute the distance of the current node.
3. Traverse the trace from the sink to the source to recover the trace critical path by following the precedent node of each current node.

3.2.2.4 Data transfers

Actors communicate between each others thanks to tokens through connections. Knowing the amount of data flowing between actors through these connections is useful for estimating the necessary bandwidth between the different partitions of the implemented system. This metric also guides the partitioning of the CAL program, trying to minimize the communications between actors because the exchange of tokens implies communications overhead which is not negligible at the implementation level.

Knowing the number of execution of each action and the size in bits of each of the ports, it is easy to compute the data transfers between actors thanks to Algorithm 2. Data transfers are extracted by the CROSSCAL tool (Section 3.5.4).

Algorithm 1 Extraction of the trace critical path from the causation trace

Let G be the causation trace (G) of the execution ;
Let $W(v)$ be the weight of node v ;
Let cp be the ordered list of the nodes belonging to the trace critical path ;
Let $lengths$ be the array with $|V(G)|$ elements of type int with value 0 ;

for vertex v in $topologicalOrder(G)$ **do**
  for edge (v, w) in $Edge(G)$ **do**
    if $lengths[w] \leq length[v] + W(w)$ **then**
      $lengths[w] = lengths[v] + W(w)$;
      $pred(w) = v$;
    end if
  end for
end for

$u = sink(G)$;
while $u \neq source$ **do**
  $add(u)$ to cp ;
  $u = pred(u)$;
end while

Algorithm 2 Profiling the data transfers

Let $N(a)$ be the number of execution of action a ;
Let $Port(a)$ be the output ports of action a ;
Let $Size(p)$ be the size in bits of port p ;
Let $C(p)$ be the connection linked to the port p ;
Let $DT(c)$ = amount of data flowing through connection c ;

for action a in the CAL program **do**
  for port p in $Port(a)$ **do**
    $bits = N(a) \times Size(p)$;
    $DT(C(p)) += bits$;
  end for
end for

3.3 Optimization strategy: trace critical path minimization

In large and complex applications, the main problem remains in finding which functions need to be optimized in order to have the best speed-up for a minimum effort. The trace critical path is a very useful metric in case of parallel programs because it indicates the functions that are critical and really affect the completion date of the execution, as discussed in Section 3.2.2.3. Because any change in the execution time of these actions belonging to the critical path impacts directly the completion date of the execution, these actions are the best starting points for the optimization.

The optimization consists in reducing the contribution of these actions to the critical path so that their reduction of their execution times reduce the general duration of the execution. By doing so for the most critical actions of the critical path, the designer can significantly optimize the CAL program. More generally, the optimization consists in minimizing the trace critical path of the execution.

Section 3.3.1 presents how to compute the contribution of an action to the critical path and defines the most critical action, the action with the largest contribution to the critical path. Section 3.3.2 presents an algorithm for predicting the successive most critical actions during the optimization process. Section 3.3.3 proposes several techniques for helping the designer in the refactoring of the CAL program for getting rid of the detected bottlenecks of the program.

3.3.1 Definition of the most critical action

The trace critical path is the list of actions which impact the most the completion date of the execution. One can sort these actions according to their contribution to the trace critical path. The sorting can be based on the number of occurrences of each actions. But it is not because an action appears the highest number of times that it contributes the most to the trace critical path. The execution time of these actions must be taken into account to decide which actions are the largest contributors. The action having the largest contribution to the trace critical path is the one cumulating the largest computations in the trace critical path and thus is a good starting point for the optimization.

Definition 3. *The **most critical action** is the one which has the largest contribution to the trace critical path in terms of computational load for a given execution. Let $Contrib(action)$, the contribution of an action to the trace critical path, the most critical action can be defined in Equation 3.3:*

$$MostCriticalAction = \max_{a \in Actions} \{Contrib(a)\} \tag{3.3}$$

*Let a an action belonging to the trace critical path, $CL(a)$ its computational load and CP the set of all the actions of the trace critical path, the **contribution***

of an action *a is defined by Equation 3.4:*

$$Contrib(a) = |a \cap CP| \times CL(a) \tag{3.4}$$

At the actor level, Equation 3.5 defines the ***contribution of an actor*** *A, which is the ratio between the sum of the contribution to the trace critical path of its actions and the total computational load of the actor.*

$$Contrib(A) = \frac{\sum_{a \in A}\{Contrib(a)\}}{CL(A)} \tag{3.5}$$

3.3.2 Critical Actions Detection algorithm

Being able to identify systematically the critical actions of a CAL program for a given execution makes possible the estimation of the necessary optimization of the action so that it is no longer critical. This information indicates to the designer how much the action needs to be optimized. But as long as the actions belonging to the trace critical path are optimized, the trace critical path may change completely, highlighting the new critical actions. By computing the trace critical path iteratively with decreasing values for the computational load of the current most critical action until it is no longer in the trace critical path, one can extract the list of the N next critical actions and their necessary optimization. Algorithm 3 provides the list of the critical actions and is implemented in the CROSSCAL tool (Section 3.5.4).

Algorithm 3 Critical Actions Detection algorithm

Let B_n be the most critical action at iteration n ;
Let $CL(A)$ be the current value of the computational load of action A ;
Let $CL_i(A)$ be the initial value of the computational load of action A ;
Let k be the optimization step (in %) ;
Let OPT be the number of minimum optimization (in %) ;

$CP_i = CriticalPath()$; {Value of the initial trace critical path}
$B_0 = MostCriticalAction()$;
$CL(B_0) = k \times CL(B_0)$;
$n = m = 0$;
while $(v < OPT$ **or** $m \neq N)$ **do**
  $n++$;
  $B_n = MostCriticalAction()$;
  if $(B_n \neq B_{n-1})$ **then**
    Add action B_{n-1} to the list with opt. $= 100 \times (1 - \frac{CL(B_{n-1})}{CL_i(B_{n-1})})$ % ;
    $m++$;
    $v = 100 \times (1 - \frac{CriticalPath()}{CP_i})$;
  end if
  $CL(B_n) = k \times CL(B_n)$;
end while

3.3.3 Refactoring techniques

After having identified a set of actions as critical, there are several techniques aiming at refactoring the dataflow program in order to get rid of these bottlenecks. The aim is to reduce the contributions of the most critical actions so that they do not belong to the critical path any more, or they are at least not critical.

The Critical Actions Detection algorithm (Section 3.3.2) extracts the N most critical actions which are the main bottlenecks of the program. The programmer needs to focus on these particular actions in order to improve the performances of the whole system. Thanks to the advantageous modularity features of CAL programming, designers can work concurrently on the refactoring of the different bottleneck actors. As long as the behavior of actors in terms of input and output tokens requirements is respected, the newly refactored actors can be interchanged seamlessly with the old ones.

Equation 3.4 defined the contribution of an action to the critical path. There are mainly two ways for reducing the contribution of an action:

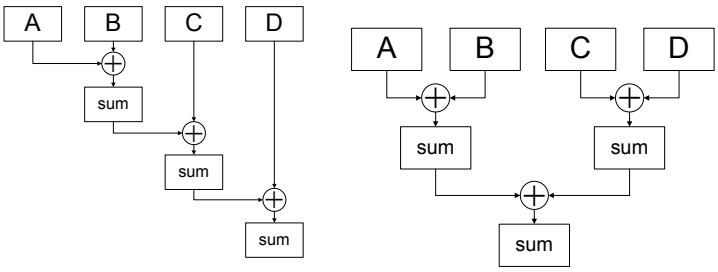
By minimizing $|a \cap CP|$ by breaking the sequentiality The programmer may have introduced unnecessary sequentiality in a critical action whose impact on the performances is important. Being embodied in a critical action, this sequentiality slows down the execution of the program. Breaking this sequentiality by splitting the most critical actors helps in minimizing the global critical path (Section 3.3.3.1).

By minimizing the computational load $CL(a)$ of the action It consists in rewriting the actor/action in a more efficient way so that the functionality of the action is fully kept but by using less computing resources (Section 3.3.3.2).

3.3.3.1 Breaking unnecessary sequentiality

Every application contains an incompressible and inherent sequentiality. Nevertheless, some unnecessary dependencies may have been introduced by the developer while writing the application. When writing parallel programs, it is important to remove the unnecessary dependencies. For instance, the sum of four numbers a, b, c and d can be implemented in two different ways. In the sequential implementation (Figure 3.4(a)), each number is added one after the other, creating useless dependencies between each step. In the parallel implementation (Figure 3.4(b), intermediate sums (`Sum1` and `Sum2`) can be calculated in parallel.

The dependencies introduced by the programmer while writing the application hide the task and data parallelisms (see Section 3.1.2) which can be exploited by the platform. Thus, these dependencies may be a real bottleneck in the parallelization of the application because they cannot be guessed by compilers even if this field of research has been active for decades. Removing it directly at the application level is a guarantee of producing solutions which can really fit to parallel platforms.



(a) Sequential implementation. (b) Parallel implementation.

Figure 3.4: Different implementations of the sum of four numbers.

3.3.3.2 Minimization of the computational load

3.3.3.2.1 Action and actor rewriting The first idea is to rewrite the action that has been detected as the most critical. It consists in rewriting the action in a more efficient way, using less computation-intensive processes to perform the same functionality. But it is not always possible to do so and the refactoring of the whole actor is necessary. The complete refactoring of the Finite State Machine of the actor is sometimes compulsory.

3.3.3.2.2 Merging actors The refactoring of the actor only may be not enough for the optimization of the critical part of the program. For example, in order to improve the ratio between computations and communication (because communications have a non-negligible cost), it may be necessary to merge connected actors in order to reduce the overhead due to the communication at the price of less exposed parallelism.

3.3.3.2.3 Reducing implementation overhead CAL allows to specify applications at a high level of abstraction, it is not an implementation language. CAL programs cannot be executed directly on target platforms. Thus, they need to be converted in the respective native language of the platforms, i.e. VHDL/Verilog for hardware platforms and C/C++ for software platforms.

The conversion of a high level language as CAL into low-level software languages reveals necessary overheads that must be considered, both in hardware and software. Additional macros need to be inserted in order the manage low-level details that are not explicitly specified into CAL programs. This extra cost is called the *implementation overhead*. The reader is referred to Section 4.4 for further details on code generation.

In **multi-processing-units software systems**, communication channels between actors are turned into real FIFO implementations which may introduce additional synchronization protocols in case of concurrent memory accesses. The implementation of actors introduces a controller (namely action scheduler) which is responsible for deciding which action must be fired next according to

the state of the actor, the action guards, the token availability at the input and the status of the FIFO. In case several actions can be fired, the action scheduler needs to choose only one. Thus, a well-defined scheduling policy needs to be defined and implemented.

In **hardware systems** (FPGA, ASIC), the level of available parallelism is much higher than in software systems, allowing each actor to run in parallel as in CAL programs. Thus, each actor implemented in HDL on the target can run concurrently with each others. The system can be self-scheduled, driven by the production of tokens and the concurrent work of the actors. Additionally, protocols need to be implemented for managing the accesses to the FIFOs when consuming or producing tokens. At the actor level, an action scheduler is necessary in order to decide which action must be fired next. These controllers introduce overheads in terms of processing time and area.

Developers can design systems at a high level of abstraction (in CAL) without worrying about low-level implementation details but it is not a reason for ignoring the underlying mechanisms at the implementation levels. Brilliant C/C++ developers are aware of how the code is turned into assembler after compilation. The same idea can be applied concerning CAL, designers need to be aware of the overhead due to the implementation: whatever the target is, any execution of an action implies four steps:

1. **The scheduling** determines:
   (a) at the actor level, which action to fire next,
   (b) at the processing unit level, which actor to fire next.
2. **Input memory accesses**: the action consumes token(s) at the input.
3. **Execution of the body**: computations of the application.
4. **Output memory accesses**: the action produces token(s) at the output.

Obviously, applying these transformations may become useless if the implementation overhead is reduced in the future versions of the code generators. Section 3.3.3.2.3 and Section 3.3.3.2.3 expose respectively the different refactoring techniques aiming at reducing the implementation overhead due to the scheduling and the memory accesses.

Minimizing scheduling overhead In CAL, at the actor level, these are different mechanisms for scheduling the actions. **The Finite State Machine (FSM)** - if any - is a behavior model composed of a finite number of states and transitions between them (i.e. the actions). FSM are similar to a flow graph in which one can control the actions firing when certain conditions are met (token availability, internal state of the actor, guards). The **guards** are conditions on the value of internal variables that need to be satisfied for an action to be fired. **Priorities** allow defining which action to fire in case several actions are fireable.

All these mechanisms are costly in terms of processing time at the implementation level because checks (guards, input tokens, priorities) need to be

performed before firing any action. This is the reason why it is recommended to remove dynamic structures whenever is possible. Obviously, dynamic actors like the parser need these constructs to be functional.

Scheduling-dominated actors need to be refactored in order to minimize the workload of the scheduling functions. There are different techniques to minimize this scheduling overhead:

Reduce (or remove) Finite State Machine One can merge actions or duplicate a given action into several others, make use of priorities, etc. This will reduce the code needed to schedule actions.

Remove guards implies checking less conditions at each test of action firing.

Turn the actor static By preferring static actors, the generated code will be more efficient because the scheduling functions will be minimized. If the runtime has a static scheduling policy, the overhead can even be null if the actor is part of a sequence of static sequence of actors. The Cal Static Analyzer provides the information on the type of actor (static, dynamic, time-dependent) and is a first indication of which actor to focus on.

Minimizing memory accesses to FIFO Too many memory accesses is a problem only when the granularity of the actor is too low, i.e. there is too much communications compared to computations. One possible solution is too increase this ratio, which corresponds in fact to merge this actor with its neighbors in order to avoid communications.

The amount of data exchange is not very important while it is under a given threshold. Above this threshold, the amount of data begins to have its importance, but below, the most important thing is that there is a communication, the cost of locking the FIFO, reading writing and accessing the FIFO is dominant. Then, sending one or one hundred tokens for each firing does not make a difference. Section 4.3.1.1 shows that below one thousand, the size of the data transfers does not influence the total execution time.

Once the amount of data exceeds the threshold, one solution could be to merge the tokens into larger ones. The efficiency of the transfer is improved. But one must be careful of the modularity of the design if tokens are too much conglomerated.

3.4 Discussion

In order to optimize the execution of a Cal program, one needs to minimize the overhead due to the scheduling and memory accesses so that the real computations of the application have a larger part in the whole execution. One can define the efficiency of an actor A as being the ratio between the computations of this actor and the overhead due to the scheduling and the communications

of this actor as shown in Equation 3.6:

$$Efficiency(A) = \frac{ComputationalLoad(A)}{SchedOverhead(A) + CommOverhead(A)} \tag{3.6}$$

In order to increase the efficiency of an actor, either the scheduling and/or the communications must be reduced or actions must be more computational-intensive. Consequently, a CAL program composed of a single actor would have the best efficiency. But the parallelism of the program is no longer exposed and thus cannot be exploited by the platform, loosing the potential achievable speed-up by the parallel execution of the program.

Thus, a trade-off must be found in order to achieve the best performance. On one side, the efficiency of the different actors must be maximum and on the other side, parallelism must be exposed so that it can be exploited by the platforms. Figure 3.5 illustrates the trade-off to be found out.

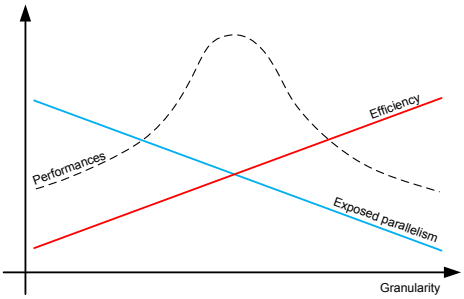


Figure 3.5: Finding the right trade-off between the efficiency of actors and the exposed parallelism.

If the granularity of the actors is too low, the efficiency of these actors will be too low and will lead to poor performance at implementation because of the overheads. If the granularity is too high, the actors are too large and the achievable speed-up by the platform will be low. But, finding the good trade-off in which the efficiency of the actors are maximized while exposing enough parallelism is the key point for designing efficient CAL programs.

The technique aiming at removing the unnecessary dependencies increases the exposed parallelism while removing the implementation overhead at the CAL level and shortening the trace critical path aims at improving the efficiency of the actors.

The exposed parallelism of the application should be at the same level as the one exploitable by the platform. Figure 3.6 illustrates the concept. If too much parallelism is exposed in the application, mapping it on a sequential processor implies a large scheduling overhead. On the contrary, if too little parallelism is exposed in the application, mapping it on a highly parallel platform will be inefficient. However, there exists some techniques (such as static scheduling)

that allow to keep the exposed parallelism at the CAL level and to reduce the overhead at the implementation (see Section 4.2.4).

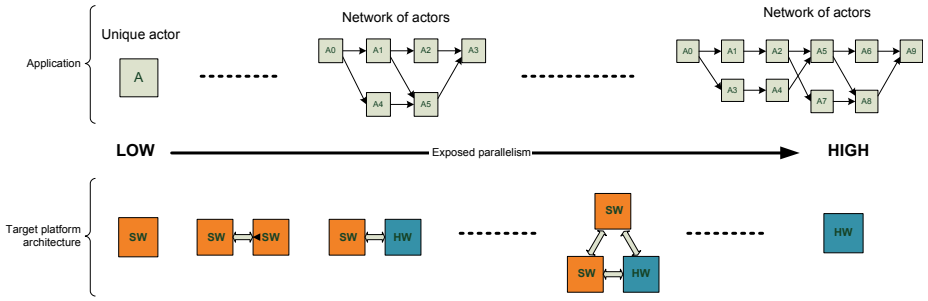


Figure 3.6: The level of exposed parallelism should be coherent with the degree of parallelism of the platform.

3.5 Tools

3.5.1 CAL Static Analyzer

This tool performs one static analysis of CAL programs:

The actors classification classifies the actors in three categories by analyzing the internal behavior of each actor taken separately:

- *Static* if a (possibly periodic) static sequence of action firings can be determined,
- *Dynamic* if the sequence of action firings is dependent on the (values of) inputs that an actor receives, but not the arrival time of the inputs,
- *Time-dependent* otherwise.

Additionally, this tool gathers some structural information of the CAL program on actors, actions, connections between actors, ports size and token rates. The aim is to pre-process the CAL program in order to gather these structural information in a single file, avoiding the other tools to parse each time the same CAL source files. For each actor instance, the following information is gathered into a single output file:

- Actors instances and actions are identified with a unique hierarchical identifier:
  - Actors : */top/network1/network2/instancenameofactor*
  - Actions : *uniqueID[$tag]* (the *uniqueID* are in lexical order, starting from 0)

- The list of ports and their size in bits.
- The list of actions with their respective tokens consumption and production on the ports.
- The connections between actors.

The Cal Static Analyzer is written in Java using the Eclipse RCP framework. This tool is exportable as an executable on different operating systems: Windows, Linux and MacOS. Figure 3.7 is a screenshot of the tool.

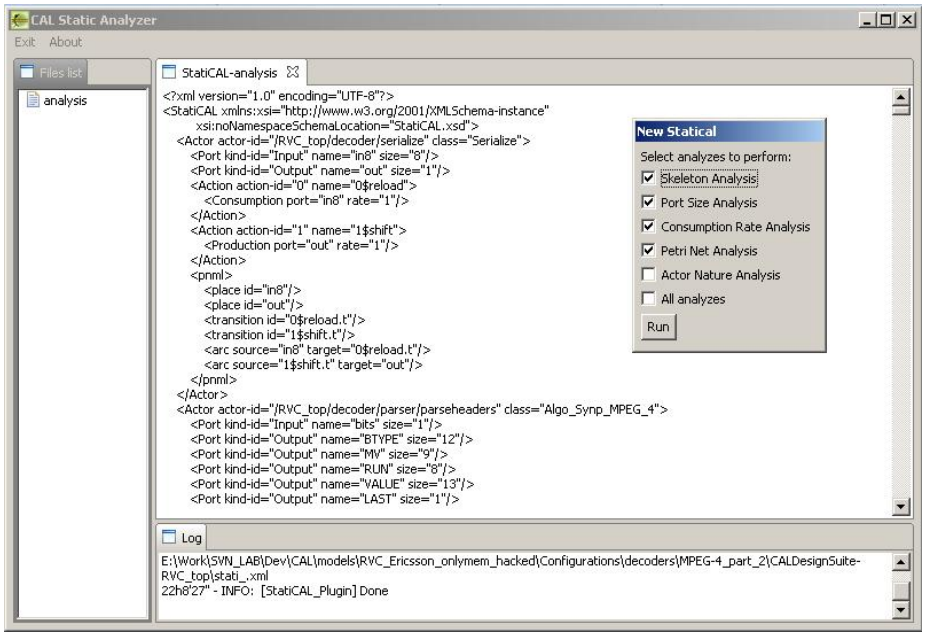


Figure 3.7: Graphical User Interface of the Cal Static Analyzer

3.5.1.1 Inputs

The **input** is the Cal source code.

3.5.1.2 Outputs

The **output** is a single XML file containing the results of the actor classification and the pre-processing of the source code.

3.5.2 Cal Dynamic Analyzer

Cal Dynamic Analyzer is a Pintool written in C++ for Pin [47], a framework for dynamic instrumentation of binaries. Pin was designed to provide the means to inject code dynamically at different places in the binary file and to record

the information while running the executable This tool can be also used under different operating systems: Linux, Windows and MacOS.

3.5.2.1 Inputs

The binary of the application compiled from the generated code produced by the C++ back-end of the ORCC framework (Section 4.4.1).

The names correspondence between the names of the functions in the generated C++ code and the names of the CAL actions in order to know which functions to profile.

3.5.2.2 Outputs

The profiling of the actions For each action of the program, it outputs its number of execution, its total number of computations and its average computational load in terms of instructions. It outputs the results in a single XML file.

The causation trace as defined is Definition 3.2.2.2. One XML file per actor instance is created and lists the firing of its actions in the chronological order. This format of the trace does not contain the dependencies between the actors but can be retrieved thanks to the connections contained in the output file of the CAL STATIC ANALYZER. The choice of this format has been motivated by size-related issues. Keeping the dependencies into the XML file produces huge causation traces which were tricky to handle.

3.5.3 PROFICAL

The PROFICAL tool has the same features as the CAL DYNAMIC ANALYZER but profiles actions at the C/C++ operator level of a CAL application instead of the instruction level. The instrumentation of the source code is made thanks to the Software Instrumentation Tool (SIT) [42].

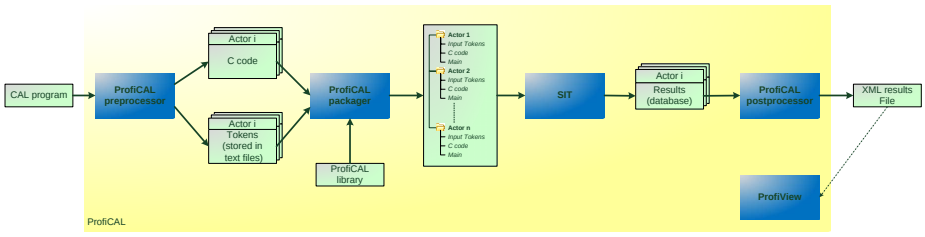


Figure 3.8: Architecture of the PROFICAL tool.

The CAL application is converted in C++ code, using the C++ backend of ORCC. The generated code is executed, and the tokens are recorded during the execution. The whole is input to the SIT which outputs the results in form of a

small database. The profiling of the actions are extracted in terms of C/C++ operators.

3.5.3.1 Input

The **input** is the CAL program and its input data.

3.5.3.2 Outputs

The profiling of the actions For each action of the program, it outputs its number of execution, its total number of computations and its average computational load in terms of C/C++ operators. It outputs the results in a single XML file.

The causation trace is also extracted, the same as in the CAL DYNAMIC ANALYZER.

3.5.4 CROSSCAL

The CROSSCAL tool aims at providing an environment for easily crossing the results of the different metrics and profiling from the other tools. It results in advanced metrics that are very valuable for the design of CAL programs. CROSSCAL performs three analyzes:

Critical Actions Detection algorithm consists in detecting and listing the potential critical actions of a CAL program executed with given input data (Section 3.3.2).

The extraction of the trace critical path from the causation trace (Section 3.2.2.3). The weights of the nodes (i.e. actions) are computed by the WEIGHTCAL tool (Section 3.5.5).

The profiling of the data transfers outputs the size of the data exchanges between actors. For each connection, the contribution of each action to the total amount of data is also specified (Section 3.2.2.4).

The tool is written in Java using the Eclipse RCP framework. This tool is exportable as an executable on different operating systems: Windows, Linux and MacOS. A screenshot of the tool is shown in Figure 3.9.

3.5.4.1 Inputs

The results of the static analyzes output by the CAL STATIC ANALYZER.

The causation trace provided by the CAL DYNAMIC ANALYZER.

The profiling of the actions provided by PROFICAL or the CAL DYNAMIC ANALYZER.

The execution time of actions provided by WEIGHTCAL.

3.5.4.2 Output

The **output** is a single XML file containing the results of the different analyzes.

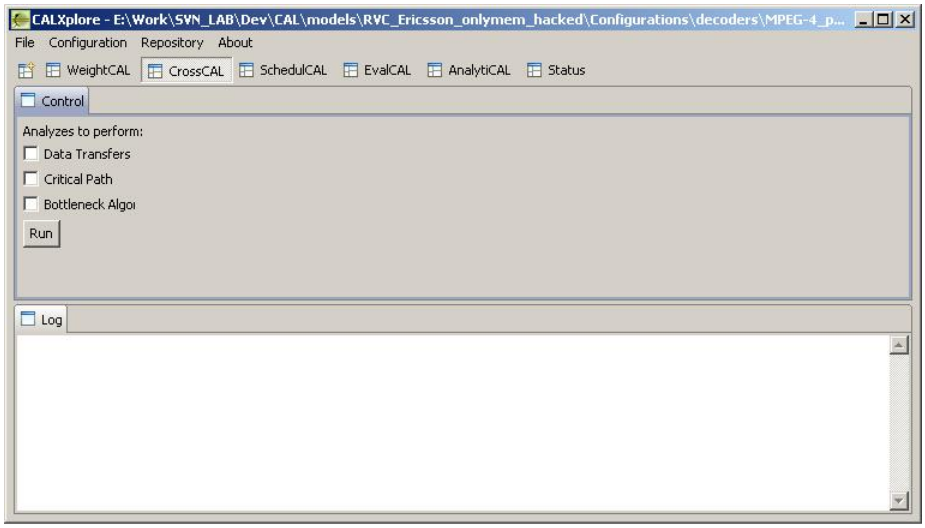


Figure 3.9: Graphical User Interface of the CrossCal tool

3.5.5 WeightCAL

WeightCAL is responsible for setting the execution time of the actions given the different profiling available. Three schemes are possible:

By considering only action bodies either in terms of C/C++ operators (provided by ProfiCal) or in terms of instructions (provided by the Cal Dynamic Analyzer).

By considering body of actions and scheduling overhead Additionally to the profiling, a part of the scheduling overhead is estimated and added to the execution time.

From Gprof The executable of the application compiled with ORCC can be profiled with Gprof [117] which outputs the results in a `gmon.out` file. This file must be converted into a CSV file to be compatible with WeightCAL.

The tool is written in Java using the Eclipse RCP framework. This tool is exportable as an executable on different operating systems: Windows, Linux and MacOS. A screenshot of the tool is shown in Figure 3.10.

3.5.5.1 Inputs

The profiling of the actions provided by:

- Cal Dynamic Analyzer for a profiling in terms of instructions,
- ProfiCal for a profiling in terms of C/C++ operators,
- Gprof for a profiling in terms of real execution time.

The partial profiling of the scheduling overhead provided by the Cal Dynamic Analyzer, ProfiCal or Gprof.

