High-level synthesis of dataflow programs for heterogeneous platforms: design flow tools and design space exploration

THÈSE N^{O} 6653 (2015)

PRÉSENTÉE LE 29 MAI 2015

À LA FACULTÉ DES SCIENCES ET TECHNIQUES DE L'INGÉNIEUR

GROUPE SCI STI MM

PROGRAMME DOCTORAL EN MICROSYSTÈMES ET MICROÉLECTRONIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Endri BEZATI

acceptée sur proposition du jury:

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

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

The philosophers have only interpreted the world, in various ways.
The point, however, is to change it.
— Karl Marx

To my parents for giving me the chance to pursue my dreams...

Acknowledgements

First of all I would like to thank my parents, Dhimiter and Pavlina Bezati for their strong efforts as immigrants to support me during my studies in France. I would like to thank my brother, Dr. Feliks Bezati for helping me with guidance all those years. I thank my uncle Dr. Anesti Duka for his proposition to pursue my studies in France, my path would have been completely different today. I would like to thank my partner Franziska Thoms for all those years of love, happiness, and support.

I wish to express my gratitude to my supervisor Dr. MER Marco Mattavelli for his guidance, support, advice, and criticism during the last four years. I would like to thank Dr. Mickaël Raulet for believing in me and suggesting me to Marco for pursuing a PhD in my Lab. I wish also to express my gratitude to my *spiritual* mentor Dr. Jörn Janneck for his help and guidance. I would like to thank Dr. Ghislain Roquier for his patience and help that he provided me during the first years of my thesis. Also, I would like to thank Dr. Matthieu Wipliez for the Orcc compiler infrastructure, Herve Yviquel and Antoine Lorence for their maintenance and advancement in Orcc. In addition, I would like to thank all previous developers of OpenForge for their excellent work and Xilinx Inc. for open sourcing it.

I would like to thank my *partner in crime* Simone Casale Brunet, both of us have built tools that are working in perfect coordination. Also, I would like to thank all my past and present lab members.

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

Lausanne, 29 April 2015 Endri Bezati

Abstract

The growing complexity of digital signal processing applications implemented in programmable logic and embedded processors make a compelling case the use of high-level methodologies for their design and implementation. Past research has shown that for complex systems, raising the level of abstraction does not necessarily come at a cost in terms of performance or resource requirements. As a matter of fact, high-level synthesis tools supporting such a high abstraction often rival and on occasion improve low-level design. In spite of these successes, high-level synthesis still relies on programs being written with the target and often the synthesis process, in mind. In other words, imperative languages such as C or C++, most used languages for high-level synthesis, are either modified or a constrained subset is used to make parallelism explicit. In addition, a proper behavioral description that permits the unification for hardware and software design is still an elusive goal for heterogeneous platforms. A promising behavioral description capable of expressing both sequential and parallel application is RVC-CAL. RVC-CAL is a dataflow programming language that permits design abstraction, modularity, and portability. The objective of this thesis is to provide a high-level synthesis solution for RVC-CAL dataflow programs and provide an RVC-CAL design flow for heterogeneous platforms. The main contributions of this thesis are: a high-level synthesis infrastructure that supports the full specification of RVC-CAL, an action selection strategy for supporting parallel read and writes of list of tokens in hardware synthesis, a dynamic fine-grain profiling for synthesized dataflow programs, an iterative design space exploration framework that permits the performance estimation, analysis, and optimization of heterogeneous platforms, and finally a clock gating strategy that reduces the dynamic power consumption. Experimental results on all stages of the provided design flow, demonstrate the capabilities of the tools for high-level synthesis, software hardware Co-Design, design space exploration, and power optimization for reconfigurable hardware. Consequently, this work proves the viability of complex systems design and implementation using dataflow programming, not only for system-level simulation but real heterogeneous implementations.

Key words: High-level synthesis, Dataflow Programing, Clock-Gating, Co-Design, Design Flow, Design Space Exploration

Résumé

De nos jours, les applications de traitement numérique du signal sont de plus en plus complexes dans leurs mises en œuvre et leurs conceptions pour des implantations sur des processeurs embarqués contenant de la logique programmable. Ceci demande le développement de nouvelles méthodologies basées sur un langage à haut niveau d'abstraction. Des recherches antérieures ont montré que pour les systèmes complexes, l'élévation du niveau d'abstraction n'augmente pas nécessairement les coûts en termes de ressources ou la dégradation des performances. En général, des outils de synthèse haut niveau avec une telle abstraction, souvent concurrentiels, améliorent dans certains cas la conception de systèmes de niveau hiérarchique très bas. En dépit de ces succès, la synthèse haut niveau s'appuie toujours sur le principe que les programmes doivent s'adapter à la cible souhaitée sans oublier les particularités du système de synthèse. En d'autres termes, les langages impératifs tels que C ou C ++, les plus utilisés pour la synthèse haut niveau, sont soit modifiés ou soit adaptés en un sous-ensemble de langages pour faire un parallélisme explicite. De plus, une description comportementale appropriée qui permet l'unification de conception de systèmes hétérogènes est toujours un objectif difficile à atteindre. Une des plus prometteuses, capable d'exprimer à la fois l'application séquentielle et parallèle, est exprimée en RVC-CAL. Ce dernier est donc un langage de programmation en flux de données qui permet l'abstraction de la conception, de la modularité et de la portabilité. Les objectifs de cette thèse sont de fournir une solution de synthèse haut niveau pour des programmes écrits en RVC-CAL et de fournir une solution d'implantation pour des plates-formes hétérogènes. Les principales contributions de cette thèse sont : une infrastructure de synthèse de haut niveau qui prend en charge la spécification complète de RVC-CAL, une stratégie de sélection d'action pour la synthèse haut niveau pour permettre la lecture et l'écriture en parallèle de la liste des jetons, un profilage dynamique à grain fin pour les programmes de flux de données qui sont synthétisés, une structure d'exploration de l'espace pour des programmes de flux de données qui permet l'estimation de performance, l'analyse et l'optimisation des plates-formes hétérogènes et enfin une stratégie de « clock-gating » qui réduit la consommation d'énergie dynamiquement. Des résultats expérimentaux sur toutes les étapes du déroulement de la conception, démontrent les capacités des outils de synthèse haut niveau, le Co-Design des parties matériel et logiciel, l'exploration spatiale de l'application, et l'optimisation de puissance pour des plates-formes reconfigurables. En conclusion, ce travail prouve la viabilité de la conception et la mise en œuvre de systèmes complexes en utilisant la programmation de flux de données, pas seulement pour la simulation au niveau du système, mais aussi pour les implémentations hétérogènes.

Mots clefs : synthèse de haut niveau, flux de données, clock-gating, co-design, flot de conception, exploration de l'éspace du design

Abbreviations

ALAP	As Late As Possible
ANSI	American National Standards Institute
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BDL	Behavioral Description Language
BRAM	Block RAM
CAD	Computer Aided Design
CAL	CAL Actor Language
CAM	Computer Aided Manufacturing
CDFG	Control-Data Flow Graph
CFG	Control Flow Graph
CL	Computational Load
CP	Critical Path
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSL	Domain-Specific Language
DSP	Digital Signal Processor
EMF	Eclipse Modeling Framework
ETG	Execution Trace Graph
FDS	Force-Directed Scheduling

FF Flip-Flop Register

FPGA Filed-Programmable Gate Array

FSM Finite State Machine

HDL High-Description Language

HLS High-Level Synthesis

HW Hardware

IDE Integrated Developing Environment

ILP Integer Linear Programming

IP Intellectual Property

IR Intermediate Representation

ISPS Instruction Set Processor Specification

LLVM Low Level Virtual Machine

LUB Least Upper Bound

LUT Look-Up Table

LVA Live Variable Analysis

MDE Model-Driven Engineering

MoA Model of Architecture

MPEG Movie Picture Expert Group

Orcc Open RVC-CAL Compiler

QoR Quality of Results

RAM Random Access Memory

ROM Read Only Memory

RTL Register Transfer Layer

RVC Reconfigurable Video Coding

SoC System on Chip

SSA Single Static Assignment

SW Software

UML Unified Modeling Language

VHDL VHSIC Hardware Description Language

VHSIC Very High-Speed Integrated Circuit

VLSI Very-Large-Scale Integration

XLIM XML Language Independent Model

Contents

List of Figures

List of Tables

1 Introduction

This thesis reports the research work done by the candidate with the aim of yielding a complete high level synthesis (HLS) design flow entirely based on a dataflow computation paradigm that includes functionality and optimizations supporting heterogeneous system designs. Despite continued scaling of silicon technology, individual sequential processors are not becoming faster. Thereupon, rather than building complex single processors, manufacturers have used the space gained from scaling the technology by adding more processors and incorporating reconfigurable parts onto a single chip. Thus, making multi-core and reconfigurable machines a nearly ubiquitous commodity in a full (and increasing) range of computing applications. Therefore, the performance gains of modern heterogeneous platforms are primarily due to an increase in the available parallelism. However, one of the main obstacles that may prevent the efficient usage of heterogeneous platforms is the fact that the traditional sequential specification formalisms and all existing software and IPs, legacy of several years of the continuous successes of the sequential processor architectures, are not the most appropriate starting point to program such parallel platforms. Moreover, such specifications are no more suitable for unified specifications when targeting both processors and reconfigurable hardware components. Another problem is that portability of applications on different platforms becomes a crucial issue, and such property is not appropriately supported by the traditional sequential behavioral description and associated methodologies.

1.1 Design of Complex Systems

The availability of heterogeneous parallel platforms that combines the processing features of FPGAs with multi-core CPUs in a single silicon die offers a potential amount of computing power for embedded designs. That by far exceeds what was available in the past years. However, such possibility can only be fully used if design flows can support heterogeneous architectures. Figure 1.1 depicts a typical Co-Design design flow. The design flow starts by choosing a behavioral description for implementing a chosen algorithm. A preliminary analysis for such a design flow is to decide which parts of the algorithm should be ported in either software and/or hardware processing elements. All relevant information has to be extracted

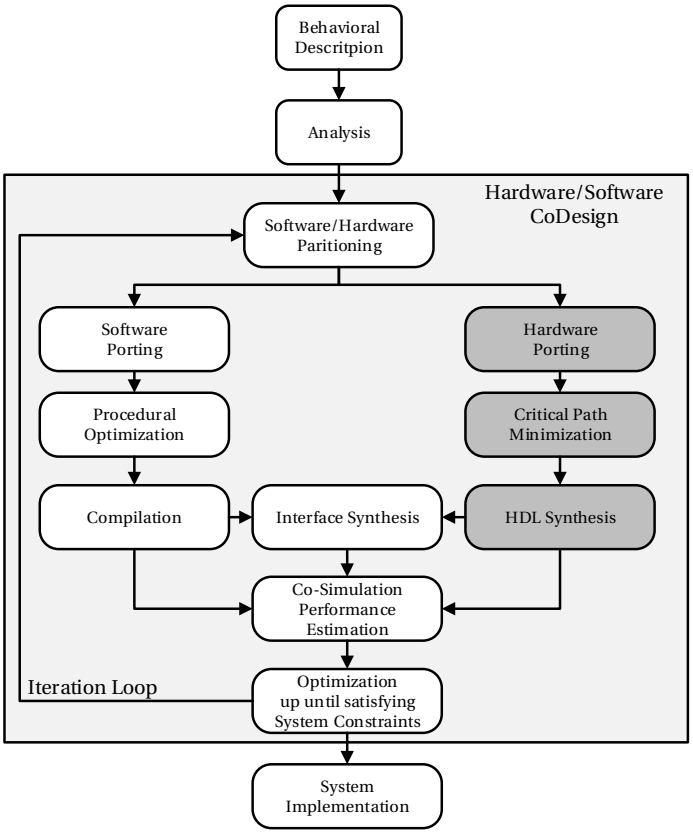


Figure 1.1 – Simplified design flow of a Hardware and Software heterogeneous system

from the behavioral description of several thousand source code lines. This initial step of the design flow has an enormous impact on the necessary optimization iteration needed to satisfy the system constraints for the final target implementation.

The selection of the parts of the algorithm to be implemented by software, or hardware components is a fundamental step that has serious consequences of the other steps of the design flow. In fact, algorithmic parts for SW processing elements require to be revised (i.e. re-written) according to platform specific SW optimization objectives. The other parts selected for execution on hardware components (i.e., FPGA or ASIC) need to undergo different, even more tedious, heavy transformations because specific timing constraints, that greatly affects the quality and required expressiveness of the HDL description, need to be explicitly introduced. In general, the optimization process of the hardware elements consists of reducing the critical path of the circuit. The critical path of a circuit is defined as the longest combinatorial path between two registers. This path represents the frequency of the circuit. Once the timing constraints are satisfied, then an interface need to be implemented for connecting the hardware and software components. An interface is characterized by two factors: its availability of it on the processing element and the interface bandwidth. Once all the system

parts are complete, then the first iteration of co-simulation begins and can give indications of an achievable performance for the developed design.

If the co-simulation results do not satisfy the defined constraints of the system, then parts of software and hardware elements should migrate from one to the other. An additional problem may also occur on the interface bandwidth or its handling. As a consequence, the design space points explode. In addition, if an initial partition is chosen poorly then the number of optimization iteration might increase rapidly. Finally, if a design space point is found that satisfies the constraints, then the system is implemented on the chosen target.

1.2 Problem Statement and Motivation

As stated, the porting of software and hardware part is difficult and a tedious work. A behavioral description must be capable of abstracting the architecture characteristics and must encompass both hardware and software design concepts. Today current design flows requires completely different abstractions for each processing element.

In addition, the average design time necessary for developing and optimizing designs for heterogeneous platforms is, as a result, much higher due to the separation of work in hardware and software parts. A common practice is to partition "a priori" a part of the design to be executed by the CPU and then discover that it does not satisfy the system constraints. In this case, it is necessary to rewrite from scratch a part of the design and rewrite it in a way that it can be executed on the hardware component of the platform. The drawback of the approach is that the two successive specifications of the design, although expressing the same semantic behavior in terms of input and output data, have to make use of two different abstractions for being executed on programmable hardware or on a processing unit. Not only is such a work error-prone, but also the functional design verification should be effectuated by combining both parts. In summary, the main problems in heterogeneous system designs are flexibility and maintainability that both can be expressed as Design Abstraction, Reuse, and Modularity.