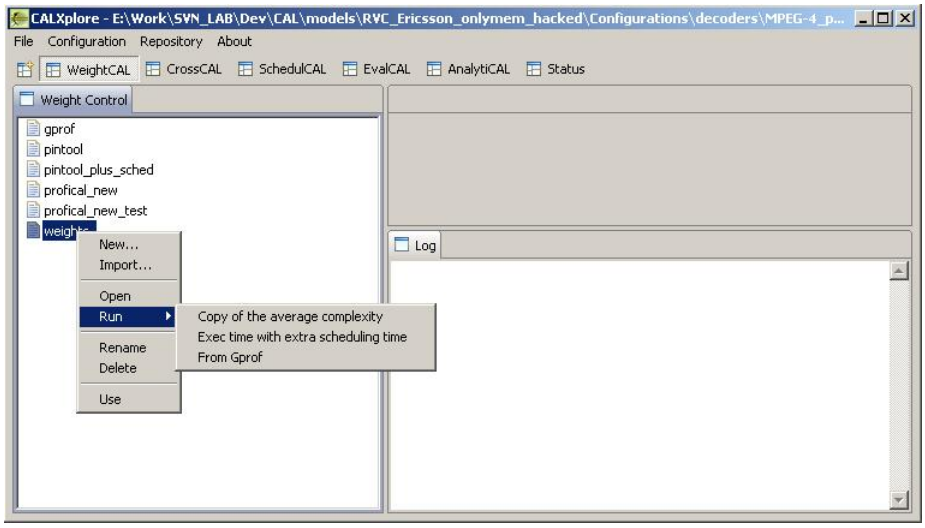


Figure 3.10: Graphical user interface of the WeightCAL tool

3.5.5.2 Output

The **output** is a single XML file containing the execution times of all the actions of the program.

Chapter 4

Mapping dataflow programs onto platforms

The *mapping* is a one-way process consisting in fitting the CAL program onto a given platform. The problem is twofold: (1) how to distribute the actors of the program on the different processing units of the platform. i.e. find the partitions that lead to an efficient implementation and (2) how to convert the high level CAL program into a program which can be executable on the given platform? Answering these questions implies to investigate on several points:

The partitioning/scheduling problem Partitioning consists in assigning actors to processing units and scheduling in determining the order of execution of the actions on each processing unit. The aim is to find the optimal partitions and schedules that maximize the performance. This combinatorial problem is NP-complete, making compulsory the elaboration of heuristics. In order to obtain valuable results, the CAL program needs to be profiled and realistic data needs to be extracted and used as input to the problem. The tools presented in Section 3.2.2 aims at providing these metrics. Section 4.1 introduces the problem and the different approaches for solving it are presented in Section 4.2.

The performance evaluation Partitioning heuristics may need to estimate its current solution in order to be able to iterate and search for a new solution that improves this current solution, based on this estimation. Thus, the estimation of the performance serves as a guide towards the solution, implying that a special attention needs to be paid on this issue in order not to guide the heuristics towards inefficient solutions. Being able to model the behavior of the CAL program at the implementation level is a challenging task. The proposed approach for the evaluation of a solution is exposed in Section 4.3.

The code generation Having an appropriate representation at high level of the application is a nice feature but is useless if there is no way to make profit of it. This is the reason why software and hardware code generators are being developed in order to translate high level CAL programs into efficient implementation. The code generation issue is discussed in Section 4.4.

4.1 The partitioning/scheduling problem: the case of Synchronous Dataflow

This section aims at introducing the partitioning/scheduling problem by studying the case of Synchronous Dataflow (SDF) before tackling the problem with CAL programs. Synchronous Data Flow (SDF) [113] is a restricted dataflow model that is well suited for static analysis. In SDF models, the tokens production and consumption rates are fixed *a priori*. In other words, an actor consumes and produces always the same amount of tokens at each actor firing. This restriction offers a higher degree of analyzability at the price of lower expressiveness. Thus, it is possible to schedule the SDF model – ordering the actions firings – and to determine the buffer requirements at compile-time. SDF models are well-adapted to a wide range of applications especially in the domain of critical real-time systems.

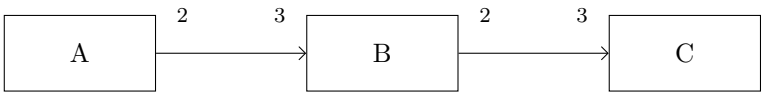


Figure 4.1: Example of a simple Synchronous Dataflow (SDF) model.

A simple SDF model is illustrated in Figure 4.1 (Source: [112]). The firing of actor A produces two tokens on its output port. The firing of actor B consumes three tokens and produces two tokens. The firing of actor C consumes three tokens. This representation reveals the task parallelism but hides the data parallelism. Another representation in form of a Directed Acyclic Graph (DAG) is used to expose both parallelisms. This DAG is also referred as the *task graph* in the literature. Each node of the DAG is weighted with the computational load of the actor and each edge with the communication cost. The corresponding DAG of the simple dataflow program of Figure 4.1 is illustrated in Figure 4.2 (Source: [112]). As the DAG reveals the data dependencies, one can notice that B_1 may be fired in parallel with A_3.

4.1.1 Analyzing Synchronous Dataflow models

As a counterpart of its limited expressiveness, SDF are highly analyzable. Checking the consistency of the model and finding for the repetition vectors are example of analyses.

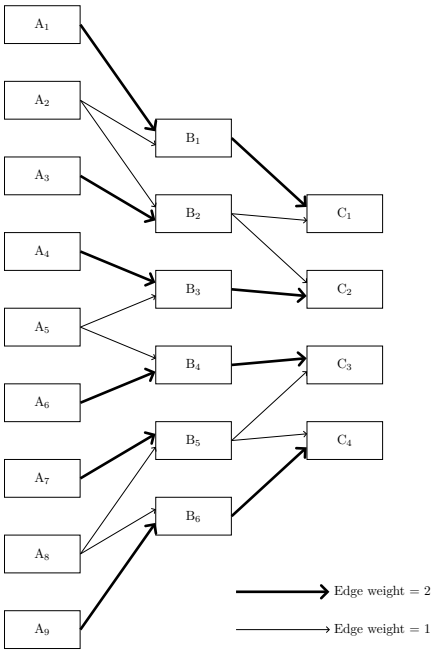


Figure 4.2: Associated Directed Acyclic Graph (DAG) of the Synchronous Dataflow model.

Checking the consistency consists in verifying that the infinite execution of the SDF model does not lead to unbounded memory requirements. A SDF model is consistent if the rank of its topology matrix equals the number of actors reduced by one. The topology matrix is the weighted incidence matrix where the weights correspond to the token production and consumption rates. The following equation (Source: [112]) checks the consistency of the simple SDF model of Figure 4.1.

$$\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ 0 & 2 & -3 \end{bmatrix} \text{ and } \text{rank}(\Gamma) = 2 = \text{card}(\{A, B, C\}) - 1$$

After having checked the consistency of the SDF model, one can compute its **repetition vectors**. It consists in determining the number of firing of each actor so that the program executes without deadlocks and with bounded buffer requirements. The repetition vector $\mathbf{q}$ is the minimal integer solution of $\Gamma \times \mathbf{q} = \mathbf{0}$. The repetition vectors of the simple SDF model indicates that there are 9 occurrences of actor A, 6 of B and 4 of C, according to the following equation (Source: [112]):

$$\begin{bmatrix} 2 & -3 & 0 \\ 0 & 2 & -3 \end{bmatrix} \times \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ that leads to } \mathbf{q} = \begin{bmatrix} 9 \\ 6 \\ 4 \end{bmatrix}$$

4.1.2 Partitioning and scheduling for multiprocessors platforms

Partitioning and scheduling a SDF model in case of multiprocessors platforms consists in finding a partition of actors (i.e. determining the allocation of actors onto processors) and a schedule of actions onto each processing unit (how the actions firing are ordered). This problem is NP-complete.

List scheduling [121] is the most common approach to solve this problem. The principle is to traverse at compile-time the DAG of the SDF model using different graph traversals (topological, breadth-first, etc.) and to assign priorities to actions using different strategies (ASAP start-time, longest processing time, critical path, etc.). At runtime, the SDF model is scheduled according to the assigned priorities. For each strategy, a well-defined criterion is optimized (e.g. throughput, latency). The most common criteria in signal processing systems are: minimization of the latency, maximization of the throughput or minimization of the resource usage for a given throughput/latency.

The maximization of the throughput of the simple SDF model leads to the Gantt chart presented in Figure 4.3 (Source: [112]). Communication cost are considered to be null and an unlimited number of processing units is considered. Obviously, it is never the case on real-world platforms.

Usually, the nodes of the DAG are weighted with a computation cost (i.e. the execution time of the actor) and the edges are weighted with a communication cost which reflects the required time for performing the exchange of tokens between two actors. Given a SDF model composed of two actors A and B.

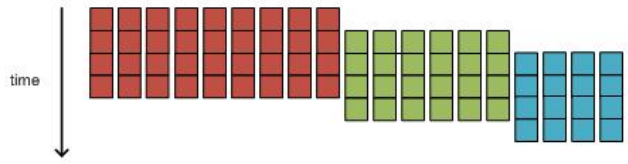


Figure 4.3: The ideal schedule to maximize the throughput.

The computation cost of each actor is equal to 2 for both processors P_1 and P_2. The communication cost is either equal to 1 if actors communicate through two different processors or is null if they lie on the same processor. Considering different optimization criteria may lead to different partitioning of the actors as shown in Figure 4.4 (Source: [112]). The first partitioning minimizes the makespan (the first iteration of the execution) ($4 < 5$), while the second maximizes the throughput ($1/3 > 1/4$).

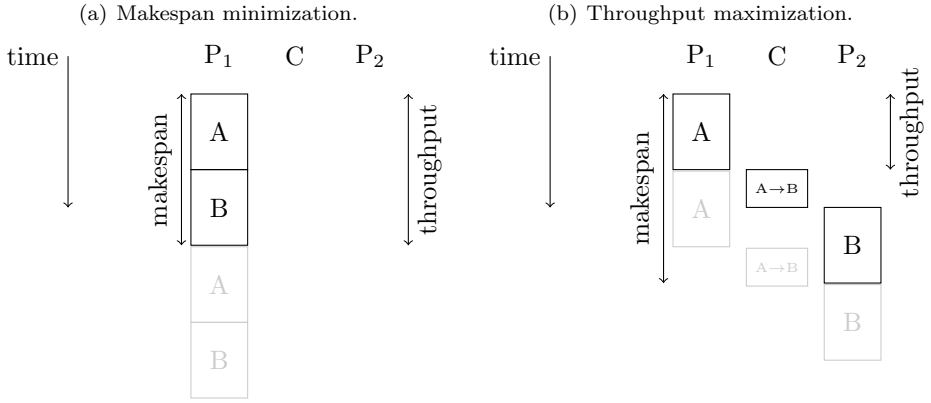


Figure 4.4: Considering different optimization criteria may lead to different partitioning of the actors onto the processors.

4.2 Partitioning and scheduling CAL programs

The expressiveness of the CAL language is higher than the one of the SDF formalism, allowing it to describe applications in several models of computations, as already discussed in Section 2.1.1. As a counter part, it is more difficult to generate efficient implementations. This potential combination of several models of computations make the partitioning and the scheduling of CAL programs harder. The principle remains the same - assigning actors to processing units and ordering the firing of actions on each processing unit - but the solution is

much complex because of the mix of static and dynamic actors. The positive point is that CAL is well-defined and formalized, which helps analyzing it.

The problem is separated into the two phases of assigning actors to processors and then sequencing the actions. The aim is to find such partition and schedule that lead to an efficient implementation. The solution space, being very complex, is split into two orthogonal spaces: the permutation space of the actors on the processors and the space specifying precedence among actions. The only argument for this leap stems from the fact that usually the number of Actors is very small in comparison to the number of nodes. Thus, a search on the partition of Actors on processors can be considered as sufficiently comprehensive to examine all possibly efficient partitions.

Several approaches will be examined:

1. The load balancing is a technique aiming at distributing the computations of the actors across processing units. A first round-robin approach is presented in Section 4.2.1.
2. The load balancing technique is kept in this second approach described in Section 4.2.2, but a simulated annealing approach taking into account communications costs is adopted.
3. Minimization of the makespan (completion date of the execution) based on the causation trace using a simulated annealing approach is described in Section 4.2.3.
4. An alternative approach, aiming at considering the non-negligible scheduling overhead, is exposed in Section 4.2.4. It consists in extracting static regions at CAL level and in computing their schedule at compile-time. At runtime, these static regions are detected, triggered and executed according to the schedule computed at compile-time. This technique aims at reducing the overhead at the implementation level.

4.2.1 Round-robin load balancing

The main idea of this first partitioning / scheduling heuristic is to load balance the total computations of the actors on the different processing units in a round-robin way. The computational load of each actor is provided by the CAL DYNAMIC ANALYZER or PROFICAL. The proposed heuristic is described by the pseudo code of Algorithm 4. The heuristic contains two phases:

1. It assigns actors to processing units: the actors are sorted in decreasing order with respect to their computational load and then are distributed in a round-robin way on the different processors.
2. It assigns priorities to actions: for each processing unit, its assigned actions are sorted in decreasing order with respect to their total number of occurrences during the execution. Then, the highest priority is given to the most called action, and so on.

Algorithm 4 Round-robin load balancing heuristic

Let a be the actions set, sorted in decreasing order w.r. to their total comp. load;
Let A be the actors set, sorted in decreasing order w.r. to their total comp. load;
Let PU be the set of processing units ;

$P = PU_0$;
$j = 0$;
while (A is not empty) **do**
  if (PU_j exists) **then**
    Assign actor A_i to processor PU_j;
    $j++$;
  else
    Assign actor A_i to processor PU_0;
    $j = 0$;
  end if
  Remove A_i from A ;
end while

$m = 0$;
while (a is not empty) **do**
  Set priority m to a_i ;
  $m++$;
end while

The heuristic complexity is $O(|A|)$ and outputs the partitioning along with a priority list scheduling. This approach has been implemented in the SCHEDULCAL tool.

Discussion Obviously, this heuristic is very fast but does not provide very efficient partitions, because of the poor accuracy of the load balancing and because it does not take into account the communication costs nor the implementation overhead.

4.2.2 Simulated annealing load balancing

This heuristic aims at improving the previous one by using a simulated annealing approach to load balance more accurately the computations of the actors onto the different processing units by taking into account the communication costs. The objective is to minimize the largest load (computation and communication) among all the processors. The objective function to minimize - *the cost of a partition* - is defined in Equation 4.1. For a given partition, the total load of an actor is defined in Equation 4.2.

$$Cost(Partition) = \max\{\sum_{Actors} Load(Actor)\}_{Processors} \tag{4.1}$$

$$Load(Actor) = CompLoad(Actor) + CommLoad(Actor) \tag{4.2}$$

$$CommLoad(Actor) = Data \times IPC \tag{4.3}$$

The communication cost of an actor is computed as the product between the total amount of exchanged data (input and output) with other partitions and the Inter-Processor Cost (IPC). The user is asked to set the IPC corresponding to the cost of exchanging one unit of data with another partition. For instance, if data transfers are defined in bytes, the IPC corresponds to the cost of transferring one byte of data through two partitions.

The computation cost of an actor is given by the profiling of its actions. It can be in terms of C/C++ operators, instructions and time.

The user must be careful about maintaining a coherency between all these units in order to obtain meaningful results. Algorithm 5 describes this new approach. The *Perturbate*() function returns an aleatory move.

This heuristic has been written by Per Persson (Ericsson) and has been integrated into the SCHEDULCAL tool.

Discussion This heuristic is slower than the first version because several thousands runs are launched. The advantages is that it takes in account the communication costs. By setting large communication costs, the heuristic behaves like a clustering heuristic, trying to gather the actions which large interdependencies. The fact of considering only the amount of data exchanged between actors does not produce as good results as expected. The reason is that it is

Algorithm 5 Simulated annealing load balancing

Let P_{cur} be the initial partition ;
Let $NbIt$ be the number of iterations ;
Let P_{min} be the best partition with the minimal cost ;
Let $Cost(P)$ be the max of the load of the processors ;
Let be T the current temperature ;

$initPartition()$; {Random}
$T0 = initTemperature()$;
for ($i = 0$ to $MaxRound(T0)$) **do**
  for ($j = 0$ to $NbIt$) **do**
    $P_{next} = Pertubate()$;
    if ($Cost(P_{next}) < Cost(P_{cur})$) **then**
      $P_{cur} = P_{next}$;
      if ($Cost(P_{next}) < Cost(P_{min})$) **then**
        $P_{min} = P_{next}$;
      end if
    else
      if ($random() < e^{-(P_{next} - P_{cur})/T}$) **then**
        $P_{cur} = P_{next}$;
      end if
    end if
  end for
  $T = T \times 0.95$;
end for
return P_{min} ;

mostly the fact that actors are connected together - *whatever the amount of tokens transfered, under a certain limit* - which is dominant and not the amount of data. The experiment of Section 4.3.1.1 assesses this hypothesis.

4.2.3 Causation trace scheduling

The Load balancing heuristics presented in Sections 4.2.1 and 4.2.2 distribute the amount of computations at the actor level on the different processors but ignore the dependencies between actions. However, dependencies are important to be considered.

The dependencies between actions firings can be retrieved from the causation trace extracted by the Cal Dynamic Analyzer or ProfiCal. Distributing the nodes of the causation trace (i.e. the actions) on the different processors according to a given partition can provide an estimation on the resulting performance. The problem is to find the best actors partition and actions scheduling on a multiprocessor system in order to minimize the makespan which can be defined as follows:

Definition 4. *The **makespan** corresponds to the latest finishing time of any action firing.*

4.2.3.1 Problem statement

The problem is based on the typical scheduling problem with the objective of minimizing the makespan. Additionally, all actions of a single actor must run entirely on a single processor. This constraint is essential for seamless maintenance of the state and cache-friendliness inside an actor. It is also this restriction that makes the problem unique from the collection of studied variants of scheduling problems. The problem would be a very typical scheduling problem if it were not the constraint for which each actor runs entirely on a single processor, that is, if $actor(u) = actor(v)$, then $processor(u) = processor(v)$.

This simple problem, the partitioning for non preemptive tasks with the minimization of the makespan, with this additional constraint, does not seem to have been formulated and analyzed in literature before and lacks any results regarding hardness, approximatively or heuristic solutions [122].

The problem is abstracted as a scheduling problem with the extra constraint that groups of actions belonging to the same actor must run on the same processor, the objective being to minimize the makespan.

Let:

- The set of actors A,
- Each actor A has actions labeled $\{a_0, .., a_i\}$,
- A set of P processors,
- The required execution time of an action of an actor on a processor p is defined by $runtime(actor, action, p)$,

- For each pair of processors p and q, $comm(p, q)$ is the delay of sending one byte from p to q.

The aim is to schedule the causation trace on the processors such that:

- Each action firing v is assigned to exactly one processor, called $proc(v)$.
- Tasks running on the same processor do not overlap.
- Tasks are not preempted, an action firing v runs from $start(v)$ to $end(v) = start(v) + runtime(actor(v), action(v), proc(v))$.
- Each action firing v assigned to a processor p runs for exactly a period of time defined by $runtime(actor(v), action(v), p)$.
- The starting time of an action firing v, given the dependency (u, v), is given by $start(v) \geq end(u) + comm(proc(u), proc(v)) \times data(u, v)$.
- Each actor runs entirely on a single processor: let u and v two actions firing, if $actor(u) = actor(v)$, then $proc(u) = proc(v)$.

The inputs are:

- The causation trace $CT(N, E)$ in which nodes N are action firing and edges E and dependencies between these action firing.
- Each action firing v is labeled with a pair $(actor(v), action(v))$.
- Each edge (u, v) is labeled with $data(u, v)$ corresponding to the number of bytes action u sends to action v.

Problem size It is necessary to have an idea of the expected sizes of the input parameters of the problem to be able to judge the success of the discussed algorithms to solve the problem efficiently. The CAL networks that have been studied and are specified in the new Reconfigurable Video Coding (RVC) standard do not exceed one hundred actors. However, the number of action firings can be arbitrarily large. A causation trace may easily contain hundreds of millions of action firings. Therefore, it is absolutely necessary that none of the discussed algorithms is superlinear in the number of nodes of the causation trace. The algorithms are even required to have a complexity with a small constant in the term proportional to the number of nodes.

4.2.3.2 Lower bounds

Most of the problems considered are computationally hard and optimal solutions for realistic data sets are hard to compute. However, it is necessary to get an idea of how close the obtained results are to the optimal. Since all our problems are minimization problems, lower bounds on the optimal are needed. This section discusses several lower bounds.

Trace critical path The first estimation on the optimal schedule is obtained by computing the longest computational path in CAL programs (see definition of the trace critical path in Section 3.2.2.3). It is obvious that no schedule can lead to a smaller makespan than the time consumed by the longest

chain of dependent actions running in sequence. It is not clear, however, which running time to choose for each action (on which processor). The estimate can choose to either take average values or worst case values. The former is not a guaranteed lower bound while the latter is not a realistic bound. Luckily, choosing one or the other does not affect the values significantly in the tested data. It is noteworthy that computing the longest path can be done in linear time (Algorithm 1). The communication costs are considered to be zero.

Minimum processor load Another very good lower bound is the minimum possible load on a processor. This estimate is only applicable on architectures with identical processors. The estimation remains accurate if the processors are not very different. The estimation is obtained by summing up the running times of all firings of actions (the running time is either the same on all processors or the average is taken) and this sum is divided by the number of processors. This number is obviously a lower bound (or an accurate estimation) on the optimal solution.

4.2.3.3 Heuristic

Algorithm 6 describes the approach. From an aleatory partition, the heuristic computes new partitions based the swapping of two actors ($SwapActors()$). The heuristic stops when the specified number of iterations has been reached. The evaluation function $Evaluate(P, CT)$ returns an estimation of the makespan of the execution given the computed partitioning and scheduling. This evaluation is further discussed in Section 5.3 and is a major issue in the success of this approach.

Algorithm 6 Minimization of the makespan

For each node v, $label(v) = runtime(v) + \max\{label(predecessor_i(v)\}_i$;
Let P_{best} be the partition leading to the best makespan ;
Let $makespan(P)$ be the value of the makespan for a given partition P ;
Let $NbIt$ be the number of iterations ;

while $(i < NbIt)$ **do**
  $P = SwapActors()$; {Swap two actors}
  $makespan = Evaluate(P, CT)$ {Evaluation Function}
  if $(makespan < bestmakespan)$ **then**
    $P_{best} = P$;
    $bestmakespan = makespan$;
  end if
end while
return P_{best} ;

This heuristic has been implemented in the SchedulCal tool.

4.2.3.4 Discussion

This success of this approach is strongly dependent on the accuracy of the evaluation of the performance of the solution. This heuristic may be mis-guided if the results given by the estimation are too far from the real behavior. Another point is that the performance evaluation of a solution takes some time. If the accuracy of the performance evaluation is improved, the complexity of this process increases because more computations are then involved. In other words, the time needed to estimate a solution may increase with respect with its accuracy. Thus, for this approach to be efficient, a fast process to evaluate the performance of the solution is needed at the risk of producing poor results or of waiting a long time for producing precise results.

What could be wrong with this approach is to be stuck to the causation trace, which is the results of the execution of the CAL program using fixed input data. The resulting partitioning may be efficient for this given input data but not for other. This drawback can be faced by using appropriate input sequence, which is statistically representative of a common execution of the application.

This heuristic has been improved by Abdallah Elguindy [122] by considering the swap of three actors. Furthermore, the assignment of the priorities is also modified. Actions with longer chains of subsequent dependencies are given priority. In other words, each node is labeled by the length of the longest path in the graph starting from this node. Precedence between nodes ready to run is determined by this value, the larger takes priority.

4.2.4 Static regions scheduling

This section exposes an alternative approach for tackling the partitioning and scheduling problem. The heuristics presented in the previous chapters ignore the overhead due to the scheduling process. In the last approaches, it was assumed that the execution of the actions dominates any scheduling overhead. This is not exactly the case as shown in Figure 4.5.

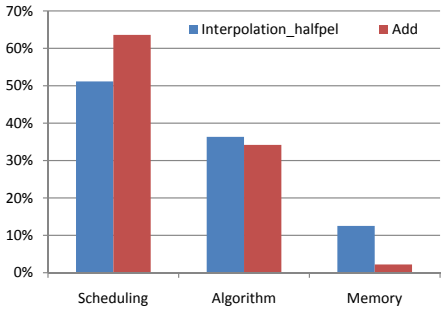


Figure 4.5: Proportion of scheduling, algorithm and memory during actor execution.

In the currently studied CAL programs, actions are defined at a quite low

level of granularity and thus the scheduling overhead is often non negligible. It makes sense to try to find good enough schedules that are easy to find at compile-time without incurring much overhead. The main idea lies in splitting the program into segments (static regions) that are repeated and computed once for all efficient schedules for these segments.

One can define a static region as follows:

Definition 5. *A **static region** is a set of actions for which their mutual dependencies at runtime (state and tokens dependencies) do not depend on input data nor time. It other words, static regions corresponds to invariant patterns in the causation trace with respect to input data and time, making possible their scheduling at compile-time.*

A Cal program contains actors that do not necessary belong to the same model of computation. Actors may be static (SDF, CSDF), dynamic (DDF) or time-dependent. Thanks to the classification of the actors (Section 3.2.1.1), one can detect sequences of connected static actors that can be scheduled statically. But when dynamic actors are finely analyzed, one can notice that they may also reveal several static behaviors according to the value of the input tokens. Let call these static behaviors **modes**. These modes are triggered by guarded actions which fire only when the tokens requirements and the guard conditions are fulfilled. In a given mode, a unique sequence of actions is executed, behaving like a static region which can be scheduled in an efficient way using the SDF techniques and thus avoiding the overhead due to the scheduling of these hidden static regions. One problem is the detection of such static regions in these dynamic actors. For instance, the Finite State Machine of Figure 4.2.4 (Source: [146]) describes the behavior of such an actor.

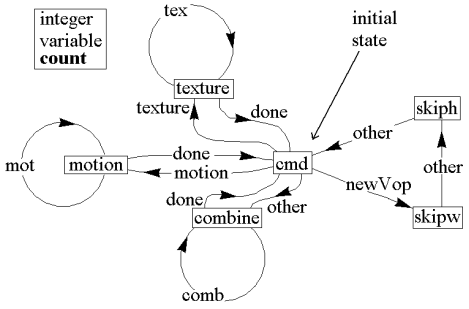


Figure 4.6: The actions *texture, motion* and *other* triggers three different modes which are respectively composed of a sequence of 64 firings of the actions *tex, mot* and *comb* plus the firing of action *done*.

The main challenge of this approach lies in:

1. Splitting the program into static regions that are repeated (Section 4.2.4.1).
2. Computing efficient schedules for these static regions (Section 4.2.4.2).
3. Finding a scheme to identify these patterns at runtime (Section 4.2.4.3).

4.2.4.1 Finding static regions at compile time

The first idea would be to search in the causation trace some repeating patterns. But comparing two subgraphs of the causation trace graph for isomorphism is a hard problem [123]. Furthermore, the size of the trace is so large that it makes this approach intractable. Detecting repeating patterns in the causation trace does not guarantee that this sequence of actions firing is really static. The input data can be such that it induces similar patterns of action firing.

By analyzing further the problem, actions which have guards checking values of input tokens are good starting points for these repeated patterns. As an example, the MPEG-4 Simple Profile decoder contains different decoding modes, reflected in the actors of the program. The actor `ADD` (Figure 4.2.4) has three running modes: *texture only, motion only, combine*. Figure 4.7 (Source: [146]) has been obtained by extracting from the causation trace the static regions of different actors running in a same mode and by merging them to obtain an overview of the cross-actor static region.