- **Design Abstraction**: What level of abstraction should be used for the specification, the design, and the implementation? In the case of heterogeneous platforms, the question is not trivial given the diverse nature of the platform. Different levels of abstraction may be employed depending on the nature and level of requirements and constraints. Thus, behavioral descriptions should be able to seamlessly express both parallel and sequential computation paradigms.
- **Modularity**: In modular design, such as dataflow designs, the system functionality is split into communicating components that divide the complexity of the overall application. The design abstraction should be able to support modularity as a data and task parallelism.
- **Reuse**: Design abstraction and the modularity of an application should permit the reuse

of components that can be described for hardware and software parts and at the same time allow the reuse of parts of the same application family. As an example, many audio codecs share part of the same functional units such as a direct cosine transform, which makes the modularity a necessity for a developer who maintains a library of audio codecs.

The essential difference between hardware and software is the execution model. In hardware everything is executed concurrently; conversely software follows a sequential memory based execution model which is derived from Turing machines. Sequential execution is very efficient in software but a bad choice for hardware in most of the cases. This has a serious implication when designing hardware from software programmers, their familiar algorithms are mostly expressed sequentially. Even thought multi-core and many-core platforms are evolving and becoming the mainstream in all computing devices (even low-cost ones), the last half of the century programmers codes sequentially. C and C-likes programming languages have conquered the sequential software programming, but today there is disagreement about the preferred parallel model of computation that takes on consideration of different parallel architectures. In addition, C or C-like languages have become the mainstream programming languages for heterogeneous computing. Many vendors provide high-level synthesis solutions based on C for the hardware part of their heterogeneous toolchain. As stated in [1], C and C-like languages have the following problems :

- **Sequentiality** : One fundamental problem with C-like High-Level Synthesis tools today is that they encourage the programmer to program algorithms sequentially with the promise that the tool has techniques that will automatically expose the parallelism of a sequential code. Unfortunately those techniques are limited to descriptions that contains few operation dependencies. Concurrency is either supported by libraries or pragmas or automatically detected by the tool, to use non standard C types and to deal with different communication issues. Although a lot of research has been done to support C features and make it as the default language for HLS, in the end all tools have a different implementation of C except some tools that supports a subset of ANSI C. Although SystemC is supporting all features above and it is a useful language for high-level synthesis it is not ideal for multi-core/many-core programming due to its library that is intentioned for circuit simulation. All these different implementation have lead the research community to develop languages based on model of computations that adapts to the hardware development specifics and parallel programming but also to be portable on different platforms.
- **Language Limitations**: Due to the C language concurrency limits, most HLS extends C with statements and/or parallel constructs, pragmas and libraries. Extending C with statements or adding non standardized constructs introduces a fundamental and far-reaching change to the language which makes it incompatible with standard C compilers and other C HLS tools. A less intrusive way is to use tool specific pragmas. Finally, the

use of tool specific libraries locks the developer to a single tool which makes it very difficult for a company to switch HLSs.

- **Bitwise Types**: Each base type in C or C++ is one or more bytes stored in memory. C types can be implemented in hardware but types smaller than a byte can not be specified, expect for defining the number of bits in the field of a structure. HLS tools approach of supporting bitwise types is exactly the same as concurrency. They either change the C language or their provide proprietary libraries.

A potential candidate which is not limited only to hardware development is the CAL dataflow programming language. CAL offers the hardware developer the necessary constructs for expressing parallel and sequential code, bitwise types, a consistent memory model, and a lossless communication between parallel tasks through queues. Thus, CAL can be used as a single behavioral description for SW and HW processing elements. Most of all, CAL comes with a model of computation that enables the programmer to express applications as network processes. The following points describe the properties of CAL:

- **Concurrency & Parallelism Scalability**: In parallel programming, programs either scale with the size of the problem or with the size of the code. Developing concurrent parts of a program without much interference is a well known problem for von Neumann architectures. As a solution, the explicit concurrency of the actor model provides a parallel composition mechanism that tends to lead to more parallelism as the size of an application grows.
- **Modularity**: The hierarchical structure and the encapsulation of actors provides high potential of parallelism. In addition, changing an actor will not have an impact on other actors.
- **Scheduling**: Like procedural programming languages, actors offers a full control on the order of execution of actions (i.e. the imperative part of actors).
- **Portability**: For heterogeneous platforms portability is still en elusive goal. Using a single representation permits the maintainability and reuse of code between HW and SW processing elements.

CAL dataflow programming is a challenging programming paradigm, it offers a flexible development approach to deal with the increasing complexity of the applications, and offers a large degree of parallelism to exploit the massive parallel capabilities available in modern architectures. The use of a programming language based on the dynamic dataflow model is more advantageous compared to static approaches. This is due to the fact that the developer can be more flexible when expressing their design. Dynamicity, facilitates the conception of complex applications with non-constant data structures. Moreover, these dynamic dataflow

languages offer a large expressive power along with a practical syntax that are both required for an industrial-scale development.

Thesis: *Dataflow Programming contains the necessary features for heterogeneous computing that circumvent the imperative (MoC) limitations of C or C-like programming language because it offers* ***abstraction***, ***concurrency***, ***modularity***, ***analyzability*** *and* ***portability*** *for different processing elements.*

To support this statement this dissertation provides the following contributions:

1. A High-Level Synthesis solution of dataflow programs for heterogeneous platforms.
2. System methodologies for optimization of implementations at high abstraction level.
3. An iterative design space exploration with the purpose to minimize the initial partitioning and assignments to software and hardware processing elements.
4. A design flow supporting all the above features.

Moreover, the thesis describes the implementation of the integrated high-level design flow that embeds new features, optimization tools, and methodological advances into an integrated open source HLS solution for programming heterogeneous platforms. The complete environment called Xronos is available under open source license at: https://github.com/orcc/xronos and has been used by two European projects (ACTORS and VAMPA) and research groups such as INSA of Rennes, Lund University, University of Oulu, University of Cagliari, and Harriot Watt University.

1.3 Design Flow for Dataflow Programs

The architecture of the high-level synthesis design flow and associated design space exploration that offers a complete design flow methodology for programming heterogeneous platforms is illustrated in Figure 1.2. A design contains a set of stages by which, from an abstract representation of the application (i.e. the dataflow program), it is possible to accomplish the synthesis of an integrated circuit or the implementation onto a SW processing unit, or any combination of both elements. Each stage is composed by a single tool or the integration of a composition of tools.

The RVC-CAL design flow is composed by eight stages. The stages are respectively:

1. **Behavioral Description, Architecture, and Constraints**: The design is expressed by the RVC-CAL dataflow programming language based on process network principles. The architecture defines on which kind of platform the design is implemented. The architecture contains operators, media, and links. An operator defines the type of the

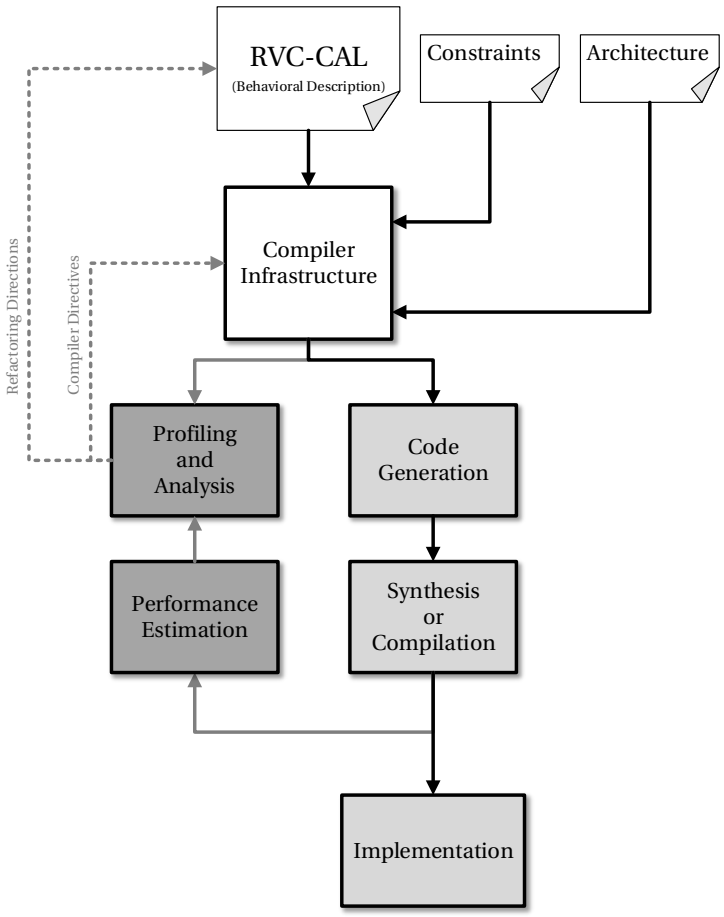


Figure 1.2 – RVC-CAL Design Flow. Two directional flows, in *black* top to down implementation and in *grey* the iterative feedback.

processing element; the media defines the way that this platform is communicating, and the links are the connection between operators and media. In addition, the constraints are applied to the architecture and defines the clocking for each operator and the input and output data consumption and production of the design.

2. **Compiler Infrastructure**: The abstract design specification is verified for algorithmic correctness. The design can also be statically profiled for complexity analysis and for identifying the longest computational path occurring when a set of input vectors are processed.

3. **Code Generation**: According to the architecture defined at the abstract level, the code generation stage generated the source code for execution on the SW architecture and HDL code for configuring the HW architectures, FPGAs and/or ASICs.

4. **Synthesis or Compilation**: Generates code which is then synthesized or compiled using

standard tools to obtain software executables and/or hardware binary files/netlists for physical implementations.

5. **Performance Estimation**: At this stage platform specific software profilers and/or HDL testbenches are used to measure the performance of individual dataflow processing components.

6. **Analysis & Profiling**: At this stage, the design bottlenecks are iteratively identified and analyzed. Initially, in the early phase of the design process, the buffer size for the different dataflow elements, the memory defined in the architecture given by the constraints are estimated and allocated. In a second phase, a more in-depth design space exploration including all dataflow components is applied to the design and profiling information is extracted. For a software architecture, the design can be partitioned into different processing units according to an optimization objective functions for the given set of an input vector and design constraints. For hardware architectures, different multi-clock domain partitioning are identified with the goal for reducing power dissipation and respecting throughputs constraints. Finally, the performance of the composition of hardware and software architectures can be analyzed with the purpose of verifying the satisfaction of the overall system design constraints.

7. **Code Refactoring Directions and Compiler Directives**: After the Profiling & Analysis stage, feedback is provided on how the dataflow program components, at a high abstraction layer, should be modified to satisfy the design constraints.

8. **Implementation**: The structure of the design flow is composed by two main paths. The first is a direct path from the top to bottom linking the high-level dataflow program abstraction to the synthesized executable implementation and the second is the iterative system-level design exploration and optimization cycle.

The structure of the design flow is composed by two main paths. The first is a direct path from the top to bottom linking the high level dataflow program abstraction to the synthesized executable implementation and the second is the iterative system-level design exploration and optimization cycle.

1.4 Thesis Contributions and Organization

The main contributions and the publications (related to each original contribution) of the thesis can be summarized as follows:

1. A high-level synthesis compiler infrastructure that supports the full specification of RVC-CAL dataflow programming language. The compiler infrastructure is based on two open source projects: Orcc for the RVC-CAL fronted, and OpenForge for the Verilog

backend. Research in [2, 3, 4, 5, 6] extends Orcc to a state-of-the-art compiler and provides the necessary transformations and optimizations to produce a close to hardware intermediate representation for OpenForge.

2. An Action selection algorithm for the actor execution model. Research in [3, 4] provides support for RVC-CAL "repeat" statements and guarded conditions on values of an input list. In addition, parallel read and write of list tokens for hardware synthesis accelerates the consumption and production of tokens.

3. Static and dynamic fine-grain profiling data extraction from RTL simulation of synthesized RVC-CAL dataflow programs [3, 7]. The structure of the generated code permits the extraction of timing profiling of the individual execution of each action.

4. A design flow that contains an iterative design exploration framework for RVC-CAL dataflow programs for heterogeneous embedded platforms (FPGA + multi-core CPU). Research in [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] offers a complete design flow for hardware and software from a single RVC-CAL behavioral description. In addition, the design flow includes an open source design space exploration tool that reduces the refactoring iterations for meeting the design constraints.

5. A clock gating strategy that reduces the dynamic power dissipation in synthesized circuits for processes that communicates through queues [19].

This dissertation is divided into six parts that represent each step of the design flow:

Chapter 2 describes the state of the art on the high-level synthesis and design flows for heterogeneous platforms. An introduction to different types of synthesis and a brief description of heterogeneous architectures is discussed. Moreover, the definition of high-level synthesis is provided as well as the high-level synthesis tools evolution. In addition, the state-of-the-art of principal high-level synthesis blocks such as scheduling, pipelining, and power optimization is given. Furthermore, the state of the art of design flow for co-design is also provided. The chapter concludes, by justifying why the RVC-CAL dataflow programming language is chosen for describing the behavioral description of the proposed design flow.

Chapter 3 Reports CAL and the standardized version of it the RVC-CAL. After that, examples that demonstrate the expressiveness of CAL are given. Furthermore, Orcc, the compiler infrastructure for RVC-CAL is presented and its Dataflow and Procedural Intermediate Representations are analyzed in depth.

Chapter 4 describes Xronos the high-level synthesis and embedded software synthesis tool of dataflow programs for heterogeneous platforms. The evolution of Xronos tool is discussed. After that, the advancements and completion that were introduced to Orcc's compiler infrastructure for enabling hardware synthesis are covered. An analysis in depth is also given for the construction of the *Action Selection* procedure that allows a hardware friendly execution model of the actor transition system that support the concurrent reading and writing

of multi-tokens in actions. In addition, the Language Independent Model (LIM), a close to hardware intermediate representation, and the mapping of the Dataflow and Procedural IR to LIM is presented and examined. With the purpose to give a detailed overview of the Xronos synthesis process. Subsequently, the embedded software and synthesizable SystemC code generation are described. Finally, experimental results demonstrate the capabilities of the tool for behavioral synthesis and heterogeneous software and hardware synthesis.

Chapter 5 describes the TURNUS tool for the iterative design space exploration of dataflow programs. The chapter focuses on design exploration and optimization functionalities. It is also demonstrated how run-time profiling data is extracted from heterogeneous platforms. Design performance is estimated by the use of an execution trace post-processor. It is illustrated how estimation results are used in order to guide the optimization heuristic during the exploration phases. Furthermore, the concept of design space critical path is defined and used as primary metric of the optimization heuristics that are incorporated through TURNUS. Finally, the iterative design space exploration methodology is presented, and experimental results demonstrate the tool capabilities and its usefulness for optimizing the throughput of a dataflow program for hardware synthesis.