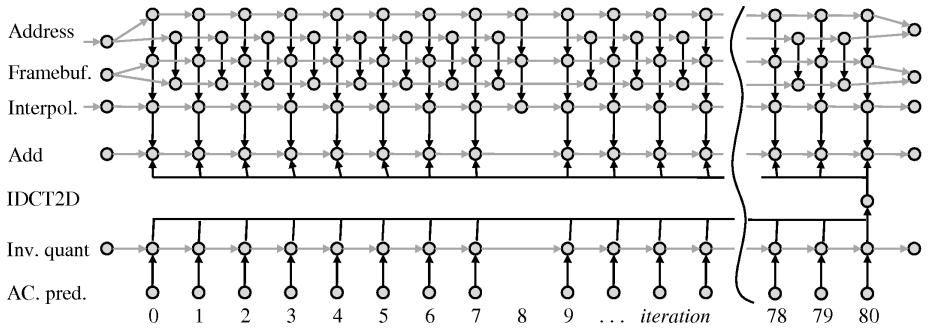


Figure 4.7: A static region can be represented as a DAG ; example taken from the MPEG-4 SP decoder.

4.2.4.2 Scheduling static regions at compile time

The Directed Acyclic Graph (DAG) presented in Figure 4.7 represents all the dependencies that admissible schedules must respect. Depending on the requirements of the application, it might be necessary to minimize the memory resources, the latency or to maximize the throughput. For each of these criteria, different partitions and schedules are computed at compile-time from this dependencies graph. Constraint programming can be used to schedule these static regions optimally according to different criteria. These DAG are equivalent to SDF graphs. Some scheduling techniques of SDF models are presented in [113].

4.2.4.3 Triggering static regions at runtime

The second challenge is to identify at runtime the entry points of these static regions, which have been identified and scheduled at compile-time. The scheduler

has the following options:

1. It can check the first few elements of the sequence and check it against previously stored patterns. If they match, then the stored sequence is immediately scheduled. If the sequences differ (and the match was a false positive), an exception is thrown and any scheduling technique is used.
2. A suffix tree is maintained with the leaves having prepared schedules, Actions are tentatively scheduled until the scheduler is sure that the sequence was seen before.

As shown in Figure 4.8 (Source: [146]), when a static region has been detected, it is scheduled on one or several processors.

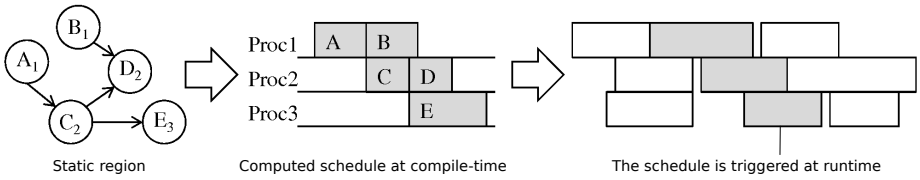


Figure 4.8: Schedules computed at compile-time and executed at runtime.

4.2.4.4 Discussion

The design heuristic aiming at removing the implementation overhead to turn actors static (Section 3.3.3.2.3) is in line with this approach because the static scheduling is only possible with static actors and dynamic actors should be avoided. This work at the CAL level is very precious because it has non-negligible impact on the performance.

The advantage of static scheduling is that the the modularity of the CAL program is kept because actors are still distinct.

Gu et al. [124] show that exploiting these static regions at the implementation level leads to better performance. A preliminary work on how to find these static regions has been done by Boutellier et al. [146, 147].

However, there are still some limitations to this approach: (1) it lacks a clear and systematic methodology for extracting from the CAL program the different static regions, which can be numerous and difficult to extract. Furthermore, the way the CAL program is written can hide its inherent static portions because of the insertion of some dynamic actor into it and (2) how to detect the static regions at runtime.

4.3 Performance evaluation heuristics

Being able to estimate the performance of a CAL program given a partitioning on a target platform is a very challenging task but can be very powerful. If

the behavior can be predicted, the partitioning heuristics can be guided by these estimations of the performance and can iterate over and over towards efficient partitioning. At the contrary, if the behavior is wrongly predicted, these heuristics output inefficient solutions.

Predicting the behavior of the program is a hard task because multiple code transformations between the CAL program and assembler code are performed and because of all the mechanisms that occurs on parallel systems, making hard to consider all these aspects in the performance evaluation step. The difficulty resides in the fact to find the right level of details for the performance evaluation, not too simple in order to reflect the reality but not too complex at the risk of being lost and outputting false estimations.

The solution chosen is to build a Gantt chart of the execution of the CAL program. One can reconstitute the execution of the CAL program by distributing the actions firing of the causation trace across the different processing units specified in the partitioning computed by the heuristics. The duration of the events of the Gantt chart is given by the profiling of the actions in terms of instructions, C/C++ operators or time. Figure 4.13 illustrates such a Gantt chart. The procedure for building the Gantt chart is detailed in Algorithm 7.

Algorithm 7 Performances evaluation: Gantt chart

```
Set the set runnable = {}. ;
for all node u in CT(V, E) do
  if (indegree(u) = 0) then
    add node u to runnable ;
  end if
  Sort(runnable) ; {Sort items by the label value in descending order}
  while (runnable is not empty) do
    Schedule the first node of runnable ASAP on Processor(u) ;
    Remove this first node of runnable ;
  end while
end for
```

As the causation trace contains only the firing of the actions, the Gantt chart contains only actions. However, in order to keep in view the characteristics of the runtime, the execution times of the actions are modified as shown in Equation 4.4. The execution time t_{exec} of an action can be seen as the sum of the time elapsed for scheduling (t_{sched} which depends on the partitioning P, the scheduling policy SP and the size of the FIFOs FS), for the application (t_{algo} which is constant) and for the memory accesses (t_{mem} which depends on the partitioning P).

$$t_{exec} = t_{sched}(P, SP, FS) + t_{algo} + t_{mem}(P) \tag{4.4}$$

Where :

- t_{algo} is the profiling of the running time of the actions, provided by the CAL DYNAMIC ANALYZER, PROFICAL or Gprof.

- $t_{mem}(P)$ is estimated as shown in Section 4.3.1.
- $t_{sched}(P, SP, FS)$ is estimated as shown in Section 4.3.2.

4.3.1 Communication model

Refining the communication model aims at improving the accuracy of the Gantt chart in the purpose of predicting as close as possible the performance of the solution.

4.3.1.1 Experiment: estimation of the communication cost

The extra cost due to the data communication through different processing units is platform-dependent. The following experiment aims at evaluating the communication cost in case of the Hewlett Packard 6710b (Intel®Core™2 Duo CPU T8300 at 2.4 Ghz) platform. It consists in profiling the communications of a simple CAL program composed of two actors (A and B), each actor being mapped to a distinct processor. Both actors are composed of a single action with a constant computational load. The firing of actor A produces N tokens entirely consumed by actor B. The communication cost during the execution of 10^5 firings of actors A and B is profiled.

The profiling of the communication cost has been performed for increasing values of the number of tokens ($N = \{1, 10, 10^2, 10^3, 10^4, 10^5, 10^6\}$) exchanged at each firing of actors A and B. Figure 4.9 plots the duration (in seconds) of the full execution (10^5 firings of A and B), according to the number of tokens N exchanged at each firing of actor A and B. The experiment is repeated considering:

- Actors A and B on the same processor (blue curve, single core)
- Actors A and B on two processors (red curve, two cores)

The communication cost really depends on the platform and the results of these experiments (Figure 4.9) are valid only for the Hewlett Packard 6710b platform (Intel®Core™2 Duo CPU T8300 at 2.4 Ghz).

The results confirm that using a FIFO which is accessible by several actors mapped on different processors has a non-negligible cost. The extra time is due to the use of locking primitives for the shared FIFO which is more expensive than a simple unshared FIFO in terms of computations. The communication cost remains unchanged when the actors A and B exchange from 1 to 10^2 tokens at each firing. When $N > 10^3$, it evolves linearly with the number of exchanged tokens.

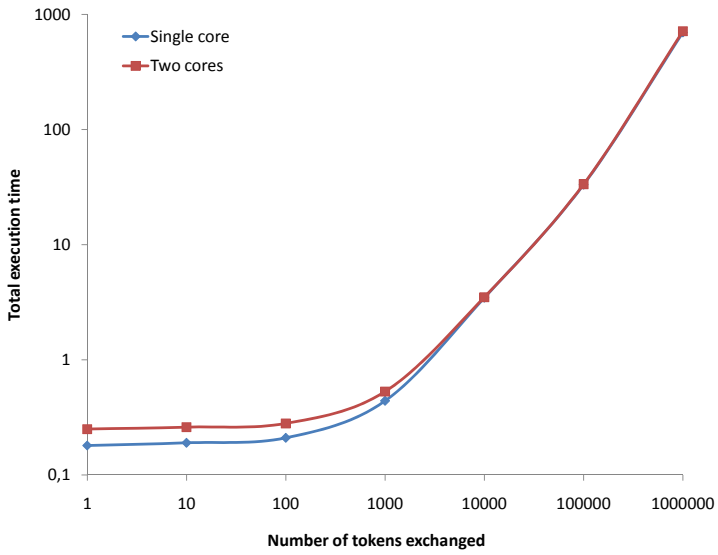


Figure 4.9: The communication cost remains identical when data transfers are smaller than 100 tokens. Above this threshold, it evolves accordingly with the size of the data transfers (Platform: Hewlett Packard 6710b Intel®Core™2 Duo CPU T8300 at 2.4 Ghz)

4.3.1.2 Model

From the results of the experiment, one can define a communication model. In case $N < 10^3$, the communication model for the HP 6710b platform is:

$$t_{exec}(P) = \begin{cases} t_{action} \text{ if the action is not connected with another partition.} \\ t_{action} + k \times t_{commcost} \text{ otherwise.} \end{cases} \tag{4.5}$$

Where:

- $t_{exec}(P)$ is the execution time of the action taking into account communication costs. It depends on the partitioning P because of the communication cost.
- t_{action} is the execution time of an action without communication cost.
- $t_{commcost}$ corresponds to the communication cost in terms of extra time due to the use of a shared FIFO. It is constant.
- k is the number of ports through which the action exchanges tokens with an actor of another partition. For example, if an action inputs and outputs tokens from two different partition, $k = 2$.

The communication cost is considered as constant ($t_{commcost}$) for any tokens transfer, whatever its size inferior to 10^3. The communication model is very

simple to implement in the tools because for each execution of an action communicating with another action on another partition, the extra cost ($t_{commcost}$) can be added to the current execution time of the action.

4.3.2 Scheduling model

Whatever the runtime, the firing of an action needs several steps which are not negligible. The question is how to reflect this scheduling overhead through the Gantt chart. In ORCC, only the functions `isSchedulable()` are called before each action firing, making possible to consider their computational load in the current version of the Gantt chart. But for the other scheduling steps, it is not obvious how to determine the extra scheduling cost for a given action execution because any firing of action depends on the following parameters:

The scheduling policy The decidability of actions firing can be more or less computations-intensive according to the chosen policy.

The partitioning According to the number of actions assigned to a processor, the scheduling of these actions will not be the same, and the overhead differs from different partitioning.

The size of the FIFO connecting the actors The round robin scheduling policy executes an action until it is no longer possible to fire any action of the actor, making the status of the FIFO an important parameter to decide which action must be scheduled next.

In order to improve the accuracy of the performance evaluation, a new architecture of the Gantt chart engine is proposed as future work in Chapter 8.

4.4 Code Generation

Having a high level representation of the application is a advantageous feature but is useless if there is no way to make profit of it. Code generators are the missing link between the high level specifications and their implementation, thus bridging the implementation gap by turning any high level specification into a runnable implementation on a target platform. Several research groups are currently working on code generators in order to improve the efficiency of the generated code.

Moreover, generating code supposes a systematic process which links clearly any high level specification in CAL to its implementation. By understanding well this relation and by modeling it, one can guess the behavior of any CAL program at high level without executing it on the platform. If designers need to implement themselves the application described at a high level of abstraction, it is impossible to get this link between the high level specification at which design space exploration occurs and the implementation. Another crucial point is that code generators are primordial for extracting metrics at the implementation level and for bringing them up to higher levels in order to perform realistic design space exploration. This link, created between the high level specification and

the low level implementation, is crucial for this purpose. Figure 4.10 illustrates this necessary link between specification and implementation.

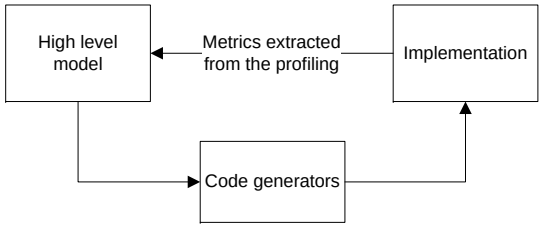


Figure 4.10: Code generators are making the link between high level specification and implementation.

The capability to estimate the performance of a CAL program given a partitioning makes meaningful the elaboration of design space exploration methodology aiming at finding the best partitioning of a CAL program onto a given platform. Advanced partitioning heuristics (Section 4.2) need this performance evaluation of the solution at high level in order to iterate and to output efficient partitions.

But the implementation of a high level language as CAL into low-level software languages - such as C/C++ or VHDL/Verilog - reveals necessary overheads that must be considered.

In software Communication channels between actors are turned into real FIFO implementations which may introduce additional synchronization protocols in case of concurrent memory accesses. Indeed, the implementation of actors introduces a controller (namely action scheduler) which is responsible for firing actions. Based on the current state of the actor, the token availability and the guards, the action scheduler selects the next action to fire. At a network level, if several actors are mapped on a single processing unit (PU), a scheduling policy must be defined to execute actors in the right order.

In hardware - FPGA, ASIC - the system is "self-scheduled" since all actors can run in parallel. Actors are executed concurrently, managed by a controller (namely action scheduler). All of these controllers introduce overheads in terms of processing time, silicon usage, etc.

The existing code generators for converting CAL programs into software and hardware implementation are presented in the following paragraphs.

4.4.1 Open RVC-CAL Compiler (ORCC)

The Open RVC-CAL Compiler (ORCC) is a compiler infrastructure for converting RVC-CAL programs into other languages: C, C++, Java, LLVM, VHDL and

XLIM are the current languages available. Only source code is generated, and programs must be compiled with the usual tools. Specific libraries need to be developed for each language so that it can be compiled and run. The generated code is cross-platform. The ORCC framework is fully described in [125] and the reader is referred to [126] for an overview. ORCC was created at IETR (INSA Rennes - France) and is now being developed by industrials and research laboratories worldwide.

4.4.2 OpenForge

OpenForge [127] is a behavioral synthesis tool infrastructure and software framework, translating applications specified in CAL into hardware descriptions expressed in VHDL/Verilog. This is a promising tool in the sense that the code generated from the CAL program outperforms a commercially produced VHDL design in terms of both throughput and silicon area [128]. Recently, a couple of students finished their Master's thesis [129] on generating low power hardware implementation from CAL. The fundamental properties of CAL make it a promising candidate for Globally Asynchronous Locally Synchronous (GALS) design.

4.4.3 Ericsson Code Generator

As part of the European project ACTORS , Ericsson has also developed its own C code generator targeting ARM processors. This software tool called Xlim2C translates an intermediate representation of a CAL actor - called XLIM - into a C program. This tool compiles directly the generated code into an executable. This code generator works only under Linux and only ARM processors are targeted. The reader is referred to [130] for further details.

4.4.4 Co-Design Tool

The idea of a co-design tool comes from the work on a rapid prototyping platform [148–150] aiming at easing the design of hardware systems. Software and hardware code generators (ORCC and OpenForge) are used for the respective targets [151] and a interface generation tool [131] has been implemented in order to ease the implementation. The aim is to be able to implement any kind of application on a wide range of architectures. This tool has been successfully used for the implementation of an industrial application on an heterogeneous platform [132]. Currently, it is still under development.

ACTORS project: `http:\www.actors-project.eu`

4.5 Tools

4.5.1 SchedulCal

This tool contains the implementation of the different partitioning heuristics described in Section 4.2:

Basic load balancing In a round-robin fashion, this algorithm distributes actors on the different processing units according to their total computational load. The priority list scheduling policy is used.

Load balancing using a simulated annealing approach A more advanced load balancing heuristic has been developed, based on simulated annealing and taking into account communication costs between actors. The round-robin scheduling policy is used.

Minimization of the makespan using a simulated annealing approach Based on the causation trace, this heuristic minimizes the makespan of the execution by using a simulation annealing approach. The priority list scheduling policy is used.

Two scheduling policies have been defined :

Priority list policy Each action has a priority. In case several actions can be fired on a same processing unit, the priorities determine the next action to fire.

Round-robin policy For each processing unit, an ordered list of actor is defined. The actions of the first actor are executed until there is no more fireable action in this actor. Then, the second actor is considered until no more actions of this actor is fireable, and so on. At the end of the list, the first action is considered again.

The tool is written in Java using the Eclipse RCP framework. This tool is exportable as an executable on different operating systems: Windows, Linux and MacOS. Figure 4.11 shows a screenshot of the GUI of the tool.

4.5.1.1 Inputs

Structural information of the Cal program provided by Cal Static Analyzer.

The causation trace provided by Cal Dynamic Analyzer or ProfiCal.

Execution time of actions provided by WeightCAL.

4.5.1.2 Output

The **output** is an XML file containing the partitioning and the scheduling of the Cal program.

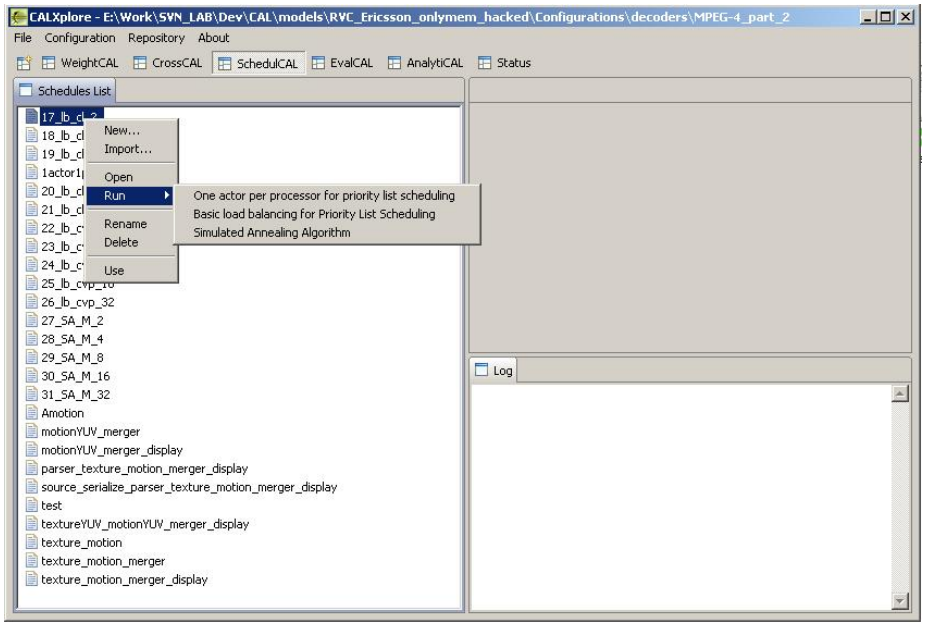


Figure 4.11: Graphical user interface of SchedulCal

4.5.2 EvalCal

The tool aims at evaluating the design by building the Gantt chart of the execution (as explained in Section 4.3), given a partitioning of actors, a scheduling of actions, the causation trace and the running time of each action of the program. EvalCal supports two scheduling policies defined in the SchedulCal tool (Section 4.5.1). Two analyzes on the Gantt chart have been implemented:

Occupancy of processors This is the estimation of the occupancy of the different processors.

Density of operations It reports how dense are executed the actions of the processors. The results are reported as distributions of the size of the idle time between two executions of actions.

The tool is written in Java using the Eclipse RCP framework. This tool is exportable as an executable on different operating systems: Windows, Linux and Mac Os. A screenshot of the tool is shown in Figure 4.12. Figure 4.13 shows an example of Gantt chart that can be generated by the tool.

4.5.2.1 Inputs

Structural information of the Cal program provided by the Cal Static Analyzer.

The causation trace provided by Cal Dynamic Analyzer.

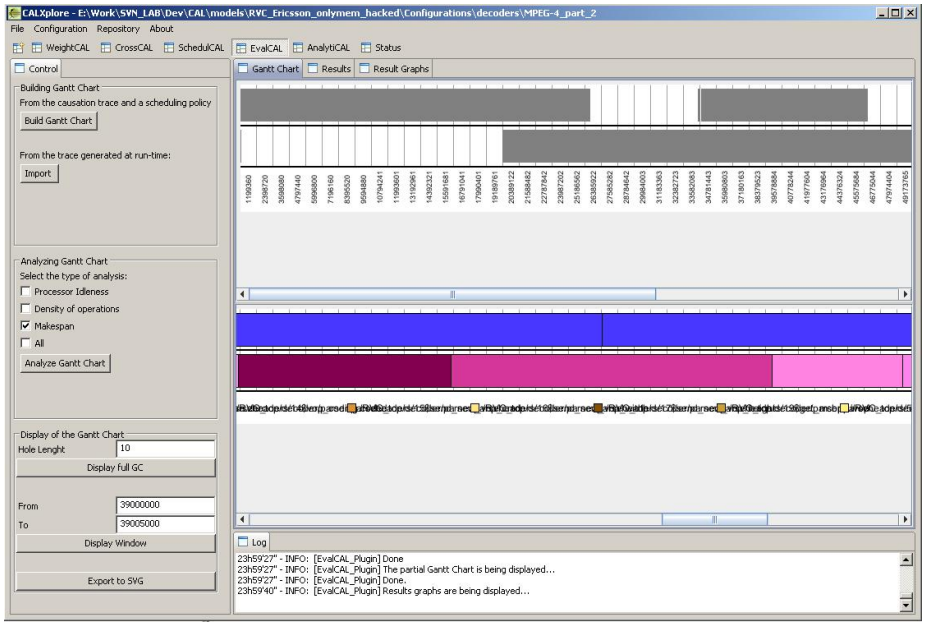


Figure 4.12: Graphical user interface of EvalCal.

Partitioning and scheduling of the CAL program, specifying the assignment of actors onto the processing units and the scheduling policy on the processing units.

Execution time of actions provided by WeightCAL.

4.5.2.2 Outputs

Gantt chart in XML The Gantt chart of the execution, given the chosen partitioning.

Gantt chart in SVG Optionally, the tool can generate the Gantt chart in a Scalable Vector Graphics (SVG) format.

XML result file containing the result of the different analysis: occupancy of each processor, density of operation on each processor.

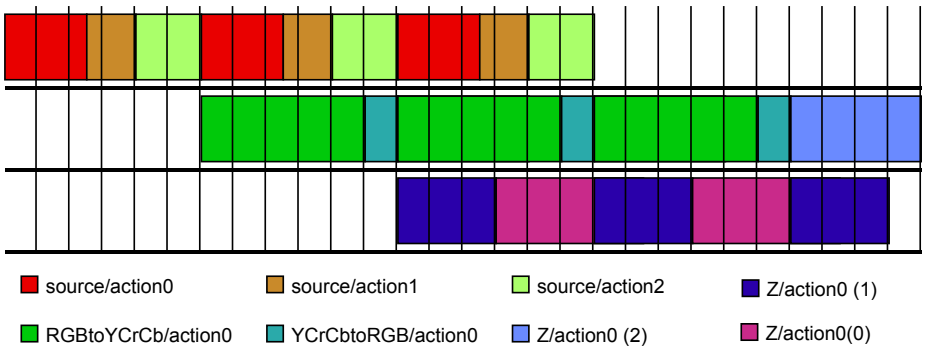


Figure 4.13: Example of a Gantt chart of an execution.

Chapter 5

A design flow for high level exploration of the design space

Digital systems design consists in fitting one or several applications onto a given platform with constrained resources while satisfying predefined criteria. This work implies optimizing the applications at an algorithmic point of view. Chapter 3 exposed some rules for writing well-shaped Cal programs. The mapping of the application onto the target platform consists in assigning actors to the different processing units and in converting the Cal code into implementation code with respect to their assigned target. Chapter 4 presented heuristics and methodologies for mapping Cal programs onto target platforms.

The work of the designers is to find the best matching between the application and the architecture in order to obtain an efficient system while minimizing the resources consumption. This task implies working at different levels of abstraction: (1) at the application level, at which the developer writes an application whose parallelism is coherent with the level of exploitable parallelism of the platform, (2) at an intermediate level at which the Cal program is weighted with platform-dependent metrics whose analysis leads to the detection of the bottlenecks of the system and allows the designer to refactor the Cal program accordingly and (3) at the implementation level, at which every actor is translated in the native implementation language corresponding to its target processing unit.

This chapter describes a systematic design flow supporting designers in this difficult task of matching complex applications onto parallel architecture, by making them navigate through the different levels of abstraction and by guiding them during the program refactoring process. Section 5.1 introduces the notion of design space, Section 5.2 presents how a design can be located into the design space, Section 5.3 explains the proposed design flow and the tools supporting it and Section 5.4 presents the tools infrastructure.

5.1 How to represent the design space?

In order to be able to explore the different solutions for the implementation of an application onto a target platform, there should be a mean for comparing the different architectural solutions in order to choice the one which fulfills the requirements in terms of speed, resources, energy consumption, etc. In the context of this research work, the design space represents the set of all possible implementations of a CAL application onto a target architecture.

Exploring the design space means evaluating the various possible solutions with a given range of possible partitioning, scheduling and refactoring. It also consists in optimizing the solution with respect to the criteria defined for validating a solution towards a solution which fulfills the requirements.

The design space is commonly represented using graphs so that the different architectural solutions can be properly compared according to a given number of criteria (throughput, resources, cost, etc.). For instance, Figure 5.1 illustrates a graph representing a design space according to two criteria. The reader is refer to the review [64] for further details on design space exploration.

The different components of the design space are:

Axes Systems design is guided by constraints: performance, resources, size, cost, etc. Designers try to optimize the design according to these criteria. The number of criteria defines the dimension of the design space, e.g. respectively 2D or 3D if two or three criteria are considered and so on. Each criterion is represented by an axis of the design space. Generally, a 2D design space representation is sufficient to cover common design constrains. Usually, throughput and resources are the criteria that define the axis of the 2D design space representation as shown in Figure 5.4.

Design point In this research work, each point in the design space corresponds to a triplet {CAL *program, partition, schedule*}. The *partition* is a one-to-one correspondence between actors and processing elements. The *schedule* represents the ordering of actions onto each processing unit. The solution is evaluated according to the chosen criteria.

Target Region Depending on the maximization, minimization or lower/upper bound conditions, specific regions (segment, area or volumes) of interests can be defined. The aim is to reach a design which belongs to this target region by fulfilling the constraints.

5.2 How to evaluate the performance of a solution?

The performance evaluation can be performed at different levels of abstraction (Section 5.2.1) and in several different ways (Section 5.2.2).

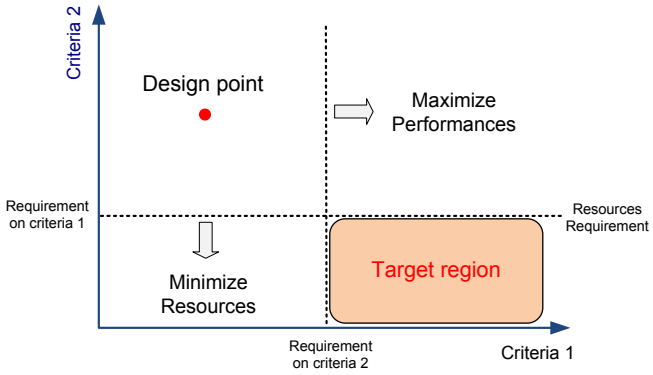


Figure 5.1: Representation of the design space according to two criteria.

5.2.1 At different levels of abstraction

High: at the level of the CAL program The metrics that position the solution in the design space are obtained by running the CAL program under a simulation environment interpreting the execution of the program according to an asynchronous data flow computation model. Thus, the dimensions of the design space are for instance the token throughput between actors, the minimum FIFO size for concurrent execution of all actors, the amount of computations for a given input sequence, etc. Currently, there is no tool for performing such a profiling at the level of CAL operators.

Intermediate: at the level of the source code of the implementation The performance evaluation process inputs the the partitioning and the scheduling of the CAL program and an architecture model of the target platform. The profiling of the CAL program (computational load of actions, communication costs) is provided by the analysis of the implementation source code (C/C++). Because the profiling of the actions is performed on the implementation code that is going to be executed on the platform, it results in a more accurate performance evaluation. It is the level chosen for running the partitioning and scheduling heuristics in the proposed approach.

Low: at the level of the implementation on the platform The performance of the real implementation of the CAL program can be measured directly on the target platform and the positioning of the design point in the space is characterized by dimensions such as execution time, CPU occupancy, area, etc. Such position is only possible after the compilation of the software code and the synthesis of the hardware code using the appropriate code generators [133] [152]. This level is used in the proposed approach to validate the results and to confirm the steps of the design space exploration.

5.2.2 Several methodologies

There are several methods for evaluating a design, from high to low levels of abstraction.

Evaluation from pure analytical models can be too pessimistic (and thus not representative of the reality) since these models often consider the worst-case only. Considering Worst Case Execution Time (WCET) is not very representative, especially in signal processing algorithms. This type of analysis is suitable for CAL programs composed of static actors because there is no uncertainties, their behavior is totally deterministic. Thus, this methodology has not been used in the proposed design flow because we should be able to analyze dynamic programs.

Simulations-based evaluations are well suited to study dynamic and unforeseeable effects in CAL programs whereas formally verifiable CAL programs require a deterministic behavior, given any stimuli. This type of analysis is necessary for CAL programs composed of dynamic actors because of the uncertainties due to the dynamic behavior of these CAL programs. Currently, apart from pure functional evaluation, there exists no tool that performs advanced analyzes of CAL programs based on a simulation.

Trace-based evaluation consists of simulation the behavior of the CAL program on a given target platform from the causation trace of the execution. The problem is that the evaluation is based on a given stimuli and may not reveal the real evaluation of the program for any stimuli. The designer must be very careful on the choice of the input data which must be representative of a usual execution of the CAL program. As for the simulation-based evaluation, this type of analysis is necessary for CAL programs containing dynamic actors. This is the methodology chosen in the proposed approach and tools have been implemented accordingly.

Cycle-accurate evaluation provides a good accuracy of the performance because the level of refinement is defined by a single clock-cycle. It means that at any given clock cycle, the state of the simulator must be identical with the state of the evaluation. It is at a very low level of abstraction and is far from the CAL program. This is very hard to obtain such a precision in the current evaluation process. Thus, in the proposed approach, the cycle-accurate evaluation is not respected.

5.3 How to explore the design space?

What is penalizing during the design space exploration on heterogeneous multi-core/multiprocessor platforms using the traditional (sequential) methodologies is that every refactoring and every change in the partitioning of the program is

very resource demanding in terms of code rewriting. Testing a new partitioning of a program onto processors is time-consuming and error-prone. Thus, there is a large gap between the idea of the new solution and its real test on the platform. The design flow presented in this chapter (and presented in [153]) aims at reducing the size of this gap by guiding the designer into steps directed towards a better solution. The abstraction features of the CAL language and the associated metrics that makes possible such design space exploration lead to efficient implementation solutions. Applying a new partitioning of actors or refactoring some actors are example of exploration steps. By iterating these steps, the designer can drive a system transformation until it fulfills the desired design requirements.

Code generators are primordial in two ways: (1) they enable designers to work at a high level of abstraction without the cost of dealing with low level implementation details and (2) they allow the high level estimation of the performance of the implementation on a given platform, enabling the elaboration of technique for automatic mapping of CAL program.

Figure 5.2 illustrates the proposed design flow along with the tools for exploring the design space at high level.

There are different types of systems requirements and they can vary according to the domain and/or the application. There are many possible optimization criteria, but performance and resources are the most common. Maximizing performance of the system or achieving a given performance while minimizing resources consumption are the main optimization criteria.

Sequential to parallel program transformation There are several ways to specify an application: textual description, C/C++ models. UML models and many others. This sequential specification has to be firstly translated in a CAL program. There is no automatic way to translate a C/C++ specification into a CAL program. The Software Instrumentation Tool (SIT) [42] is a tool providing capabilities to analyze complex sequential C/C++ applications in order to get the necessary metrics to translate it into a parallel program in CAL. The critical path analysis and the data transfers between functions are examples of analyzes that allow the designer to operate this conversion. A successful translation of the C/C++ specification of the MPEG-4 Simple Profile video decoder into a CAL program is reported in [1]. This step is not in the scope of this research work.

Characterization The characterization of the CAL program consists in extracting the metrics which are the basis information for the design space exploration. The analyzes described in Section 3.2 contribute to characterize CAL programs:

- Nature of the actors,
- Computational load of actions in terms of instructions, C/C++ operators or time,
- Causation trace,

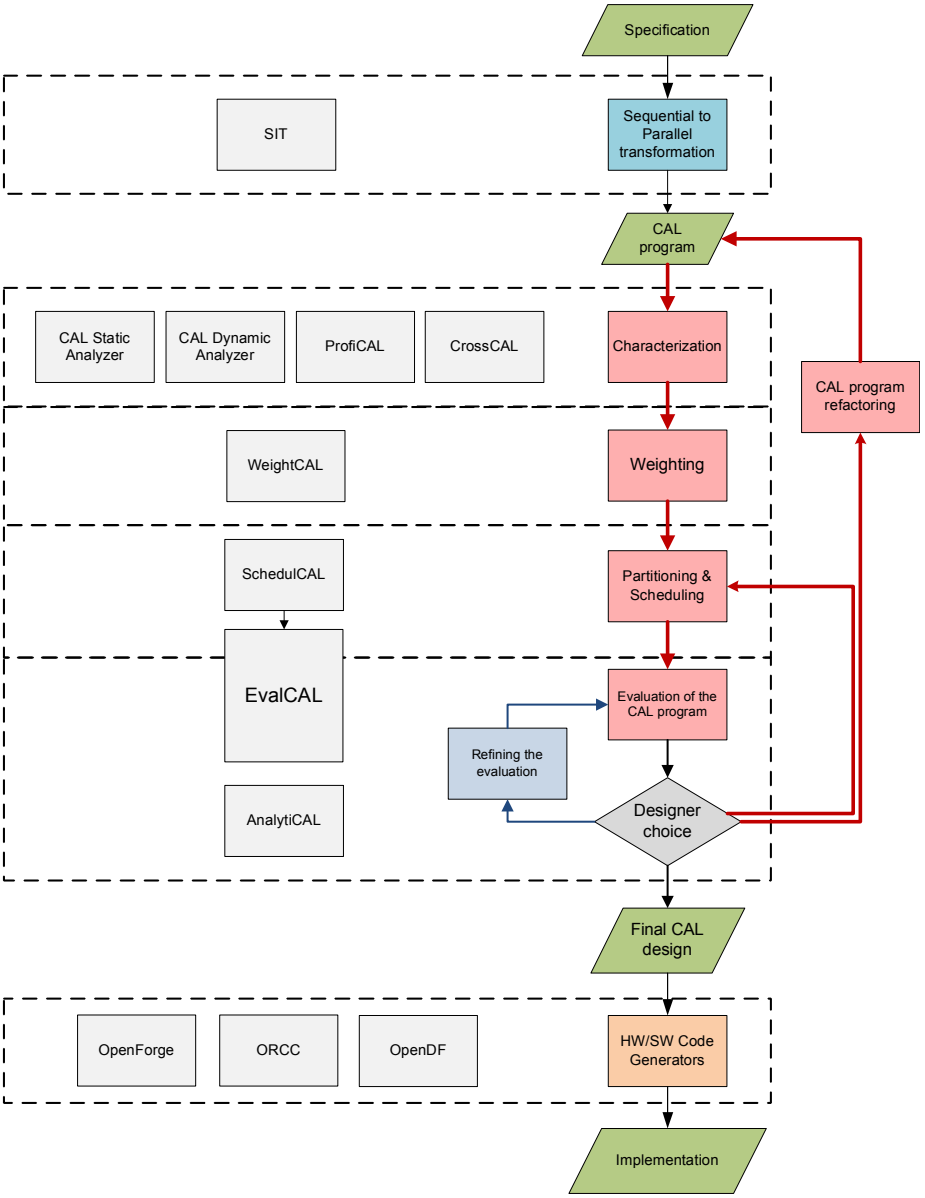


Figure 5.2: Illustration of the proposed design flow and the supporting tools.

- Trace critical path,
- Data transfers between actors.

The Cal Static Analyzer, Cal Static Analyzer, ProfiCal and CrossCal are the tools that have been implemented in order to extract these metrics.

Weighting This step consists in estimating the execution time of actions according to the target platform. The specificities of the target platform must be clearly taken into account in the design flow in order to estimate more precisely the performance of the design. As an example, in the ARM 7500FE processor, a General Purpose Processor (GPP), the multiply instructions take one instruction fetch and m internal cycles, m being the number of cycles required by the multiply algorithm, which is determined by the contents of the registers. In Digital Signal Processors (DSP), a Multiply-Accumulate operation costs only one clock cycle. In other words, this step aims at adapting the raw metrics output by the characterization tools to the target platform in order to predict as close as possible the performance of the design.

WeightCAL is the tool that has been implemented for setting the execution time of the actions. Currently, it is capable of choosing from different sources: Cal Dynamic Analyzer (instruction level) or ProfiCal (C/C++ operator level) or Gprof (time). It is not capable of determining the weight of each operator independently but it is one of the potential improvement of the tool.

Partitioning and scheduling The partitioning consists in allocating actors to processing units and scheduling in setting the order of execution on actions of the processing units. Section 4.2 exposes the heuristics for partitioning and scheduling Cal programs:

- Round-robin load balancing,
- Simulated annealing load balancing,
- Causation trace scheduling,
- Quasi-static scheduling.

All these heuristics have been implemented in the SchedulCal tool.

Performance evaluation The performance evaluation step consists in predicting the behavior of the partitioned and scheduled Cal program onto the target platform, taking into account the maximum number of specificities of the target platform. In the current version of the tools, only the communication cost is considered. The estimation of the performance is performed by reconstructing a Gantt chart of the execution, given the running time of actions, the causation trace and the partitioning/scheduling. The techniques presented in Section 4.3 have been all implemented in the EvalCal tool.

This step is crucial because it allows to assess or not the refactoring performed during the design space exploration, and the partitioning computed for a given program. The performance evaluation step guides the designer in the design space.

Designer choice In case the predicted performance do not fill the requirements, the designer has several choices:

- Refactor the Cal program,

- Partition and schedule the CAL program by using a different heuristic of by specifying it by hand.

CAL program refactoring The performance of the implementation is directly linked to the quality of the CAL program. Chapter 3 describes a strategy for optimizing CAL programs at the algorithmic level in order to reach the desired target region. It includes detecting the most critical actions of the programs (using the Critical Actions Detection algorithm presented in Section 3.3.2) and to propose appropriate refactoring techniques (Section 3.3.3). The aim of this step is to remove the potential bottlenecks of the programs by guiding the design in the refactoring. The Critical Actions Detection algorithm has been implemented in the CROSSCAL tool.

Code generation Once the design is ready for the implementation, the code generators convert the final CAL program into the corresponding native platform language (i.e. HDL for FPGA, DSP code, C/C++, etc.). Designers have at their disposal several code generators that generate C, C++ or VHDL/Verilog implementations from CAL programs. The different code generators are described in Section 4.4.

5.4 Tools infrastructure

The tools infrastructure implementing the design flow is shown in Figure 5.3. There are two types of tools. The **characterization tools** (CAL STATIC ANALYZER, CAL DYNAMIC ANALYZER and PROFICAL) analyze CAL programs by performing static analyzes and profiling. The **exploration tools** (CROSSCAL, WEIGHTCAL, SCHEDULCAL, EVALCAL, ANALYTICAL) allow to explore the design space, enabling designers to test different solutions with minimal efforts.

The main reason of having structured the tools such a way is the modularity. By defining clearly the interfaces of each tool, it is simpler for the developers of the tools to concentrate on the different aspects of the problem without being disturbed by the modification of other tools. The choice of XML for the intermediate files is motivated by their seamless integration within a Java environment.

The choice of the Eclipse RCP platform is motivated by the capability to package the tools into stand-alone applications for the ease of use and of distribution and to export them on different operating systems: Windows, Linux or MacOS.

The exploration tools are gathered into an integrated environment called CALXPLORE, each tool previously described being a perspective in this framework. The screen-shots of the different tools are in fact the perspective of each tool. The CALXPLORE framework contains CROSSCAL, WEIGHTCAL, SCHEDULCAL, EVALCAL and ANALYTICAL.

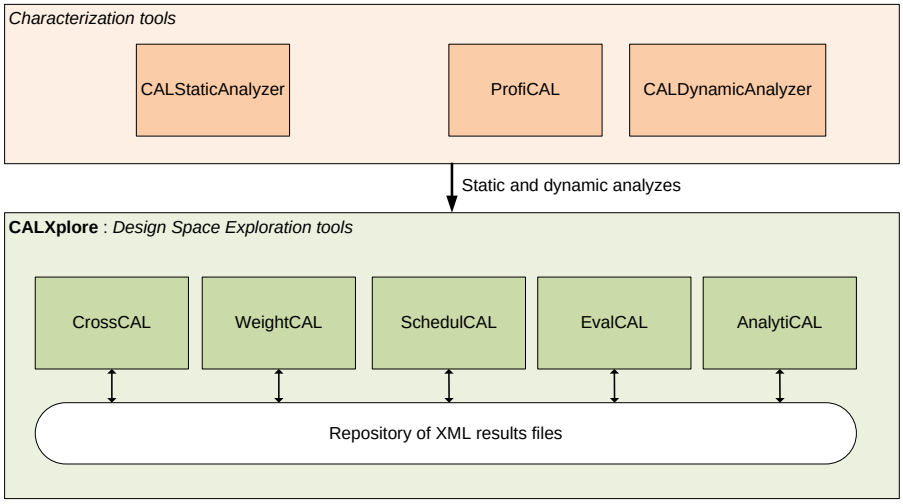


Figure 5.3: There are two types of tools: characterization and explorations tools.

5.5 Summary

The developed tools (Section 5.4) support designers in the exploration of the design space (Figure 5.4) using the proposed design flow (Figure 5.2).

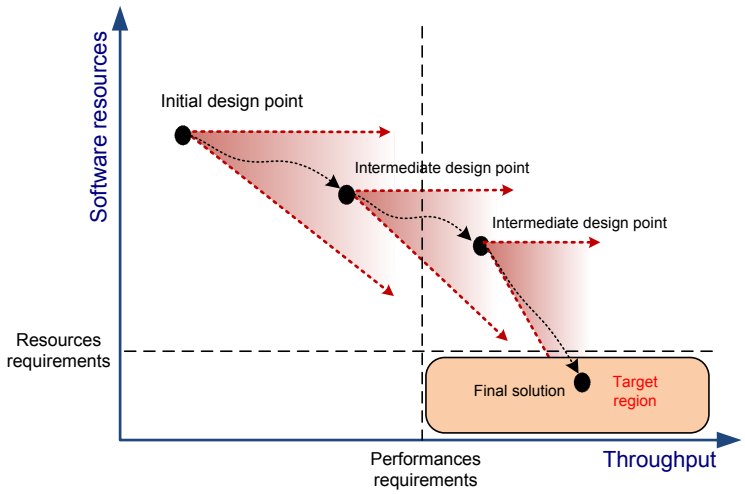


Figure 5.4: The design flow aims at guiding the designer in the exploration of the design space in order to reach a satisfactory implementation.

Thanks to the metrics extracted by the tools and the heuristics aiming at

partitioning, scheduling, detecting bottlenecks, profiling the execution, the designer has enough information and means for implementing quickly a solution for matching the application onto the target architecture. Each step in the exploration is either the refactoring the CAL program or the choice of a new partitioning and/or scheduling of the program. Chapter 6 illustrates how this design flow has been successfully applied to a real world application using the developed tools.

Chapter 6

Design case study: MPEG-4 SP

This chapter illustrates a concrete example of optimization and mapping of a real world application by applying the proposed methodology (Section 5.3). The chosen application is the MPEG-4 Simple Profile video decoder described by the standard document ISO/IEC 14496. Figure 6.1 depicts the initial version of the decoder (referred as *Serial-v0* in the text) written by Xilinx Inc.

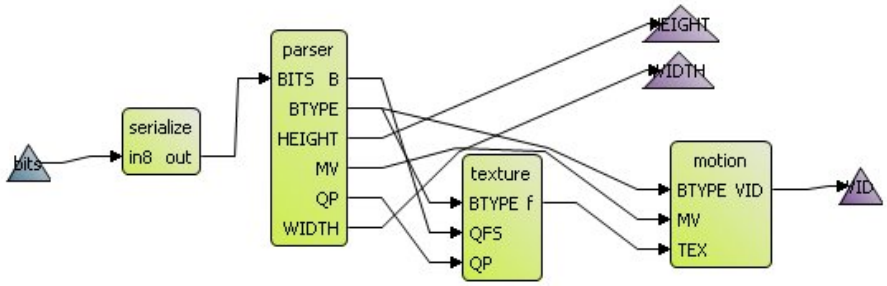


Figure 6.1: The initial version of the decoder.

Figure 5.2 provides a graphical representation of the different steps composing the design flow. At each iteration of the design flow due to the refactoring of the CAL program, the steps are:

Profiling of the actions The input sequence used during the design flow is Foreman in QCIF format (176x144) of 5 frames. The actions are profiled in terms of instructions using the CAL DYNAMIC ANALYZER.

Weighting The execution times of the actions are set according directly to the profiling of the actions, in terms of instructions.

Partitioning Two heuristics have been used to find the best partitioning: load balancing using the simulating annealing approach (Section 4.2.2) and the makespan minimization technique (Section 4.2.3).

Performance evaluation The Gantt chart is built in order to evaluate the performance of the current solution. It reconstructs the execution on 5 frames of the input Foreman sequence in QCIF format.

Refactoring The optimization strategy described in Section 3.3 has been applied.

Code Generation The C++ backend of the ORCC framework is used for generating the code either for profiling and for implementation.

Three target platforms have been considered:

Two cores Hewlett Packard 6710b Personal Computer, Intel®Core™2 Duo CPU T8300 at 2.4 Ghz.

Four cores Hewlett Packard ProLiant ML350 G6 Server, Intel®Xeon®CPU E5504 at 2 Ghz.

Eight cores Freescale QorIQ Platform, P4080 processor at up to 1.5 Ghz.

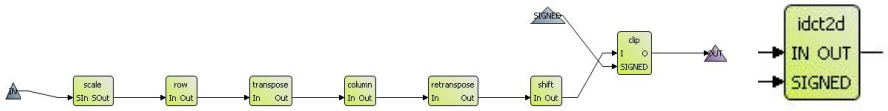
Five refactoring of the decoder have been performed: merging of actors for reducing implementation overhead (Section 6.1.1), splitting of actors for removing unnecessary dependencies (Section 6.1.2) and the removing of three bottlenecks thanks to the optimization strategy (Section 6.1.3).

6.1 The steps of the design space exploration

6.1.1 Improving the efficiency of actions

As discussed in Section 3.3.3.2.3, the way actors are written has an important impact on the implementation overhead. The initial version of the decoder (*Serial-v0*) contains a sequence of static actors that can be merged in order to reduce the scheduling overhead. The classification of the actors performed by the CAL STATIC ANALYZER outputs that the actors `scale, row, transpose, column, shift and clip` of the IDCT sub-network are static. Figures 6.2(a) and 6.2(b) illustrate the refactoring of the CAL program.

After partitioning and scheduling the refactored version of the decoder (*Serial-v1*), the resulting Pareto points in the design space defined by the throughput and resources criteria are plot in Figure 6.11. This refactoring improves the decoder when mapped on a small number of processors but the performance is still decreasing when mapped on more processors. The merging of the actors increases the efficiency of the IDCT computations but reduces the level of exposed parallelism, resulting in smaller performance on a higher level of processors. In order to improve the performance of the decoder on more than two cores, a major refactoring aiming at exposing more parallelism is needed.



(a) Initial version

(b) Refactored version

Figure 6.2: The static actors of the IDCT sub-network have been merged in order to increase the efficiency of the actors and reduce the implementation overhead.

6.1.2 Removing unnecessary dependencies

As explained in Section 3.3.3.1, unnecessary dependencies need to be removed from the CAL program in order to expose more parallelism. In MPEG-4, each QCIF frame of a video contains 99 macroblocks. A macroblock is composed of six blocks of 8x8 pixels: four blocks for the luminance (Y) and one block for each chrominance (U and V). The initial version of the decoder (Figure 6.1) has been implemented such that each 8x8 block composing a macroblock of the frame is processed serially by the different actors although the decoding of these Y, U and V macroblocks do not imply any dependencies. As a consequence, these macroblocks can be decoded in parallel, independently.

This refactoring consists in splitting the `Texture` and `Motion` sub-networks into three branches, each one processing a unique type of macroblocks. Additional actors need to be inserted in order to distribute the tokens output by the parser to the different branches and to reconstruct the decoded macroblocks for the display. Figure 6.3 depicts the resulting parallel video decoder.

The parallelization of the decoder has been eased by the modularity capabilities of the CAL language: the `Texture` and `Motion` sub-networks have been duplicated for the three branches Y, U and V. This refactoring allowed to reduce the trace critical path of the execution from 140.43×10^6 to 94.40×10^6, assessing that sequentiality has been removed. The trace critical path is expressed in terms of executed instructions as the profiling of the actions have been performed at this level.

After partitioning and scheduling this version of the decoder (*RVC-v0*), the resulting Pareto points in the design space defined by the throughput and resources criteria are plot in Figure 6.11. This refactoring improves the performance of decoder when mapped on more than three processors but because of the additional scheduling overhead due to the addition of many actors, the performance of the decoder with less than three processor remains low.

6.1.3 Refactoring of the most critical actions

The performance of the *RVC-v0* parallel version is better than the previous serial versions when mapped on more than three processors, but they are not improved

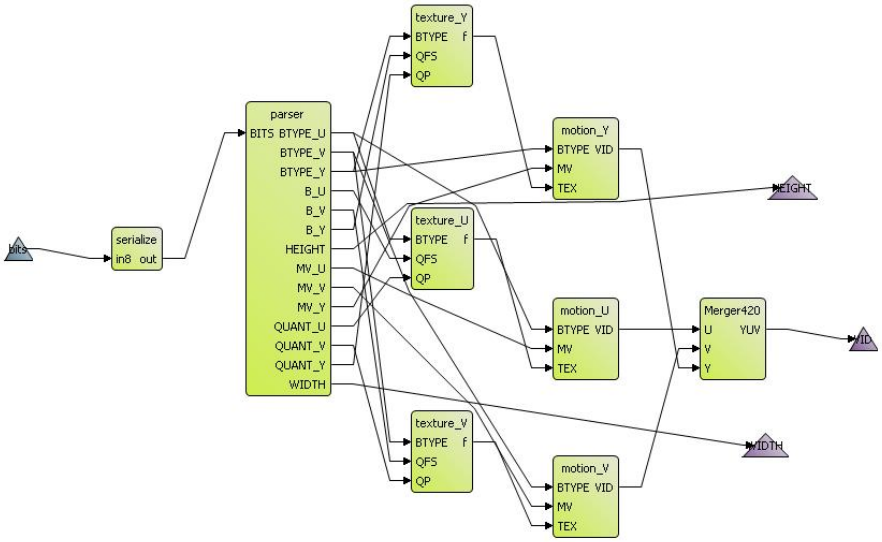


Figure 6.3: This version of the decoder (*RVC-v0*) results from the splitting of the *Serial-v1* version into three branches (Y, U and V) in order to expose more parallelism.

when increasing the number of processors. It means that some bottleneck is slowing down the decoder and the exposed parallelism of the platform cannot be exploited. The Critical Actions Detection algorithm (Section 3.3.2) based on the analysis of the trace critical path, one can highlight the actions that really affect the performance of the decoder. Thanks to this heuristic, three bottlenecks have been detected and removed, leading to better implementations.

6.1.3.1 Iteration 1: *Addressing* in motion compensation

Table 6.1 lists the critical actions of the *RVC-v0* version, extracted by CrossCal.

Critical actions	Necessary optimization
decoder/motion_Y/address/11$read_addr_Y	50
decoder/motion_Y/interpolation/3$other_Y	10
decoder/motion_Y/address/10$write_addr_Y	10
decoder/motion_Y/interpolation/3$other_Y	10
decoder/motion_Y/address/10$write_addr_Y	10

Table 6.1: List of the detected critical actions and their necessary optimization in the *RVC-v0* version.

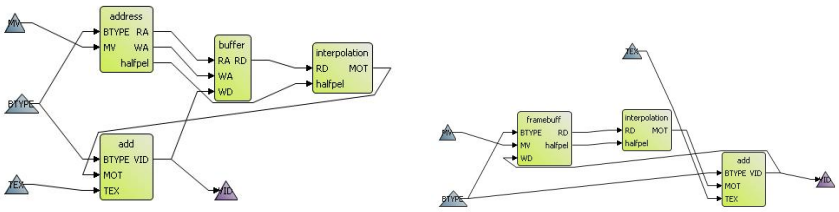
According to the results, the most critical action of the *RVC-v0* version is

the action `read_addr_Y` of actor `Address` in the motion compensation network (Y branch). The actor `address` exchanges a large amount of tokens with the `framebuffer` actor. Table 6.2 lists the five most expensive communications in terms of bits between actors. One can notice that the communication between the `address` and `framebuffer` actors are on the top of the lists. Thus, in order to get rid of the bottleneck, these actors are merged so that the communication costs are removed, resulting in more efficient computations. Figure 6.1.3.1 illustrates the refactoring. Thanks to the advantageous modularity capabilities of CAL, the `address` and `framebuffer` actors in the U and V branches have been also replaced by the merged version.

After partitioning and scheduling this version of the decoder (*RVC-v1*), the resulting Pareto points in the design space defined by the throughput and resources criteria are plot in Figure 6.11. The performance has been incredibly improved when mapped on more than three processors and small improvement is observed on two processors.

Source (Port)	Destination	Size (Mbits)
motion_Y/address (RA)	motion_Y/buffer	3.080
motion_Y/address (WA)	motion_Y/buffer	3.042
Merger420 (YUV)	display	1.521
parser/blkexp (OUT)	parser/splitter_420_B	1.242
motion_Y/buffer (RD)	motion_Y/interpolation	1.155

Table 6.2: Top 5 most expensive communications in the *RVC-v0* version.



(a) Initial version (b) Refactored version

Figure 6.4: The bottleneck of the version *RVC-v0* has been removed by merging the `framebuffer` and the `address` actors in the motion compensation network.

6.1.3.2 Iteration 2: *Interpolation* in motion compensation

By applying again the Critical Actions Detection algorithm to this new version of the program (*RVC-v1*), new critical actions are detected, listed in Table 6.3.