Chapter 6 describes a clock-gating strategy that reduces the dynamic power dissipation on systems described as processes that communicates through buffers. Moreover, the FPGA clocking architecture is given, and the kind of clock buffers used for enabling coarse-grain clock gating is described. Finally, experimental results demonstrate that the throughput of a particular design is not modified by the clock-gating strategy, and does not at all affect the design flow.

Finally, chapter 7 summarizes the thesis and highlights future work and potential improvements in the overall design flow for hardware, software, and interface synthesis.

2 State of the Art

2.1 Introduction

As the race of minimizing the transistor footprint close to a single atom still goes one, the number of transistor on future chip will furthermore increase. The need for new design automation methodologies on more abstract levels, where cost to market/functionality and trade-offs is easier to comprehend, is a must for developing future chips generations. Today, VLSI technology has reached maturity level, and it is well understood and no longer provides a competitive edge by itself [20]. The industry now is focusing at the product development cycle with the purpose to increase productivity, where high-level synthesis (HLS) plays a central role. Thus, enabling the automatic synthesis of high-level untimed or partially timed to low-level cycle-accurate RTL specifications for efficient implementation in reconfigurable hardware.

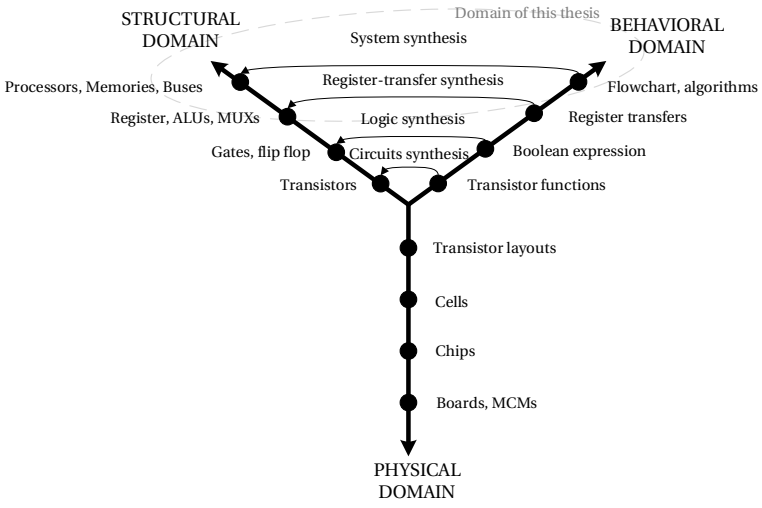


Figure 2.1 – Gajski's Y-Chart, for different types of synthesis.

Driven by this complexity, there has been a renewed interest (from 2000) in high-level synthe-

sis of digital circuits from behavioral descriptions. A key change that has taken place since HLS was first explored in the 70s is that now RTL languages such as Verilog and VHDL are widespread accepted. Design synthesis is a process that translates a behavioral description into a structural one. In [21, 20] Gajski and al. used the Y-chart (Figure 2.1), a tripartite representation of design. The axes in the Y-chart represents three different domains of description: behavioral, structural, and physical. Gajski describes that the level of description becomes more abstracts as we move farther away from the center of the Y-chart. So a design tool and what information is used by the tool can be represented as arcs along domain's axis or between the axes. Today, the industry masters solutions for both the structural and physical domains, but in behavioral domain of high-level synthesis there are still open problems that have not yet being fully solved. Despite the past failure of early generations of commercial HLS tool, there is a demand for high-quality HLS solutions for:

- **An increasing silicon capacity**: requires a higher level of abstraction. Design abstraction is one of the most effective method for controlling complexity, maintainability, reuse, modularity, and improving design productivity. A recent study [22] shows that code density can be reduced 7 to 10 times when moved from an RTL to a high-level C specification.
- **Embedded processors in SoC**: FPGA market leaders offers heterogeneous processors and reconfigurable logic on the same die. In addition, the programmable logic part offers digital signal processors (DSPs), memories and custom logic. That is to say, more software elements can be involved in the process of designing complex embedded heterogeneous systems. An HLS, included in a design flow, allows developers to specify functionality in high-level behavioral description for both embedded software and reconfigurable hardware. In other words, developers can quickly experiment software/hardware partitioning and explores trade-offs such as performance,area, and power from a single behavioral description.
- **IP reuse**: improves design productivity. As opposed to RTL Intellectual Property (IP) which has a fixed interface protocols and mirco-architecture, behavioral IP can be repurposed to different technologies (a greater range of FPGA families and constructors) or system requirements.

The rest of this chapter summarizes the state-of-the art of HLS and hardware and software design flows that partially solves the previous problems. In the following, the HLS tools generations through the years, and cites previous work that have been effectuated for each step in high-level synthesis flow. In addition, several dataflow design flows that permits the HW and SW co-Design are also cited. This chapter concludes on the need for a new behavioral description programming language that does not only fit for HLS but also for parallel architectures such as many cores/multi-cores and hybrid hardware and software architectures.

2.2 Heterogeneous platforms

FPGA or Filed-Programmable Gate Array is an integrated circuit designed to be a reconfigurable circuit. The very first FPGA, the XC2064, was constructed by Xilinx and announced on November 1, 1985. An FPGA is essentially a matrix of logic cells called slices. A generic FPGA architecture is depicted in Figure 2.2. Slices, depicted in Figure 2.3, are connected among themselves with Input/Output (IO) blocks through routing channels. Therefore, they are distributed in the FPGA in horizontal and vertical form, and its connection are fixed using a programmable switch matrix.

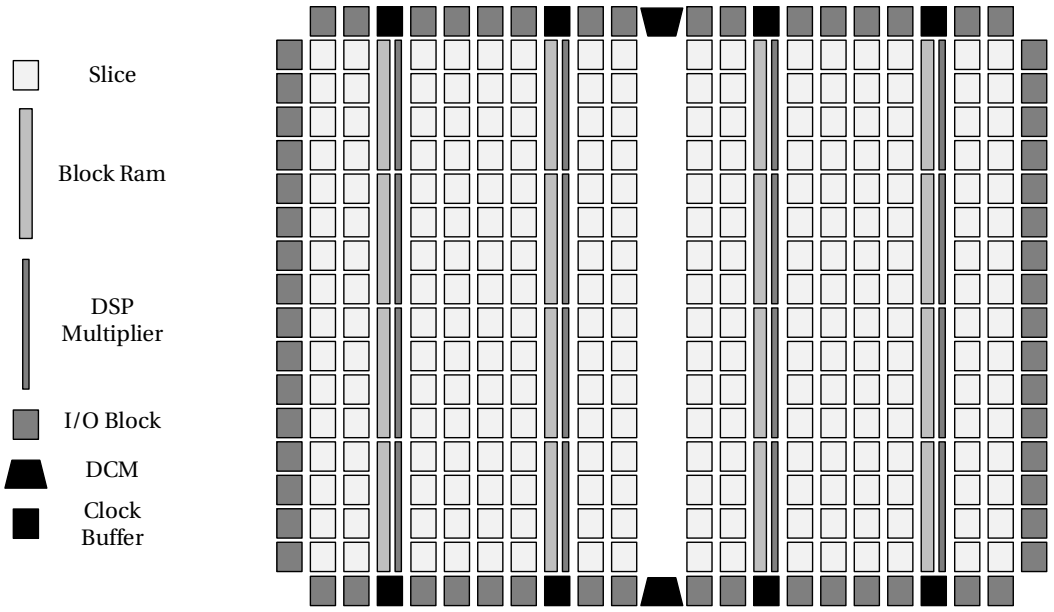


Figure 2.2 – Generic FPGA architecture.

Each slice contains Look-Up Tables (LUT) and several Flip-Flops (FF) and programmable multiplexers (MUX). These elements vary for each FPGA family. General parallel circuits and complex functions are implemented in FPGA when these elements are associated among themselves. Also, FPGAs contains RAM blocks (BRAM) that can be configured in different combination and also acts as ROMs and hard-cabled arithmetic circuits. New Xilinx architecture have added the notion of Configurable Logic Block, which contains two kinds of Slices: SLICEM and SLICEL. SLICEL and SLICEM support LUTs, eight storage elements, wide-function multiplexers and carry logic. In addition, SLICEM supports two additional functions: storing data using distributed RAM (memory in LUTs) and shifting data with 32-bits registers. In Altera FPGAs, the equivalent to SLICE is called ALM or Adaptive Logic Module.

All new Xilinx FPGAs incorporate DSP48s arithmetic modules. A DSP48 slice has a two-input multiplier connected to multiplexers and a three-input adder/subtractor. The multiplier accepts two 18-bit, two's complement operands producing a 36-bit, two's complement result.

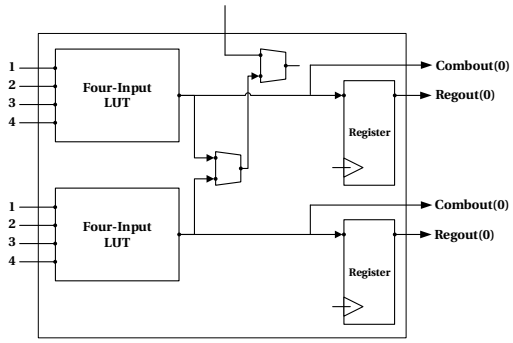


Figure 2.3 – Slice found on Virtex-4 FPGAs

The result is a sign-extended to 48 bits and can optionally be fed to the adder/subtractor. The adder/subtractor accepts three 48-bit, two's complement operands and produces a 48-bit two's complement result. In addition, Altera is adding cabled-circuit floating-point DSP in their FPGAs.

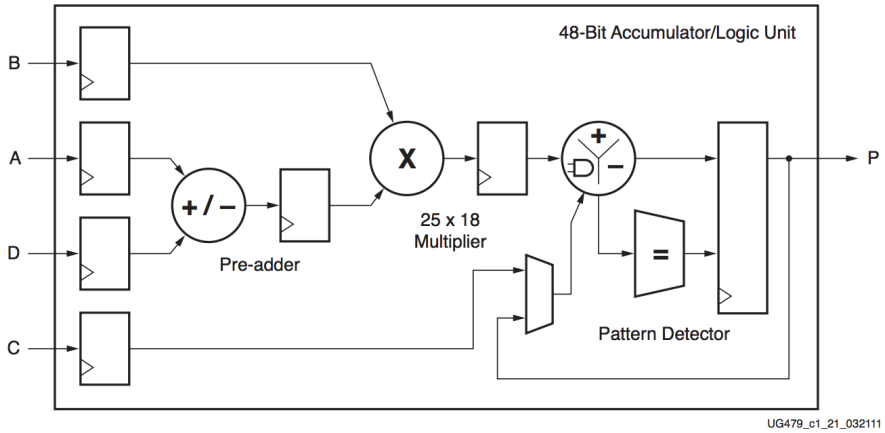


Figure 2.4 – Xilinx DSP48E1 (image courtesy of Xilinx Inc.)

A new trend in both Xilinx and Altera is to add ARM cores along in the same die with the FPGA. The architecture of Xilinx Zynq 7000 is depicted in Figure 2.5 Both of them had already added PowerPC cores but back then, those FPGAs were too expensive, and they had a poor adoption. New Altera SoCss and Zynq FPGA have seen a great adoption by the open source community and companies for co-design applications. The design flow presented in Figure 1.2 fits these architectures perfectly because using a single behavioral description it is possible to target both the CPUs cores and the FPGA.

Finally, most of the commercial HLS tools supports both FPGA and Application Specific Integrated Circuit (ASIC) as a target architecture. An ASICs is an integrated circuit customized for a particular use that is not reconfigurable. Those ICs exists in different types such as Standard Cell, Full Custom ASIC, Gate-Array ASICs. It should be noted that the design flow

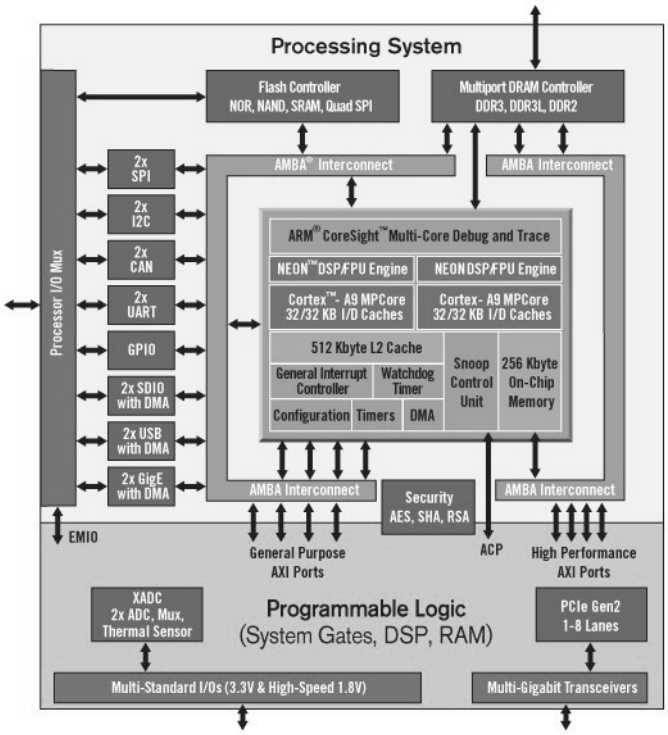


Figure 2.5 – Xilinx Zynq 7000 Architecture (image courtesy of Xilinx)

of this thesis is oriented for FPGAs, but it can also be applied with ASICs too. Even though, it has not been not verified by the author. The interested reader may consult the following references for more information on the ASIC architecture [23] and HLS for ASICs in [20, 24].

2.3 High-Level Synthesis

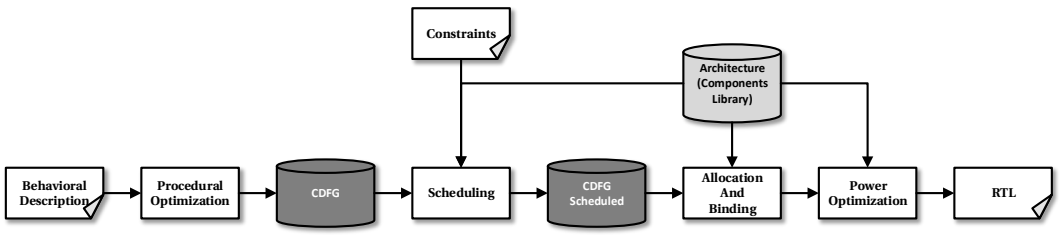


Figure 2.6 – A generic High-Level Synthesis Flow.