According to the results, the most critical action is the action `start` in actor `Interpolation_halfpel` belonging to the `Motion` network (Y branch). In *RVC-v1*, for each 8x8 block of a macroblock, the interpolation actor fires the following

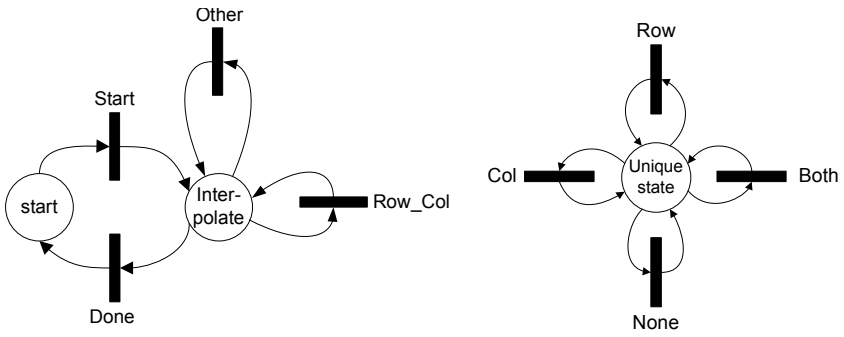
Critical actions	Necessary optimization
decoder/motion_Y/interpolation/3$other_Y	60
decoder/texture_Y/idct2d/1$inter_Y	10
decoder/motion_Y/interpolation/3$other_Y	20
decoder/texture_Y/idct2d/1$inter_Y	10
decoder/motion_Y/interpolation/3$other_Y	10

Table 6.3: List of the detected critical actions and their necessary optimization in the *RVC-v1* version.

sequence of actions:

$$\{start, row\_col \times 9, \{row\_col, other \times 8\} \times 8, done\} \tag{6.1}$$

Thus, for each 8x8 block, 83 actions are fired. The interpolation actor can be refactored such that for each 8x8 block, this sequence of action is replaced by only one unique action firing. Figure 6.1.3.2 illustrates the modification of the Finite State Machine. This refactoring has been done by colleagues in Ericsson Research.



(a) Initial version (b) Refactored version

Figure 6.5: The bottleneck of the *RVC-v1* version has been removed by rewriting the `Interpolation_halfpel` actor in a more efficient way.

Tables 6.4 and 6.5 compare the initial and refactored versions of the interpolation actor in terms of calls and total computations. In the new version, actions are bigger and called less, resulting in a great improvement in terms of total computations. After partitioning and scheduling this new version of the decoder (*RVC-v2*), the resulting Pareto points in the design space defined by the throughput and resources criteria are plot in Figure 6.11. The performance is largely improved on any platform.

	Y			U or V		
Action	**Calls**	**Tot.**	**Avg.**	**Calls**	**Tot.**	**Avg.**
other	111 376	50 859 834	456	30 272	13 860 456	457
row_col	29 592	83 832 41	283	8 041	22 779 68	283
done	1 740	120 060	69	473	32 637	69
start	1 741	240 258	138	474	65 412	138
Total	144 449	59 603 393	-	39 260	16 236 473	-

Table 6.4: Profiling of the actions of the `Interpolation_halfpel` actor before refactoring.

	Y			U or V		
Action	**Calls**	**Tot.**	**Avg.**	**Calls**	**Tot.**	**Avg.**
both	224	1 337 280	5 970	107	638 790	5 970
col	324	1 492 344	4 606	71	327 026	4 606
row	132	616 440	4 670	34	158 780	4 670
none	1 060	4 043 900	3 815	261	995 715	3 815
Total	1 740	7 489 964	-	473	2 120 311	-

Table 6.5: Profiling of the actions of the `Interpolation_halfpel` actor after refactoring.

6.1.3.3 Iteration 3: *Inverse Scan* in texture decoding

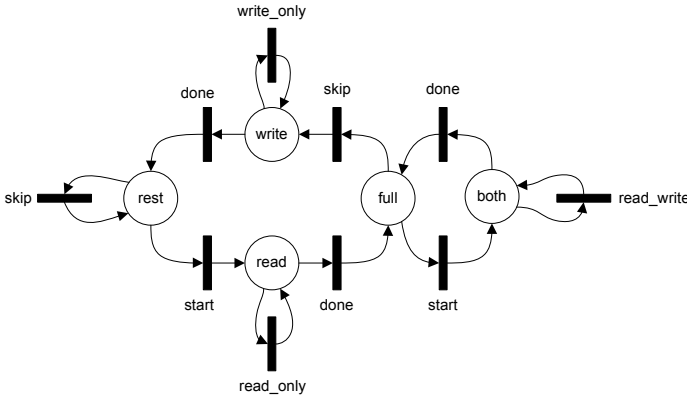
By applying again the Critical Actions Detection algorithm to this new version of the program (*RVC-v2*), new critical actions are detected, listed in Table 6.6.

Critical actions	Necessary optimization
decoder/texture_Y/idct2d/1$inter_Y	40
decoder/texture_Y/IS/5$read_write_Y	10
decoder/texture_Y/idct2d/1$inter_Y	20
decoder/texture_Y/IS/5$read_write_Y	10
decoder/texture_Y/idct2d/1$inter_Y	10

Table 6.6: List of the detected critical actions and their necessary optimization in the *RVC-v2* version.

According to the results, the most critical action is the action `inter_Y` in the IDCT actor in the texture decoding network (Y branch). IDCT has been already the scope of an optimization (Section 6.1.1). Thus, the designer can focus on the second critical action which is the action `read_write_Y` in the `IS` actor (Inverse Scan). Figures 6.6(a) and 6.6(b) illustrate the refactoring of the actor. All the actions (except `skip`) have been merged into a single larger action.

Tables 6.7 and 6.8 compare the initial and refactored versions of the inverse scan actor in terms of calls and total computations. In the new version, ac-



(a) Initial version

(b) Refactored version

Figure 6.6: The bottleneck of the *RVC-v1* version has been removed by rewriting the `IS` (Inverse Scan) actor in a more efficient way.

	Y			U or V		
Action	**Calls**	**Tot.**	**Avg.**	**Calls**	**Tot.**	**Avg.**
read_write_Y	58 244	22 598 704	388	6 300	2 444 400	388
write_only_Y	17 325	4 322 241	249	1 197	299 124	249
read_only_Y	17 388	4 043 025	232	1 197	277830	232
done_IS_Y	1 475	195 646	132	138	18458	133
start_Y	1 201	136 914	114	120	13 680	114
skip_Y	874	88 274	101	415	41 915	101
Total	96 507	31 384 804	-	9 367	3 095 407	-

Table 6.7: Profiling of the actions of the `IS` (Inverse Scan) actor before refactoring.

	Y			U or V		
Action	**Calls**	**Tot.**	**Avg.**	**Calls**	**Tot.**	**Avg.**
rw	1 199	3 894 352	3 248	119	386 512	3 248
skip	874	88 274	101	413	41 713	101
Total	2 073	3 982 626	-	532	428 225	-

Table 6.8: Profiling of the actions of the `IS` (Inverse Scan) actor after refactoring.

tions are bigger and called less, resulting in a great improvement in terms of total computations. After partitioning and scheduling this new version of the decoder (*RVC-v3*), the resulting Pareto points in the design space defined by the throughput and resources criteria are plot in Figure 6.11. The performance is improved when mapped on more than four cores.

6.1.3.4 Final version

The AnalytiCal tool allows browsing and visualizing the metrics characterizing this resulting version of the decoder. Figure 6.7 is a screenshot of the tool: the size of the nodes (actors) represent the total computational load of actors, the blue nodes are the actors belonging to the trace critical path and the size of the edges represents the size of the data transfers.

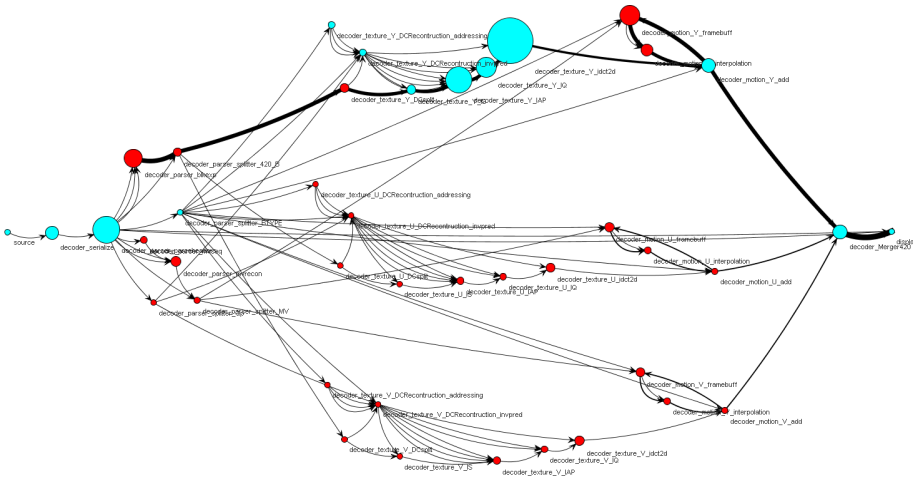


Figure 6.7: The AnalytiCal tool allows visualizing graphically the metrics extracted from Cal programs. This screenshot corresponds to the *RVC-v3* version.

By applying again the Critical Actions Detection algorithm to this new version of the program (*RVC-v3*), the new critical actions are detected and listed in Table 6.9.

Critical actions	Necessary optimization
decoder/texture_Y/idct2d/1$inter_Y	50
decoder/texture_Y/idct2d/0$intra_Y	10
decoder/texture_Y/idct2d/1$inter_Y	10
decoder/texture_Y/idct2d/0$intra_Y	30

Table 6.9: List of the detected critical actions and their necessary optimization in the *RVC-v3* version.

6.1.4 Searching for efficient partitioning solutions

The optimizations being over, one can search now for efficient partitions of the decoder on the three target platforms.

Figures 6.8 and 6.9 illustrate how an efficient partition has been found.

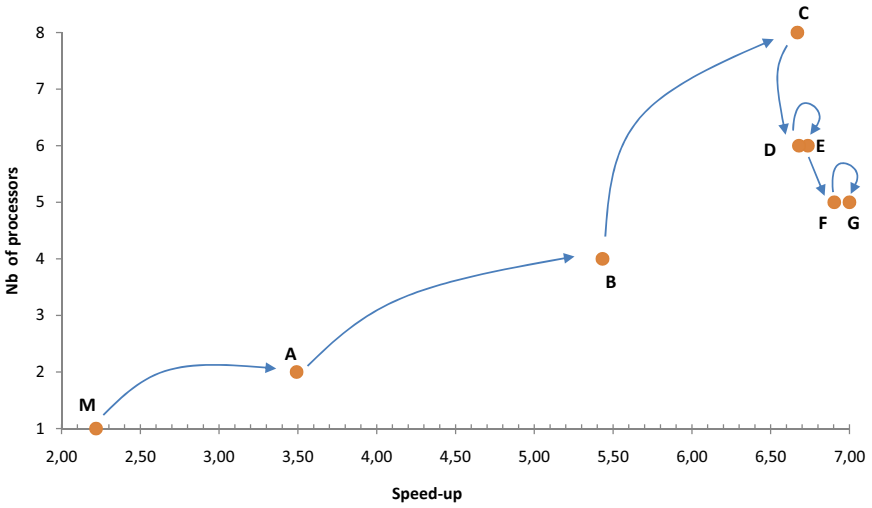


Figure 6.8: Different points in the design space, corresponding to different partitioning solutions.

M → **A** Actors with the largest computational load (i.e. texture decoding and motion compensation of the luminance branch, respectively named TY and MY) are assigned to a second core.

A → **B** Using 4 cores, motion compensation Y and texture decoding Y are assigned to two different cores. Motion compensation and texture decoding of the U and V branches are assigned to a single core, all together. The rest of the actors are on another core.

B → **C** Using 8 cores, the whole decoder is pipelined. Each of the following sets are assigned to one core: {source, parser}, {texture decoding Y}, {motion compensation Y}, {texture decoding U}, {motion compensation U}, {texture decoding V}, {motion compensation V} and {merger, display}.

C → **D** In order to save cores, motion compensation U and texture decoding U are assigned to the new core, as well as motion compensation V and texture decoding V.

D → **E** In order to better load balance the partitions in terms of computational load, the IDCT actor in the luminance branch is assigned to a single core and {texture decoding U, motion compensation U, texture decoding V, motion compensation V} are assigned to the same processor. The results shown in AnalytiCal (Figure 6.7) serve for the load balancing.

E → **F** In order to continue saving resources, the merger and the display actors are assigned to the same core as texture decoding U/V and motion compensation U/V.

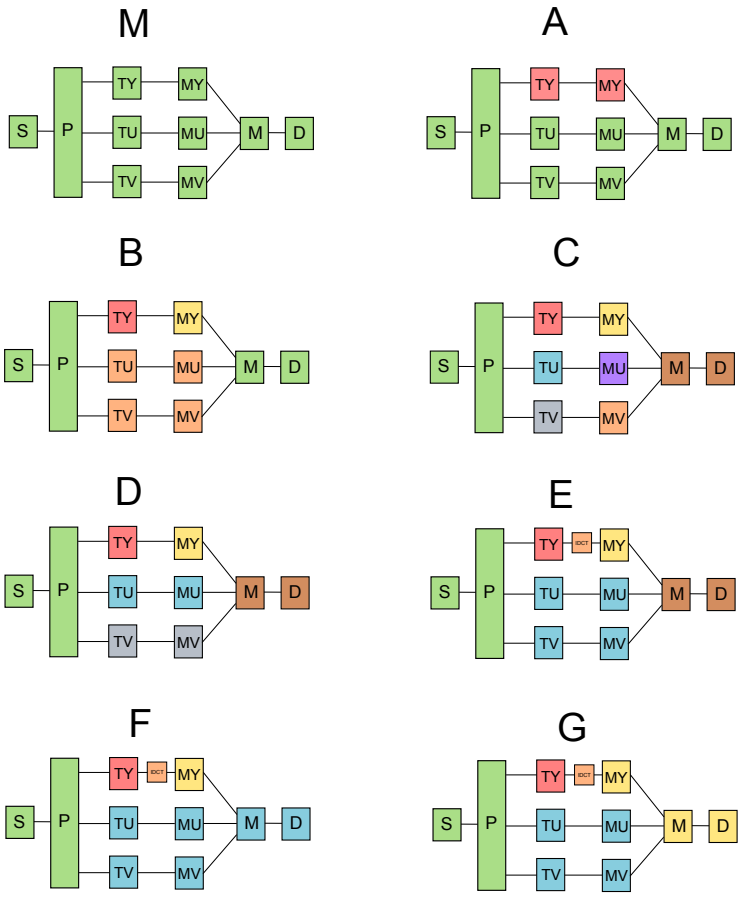


Figure 6.9: The different partitions while searching for an efficient partitioning.

F → **G** To re-adjust the load balancing between partitions, the merger and display actors are now attached to the same core as texture decoding Y and motion compensation Y. It is the partition that offers the best performance.

The performance of the resulting CAL program (*RVC-v3*) is shown in Figure 6.11.

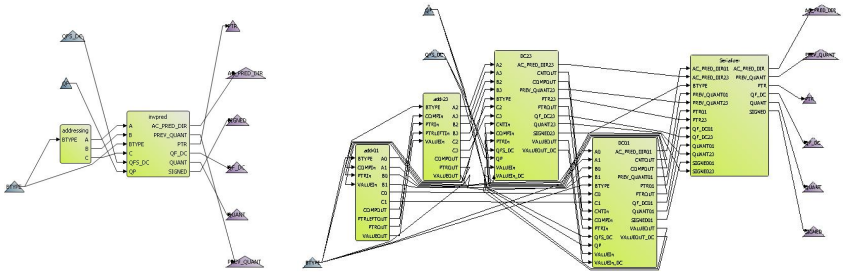
6.1.5 Splitting for better load balancing

Table 6.10 illustrates how the computations are distributed among the different clusters of actors in the *RVC-v3* version. One can notice that the Y branch contains half of the computations. Thus, its parallelization would be beneficial for speeding-up this part of the decoder. The `DCReconstruction` network has been

parallelized by splitting the `addressing` and `invpred` actors. Figures 6.10(a) and 6.10(b) illustrate this refactoring. The IDCT actor (only in the Y branch) has been replaced by its initial version, i.e. the pipelined version (Figure 6.2(a)). The obtained version of the decoder is referred to *RVC-v4*. Figure 6.13 compares the performance of the *RVC-v3* and *RVC-v4* versions and Section 6.2 discusses the results.

	Parser	**Y branch**	**U branch**	**V branch**	**Merger**
Total	53 119 469	118 621 592	16 202 189	16 422 733	9 419 697
Total (%)	25 %	55 %	8 %	8 %	4 %

Table 6.10: Distribution of the computations among the different clusters of actors of the *RVC-v3* version.



(a) Initial version (b) Refactored version

Figure 6.10: Parallelization of the `DC Reconstruction` network.

6.2 Results and discussion

The different techniques presented in Section 3.3 drove the design space exploration by:

- Removing scheduling overhead by merging actors (Section 6.1.1)
- Splitting actors to expose more parallelism (Section 6.1.2)
- Shortening the trace critical path:
  - By optimizing actions bodies (Section 6.1.3.2)
  - By merging actors (Section 6.1.3.1)

Efficient design space exploration Six versions of the decoder, three different platforms with different architectures and level of parallelism (from one to eight processors), the proposed approach with the supporting tools allows exploring a large design space as shown in Figure 6.11. The effort for exploring

such a design space is small compared to what it would have been if using traditional approaches.

The different refactoring have led to important improvements of the performance of the implementation of the MPEG-4 decoder on several parallel platforms. Figure 6.11 illustrates the performance of the different versions of the decoder. The results have been obtained using a QCIF Foreman input sequence.

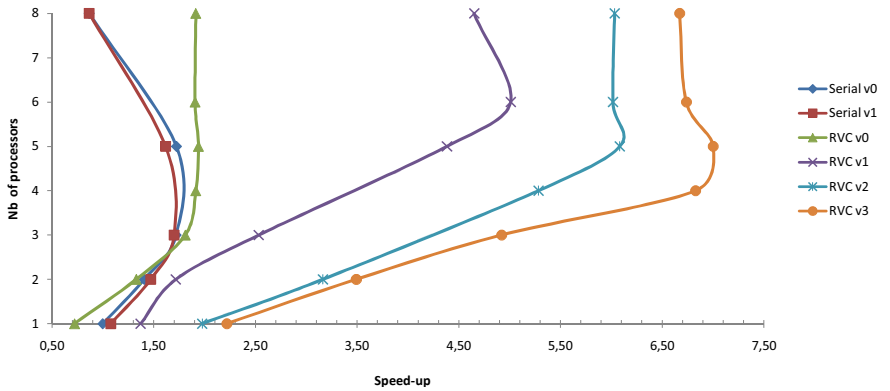


Figure 6.11: Performance of the different versions of the MPEG-4 SP decoder on the different platforms. The points corresponds to Pareto points.

The first refactoring (*Serial-v0* to *Serial-v1*) improved slightly the decoder when mapped on a small number of processors. The actors of the IDCT being merged, it resulted in more efficient actions, removing the scheduling overhead and communication costs due to individuals actors. The second refactoring (*Serial-v1* to *RVC-v0*) slowed down the decoder when mapped on a small number of processors because the splitting of actors introduced overhead but speeded-up the decoder when mapped on more than five processors because the removing of the unnecessary dependencies enabled the parallelism of the application to be really exploited by the platform. The third refactoring (*RVC-v0* to *RVC-v1*) improved a lot the performance of the decoder because a first serious bottleneck has been removed. The addressing process for the frame buffer in motion compensation implied too much communications compared to computations. Thus, the merging of the two actors allowed increasing this ratio between computations and communications. The fourth and fifth refactoring (respectively from *RVC-v1* to *RVC-v2* and from *RVC-v2* to *RVC-v3*) led also to better performance thanks to the removing of the bottlenecks, based on the same principles as for the third refactoring.

Validation of the optimization strategy Figure 6.12 compares the total computations and the trace critical path of the execution for each version of the decoder. The trace critical path values and the total number of compu-

tations can be considered respectively as the lower and upper bounds on the performance. If the application is entirely mapped on a single core, the execution time of the execution evolves directly with the total number of computations and cannot be larger. At the contrary, if each actor of the program is mapped on a single core, the execution time of the execution corresponds directly to the trace critical path - without considering the communications costs - and cannot be smaller. This graph shows the potential range of expected performance.

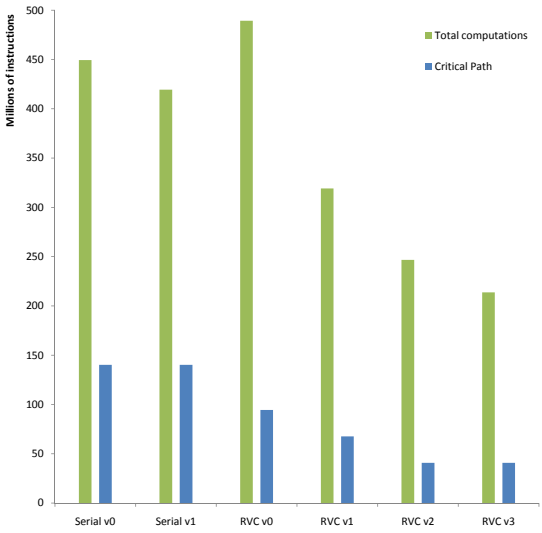


Figure 6.12: The trace critical path and the total number of computations can be considered as the lower and upper bounds on performance.

It should be noticed that each refactoring leading to better performance has also a smaller trace critical path. Thus, these results validate the proposed approach (Section 3) which consists in guiding the designer in the refactoring of the CAL program thanks to the minimization of the trace critical path.

Fine-grain parallelism is hardly exploitable In case of software systems, fine-grained parallelism is hardly exploitable by platforms because the communication cost takes the advantage on the speed-up gained by the parallelization. The refactoring presented in Section 6.1.5 confirms the idea. Figure 6.13 compares the obtained performance of the *RVC-v3* and *RVC-v4* versions.

Parallelization does not necessary mean speed-up on software platforms. The difference between the parallelization of the version *Serial-v1* (which speeds-up the processor) and this parallelization (which does not) lies in the level of granularity at which the parallelism is exposed. In the first case, the new parallelism was at the macroblock level whereas in the second case, was at a much deeper level, which makes it impossible to exploit with the current status

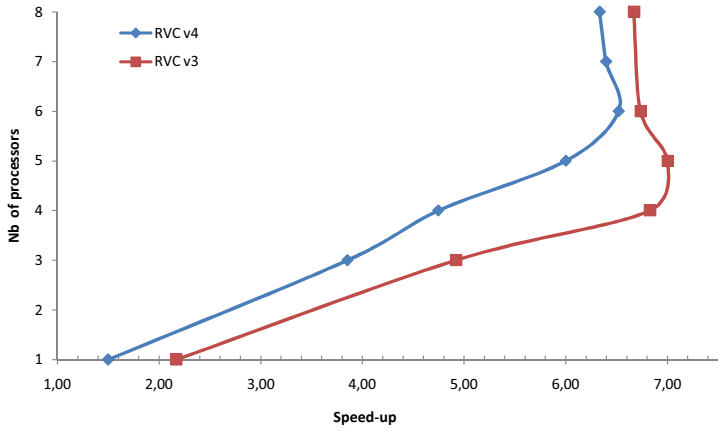


Figure 6.13: Fine-grain parallelism is hardly exploitable: the parallelization of the Y branch does not improve performance on software platforms.

of the tools and the given platforms. However, the parallelization of the Y branch allows the platform to exploit more parallelism. For *RVC-v3*, the best performance is found when the decoder is mapped on five processors whereas for the *RVC-v4*, six processors are necessary. But the cost of the parallelization is higher than the gain from the parallelization. The cost is mainly due to communications.

This experiment highlights the fact that communication cost due induced by any splitting must be carefully analyzed. The level of granularity needs must be superior to a given threshold so that the parallelization process leads to more efficient implementations. This level may change according the the future optimizations of the code generations for which their development consists in minimizing the overhead due to the handling of the actions and their synchronization. In hardware, the threshold of the level of granularity may be much lower than the one for software systems. The reason is that the exploitable parallelism by hardware platforms is much higher than software systems because all the actors can effectively run concurrently with each others.

The choice of the input sequence As the design space exploration is mostly based on dynamic analysis of the CAL programs, the choice of the input sequence is important. It has been chosen to take as input sequence the foreman sequence because it contains both a static background and a moving object. A non-statistically meaningful analysis does not provide the expected results.

For example, if the profiling of the actions is limited to the INTRA pictures and the design space exploration is performed from these metrics, the performance may be different from the one expected because the INTER pictures have not been analyzed and as the INTER pictures are generally dominant in video sequences, it will lead to non-efficient implementation. The optimal solu-

tion computed from the metrics extracted from the analysis of INTRA pictures has a good chance not to be the same when dealing with a complete sequence comprising also INTER pictures.

Algorithmic exploration: a key point Dataflow languages have one important benefit that cannot be ignored: their ability to embed computations in independent entities that can be run on a set of independent processing units. In case of CAL, the embedding of each actor coupled with a graphical visualization of the program reduces parallel programming pain and contributes in bringing parallel programming to the average programmers. As a consequence, writing a parallel program is almost as easy as thinking of some code for filling the sketched boxes that any designer draws with a pen on a sheet of paper to model the application.

The easiness for conceiving parallel programs should enable developers to really focus on designing optimal applications at an algorithmic point of view. The MPEG-4 video decoder case study confirmed that the way applications are written has important impacts on the implementation performance. Consequently, design space exploration at an algorithmic level (i.e. CAL level) is primordial for achieving efficient designs. But one needs to define the *optimality* of a program at the pure algorithmic level. This optimality is closely linked to the target platform. Algorithmic efficiency is the process aiming at reducing the completion time of the execution, the resource consumption, and in case of parallel programming, exposing the right level of parallelism that can be fully exploited by the platform. The procedure for obtaining optimal programs is not trivial and should be investigated in order to find metrics and heuristics that efficiently guide designers in the design space exploration.

CAL has this specificity to allow certain degrees of freedom in the application specification. A special type of non-determinism (time-dependency) can be introduced in the specification without affecting the global deterministic behavior of the application. This is the case for the frame buffer actor in the MPEG-4 SP decoder. This degree of freedom should be better exploited at the implementation stage, according to the target platform. This voluntary underspecification of the application - which does not affect its accuracy - is a source of optimization.

Does CAL really unify hardware and software worlds? The hardware and software worlds are known to be really different because of their underlying model of computations: the first is massively parallel while the second is sequential. With the advent of the muticore era, the software world is now faced to the parallelization of the applications because developers can no longer profit from the incredible constant increase of the performance of the sequential general purpose processors. The consequence is that these two worlds are now getting closer.

The time has maybe come to find a new formalism for unifying the specification of applications in these two worlds. The CAL language seems to be good

candidate for being the base of this unification. It has been already proven that it really suits for hardware synthesis [128] for which the generated code outperforms the manual implementation. Now, it needs to be proven that efficient software code can be generated from CAL. This objective will be quickly reached as the optimizations of the existing code generators are going well.

However, the question of a unique and optimal representation of an application remains opened. Currently, a CAL program leading to an optimal solution in hardware does not imply that it is also the optimal solution in software because the level of parallelism that can be exploited by the software and hardware platforms are not comparable. CAL offers valuable modularity capabilities for interchanging actors in a very seamless way. An actor optimized for software can be substituted by its hardware version (and vice-versa) very easily as far as the input and output tokens requirements are compatible. This kind of operation can even be automatic according to the processing unit to which the actor is assigned. The compiler can choose the appropriate representation of an actor with respect to the platform. In case of hardware platforms, a tool is currently being developed at the laboratory for automatically create a pipeline structure of the most complex actions by means of automatic CAL transformations.

The approach adopted by the C-to-Gates tool, i.e. converting sequential code onto massively parallel platforms, has been largely criticized. But, one can ask - in the same way - if CAL does not have the similar problem with software implementation, the other way round. Dataflow programming is closer to the hardware model of computation in which each actor can run concurrently. Where parallelization is the main problem in the C-to-Gates approaches for mapping sequential programs on hardware, sequentialization becomes a crucial issue for mapping massively parallel CAL programs on less parallel platforms. As a consequence, this issue needs to be deeply investigated. Hopefully, the advent of the parallel era increases the level of exploitable parallelism in the software world and brings closer the hardware and software worlds, reducing the importance of the sequentialization issue.

Keeping a close link between specification and implementation through code generators Mapping seamlessly parallel applications onto heterogeneous platforms is an incredible feature for today's manufacturers. The refactoring of high level programs is valuable only if it has positive effects on the performance of the implementation. Estimating precisely the gain of the refactoring without implementing it on the real platform is a crucial issue because heuristics aiming at guiding the designers through the design space are based on this performance evaluation.

The accuracy of the performance evaluation is possible only if a close link between the CAL program and its implementation is maintained through code generators. How to evaluate the impact of any refactoring of the program if this latter can lead to different implementation solutions? There must be a one-to-one correspondence between the high level program and its implementation. This close link allows also attaching to the high level program low level metrics

that reflect its real behavior on target platform. These realistic metrics are necessary to have a precise performance evaluation and thus an efficient design space exploration.

Chapter 7

The shift of paradigm for systems specification: the case of Reconfigurable Video Coding

The proposed design space exploration methodology using dataflow programming and CAL language has been shown to be efficient but it supposes the existence of the CAL programs. Translating sequential reference software into a dataflow representation is still a hard task and time-consuming task and this is the reason why there was an important effort by several teams worldwide to push a new standard for changing the paradigm at the specification level.

The emergence of the multicore technology allows the development of platforms capable of supporting multiple coding standards and multiple profiles. These standards may have commonalities at the level of the coding tools but there is currently no explicit way to exploit such commonalities at the level of the specification nor at the level of implementation.

The lack of capabilities for handling parallelism and for exploiting the commonalities between codecs of the traditional methodologies led to the development of the Reconfigurable Video Coding (RVC) standard [154–156] by the Moving Picture Experts Group (MPEG) – a working group of experts formed by ISO and IEC – in order to change the way the video coding technology is specified.

The RVC standard provides a framework which aims at deploying the video coding technology, at easing the implementation of complex video coding application and at promoting the adoption of a more adapted paradigm for the specification, design and implementation of applications which target parallel platforms. Modularity, flexibility and portability are some characteristics that make RVC a very attractive approach.

7.1 Overview

The novelty of the RVC standard lies in its ability to specify new codecs by combining blocks called Functional Units (FUs). Unlike previous video coding standard, it does not specify rigidly new video coding algorithms but it allows the combination of a multitude of video coding tools (FUs) defined in Video Tools Libraries (VTLs) to specify new codecs in a very flexible and modular way. Obviously, it eases drastically the design of multi-standards video coding application by enable reuse of the FUs across the different standards. In the context of RVC, there are two standard documents:

ISO/IEC 23001-4 (or MPEG-B part 4 [21]) describes the whole framework and the internal processes and specifies the languages

ISO/IEC 23002-4 (or MPEG-C part 4 [134]) specifies the elementary video coding tools (the so-called Functional Units) of the standard MPEG Video Tool Library (VTL).

The principle of RVC is illustrated in Figure 7.1. An RVC decoder is built with the interconnection of FUs. The decoder inputs the Decoder Description: it is the description of how the FUs are connected together to build the decoder. Each FUs can be instantiated either from the standard MPEG Video Tool Library (VTL) or from a proprietary VTL. ISO/IEC23002-4 specifies the functionality of each FU and clearly describes their input/output tokens requirements. Thanks to this flexibility, decoders specified within the RVC standard (i.e. conform to ISO/IEC23001-4) can be of different types:

Type-1 The decoder is built only by using FUs in the MPEG VTL. Thus, this decoder conforms also to ISO/IEC23002-4.

Type-2 The decoder is built using FUs from the MPEG VTL and proprietary VTLs. Thus, this decoder does not conform to ISO/IEC23002-4.

Type-3 The decoder is built using only FUs from proprietary VTLs. Thus, this decoder does not conform to ISO/IEC23002-4.

The traditional approaches using imperative languages are not appropriate for supporting such modular specifications of video codecs. CAL is the language that has been chosen for defining Functional Units in the VTLs.

Figure 7.2 describes more precisely the different components and processes within the RVC standard. The top level precises the normative procedure, how the Abstract Decoder Model (ADM) is built and the bottom part explains the non-normative part, how an RVC decoder can be implemented from the ADM.

7.1.1 Normative part

A decoder defined within RVC can be distinguished from decoders rigidly specified by traditional video coding standards because the description of the decoder, called Decoder Description, is sent along with the encoded data. This Decoder Description fully specifies the composition of the decoder and the structure of the incoming bitstream thanks to:

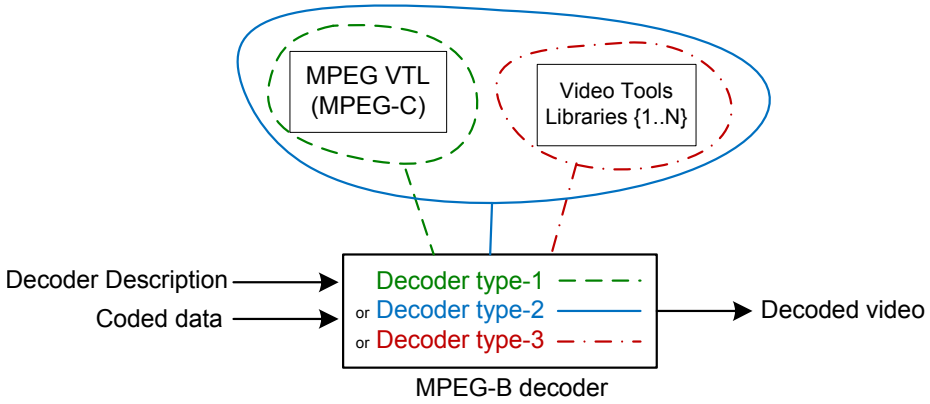


Figure 7.1: An RVC decoder is specified by the interconnection of Functional Units (FUs) instantiated from the standard MPEG and/or from proprietary Video Tools Libraries (VTLs).

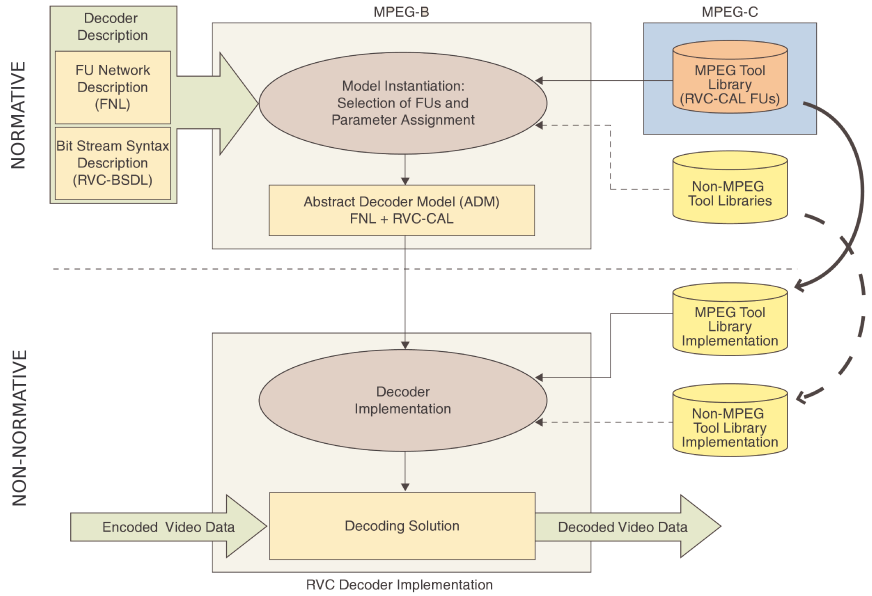


Figure 7.2: Illustration of the normative and non-normative parts of the Reconfigurable Video Coding framework.

The Bitstream Syntax Description (BSD) which describes the structure of the bitstream by listing all the syntax elements (with type and size) contained in the bitstream. The BSD is specified in the RVC-BSDL language

(Section 7.1.3.3).

The FU Network Description (FND) which describes the network of the coding tools (FUs) and the parameters necessary to instantiate the different FUs of the decoder. The FND is specified in the FU Network Language (FNL) (Section 7.1.3.2).

The aim of the normative process is to specify the Abstract Decoder Model (ADM) which is the normative behavioral model of the decoder. The ADM is built thanks to the Decoder Description and the FUs defined in Video Tools Libraries (MPEG or proprietary).

Video Tool Library (VTL) The Functional Units (FUs) of the VTL are specified using a subset of the CAL language called RVC-CAL (Section 7.1.3.1). The chosen level of granularity for the specification of the Functional Units is very important and must be such that it allows the reuse of the different video coding tools (FUs) in future codecs implementations. If FUs are defined with a too coarse granularity, it will result in too large modules which will be unusable in new codecs because it does not perfectly fit the required functionality. At the contrary, if the granularity is too fine, the number of modules explodes and the configuration of the decoder becomes too complex because of the numerous interconnections of FUs, even obscuring the desired high-level and modular description of the RVC codec.

7.1.2 Non-normative part

The implementation process of the decoding solution from the ADM (defined by the normative part) is non-normative. The Decoder Description establishes the interconnections between the FUs. These FUs are specified in RVC-CAL into the MPEG and proprietary VTLs and need to be converted into components described in native implementation code, C/C++ for targeting software platforms or Verilog/VHDL for hardware platforms. Existing software and hardware code generators [128, 135] translate directly the FUs defined in RVC-CAL into native implementation code. The formalism chosen in RVC guarantees that as far as the implementation of the FUs conforms to the description of the VTLs in terms of input/output tokens requirements, the resulting implementation will be consistent with the ADM.

Hence, as far as the input/output tokens requirements at the FU level is respected, nothing prevents the designer to build different versions of the FU in implementation code. This modularity is a very valuable feature for the design space exploration when implementing any RVC decoder onto a wide range of target platform. As discussed in the previous chapters, the future challenge of digital systems design is how to implement efficient complex applications onto massively parallel platforms. In RVC, designers can build different versions of each FU, exposing more or less parallelism to target a maximum range of platforms. A less-parallel implementation of a FU better suits software platforms because of the restricted number of available processors. At the contrary, a

massively parallel implementation better suits hardware platforms which can fully exploit the exposed parallelism. Thus, special VTLs targeting different type of platforms (multicore, FPGA, GPU, etc.) can be built in order to make profit of the flexibility offered by RVC to reach efficient implementation of RVC codecs.

7.1.3 Languages defined within RVC

In order to ensure the generation of efficient implementations of the FUs of the VTL by means of code generators, only a subset of CAL has been standardized within RVC. Having a well-defined subset of a language is necessary to ensure the full compatibility of the language with the underlying tools. This subset is called RVC-CAL and specified in Section 7.1.3.1. This subset has only small differences with CAL. For the same reason, BSDL has been restricted and extended with new constructs in order to guarantee that this subset has only the required features to describe the syntax so that parsers can be generated efficiently. RVC-BSDL is described in Section 7.1.3.3.

7.1.3.1 RVC-CAL

RVC-CAL is a large subset of the CAL language and is specified in MPEG-B (ISO/IEC 23001-4 [21]). The main difference between CAL and its subset is that RVC-CAL only deals with fully typed actors. Additional minor restrictions on the CAL language constructs are necessary in order to have efficient hardware and software code generations without changing the expressivity of the algorithm. For an in-depth description of the subset of the language, the reader is referred to the Annex C of ISO/IEC standard [21].

7.1.3.2 FU Network Language

The FU Network Language is an XML dialect specified in MPEG-B (ISO/IEC 23001-4 [21]) and describes the network of FUs composing the ADM, their parametrization and their interconnections. It corresponds to the FU Network Description (FND). Figure 2.3 shows an example of FND. A graphical editing framework called Graphiti [136], a generic graph editor under BSD license created by IETR/INSA Rennes, is available to create, edit, save and display networks.

7.1.3.3 RVC-BSDL

RVC-BSDL is an XML dialect specified in MPEG-B (ISO/IEC 23001-4 [21]) and is a subset of the BSDL language [137]. RVC-BSDL is the language aiming at describing the Bitstream Syntax Description (BSD) which is an XML schema. This schema describes the structure of the incoming bitstream compatible with the decoder. Listing 7.4 is a piece of the BSD describing the incoming bitstream for the Simple Profile of the MPEG-4 video decoder.

```
<XDF name="Sum">
   <Port kind="Input" name="In"/>
   <Port kind="Output" name="Out"/>
   <Instance id="add"/>
   <Instance id="z">
      <Class name="Z"/>
      <Parameter name="v">
         <Expr kind="Literal" literal-kind="Integer" value="0"/>
      </Parameter>
   </Instance>
   <Connection dst="add" dst-port="A" src="" src-port="In"/>
   <Connection dst="add" dst-port="B" src="z" src-port="Out"/>
   <Connection dst="z" dst-port="In" src="add" src-port="Out"/>
   <Connection dst="" dst-port="Out" src="add" src-port="Out"/>
</XDF>
```

Figure 7.3: FNL representation of the *Sum* network.

```
<xsd:complexType name="VideoPacketHeaderType">
 <xsd:sequence>
  <xsd:element name="resync_marker" type="ResyncMarkerType"/>
  <xsd:element name="macroblock_number" type="MBNumberType"/>
   <xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape!=&binaryOnly;">
    <xsd:element name="quant_scale" type="VOPQuantType"/>
   </xsd:sequence>
   <xsd:sequence minOccurs="0" bs2:if="$m4v:video_object_layer_shape=&rectangular;">
    <xsd:element name="header_extension_code" type="bs1:b1" bs0:variable="true"/>
   </xsd:sequence>
   <xsd:element name="VPHeaderExtension" minOccurs="0" bs2:if="$header_extension_code=1"
      type="VPHeaderExtensionType"/>
 </xsd:sequence>
</xsd:complexType>
```

Figure 7.4: A piece of the Bitstream Syntax Description of the MPEG-4 SP decoder.

For the sake of clarity, it is possible to define groups of elements of syntax. For example, the group *VideoPacketHeaderType* is composed of a sequence of several elements of syntax: a first element *resync_marker* comes, followed by a second element *macroblock_number*, followed by a third element *quant_scale only* if the element *video_object_layer_shape* previously decoded is of not of type *binaryOnly*, and so on. The RVC-BSDL subset contains only the minimum set of necessary constructs (XML elements and attributes) like *bs2:if, bs2:ifNext, minOccurs, bs2:nOccurs* (non-exhaustive list) in order to be able to fully describe complex bitstreams. The reader is refered to [21] for further details.

7.2 Promises

The relentless increase of the codecs complexity poses a new challenge in their implementation onto parallel platforms. Traditional methodologies lack flexibility to efficient map complex codecs onto platforms with different levels of parallelism, different native implementation codes and different technologies. A shift of paradigm is needed at the specification level and RVC provides a solution to this challenge.

"Data dependencies were not considered important factors during standardization in the sequential processor age, however, now they can become the real obstacle for an efficient multicore implementation". [154]. The adoption of the RVC standard is intended to have a significant impact in the field of multimedia systems design by proposing a new paradigm capable providing portable and scalable codecs specifications. The direct consequence would be the larger deployment of video coding technology onto a wide range of parallel platforms.

7.2.1 Towards portable and scalable parallelism

CAL specifications do not imply any specific assumption on the underlying architecture.CAL programs are composed of computational entities, independent from each others (encapsulation property) and share data only through the exchange of tokens. In case of multicore platforms, there is no need for cache coherency protocols as far as the states of the actors are not shared. As a consequence, CAL specifications can be translated into native implementation codes on a wide range of platforms thanks to software and hardware code generators [128, 135].

Additionally to its portability, the parallelism of a RVC specification is also scalable in the sense that designers have the freedom to choose a different implementation of a Functional Unit, exposing different levels of parallelism by using different Video Tools Libraries in order to fit the available parallelism of the platform.

Figure 7.5 (Source: [154]) illustrates this concept. The parallelism of the RVC specification is naturally shown by the interconnections of the Functional Units (IQ, Adder, DF, etc.), as each FU can potentially work concurrently. At the level of the Functional Unit (e.g. 2-D IT), the designer can choose, in a proprietary library, a version of this FU that exposes more parallelism (1D-IT, Transpose and 1D-IT) in order to better exploit the processors let vacant by the less-parallel implementation of this FU provided by the standard MPEG VTL.

7.2.2 Deployment of video coding technology

The RVC standard provides an undeniable flexibility for the implementation of codecs onto the next generation of parallel platforms thanks to the adoption of a new paradigm (dataflow programming and CAL language) for the specification of codecs and thanks to the definition of decoder descriptions (FND) and bitstream descriptions (BSD) as input to any RVC decoder. The degree of freedom offered by varying the different encoding/decoding schemes, by choosing different FUs instantiations and by partitioning the ADM in different ways is such that there is a theoretically unbounded set of implementation possibilities. Such a flexibility is paramount for the efficient implementation of codecs and to make a full profit of the potential computations power of the new parallel platforms.

Figure 7.6 (Source: [154]) illustrates a potential use of the RVC standard. At the server side, an RVC encoder produces a bitstream composed of the usual

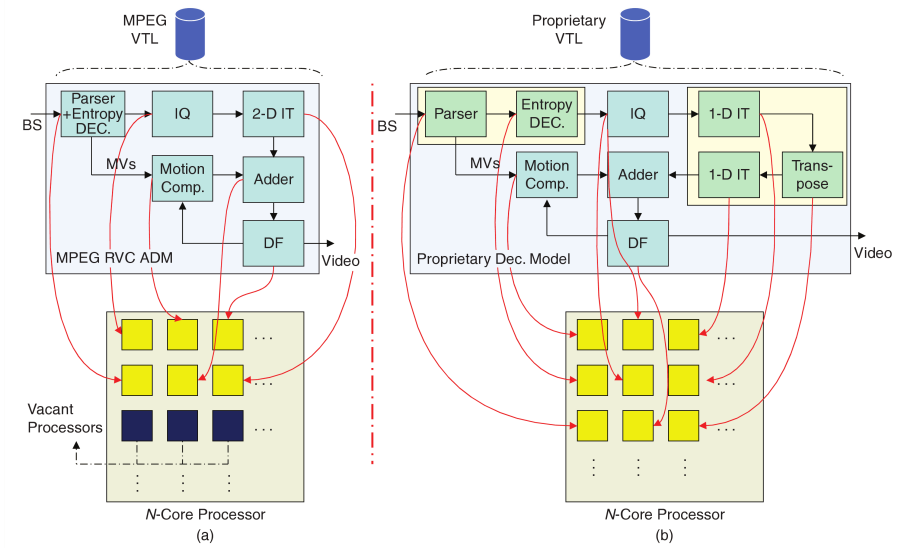


Figure 7.5: Illustration of the concept of mapping an RVC specification onto a multicore platform (a) from the MPEG Video Tools Library or (b) from user-defined proprietary Video Tools Libraries.

decoder description (FND) and the bitstream description (BSD) along with several enhancement layers, each layer providing a higher decoding quality.

The decoder-3, implemented on a small platform with low computing power, decodes only the base layer of the bitstream. The decoder-2, implemented on a more powerful platform, decodes the base and the enhancement layer 1 of the bitstream, Finally, decoder-1 decodes the entire bitstream (base + enhancement layers 1 and 2) because the platform has the required resources for decoding it.

For designers, the adoption of the RVC standard presents two main advantages in terms of scalability:

- To be able to scale the exposed parallelism of the application to fit the available level of parallelism of the platform
- To be able to better take into consideration the constraints imposed by the desired target platform (in terms of computing power, energy consumption, area, etc.) when implementing the Abstract Decoder Model (ADM) – in other words, the CAL program – onto the desired platform.

Hence, the use of the RVC standard and the appropriate tools and methodologies (as the ones presented in this research work) ensures efficient implementations of decoder specifications in spite of the large diversity of the platforms.

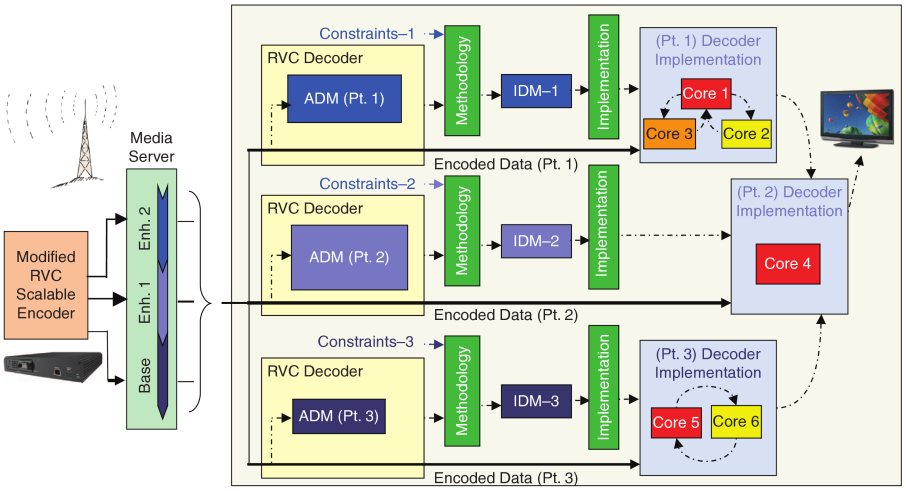


Figure 7.6: An RVC bitstream can be efficiently decoded by various platforms with various architectures.

7.3 Contributions

7.3.1 General development of the standard

I participated actively to the development of the RVC standard from the beginnings. It includes:

- Study of the compactness of an RVC bitstream containing the description of the bitstream and the description of the network of FUs [164, 165],
- Study of the feasability of defining FUs with different levels of granularity [166],
- First proposal of the FU interfaces (inputs/outputs) according to the MPEG-4 SP decoder [167],
- Proposal of a template for the textual description of the FUs of the VTL [168],
- Proposal of a classification of FUs in the MPEG VTL in order to distinguish algorithmic video coding tools from pure data management tools [169],
- Proposal of a naming convention in the MPEG VTL for a unique identification of FUs [170],
- Exploration of the classification of tokens in the MPEG-4 SP decoder for easing the connectivity of FUs [171, 172],
- Submission of the parallel version of the MPEG-4 SP decoder [173],
- Submission of new FUs in the MPEG-4 SP decoder: for dealing with the BTYPE datatype [174], for Variable Length Coding [175] and some serial versions of existing FUs [176],

- Definition of the subset of the BSDL language and description of the MPEG-4 SP bitstream in RVC-BSDL [177],
- Proposal for the implementation of multiple reference frame support for extending the MPEG-4 SP to higher profiles [178],
- Proposal of a methodology and a tool for converting bitstream description into CAL parsers [179, 180] (the reader is referred to Section 7.3.2),
- Test of the reconfigurability capabilities of decoders defined within the standard in order to prove the concepts of RVC [173] (the reader is referred to Section 7.3.3),
- Edition of the standard documents [21, 134].

7.3.2 Easing the design of parsers

The RVC framework aims at facilitating the development of new video coding tools. The flexibility offered to explore rapidly different decoder reconfigurations is primordial. However, testing new decoding solutions implies that the bitstream syntax may change from a solution to another, implying the refactoring of the parser. The parser FU may be very complex and writing it by hand is time-consuming and error prone. This section proposes a solution for generating a parser in CAL directly from the bitstream description (BSD). The reader can refer to [157–160] for further details. Many contributions to the standard have been submitted [175, 177, 179, 180].

7.3.2.1 The BSDL2CALML tool

The difficulty of such transformation remains in the fact that a *description* (the bitstream description) is converted into an *executable*. Using a XML formalism, the BSD describes the sequence of syntax elements constituting the bitstream and the parser is the process that distributes the elements of syntax to the others actors of the network. The challenge is to develop a process such that the parser is capable of handling all the possible combinations of the syntax elements allowed by the BSD.

As BSD is an XML schema and RVC-CAL can be represented in XML (CALML) XSL Transformations (XSLT) is the most appropriate way to perform this conversion.

The conversion process illustrated in Figure 7.7 is performed in five passes. For each pass, only some parts of the BSD are considered and predefined CALML templates are used to generate the parser.

First pass creates the header of the parser actor in CALML. It consists of adding constant values, initialized variables, input and output ports and the signature of the actor.

Second pass creates the actions for each syntax element of the bitstream description (the BSD schema). One or several actions can be created for each syntax element. Follows an non-exhaustive list of cases:

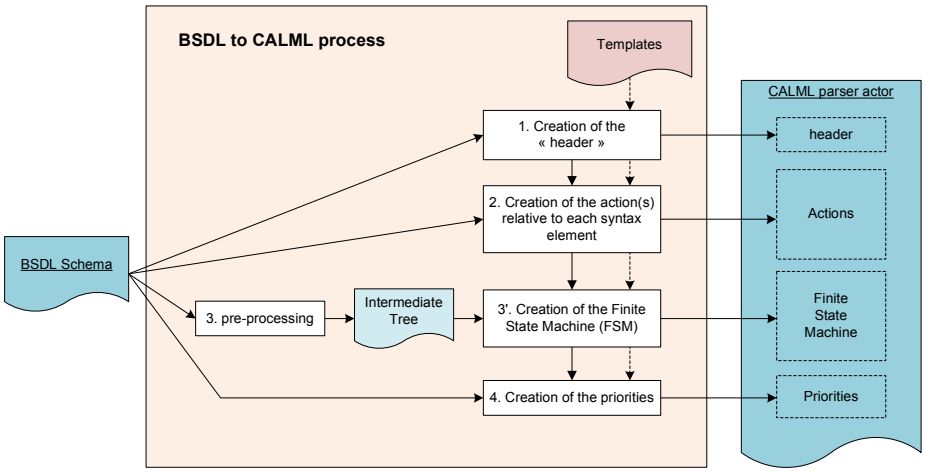


Figure 7.7: Illustration of the process converting a bitstream description (BSD) into a CalML parser.

- If the syntax element is simple (fixed sized element, without any condition on its existence), only one action is created
- If the syntax element presents some conditions on its existence, three actions will be created: the first action tests if this element exists, the second action consumes the tokens relative to this syntax element and the third action is created for jumping to the next syntax in case of this element does not exists.
- If the syntax element must be repeated several times, three actions are created. An action is needed to check if this element needs to be repeated, an action which consumes the token of the syntax element and an action which is used to jump to the next syntax in case of the element must be not repeated anymore.
- If the parser actor needs to communicate with an external actor to parse this syntax element, then several actions are created for establishing a communication protocol between these two actors (Section 7.3.2.1.1).

Third pass creates the Finite State Machine (FSM) of the whole parser. A preliminary sub-step is performed in order to build an intermediate tree which is a more convenient representation of the initial tree so that it is then easier to perform the transformation for building the FSM. The process consists of obtaining a flatten representation of the relations between all the actions in order to have a better view on how the actions follows from each others.

Fourth pass sets priorities in case several actions are fireable at the same time.

Fifth pass closes the file.

7.3.2.1.1 Dealing with unsized syntax elements Bitstreams can contains elements of syntax for which the size in not known *a priori*. In MPEG-4, it is the case with Variable Length Codes (VLC). It makes the decoding process a bit more complex because the parser needs to handle the processes for decoding these unsized elements. One solution consists in externalizing these decoding processes in an FU, which needs to be written by hand because the decoding process cannot be guessed. Figure 7.8 illustrates the procedure for decoding such unsigned syntax elements.

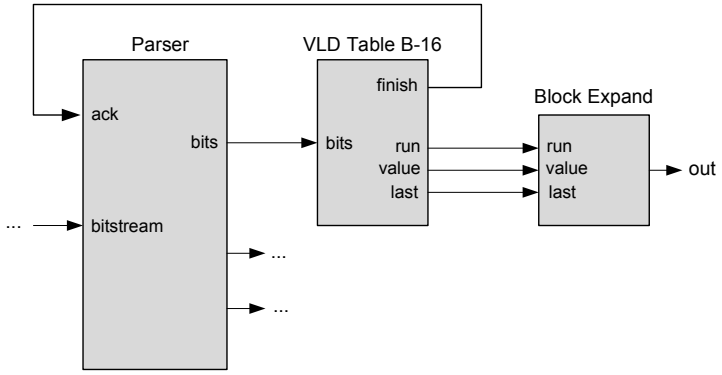


Figure 7.8: The generated Parser sends bits to the external FU (VLD Table B-16) and receives back an acknowledgment from the FU. Once the data is decoded by the FU, it is send to the next FU, Block Expand.