A High-Level Synthesis flow consists of a set of steps that from an abstract behavioral representation to a Register Transfer Layer netlist is created. The flow starts with a behavioral description, in which the designer specifies in a formal language the design algorithm. The next step is the compiler infrastructure that parses the behavioral description and produces a single or several Intermediate Representation(s) (IR) of it. In which the compiler is applying a

set of procedural optimizations for reducing and optimizing the description. One of the most used procedural optimization is the pipelining. The next is to create a graph called CDFG (Control-Data Flow Graph) which captures all the control and data-flow dependencies of the given IR. From this graph scheduling is applied. Scheduling is the process that partitions this CDFG into subgraphs so that each subgraph is executed in one control step by taking on account the designer's constraints applied to the behavioral description. Thus, then scheduling converts the behavioral description into a set of register transfers that can be described by an FSM. From the FSM it is derived the control step sequence and the conditions used to determine the following control step sequence, the data path is derived from each register transfer that is assigned to each control step. The datapath in the FSM is a netlist composed of three types of register transfer components such as functional, storage and interconnection. Functional units, such as adders, subtractors, shifts, multipliers etc., execute the operation that are specified in the behavioral description. Storage units, such as registers, RAMs, and ROMs, hold the values of variables generated and consumed during the execution. Interconnections units, such as buses and multiplexers, transports the data between functional units to other functional units and storage units to functional units. After that, allocation and binding follow. Allocation consists of selecting the number and types of components to be used in the design. Binding involves the mapping of the variables and operations in the scheduled CDFG into function, storage, and interconnection units. The next step that is optional, and not effectuated by all HLS is the power optimization. The power optimization optimizes the dynamic and static power dissipation(if an ASIC is the target architecture). The dynamic power dissipation is caused by transistors switching, and as a result charges are being moved along wires. The static is the outcome of the current leakage of the transistors. Power optimization is either optimizing both kinds of dissipations or only the dynamic one. Finally, an RTL netlist is generated by the HLS. The final step of the design is the interfacing of the RTL netlist, and the logic synthesis is applied to the final implementation.

2.3.1 HLS tools evolution

Grant Martin and Gary Smith on [25] have divided the evolution of High-Level Synthesis into three generation and a primal one. The current generation is the third, and most of the tools are C based (including C++ and SystemC). Late 1970s up until 1980s is the primal period, first generation is from 1980s to 1990s, second generation goes from mid-1990s to early 2000s and the third from early 2000s to today.

The first generation was oriented mainly on data-path research. The second generation, mainly commercial products were driven by high description languages (HDL). Third and current generation follows the trend of C-based HLS oriented on datapath applications.

The pioneer tool in "primal" period is the Carnegie-Mellon University design automation or CMU-DA HLS. The design flow is represented in Figure 5.4 and was built by Carnegie Mellon University in the 1970s [26, 27]. Their work focuses on design specification, simulation, and

RTL synthesis. Common software code-transformation, which are used by today tools, like dead-code and redundant sub-expression elimination, constant propagation, code motion are used in the synthesis process. The instruction set processor specification (ISPS) language is used as the design specification [28]. The data-memory allocators perform a mapping function from the algorithmic ISPS description to the data-path part of the hardware implementation. Then the module selection binds the abstract components to a database of specified modules and finally a controller of components is produced.

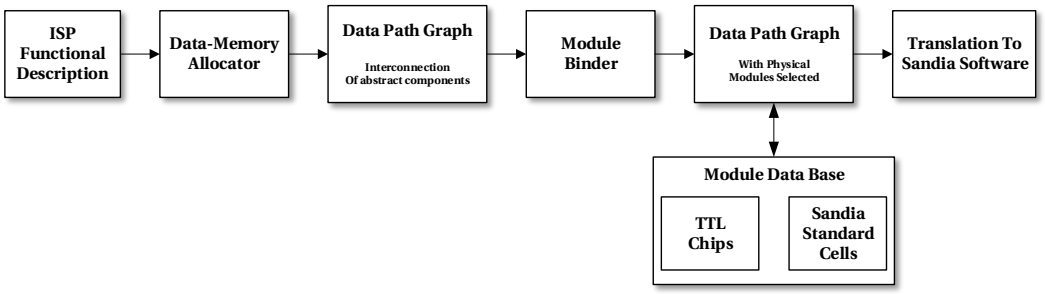


Figure 2.7 – The CMU design system, one of the earliest HLS.

Although it was a groundbreaking research, this old work had a little impact on the industrial design. This is because large electronics companies were not still using or starting adopting CAD/CAM systems at that period.

During the first generation, a plethora of tools for research and prototyping were built. MIMOLA [29, 30] a design method with a purpose to produce digital processors from high-level behavioral specification. The design system combines both compiler construction and hardware oriented concepts. Advance Design AutoMation or ADAM [31, 32] was a unified framework with restricted natural language interface (a dataflow graph representing the behavioral specification) which contained program tools which synthesized RTL designs from behavioral descriptions and the prediction tools which guides the designer in exploring the design space. In addition, the Sehwa [33] tool in ADAM can generate pipelined implementations by exploring the design space. The tool is able to synthesize and perform high-level estimates on the area-delay characteristics of designs and to determine the best design that meets the given constraints. Hardware ALlocater or HAL [34] is a data path synthesis tool with three characteristics. Firstly, it offers the analysis of the input data flow graph and attempts to evenly distribute operations with similar resources with a load balancing technique. Secondly, it provides a global pre-selection of operator cells to full fill speed constraints and register and multiplexer optimizations. Finally, HAL proposed a well-known scheduling technique called the force-directed scheduling on [35] and conflict-graph graph coloring technique for sharing resources in the datapath [34]. Flamel [36], a Pascal to gates HLS, extracts parallelism from block-level transformations.

Hercules/Hebe [37, 38] is a C to gates HLS. Hercules introduced a method called *Reference*

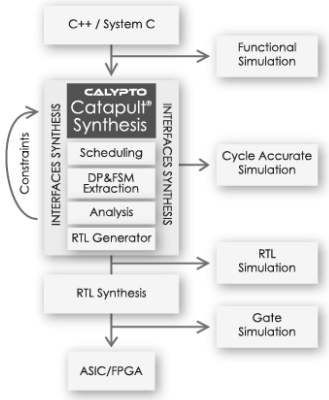
Stack a one-pass transformation on the parse tree that resolves variable/constant unfolding, conditional assignments, and multiple arguments. This transformation, also provides information to structural synthesis that minimizes the number of registers. In addition, Hercules introduces an elegant way to handle operations with unbounded delay called *relative scheduling* [39]. Hyper/Hyper-LP [40, 41] is a high-level synthesis system oriented on power minimization by using architectural and computational transformations. Cathedral/Cathedral-II was specially designed for the synthesis of digital signal processing hardware [42]. In addition, proprietary in-house tools from IBM [43], Philips [44], Motorola [45] were also developed.