A generic communication protocol needs to be defined between the parser and these FU in order to achieve the parsing. Each time a sized syntax element is parsed, the parser fires a `xxx.read` action. When it parses an unsized syntax element (e.g. variable length codes), the following protocol is followed:

1. The parser reads a bit and sends it to the FU by firing an `xxx.read` action.
2. The FU receives the bit and start the decoding process. Then, the FU acknowledges the parser according to the results of the process: either the element has been decoded or not.
3. If the parser receives an acknowledgement saying that the element has been decoded, the parser fires an `xxx.finished` action and continues with the next element of the bitstream. If the element has not been yet decoded, the parser fires an `xxx.notfinished` action, sending a new bit to the FU. This loop continues as long as the element is decoded.

In case of the MPEG-4 SP decoder, Li and al. [159] have developed a systematic way for generating in RVC-CAL the FU capable of handling the Variable Length Code (VLC).

7.3.2.2 Example

Figure 7.9 shows a simple example of BSD. The bitstream is composed of four elements: the first element is a 1-bit long element and is output on port A, the second element is 2-bit long and is an output on port B, etc. Figure 7.10 reports the CALML code generated for decoding the first element of syntax, the `read` action and the FSM for decoding the piece of BSD.

```
    <xsd:element name="bitstream_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstelement" type="bs1:b1" rvc:port="A"/>
        <xsd:element name="secondelement" type="bs1:b2" rvc:port="B"/>
        <xsd:element name="thirdelement" type="bs1:b3" rvc:port="C"/>
        <xsd:element name="fourthelement" type="bs1:b4" rvc:port="D" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

Figure 7.9: Sample of a Bitstream Syntax Description (BSD).

```
<!-- action for parsing the element "firstelement" of 1 bit -->
<Action>
 <QID name="firstelement.read">
  <ID name="firstelement"/>
  <ID name="read"/>
 </QID>
 <Input kind="Elements" port="bitstream">
  <Decl kind="Input" name="b"/>
  <Repeat>
   <Expr kind="Var" name="BS1_B1_LENGTH"/>
  </Repeat>
 </Input>
 <Output port="A">
  <Expr kind="Var" name="b"/>
   <Repeat>
    <Expr kind="Var" name="BS1_B1_LENGTH"/>
   </Repeat>
 </Output>
</Action>

[...]
<!-- Finite state machine of the parser -->
<Schedule kind="fsm" initial-state="root.firstelement_exists">

<!-- Transition for switching the actor state after having parsed the first element -->
<Transition from="root.firstelement_exists" to="root.secondelement_exists">
 <ActionTags>
  <QID name="firstelement.read">
   <ID name="firstelement"/>
   <ID name="read"/>
  </QID>
 </ActionTags>
</Transition >

[...]

</Schedule>
```

Figure 7.10: For each action, the tool generates a CAL action for parsing the element `firstelement` and the corresponding part of the Finite State Machine.

7.3.3 Testing the reconfigurability capabilities

This case study aims at proving the concept claimed by RVC [161] [173]. Being able to create, develop and test quickly new decoder configurations for easing the development of video coding technology is one of the main idea behind RVC. This is possible by exploiting the existing commonalities between decoders and be able to compose, mix and refactor them seamlessly.

Starting from the MPEG-4 Simple Profile decoder presented in Figure 7.11, this experiment consists in replacing the Inverse Quantization (IQ) and Inverse Transform (IT) FUs by the ones of Audio Video coding Standard of China (AVS). Section 7.3.3.1 presents briefly the AVS standard, Section 7.3.3.2 presents the few steps for reconfiguring the decoder and Section 7.3.3.3 discusses the results.

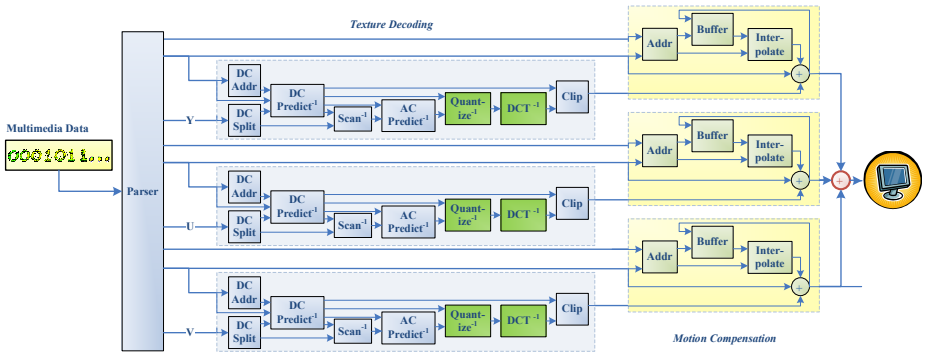


Figure 7.11: The MPEG-4 Simple Profile decoder.

7.3.3.1 Audio Video coding Standard (AVS)

Audio Video coding Standard of China (AVS) [138] is a new compression standard developed by AVS Workgroup of China. AVS Part 2 (referred as AVS for short) is addressing high-definition, high-density storage media and digital video broadcasting applications and was published as national standard for China in February 2006. Integer transform, intra and inter-picture prediction, in-loop deblocking filter and context-adaptive two dimensional variable length coding (CA-2D-VLC) are the key compression tools of AVS [139].

The Inverse Quantization (IQ) stage in AVS is characterized by a QP for which the reconstructed transform coefficients are obtained through a 1-D lookup only. Compared with MPEG-4 Simple Profile in which two inverse quantization methods are used and DC coefficients of intra coded blocks are inverse quantized in a different manner, AVS implies only one quantization type and processes coefficients of the whole block in the same way.

The 8x8 pre-Scaled Integer Inverse Cosine Transform (IICT) is used by AVS for providing a unique specification for the finite precision implementations and

for yielding significant saving in processing complexity when compared with the traditional Inverse Discrete Cosine Transform (IDCT), features that are particularly interesting for low-end processors.

7.3.3.2 Decoder reconfiguration

Although the Inverse Quantization (IQ) and Inverse Transform (IT) algorithms are different between AVS and MPEG-4, they both have the same behavior in terms of input and output tokens requirements: 8x8 blocks. The level of granularity used to define these FUs allows to interchange them seamlessly. Being able to lower the standardization level at the coding tool level is one strength of RVC.

Inverse Quantization and Inverse Transform tools (FUs in green in Figure 7.11) are taken from the AVS toolbox, and the rest of the video coding tools are the ones from MPEG-4 SP, resulting in a hybrid decoder with new properties.

The flexibility offered by RVC to interchange easily the FUs must lead to the improvement of the decoding solution. In this case study, the quantization precision of MPEG-4 SP ranges on a 5-bit scale and is extended to 6 bits applying the quantization algorithm of AVS. Bitstream syntax changes because elements such as `vop_quant`, `quant_scale` need to be extended to 6 bits in the new bitstream syntax. Thus the new bitstream format is no longer the one of MPEG-4 SP format and in RVC it has to be described by a corresponding RVC-BSDL schema. The syntax elements `vop_quant` and `quant_scale` are both defined as a syntax element of type `VOPQuantType` whose bit length is equal to `quant_precision`. For the new schema, `quant_precision` is extended to 6 bits.

The BSD presented in Figure 7.12 contains the changes that are necessary by the new IQ and IT stages.

```
<xsd:schema>
    [...]
    <xsd:annotation><xsd:appinfo>
      <bs2x:variable name="m4v:quant_precision" value="6"/>
    </xsd:appinfo></xsd:annotation>
    [...]
    <xsd:element name="vop_quant" type="VOPQuantType">
    [...]
    <xsd:element name="quant_scale" type="VOPQuantType"/>
    [...]
    <xsd:simpleType name="VOPQuantType">
      <xsd:restriction base="bs1:b9">
        <xsd:annotation><xsd:appinfo>
          <bs2x:bitLength value="$m4v:quant_precision"/>
        </xsd:appinfo></xsd:annotation>
      </xsd:restriction>
    </xsd:simpleType>
    [...]
</xsd:schema>
```

Figure 7.12: Modification of the Bitstream Syntax Description (BSD) for the new decoder.

This modification of the BSD implies that the parser needs to be also modified to parse correctly the new bitstream format. This new parser can be generated automatically using the approach presented in Section 7.3.2.

7.3.3.3 Results and discussion

The quantization precision has been extended to 6 bits in both the variable length encoder and the variable length decoder. A new bitstream is produced and provided as input into the reconfigured decoder.

The overall decoding performance is evaluated under the following test conditions: I frames only, progressive sequence coding, VOL frame rate equal to 30, sequence of 200 frames. In the MPEG-4 decoder, the quantization range is equal to 5 bits, IDCT is used for Inverse Transformation and the values of QP are {1, 2, 3, 5, 7, 11, 16, 21, 26, 31}. In the reconfigured decoder, the quantization range is equal to 6 bits, IICT from AVS is used as inverse transform and the values of QP are {1, 9, 15, 22, 29, 36, 43, 50, 57, 63}.

Figure 7.13 reports the PSNR curve versus bitrate and shows that at low and medium bitrates, the two decoders yield very similar performance. However, the performance of the two decoders differs at high bitrates where the reconfigured decoder improves gradually at increasing bitrates.

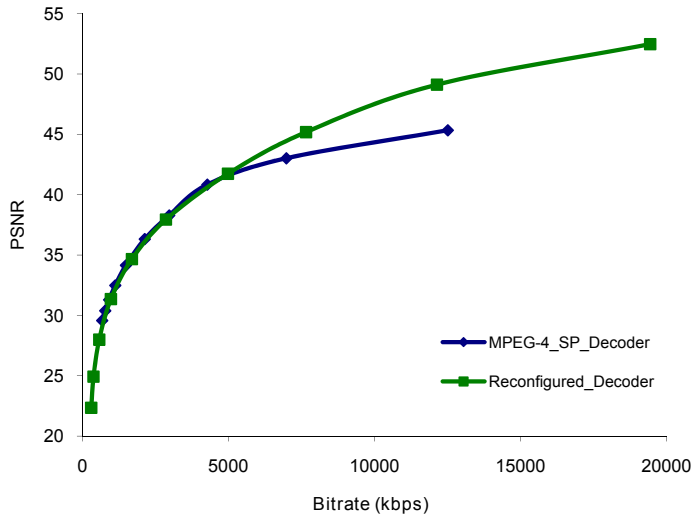


Figure 7.13: Bitrate versus PSNR for the original MPEG-4 SP and reconfigured with AVS quantization for the Foreman sequence.

In this simple example, coding tools from AVS are used by the reconfigured decoder, yielding both complexity reduction and improvements of the performance in high bitrate ranges. Obviously, other reconfigurations can be specified by selecting coding tools from different standards, such as intra prediction,

half/quarter-pixel precision interpolation, motion vectors prediction, to achieve specific complexity performance trade-offs.

This experiment shows the potential offered by the RVC framework to reconfigure easily decoders and to improve them seamlessly thanks to this new actor- and dataflow-based paradigm enabling the tool-level specification of the standard Video Tool Library and resulting in a greater modularity. It is likely that the usage of the RVC will make disappear the traditional MPEG profile-level definition. RVC provides the means to explore seamlessly new video coding algorithms.

Chapter 8

Conclusion

During my research work, I contributed to make significant advances in digital systems design by enabling efficient high level design space exploration through the use of dataflow programming and the CAL language [143, 153]. It includes:

- The development of profiling tools in order to extract metrics from CAL programs which serves as a basis for the design space exploration: execution times of actions, trace critical path and data transfers between actors (Section 3.2).
- The elaboration of an optimization strategy aiming at guiding designers in the refactoring of CAL programs by minimizing the trace critical path of the execution of programs (Section 3.3).
- The implementation of partitioning and scheduling heuristics based on the causation trace [146, 147] (Section 4.2).
- The elaboration of a holistic methodology for systematically explore the design space by gathering all the steps described previously. An important work of integration of the different tools was necessary to make all the tool compatible, and gathering the work of several teams into a single environment [153] (Section 5.3).

I also contributed to the creation of a new ISO/IEC standard, Reconfigurable Video Coding (RVC), aiming at shifting the paradigm for the specification of MPEG video coding technology [145, 154–156, 162]. It includes:

- The development of the standard from the beginnings by exploring different solutions, by proposing methodologies for classifying the FUs, by submitting new FUs, by defining the languages used in the framework, etc. [164–172, 174, 176, 178]. I took part of the writing of the two standard documents [21, 134] (Section 7.3.1).
- The development of a tool aiming at generating parsers in CAL directly from bitstream descriptions [157–160] [175, 177, 179, 180] (Section 7.3.2).

- The proof of concept of RVC by testing the reconfigurability of coding tools coming from different standards and mixing them together to test new decoder configuration [161] [173] (Section 7.3.3).

The proposed approach allows exploring the design space on different target platforms with different levels of parallelism with a reduced effort compared to what it would have been by using traditional approaches. We suceeded in optimizing the performance of a MPEG-4 Simple video decoding by a factor of seven. The optimization strategy aiming at minimizing the trace critical path of the execution of CAL programs for guiding the refactoring of CAL programs towards the target region of the design space has been validated by the results.

Future work

Some problems have been clearly identified problems in the methodology and need to be addressed in the future. The first work consists in improving the accuracy of the performance evaluation of CAL programs mapped on a target platform. It includes the improvement of the discrete event simulalor and the modeling of target platforms with realistic characteristics. The second issue relates to new partitioning and scheduling heuristics. The third topic concerns the improvement of the code generators for minimizing the effect of the scheduling on the performance.

Towards more accurate performance evaluation

The performance evaluation of a solution is performed by reconstructing virtually the execution of a CAL program by mean of a Gantt chart thanks to its causation trace (Section 4.3).

New architecture of the discrete event simulator

A more sophisticated engine for reconstructing the Gantt chart must take into consideration more characteristics of the runtime and the target platform in order to improve the accuracy of the estimation. Currently, the Gantt chart engine inputs the causation trace, the partitioning and scheduling configuration and the running time of actions. In order to better reflect the influence of the scheduling overhead - which is far from being negligible - onto the performance, the engine should be based on a model of the underlying runtime. This model should describe the different steps for scheduling the actions. As a consequence, the engine can reconstruct the Gantt chart of the actions firings with an increased accuracy by introducing in the Gantt chart the scheduling overhead induces by the modeled underlying runtime. The advantage of externalizing the runtime model from the engine is that several runtimes can be used for the performance evaluation, enabling designers to compare the performance of several runtime for a given partitioning and scheduling. Figure 8.1 illustrates

the proposed refactoring of the Gantt chart engine. This latter should have the following characteristics:

- Reflect the different steps of the runtime for scheduling and executing actions,
- Consider the scheduling policy implemented in the runtime,
- Consider the state of the FIFO when scheduling actions: if it is full, any actor connected to its input cannot fire, if it is locked, any of other actor cannot fire.
- Be fast, because performance evaluation heuristics are using it.

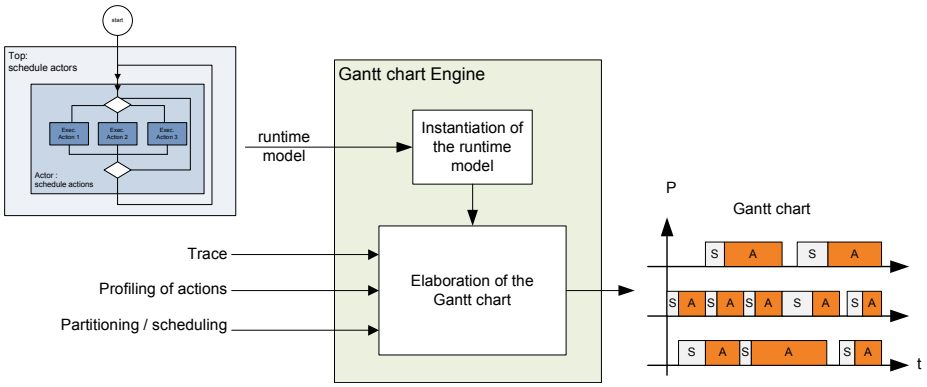


Figure 8.1: Illustration of the proposed refactoring of the Gantt chart engine.

Modeling target platforms

Performing meaningful design space exploration implies being able to evaluate correctly the performance of the implementation. Because every platform has its own intrinsic properties (speed, communication costs, memory hierarchy, computing power, parallelism, etc.), models need to be elaborated to report these characteristics at high level. It should include the characteristics that are really important for the performance evaluation at the CAL level, i.e. communications costs and running time of actions. Dedicated tools need to be developed for each type of platforms in order to profile the running time of actions and the communication costs between actors. The tool developed in this research work (the CAL DYNAMIC ANALYZER) is dedicated to Intel architectures. Experiments like the one described in Section 4.3.1.1 need to be ran on a larger range of target platforms in order to model appropriate communication models.

Alternative partitioning/scheduling approaches

Clustering partitioning

During the search for efficient partitions (Section 6.1.4), it has been noticed - given the current status of the tools - that the best partitions are the ones that present the following characteristics:

- The ratio between computations and communications should be over a given threshold,
- The clusters of actors assigned to the different processors must be chosen such as there are minimal-cuts between them in terms of communication channels,
- The partitions should be computationally load balanced.

An alternative approach, based on clustering algorithms, for partitioning the actors on the different processors can be elaborated from this observation.

Cache-friendly scheduling

In order to minimize the overhead due to the memory accesses, one can think of scheduling the actions such that caches misses are minimized and the hit rate is improved. This work has been started by Mirko Ferrati and Alessandro Pignotti [140]. It focuses on improving the performance of the implementation of dataflow programs by scheduling actions with a cache aware schedule. This work needs to be continued and to be integrated into the existing tools.

Improving code generators

Even if the improvements of the performance of the software code generators are important, there is still room for optimization.

The main topic is still the reduction of the overhead due to the scheduling of actions. The current solution being studied is the static scheduling of the actions belonging to static regions. These static regions (discussed in Section 4.2.4) can be detected thanks to the analysis of CAL programs through a verification modeling language called PROMELA (Process or Protocol Meta Language). This language allows for the dynamic creation of concurrent processes to model, for example, distributed systems.

Another important topic is the reduction of the discrepancy between the design of dataflow programs targeted for hardware and software. Currently, there are still some special coding rules that lead to efficient solutions in hardware but not in software and vice-versa. The aim is to try to converge to a unique dataflow program or to minimize the refactoring for switching from one to the other. It is possible thanks to smarter code generators.

The future is promising

With the advent of the multicore era, there is no clear path to a unified methodology for the specification of complex applications and their implementation onto heterogeneous platforms. There are several approaches trying to tackle the problem, some with more emphasis to the reuse of legacy code and IP blocks, others that require specific methodologies tighten to a given type of platform or technology. Very few approaches have as main objective the achievement of a true portable parallelism. We are just at the beginnings and most of the industry has not yet changed their traditional way of working fully based on sequential approach. However, there will be soon compelling reasons to change. It is the right time to develop a common technology for dealing with parallelism in systems design and this PhD work has tried to make some advances in this direction. What is sure is that breakthroughs for the adoption of new methodologies in parallel programming will be possible only if the increase of the difficulty and complexity of the problem will correspond to simplifications and support by means of a set of tools that allows a seamless use of the technology by the average designer.

Dataflow programming with CAL provides extremely interesting features to allow such simplification and adoption, thus it has the potential to bring efficient parallel systems design to the average developers. Its semantics and its model of computation enable the development of code generation tools that directly convert CAL programs into efficient implementation code, guaranteeing the correctness of data dependencies by construction without dealing with low-level programming issues. Concurrently to this PhD work, several research activities are well progressing in the direction of improving the efficiency of these code generators and of generating automatically the necessary interfaces for the target platforms.

The adoption of this new formalism by several teams worldwide shows that there is a great interest for this new technology. The numerous successful projects on different subjects from high level design exploration down to the implementation level proves that CAL can effectively be the support of a new abstraction layer for the high level design of high-demanding digital systems. The number of applications described in CAL is growing. The most known are MPEG-2, MPEG-4, AVC (part of the development of RVC). Libraries of signal processing algorithms are currently under development. Industrial applications have also been modeled using CAL: a wireless device [141], a bar code reader [163] [132] and a robot controller [142].

However, it is not enough to be able to build efficient code generators to make the approach attractive and functional to wider use in systems design. A meaningful breakthrough is possible only if the added value of the new approach answers other needs of systems design that are currently not well developed in the current practice. Indeed, CAL, by raising the abstraction level and by the nature of the abstractions themselves enable new dimensions in the design space exploration and in the development and validation of applications. Enabling design space exploration at a higher level of abstraction and being able to im-

plement high level programs onto a wide range of platforms are very attractive features for complex systems designers. Raising the level of abstraction has been always being synonym of productivity increase.

This research work has explored the potential of dataflow programming for supporting the design space exploration of complex digital systems. A complete design exploration framework has been developed: it includes profiling tools, partitioning and scheduling heuristics, optimization methodologies and performance evaluation. The successful use of the developed tools and methodologies on real world applications has proved that CAL approach can support efficient design exploration and portability. This research work is the starting point for the development of advanced heuristics, methodologies and tools for exploring the design space at high level of abstraction. The opened topics are numerous and there is plenty of room for several research areas: partitioning and scheduling problems, profiling tools, architecture characterization, algorithmic optimization and code generation. The results obtained in digital system design are very promising, particularly if considering that are still many foreseen improvements and optimizations that have not been applied yet. Further improvements in genericity, portability and efficiency can be expected for the near future.

Bibliography

[1] C. Clerc, *A profiling framework for high level design space exploration for memory and system architectures.* PhD thesis, Lausanne, 2006.

[2] A. Jantsch, "Models of Embedded Computation," in *Embedded Systems Handbook*, ch. 4, 2005.

[3] L. Gomes, J. Paulo Barros, and A. Costa, "Modeling Formalisms for Embedded System Design," in *Embedded Systems Handbook*, ch. 5, 2005.

[4] E. F. Moore, "Gedanken Experiments on Sequential Machines," in *Automata Studies*, pp. 129–153, Princeton University, 1956.

[5] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level synthesis: introduction to chip and system design.* Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[6] D. Harel, "On visual formalisms," *Commununication ACM*, vol. 31, pp. 514–530, May 1988.

[7] D. Harel, "Statecharts: A visual formalism for complex systems," *Journal of Science of Computer Programming*, vol. 8, pp. 231–274, June 1987.

[8] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, pp. 366 –390, mar 1997.

[9] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-Software Codesign of Embedded Systems," *IEEE Micro*, vol. 14, pp. 26–36, August 1994.

[10] F. Vahid, S. Narayan, and D. Gajski, "SpecCharts: a VHDL front-end for embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 14, pp. 694 –706, June 1995.

[11] "SpecC System," http://www.cecs.uci.edu/ specc/.

[12] C. A. Petri, *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[13] "Esterel Technologies," http://www.esterel-technologies.com/.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, pp. 1305 –1320, Sept. 1991.

[15] A. Benveniste and P. Le Guernic, "Hybrid dynamical systems theory and the Signal language," *IEEE Transactions on Automatic Control*, vol. 35, pp. 535 –546, May 1990.

[16] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing* (J. L. Rosenfeld, ed.), pp. 471–475, 1974.

[17] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink, "YAPI: application modeling for signal processing systems," in *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, (New York, NY, USA), pp. 402–405, ACM, 2000.

[18] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, pp. 773 –801, May 1995.

[19] E. Lee and D. Messerschmitt, "Synchronous Data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, 1987.

[20] J. Eker and J. W. Janneck, "CAL Language Report," Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, Dec. 2003.

[21] "ISO/IEC 23001-4:2009 Information technology - MPEG systems technologies - Part 4: Codec configuration representation."

[22] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting style in architectural design environments," *SIGSOFT Softw. Eng. Notes*, vol. 19, pp. 175–188, December 1994.

[23] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, "Using object-oriented typing to support architectural design in the C2 style," *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 24–32, October 1996.

[24] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying Distributed Software Architectures," in *Proceedings of the 5th European Software Engineering Conference*, (London, UK), pp. 137–153, Springer-Verlag, 1995.

[25] W. Tracz, "Parametrized programming in LILEANNA," in *Proceedings of the ACM/SIGAPP symposium on Applied computing: states of the art and practice*, SAC '93, (New York, NY, USA), pp. 77–86, ACM, 1993.

[26] P. Binns, M. Englehart, M. Jackson, and S. Vestal, "Domain-Specific Software Architectures for Guidance, Navigation and Control," *International Journal of Software Engineering and Knowledge Engineering*, 1993.

[27] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using rapide," *IEEE Transactions on Software Engineering*, vol. 21, pp. 336–355, 1995.

[28] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct architecture refinement," *IEEE Transactions on Software Engineering*, vol. 21, pp. 356–372, 1995.

[29] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering*, vol. 21, pp. 314–335, 1995.

[30] M. M. Gorlick and R. R. Razouk, "Using weaves for software construction and analysis," in *Proceedings of the 13th international conference on Software engineering*, ICSE '91, (Los Alamitos, CA, USA), pp. 23–34, IEEE Computer Society Press, 1991.

[31] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 213–249, July 1997.

[32] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, pp. 70–93, January 2000.

[33] "IEEE Standard for IP-XACT – Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows," http://standards.ieee.org/.

[34] "Wind River Simics," http://www.windriver.com/products/simics/.

[35] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer, "Comparing analytical modeling with simulation for network processors: a case study," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 256–261 suppl., 2003.

[36] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli, "A framework for evaluating design tradeoffs in packet processing architectures," in *Proceedings of the 39th annual Design Automation Conference*, DAC '02, (New York, NY, USA), pp. 880–885, ACM, 2002.

[37] G. Ascia, V. Catania, and M. Palesi, "Design space exploration methodologies for IP-based system-on-a-chip," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 2, pp. 364–367, 2002.

[38] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere, "System level design with SPADE: an M-JPEG case study," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, (San Jose, California, US), pp. 31 –38, 2001.

[39] P. Puschner and C. Koza, "Calculating the maximum, execution time of real-time programs," *Real-Time Systems*, vol. 1, pp. 159–176, September 1989.

[40] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *SIGPLAN Not.*, vol. 30, pp. 88–98, 1995.

[41] S. Mallat and F. Falzon, "Analysis of low bit rate image transform coding," *IEEE Transactions on Signal Processing*, vol. 46, pp. 1027–1042, 1998.

[42] M. Mattavelli and M. Ravasi, "High Level Extraction of SoC Architectural Information from Generic C Algorithmic Descriptions," in *Proceedings of the Fifth International Workshop on System-on-Chip for Real-Time Applications (IWSOC)*, (Washington, DC, USA), pp. 304–307, 2005.

[43] "GCC online documentation - GNU Project - Free Software Foundation (FSF)," http://www.gnu.org/software/gcc/onlinedocs/.

[44] J. R. Larus, "Abstract execution: a technique for efficiently tracing programs," *Software – Practice & Experience*, vol. 20, pp. 1241–1258, 1990.

[45] P. M. Kuhn, *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*. Kluwer Academic Publishers, 1999.

[46] A. Srivastava and A. Eustace, "ATOM: a system for building customized program analysis tools," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Orlando, Florida, United States), pp. 196–205, ACM, 1994.

[47] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[48] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, vol. 42, (New York, NY, USA), pp. 89–100, ACM, June 2007.

[49] J. R. Larus and T. Ball, "Rewriting executable files to measure program behavior," *Software – Practice & Experience*, vol. 24, pp. 197–218, 1994.

[50] J. Larus, "Efficient program tracing," *Computer*, vol. 26, pp. 52–61, May 1993.

[51] M. Smith, "Tracing with Pixie," Technical Report CSL TR 91 497, Computer Systems Laboratory, Stanford University, Stanford, CA, USA, 1991.

[52] J. R. Larus and E. Schnarr, "EEL: machine-independent executable editing," in *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, (La Jolla, California, United States), pp. 291–300, ACM, 1995.

[53] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, pp. 120–126, 1982.

[54] D. Sciuto, F. Salice, L. Pomante, and W. Fornaciari, "Metrics for design space exploration of heterogeneous multiprocessor embedded systems," in *Proceedings of the International Symposium on Hardware/Software Codesign*, (Estes Park, Colorado, US), pp. 55–60, ACM, 2002.

[55] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto, "Affinity-driven system design exploration for heterogeneous multiprocessor SoC," *IEEE Transactions on Computers*, vol. 55, pp. 508–19, 2006.

[56] L. Carro, M. Kreutz, F. Wagner, and M. Oyamada, "System synthesis for multiprocessor embedded applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 697–702, 2000.

[57] H. P. Peixoto and M. F. Jacome, "Algorithm and architecture-level design space exploration using hierarchical data flows," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, p. 272, IEEE Computer Society, 1997.

[58] N. B. Amor, Y. L. Moullec, J. P. Diguet, J. L. Philippe, and M. Abid, "Design of a multimedia processor based on metrics computation," *Advances in Engineering Software*, vol. 36, pp. 448–458, 2005.

[59] F. Vahid and D. Gajski, "Closeness metrics for system-level functional partitioning," in *Proceedings of the Design Automation Conference (DAC)*, pp. 328–333, 1995.

[60] F. Vahid, "Partitioning sequential programs for CAD using a three-step approach," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, pp. 413–429, 2002.

[61] J. Diguet, O. Sentieys, J. Philippe, and E. Martin, "Probabilistic resource estimation for pipeline architecture," in *Workshop on VLSI Signal Processing*, pp. 217–226, IEEE Signal Processing Society, 1995.

[62] C. Haubelt, J. Teich, K. Richter, and R. Ernst, "System design for flexibility," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 854–861, 2002.

[63] A. Sangiovanni-Vincentelli, "Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design," *Proceedings of the IEEE*, vol. 95, pp. 467–506, 2007.

[64] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integration, the VLSI Journal*, vol. 38, pp. 131–183, 2004.

[65] A. Jantsch and I. Sander, "Models of computation and languages for embedded system design," *IEEE Proceedings - Computers and Digital Techniques*, vol. 152, pp. 114–129, Mar. 2005.

[66] A. Sangiovanni-Vincentelli and M. D. Natale, "Embedded System Design for Automotive Applications," *Computer*, vol. 40, pp. 42–51, 2007.

[67] M. Duranton, "The challenges for high performance embedded systems," in *Proceedings of the EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, (Dubrovnik, Croatia), IEEE Computer Society, 2006.

[68] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," in *Proceedings of the IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, (Salzburg, Austria), pp. 9–14, ACM, 2007.

[69] A. Pimentel, A. Pimentel, L. Hertzbetger, L. Hertzbetger, P. Lieverse, P. van der Wolf, and E. Deprettere, "Exploring embedded-systems architectures with Artemis," *Computer*, vol. 34, pp. 57–63, 2001.

[70] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from Matlab for embedded signal processing architectures," in *Proceedings of the International Workshop on Hardware/Software Codesign*, pp. 13–17, 2000.

[71] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu, "A next-generation design framework for platform-based design," in *Proceedings of the Conference on Using Hardware Design and Verification Languages (DVCon)*, (San Jose, CA, US), 2007.

[72] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An Integrated Electronic System Design Environment," *Computer*, vol. 36, pp. 45–52, 2003.

[73] F. Balarin, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, "Modeling and Designing Heterogeneous Systems," in *Concurrency and Hardware Design, Advances in Petri Nets*, pp. 228–273, Springer-Verlag, 2002.

[74] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and

B. Tabbara, *Hardware-software co-design of embedded systems: the POLIS approach*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[75] A. Mihal, C. Kulkarni, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, C. Sauer, K. Vissers, and S. Malik, "Developing architectural platforms: A disciplined approach," *IEEE Design & Test*, vol. 19, pp. 6–16, November 2002.

[76] "Ptolemy Project," http://ptolemy.eecs.berkeley.edu/.

[77] "MathWorks - Simulink," http://www.mathworks.com/products/simulink/.

[78] S. Ha, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "Hardware-software codesign of multimedia embedded systems: the peace approach," in *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, (Sydney, Qld., Australia), IEEE Computer Society, 2006.

[79] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, pp. 127–144, 2003.

[80] H. Hwang, T. Oh, H. Jung, and S. Ha, "Conversion of reference C code to dataflow model: H.264 encoder case study," in *Proceedings of the Conference on Asia South Pacific Design Automation*, (Yokohama, Japan), pp. 152–157, IEEE Press, 2006.

[81] A. Antola, M. Santambrogio, M. Fracassi, P. Gotti, and C. Sandionigi, "A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse C and CoDeveloper," in *Proceedings of the Southern Conference on Programmable Logic*, (Mar del Plata, Argentina), pp. 221–4, IEEE Press, 2007.

[82] E. Khan, M. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "Applying the handel-C design flow in designing an HMAC-hash unit on FPGAs," *IEE Proceedings – Computers and Digital Techniques*, vol. 153, pp. 323–334, 2006.

[83] E. Khan, M. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "Designing an HMAC-Hash Unit on FPGAs Using Handel-C," in *Proceedings of the IEEE International Symposium on Industrial Electronics*, vol. 2, pp. 1521–1526, 2006.

[84] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the International Conference on VLSI Design*, pp. 461–466, 2003.

[85] M. Graphics, "Catapult Synthesis," http://www.mentor.com/.

[86] Celoxica, "Handel-C Language Reference Manual," 2003.

[87] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura, "A C-based synthesis system, Bach, and its application," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 151–155, 2001.

[88] G. D. Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus synthesis system," *IEEE Design & Test*, vol. 7, pp. 37–53, 1990.

[89] G. D. Micheli, "Hardware synthesis from C/C++ models," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 382–383, 1999.

[90] O. Brassard, F. Rousseau, J. P. David, M. Kastle, and E. M. Aboulhamid, "Automatic Generation of Embedded Systems with .NET Framework Based Tools," in *Proceedings of the IEEE North-East Workshop on Circuits And Systems (NEWCAS)*, pp. 165 –168, 2006.

[91] Microsoft, ".NET Framework," http://www.microsoft.com/net.

[92] "ISO/IEC 23271:2006 Information technology – Common Language Infrastructure (CLI) Partitions I to VI."

[93] J. David and E. Bergeron, "An Intermediate Level HDL for System Level Design," in *Forum on specification and Design Languages (FDL)*, (Lille, France), 2004.

[94] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: A System-Level Exploration Platform for Network Processors," *IEEE Design & Test*, vol. 19, pp. 17–26, 2002.

[95] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proceedings of the IEEE International Conference on Formal Methods and Models for Co-Design,*, pp. 69–70, 2004.

[96] J. C. Hoe and Arvind, "Hardware Synthesis from Term Rewriting Systems," in *Proceedings of the IFIP International Conference on Very Large Scale Integration: Systems on a Chip*, pp. 595–619, Kluwer, B.V., 2000.

[97] R. V. Bennett, A. C. Murray, B. Franke, and N. Topham, "Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems," *SIGPLAN Notices*, vol. 42, pp. 83–92, 2007.

[98] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, and A. Jerraya, "Flexible and Executable Hardware/Software Interface Modeling for Multiprocessor SoC Design Using SystemC," in *Proceedings of the Conference on Asia South Pacific Design Automation*, pp. 390–395, IEEE Computer Society, 2007.

[99] R. Esser, J. Teich, and L. Thiele, "CodeSign: An embedded system design environment," *IEE Proceedings – Computers and Digital Techniques*, vol. 145, no. 3, pp. 171–180, 1998.

[100] Y. Moullec, J.-P. Diguet, N. Amor, T. Gourdeaux, and J.-L. Philippe, "Algorithmic-level specification and characterization of embedded multimedia applications with design Trotter," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 42, pp. 185–208, 2006.

[101] M. Raulet, M. Babel, O. Deforges, J. Nezan, and Y. Sorel, "Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, pp. 316–321, 2003.

[102] N. Pernet and Y. Sorel, "A design method for implementing specifications including control in distributed embedded systems," in *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 2, 2005.

[103] Y. Sorel, "Massively parallel computing systems with real time constraints: the Algorithm Architecture Adequation methodology," in *Proceedings of the First International Conference on Massively Parallel Computing Systems*, pp. 44–53, 1994.

[104] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hannikainen, T. D. Hamalainen, J. Riihimaki, and K. Kuusilinna, "UML-based multiprocessor SoC design framework," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 5, pp. 281–320, 2006.

[105] P. Kukkala, M. Setala, T. Arpinen, E. Salminen, M. Hannikainen, and T. D. Hamakainen, "Implementing a WLAN video terminal using UML and fully automated design flow," *EURASIP Journal of Embedded Systems*, vol. 2007, pp. 20–20, 2007.

[106] B. Hailpern and P. Tarr, "Model-driven development: the good, the bad, and the ugly," *IBM Systems Journal - Model-driven software development*, vol. 45, pp. 451–461, 2006.

[107] "National Instruments – Labview," http://www.ni.com/labview/.

[108] K. Shanmugan, P. Titchener, and W. Newman, "Simulation based CAAD tools for communication and signal processing systems," in *Communications of the IEEE International Conference on World Prosperity Through Communications*, vol. 3, pp. 1454–1461, 1989.

[109] Synopsys, "Cocentric System Studio," http://www.synopsys.com/.

[110] FP7 European Project Proposal, "CALTOOLS," 2010.

[111] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, May 2006.

[112] C. Lucarz, P. Faure, G. Roquier, M. Mattavelli, and V. Noël, "D1A: CAL Methodology," *ACTORS Project (http://www.actors-project.eu)*, 2008-2011.

[113] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[114] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, 1996.

[115] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.

[116] "The OpenDF tool," http://opendf.sourceforge.net.

[117] "GNU Gprof," http://sourceware.org/binutils/docs-2.16/gprof/.

[118] J. Janneck, I. Miller, and D. Parlour, "Profiling dataflow programs," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, pp. 1065–1068, 2008.

[119] L.-R. Liu, D. Du, and H.-C. Chen, "An efficient parallel critical path algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 909 –919, July 1994.

[120] J. Hollingsworth, "Critical path profiling of message passing and shared-memory programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 1029 –1040, Oct. 1998.

[121] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[122] A. El Guindy, "Scheduling and partitioning dataflow programs on heterogeneous multiprocessor platforms," Master's thesis, Cairo (Egypt), 2011.

[123] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[124] R. Gu, J. W. Janneck, M. Raulet, and S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Apr. 2009.

[125] M. Wipliez, *Infrastructure de compilation pour des programmes flux de données*. PhD thesis, Rennes, 2010.

[126] M. Wipliez, G. Roquier, and J.-F. Nezan, "Software code generation for the rvc-cal language," *Journal of Signal Processing Systems*, vol. 63, pp. 203–213, May 2011.

[127] "OpenForge," http://openforge.sourceforge.net.

[128] 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*, vol. 63, pp. 241–249, May 2011.

[129] H. S. Prabhu and T. Sherine, "GALS design in the CAL dataflow language," Master's thesis, Lund (Sweden), 2010.

[130] C. Von Platen, "D2C: CAL ARM Compiler," *ACTORS Project (http://www.actors-project.eu)*, 2008-2011.

[131] R. Thavot, A. Rahman, A. A. H. Bin, R. Mosqueron, and M. Mattavelli, "Automatic mutli-connectivity interface generation for system designs based on a dataflow description," in *Proceedings of the 6th Conference on Ph.D. Research in Microelectronics & Electronics*, (Berlin, Germany), 2010.

[132] P. Faure, "D5C: Image Demonstrator," *ACTORS Project (http://www.actors-project.eu)*, 2008-2011.

[133] M. Wipliez, G. Roquier, M. Raulet, J. Nezan, and O. Deforges, "Code generation for the MPEG reconfigurable video coding framework: From CAL actions to C functions," in *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*, (Hannover, Germany), pp. 1049–1052, 2008.

[134] "ISO/IEC 23002-4:2010 Information technology - MPEG video technologies - Part 4: Video tool library."

[135] M. Wipliez, G. Roquier, M. Raulet, J.-F. Nezan, and O. Déforges, "Code generation for the MPEG reconfigurable video coding framework: from CAL actions to C functions," in *IEEE International Conference on Multimedia & Expo (ICME)*, (Hannover, Germany), 2008.

[136] "Graphiti Editor on Sourceforge.net : http://graphiti-editor.sf.net."

[137] C. Timmerer *et al.*, "Digital Item Adaptation - Coding Format Independence," in *The MPEG-21 Book* (I. Burnett *et al.*, eds.), Chichester, UK.: Wiley, 2006.

[138] C. Audio and V. S. (AVS), "GB/T 20090.2/-2006: Information technology & Advanced coding of audio and video Part2: Video,"

[139] L. Yu, S. Chen, and J. Wang, "Overview of AVS-video coding standards," *Image Communications*, vol. 24, pp. 247–262, April 2009.

[140] M. Ferrati, "Cache-aware scheduling for a dataflow software," Master's thesis, Pisa (Italy), 2010.

[141] C. Xu, C. Von Platen, and J. Eker, "D5A: Wireless Demonstrator," *ACTORS Project (http://www.actors-project.eu)*, 2008-2011.

[142] M. Kralmark and K.-E. Arzen, "D5B: Control Demonstrator," *ACTORS Project (http://www.actors-project.eu)*, 2008-2011.

Personal Publications

[143] C. Lucarz, M. Mattavelli, M. Wipliez, G. Roquier, M. Raulet, J. Janneck, I. Miller, and D. Parlour, "Dataflow/Actor-Oriented language for the design of complex signal processing systems," in *Conference on Design and Architectures for Signal and Image Processing*, 2008.

[144] R. Thavot, R. Mosqueron, M. Alisafaee, C. Lucarz, M. Mattavelli, J. Dubois, and V. Noel, "Dataflow design of a co-processor architecture for image processing," in *Conference on Design and Architectures for Signal and Image Processing*, 2008.

[145] S. S. Bhattacharyya, J. Eker, J. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *Journal of Signal Processing Systems*, vol. 63, pp. 251–263, May 2011.

[146] 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*, vol. 63, pp. 191–202, May 2011.

[147] J. Boutellier, V. Sadhanala, C. Lucarz, P. Brisk, and M. Marco Mattavelli, "Scheduling of Dataflow Models Within The Reconfigurable Video Coding Framework," in *IEEE Workshop on Signal Processing Systems*, 2008.

[148] C. Lucarz, M. Mattavelli, and J. Dubois, "A HW/SW codesign platform for Algorithm-Architecture mapping," in *Conference on Design and Architectures for Signal and Image Processing*, 2007.

[149] C. Lucarz and M. Mattavelli, "A platform for mixed HW/SW algorithm specifications for the exploration of SW and HW partitioning," in *Proceedings of the 17th International Workshop on Power and Timing Modeling, Optimization and Simulation*, vol. 4644 of *Lecture Notes in Computer Science (LNCS)*, pp. 485–494, 2007.

[150] C. Lucarz, M. Mattavelli, and J. Dubois, "A co-design platform for Algorithm/Architecture design exploration," in *IEEE International Conference on Multimedia & Expo*, 2008.

[151] G. Roquier, C. Lucarz, M. Mattavelli, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "An integrated environment for HW/SW co-design based on a CAL specification and HW/SW code generators," in *IEEE International Symposium on Circuits and Systems*, 2009.

[152] J. W. Janneck, I. D. Miller, D. B. Parlour, M. Mattavelli, C. Lucarz, M. Wipliez, M. Raulet, and G. Roquier, "Translating dataflow programs to efficient hardware: an MPEG-4 simple profile decoder case study," in *Design, Automation and Test in Europe (DATE)*, 2008.

[153] C. Lucarz, G. Roquier, and M. Mattavelli, "High level design space exploration of RVC codec specifications for multi-core heterogeneous platforms," in *Conference on Design and Architectures for Signal and Image Processing*, 2010.

[154] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J.-F. Nezan, and O. Déforges, "Reconfigurable Video Coding on Multicore: The Video Coding Standard for Multi-Core Platforms," *IEEE Signal Processing Magazine, Special issue on Multicore Platforms*, vol. 26, no. 6, pp. 113–123, 2009.

[155] C. Lucarz, I. Amer, and M. Mattavelli, "Reconfigurable Video Coding: Objectives and Technologies," in *IEEE International Conference on Image Processing*, 2009.

[156] C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. Janneck, "Reconfigurable media coding: a new specification model for multimedia coders," in *IEEE Workshop on Signal Processing Systems*, 2007.

[157] C. Lucarz, J. Piat, and M. Mattavelli, "Automatic synthesis of parsers and validation of bitstreams within the mpeg reconfigurable video coding framework," *Journal of Signal Processing Systems*, vol. 63, pp. 215–225, May 2011.

[158] 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 *IEEE Workshop on Signal Processing Systems*, 2008.

[159] J. Li, D. Ding, C. Lucarz, S. Keller, and M. Mattavelli, "Efficient Data Flow Variable Length Decoding Implementation For The Mpeg Reconfigurable Video Coding Framework," in *IEEE Workshop on Signal Processing Systems*, (Washington DC, US), 2008.

[160] D. Li, D. Ding, C. Lucarz, S. Keller, and M. Mattavelli, "Validation of bitstream syntax and synthesis of parsers in the MPEG Reconfigurable Video Coding Framework," in *IEEE Workshop on Signal Processing Systems*, 2008.

[161] D. Ding, L. Yu, C. Lucarz, and M. Mattavelli, "Video decoder reconfigurations and AVS extensions in the new MPEG reconfigurable video coding framework," in *IEEE Workshop on Signal Processing Systems*, 2008.

[162] C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. Janneck, "Reconfigurable Media Coding: A New Specification Model for Multimedia Coders," in *IEEE Workshop on Signal Processing Systems*, pp. 481–486, 2007.

[163] J. Dubois, M. Mattavelli, J. Miteran, C. Lucarz, and R. Mosqueron, "Motion estimation accelerator with user search strategy for the RVC framework," in *IEEE International Conference on Image Processing*, 2009.

MPEG Contributions

[164] C. Lucarz, J. Thomas-Kerr, M. Mattavelli, J. Janneck, D. Parlour, A. Kinane, and R. Krisha, "Implement flexible FUs according to the processing mechanism in CVC WD using CAL (Results of Core Experiment 1.1) and analysis of the compactness of RVC Abstract Decoder Model (Results of Core Experiment 1.3)." 2007.

[165] C. Lucarz and M. Mattavelli, "Compression of the RVC DDL Decoder Description with BiM (results of Core Experiment 1.3 in RVC)." 2007.

[166] C. Lucarz, M. Mattavelli, and A. Kinane, "Report on results of RVC CE 2.2: Explore the extensibility of FUs." 2006.

[167] M. Mattavelli, A. Kinane, C. Lucarz, J. Janneck, and D. Parlour, "Report on results of RVC CE 2.1 reshape the current MPEG-4 SP CAL decoder according to the current FU interface in RVC WM." 2006.

[168] M. Mattavelli, C. Lucarz, A. Kinane, K. Radha, J. Janneck, and D. Parlour, "Update of the Textual specification of Functional Units, DDL and FUs SW of the MPEG-4 SP RVC Abstract Decoder Model (Results of CE 2.1)." 2007.

[169] C. Lucarz, M. Mattavelli, A. Kinane, and R. Krisha, "A proposal for the classification and mapping of MPEG video coding technology into Functional Units for the RVC framework (Results of CE 2.2)." 2007.

[170] C. Lucarz, M. Mattavelli, A. Kinane, S. Lee, and S. Lee, "RVC Functional Units naming process proposal." 2007.

[171] D. Ding, M. Mattavelli, C. Lucarz, and L. Yu, "Update of Classification of Tokens for FUs of MPEG-4 SP and MPEG-4/AVC in RVC Framework." 2008.

[172] D. Ding, M. Mattavelli, C. Lucarz, and L. Yu, "Classification of Tokens for FUs of MPEG-4 SP and MPEG-4/AVC in RVC Framework." 2007.

[173] C. Lucarz, J. Thomas-Kerr, and M. Mattavelli, "Reconfigurability potential of the MPEG-4 SP decoder (results of CE 1.1)." 2007.

[174] D. Ding, C. Lucarz, M. Mattavelli, and L. Yu, "Function Units for Conversion from Syntax to Sequence of Tokens: BTYPE." 2008.

[175] C. Lucarz, J. Li, M. Mattavelli, and D. Ding, "Functional Units for RVC Toolbox: Variable Length Decoding." 2008.

[176] C. Lucarz, M. Mattavelli, and D. Parlour, "Serialized version of some MPEG-4 SP FUs." 2007.

[177] C. Lucarz, D. Ding, J. Li, and M. Mattavelli, "BSDL Description of MPEG-4 SP and AVC BP Bitstream Syntax for RVC Framework." 2008.

[178] C. Lucarz and M. Mattavelli, "Implementation of multiple reference frame support in RVC CAL model." 2007.

[179] C. Lucarz, J. Li, M. Mattavelli, and D. Ding, "Auto-generation of RVC Parser from BSDL Syntax Description: Variable Length Decoding." 2008.

[180] C. Lucarz, J. Thomas-Kerr, and M. Mattavelli, "A systematic procedure for the generation of a CAL parser from BDSL in the RVC framework - result CE 1.1." 2007.

Biography

Christophe Lucarz received his M.Sc. degree in electrical engineering from the Institut National des Sciences Appliquées in Lyon (INSA Lyon - France) in 2006. After a specialization in computer systems, he worked at the Processor Architecture Laboratory (LAP) in École Polytechnique Fédérale de Lausanne (EPFL - Switzerland) for his diploma project (*Virtual Memory support for MPEG-4 Accelerators on FPGA*) under the supervision of Paulo Ienne and Marco Mattavelli.

He pursued his work as PhD student within the Multimedia Architectures Research Group (GRAMM) at EPFL under the supervision of Marco Mattavelli. He has been associate editor of a new ISO/IEC MPEG standard, Reconfigurable Video Coding (RVC). He also contributed to its development and with 20 contributions. He fully took part to the ACTORS European Project from 2008 to 2011. During his work, he was the author of 20 publications in journals and conferences.

His interest is mainly how dataflow programming can be made more mainstream for the development of complex digital systems. Parallel programming is becoming compulsory for fulfilling the increasing demand of digital systems and dataflow programming seems an intuitive way to cope with the problem. Nowadays, the energy consumption issue is becoming very important and high level design methodologies should also deal with this issue.

Contact
christophe.lucarz@gmail.com

Christophe Lucarz

PhD candidate in Computer Science

Creative, communicative, autonomous, team-worker.

Objectives

Work in a dynamic team with creativity and initiatives, brainstorm in the purpose of finding new ideas towards a clearly identified objective.

Professional Experience

2006–2011 **PhD candidate**, *École Polytechnique Fédérale de Lausanne (EPFL)*, Switzerland.

My work was about using dataflow programming for the high level design space exploration of systems targeting heterogeneous platforms. I took part in the development of a new ISO standard, Reconfigurable Video Coding (RVC). During the European project (ACTORS), I have developed a framework for the design space exploration of complex systems, including profiling tools, scheduling and partitioning heuristics and performance evaluation methodology.
*Even more than technical skills, my **communication and collaboration skills** have been really improved. Furthermore, I have learned how not to be lost with a large set of data, how to cross them, how to lead constructive communication among a group, how to make efficient meetings, how to develop modular and flexible tools, how to conceptualize facts.*

Mar-Oct 2010 **Voluntary internship**, *World Wide Fund International (WWF)*, Switzerland.

My work consisted in finding solutions to reduce the ecological footprint of the offices.
This experience was very interesting in the sense that I met very interesting people battling for the future of our planet. I have been also introduced to the real problems that can face a company for reducing its footprint.

2006 **Master project**, *École Polytechnique Fédérale de Lausanne (EPFL)*, Switzerland.

I upgraded an existing platform with a virtual memory feature, in order to automate data exchanges between a host PC and the co-processor based on the FPGA platform.
*My **adaptation skills** have been greatly improved. This work was not easy because of the novelty of the language and the concepts that were tackled. It was also a large work of analysis, trying to find the best way to adapt the system given the constraints. Unlike my predecessor on the same project, I have succeeded in adapting this platform with this new feature.*

2005 **Engineer internship**, *Sedatelec*, Lyon, France.

Development from scratch of the software architecture of a medical device for detecting acupuncture points.
*First responsibility in the technical domain, I have learned to be **autonomous** and to ask question to my chief only if necessary. I became aware that any system must be considered as a whole and there are interactions between the electronic and mechanical parts. Good introduction on system design.*

4, allée du stade – 25480 Pirey - France
+33(0)6 84 54 56 85 • ✉ christophe.lucarz@gmail.com
• http://ch.linkedin.com/in/lucarz

Association Activities

2009–2010 **President of Unipoly, students association for Sustainable Development** , *EPFL*.
Responsible of a 80-members association, improvement of the internal organization of the association thought the installation and use of a Wiki, initiator of three new projects (collaborative farm, mini-meeting once a month, sustainable development forum).
It was a great experience for learning how to manage and motivate people to create new projects in sustainable development. Management and communication skills have been once again improved thanks to the weekly meetings with members and the committee. I learned how to convey the right message for motivating people for projects and to manage many projects simultaneously.

2004–2005 **President of the students cycling club**, *INSA Lyon*, France.
Organization of weekly outings among the members of the club.
First experience with such responsibility. It makes me awake about additional administrative aspects of the association world. I got a certain competence in organizing events and motivating people for riding.

Education

2005–2006 **École Polytechnique Fédérale de Lausanne (EPFL, Switzerland)**.
Master in Computer Engineering

2001–2005 **Institut National des Sciences Appliquées (INSA Lyon, France)**.
Degree in Electrical Engineering - Graduated in 2006

Languages

French	**Mother tongue**	
English	**Fluent**	*Multiple travels in English-speaking countries, daily work in English.*
German	**Basics**	*Capable of understanding small conversations and easy texts.*

Computer skills

Frameworks	Eclipse, Visual Studio, OxygenXML, Mathematica, Pin.
EDA	ModelSim, Xilinx ISE, Simplicity Synplify.
Languages	Java, C++, C, XML, XSLT, VHDL, assembly PIC.
Collaborative	SVN, CVS, DokuWiki, CMS.
Application	Microsoft Office, LaTeX.
OS	Windows, Linux, Cygwin.

Interests

Mountain sports	Regular practice of mountain biking, ski touring and hiking.
Artistic activities	Photography, web sites, drawing.
Reading	Philosophy.

4, allée du stade – 25480 Pirey - France
+33(0)6 84 54 56 85 • ✉ christophe.lucarz@gmail.com
• http://ch.linkedin.com/in/lucarz

Publications

[1] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J.-F. Nezan, and O. Deforges, "Reconfigurable Video coding on Multicore: An overview of its main objectives," *IEEE Signal Processing Magazine*, vol. 26, pp. 113–123, 2009.

[2] C. Lucarz, G. Roquier, and M. Mattavelli, "High level design space exploration of RVC codec specifications for multi-core heterogeneous platforms," in *Conference on Design and Architectures for Signal and Image Processing, DASIP*, 2010.

[3] C. Lucarz, M. Mattavelli, M. Wipliez, G. Roquier, M. Raulet, J. Janneck, I. Miller, and D. Parlour, "Dataflow/Actor-Oriented language for the design of complex signal processing systems," in *Conference on Design and Architectures for Signal and Image Processing, DASIP 2008*, pp. 168–175, 2008.

A total of 20 publications in conferences and journals and 20 contributions to the ISO/IEC MPEG Reconfigurable Video Coding standard.

4, allée du stade – 25480 Pirey - France

+33(0)6 84 54 56 85 • *christophe.lucarz@gmail.com*

• *http://ch.linkedin.com/in/lucarz*