The second generation started when major EDA companies such as Cadence, Mentor, and Synopsis begun to offer behavioral HDL to RTL synthesis. Most important such as Mentor's Monet [46] and Cadence's Visual Architect [47]. In [25] several reasons are highlighted for the failure of the second generation. First it failed to replace the established RTL design because HDL languages were not popular among system designers, and there was a need for a new steep learning curve. In addition, the quality of result were often widely variable and unpredictable, hard to validate result because no verification methods were available at that time, HLS produces poor results for control dominated algorithms, and simulation time were almost as long as RTL synthesis.

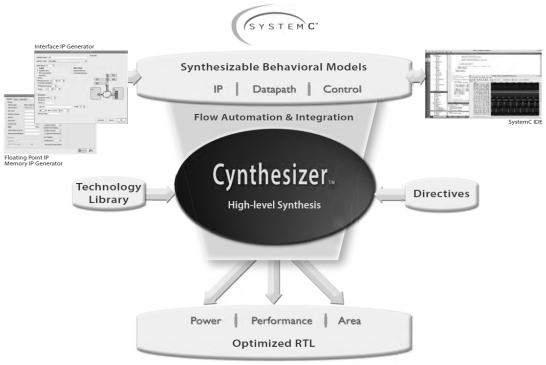
Third-generation HLS tools focuses on using C or C-like languages as behavioral descriptions. As discussed already on the Introduction chapter, the lack of expressing parallelism with C motivated academics and companies to introduce additional languages extensions and restrictions to make C more compliant to hardware synthesis. In this way, the developer is discouraged to use dynamic structures such as pointers with unknown bounds and recursive functions. Most famous third generations tools are HardwareC [39], SpecC [48], Impulse-C [49], Forte's Cynthesizer [50] now acquired by Cadence, Calypto's Catapult [51], NEC Cyber Workbench [52], Synopsis SynphonyC [53] and AutoPilot xPilot acquired by Xilinx and called Vivado HLS [54]. Often tool vendors restrict publications of benchmarking results, but indications shows that third generation HLS tools is achieving reasonable success. In addition, third generation tools are better that the second one because they focus on domain application (Dataflow and DSP design) by achieving reasonable good designs. Furthermore, they provide the "right" input languages that suits more application software developers(eg. C-like and Matlab). Finally, the quality of results are better because HLS can take advantage of compiler-based optimization (Procedural optimization).

2.3.2 Behavioral Description

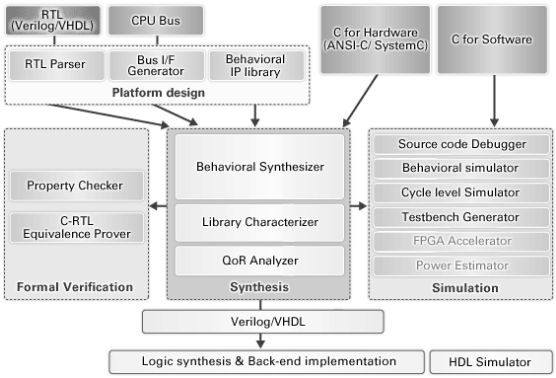
High-level languages make writing, debugging and verification of complex applications more efficient. The degree of how *high-level* a programming language can be depends on the context. For a VLSI engineer, VHDL is considered high-level when he considers custom design, for firmware engineer C is seen as high-level but for a web programmer C is *very* low-level compared to JavaScript. Each programming language is built for a need. C was developed for



(a) Calypto Catapult.



(b) Forte Cynthesizer.



(c) NEC CyberWorkbench.

Figure 2.8 – Three of the most used third generation HLS in the market. The three of them focuses on HLS for ASICs. Catapult and recently CyberWorkbench offers also FPGA support.

executing sequential code on a sequential processor and VHDL and Verilog were designed for replacing hand-written RTL designs.

Most of the programming languages today are sequential. Although, parallel processors are becoming the standard i.e. multi-core and many cores CPUs, general purpose GPUs, and

hybrids multi-core besides an FPGA. Currently there is no parallel languages that target all of them with a single representation. As discussed, for classic programming language like C, libraries are used for exploiting the parallelism, only recently there is native thread support within C++11. Currently only data-parallelism is efficiently supported by OpenCL and CUDA APIs for GPGPUs. FPGAs are natural parallel machines, and Verilog and VHDL exploit their full capability on task and data parallelism, but HDL languages are unacceptable to most application software developers.

In this section, the state of the art of HLS languages and their corresponding tool is given, with the majority of those being C based.

C-like HLS Languages

One of earliest C-to-Gates tools was Cones [55]. It synthesizes single functions, a cone, into combinational blocks. Those blocks were modeled in the C programming language, using assignments, branching, loops and iterative constraints. Ku and De Micheli developed HardwareC [56] for the input of the Olympus [38] synthesis system. HardwareC is a behavioral hardware language with C-like syntax and much larger expressing power than Cones. HardwareC has extensive support for hardware-like structure and hierarchy, supports concurrency, structural and timing constraints. The Transmogrifier C [57], now called FpgaC, is a small C subset that supports branches, loops, and preprocessor directives. As a disadvantage it does not support multiplication, division, pointers, arrays, structures, or recursion. Celoxica Handle-C extends the C language with constructs for parallel statements and Occam-like communications. CompiLogic C2Verilog or C Level Design, now part of Synopsis, the compiler supports a broad set of ANSI C. It is capable of supporting features such as pointers and dynamic memory allocation. The compiler and its transformations are described in details on this patent [58]. SpecC language by Gajski et al. [48] is a superset of ANSI C that includes many systems and hardware constructs such as FSM, concurrency and pipelining. Although, not all constructs of SpecC are synthesizable, the designer can manually refine the SpecC program into to one that can be. Bach C [59] from Sharp supports explicit concurrency and rendezvous communications. Each operation is sequenced, and arrays are supported but not pointers.

C++ can also be used as an HLS language for synthesizing to RTL, some compilers supporting it can even synthesize a subset of SystemC. SystemC is a C++ library that supports hardware and system modeling. An HLS specialized in SystemC is the Cynthesizer from Forte Design System, which was acquired by Cadence in 2014. Cyntheziser supports a strict set of SystemC's TLM. Calypto's Catapult C, previously Mentor Graphics, performs a behavioral synthesis from a strict subset of the ANSI C/C++ and SystemC. Vivado HLS, prior AutoESL Autopilot, is one of the most recent HLS tools. Vivado HLS also synthesizes to RTL from C, C++, and SystemC. It is based on the open source LLVM framework, and it uses Clang the LLVM C frontend. As an advantage, with every iteration of LLVM, Vivado HLS naturally retrieves the latest optimizations on LLVM's compilation techniques. NEC CyberWorkbench targets behavioral synthesis, and it has been used in industry for many years. CyberWorkbench supports BDL [60] and even

thought it deviates from C by adding support for I/O ports, specific types and operators, explicit clock cycles and pragmas it can also synthesize C++ and SystemC.

StepNP [61] is a system-level exploration framework based on SystemC targeted at network processors. It enables rapid prototyping and architectural exploration and provides well-defined interfaces between processing cores, co-processors, and communication channels to allow 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 [62] takes as input a SystemVerilog or a SystemC subset and manipulates it with technology derived from term rewriting systems (TRS) 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.

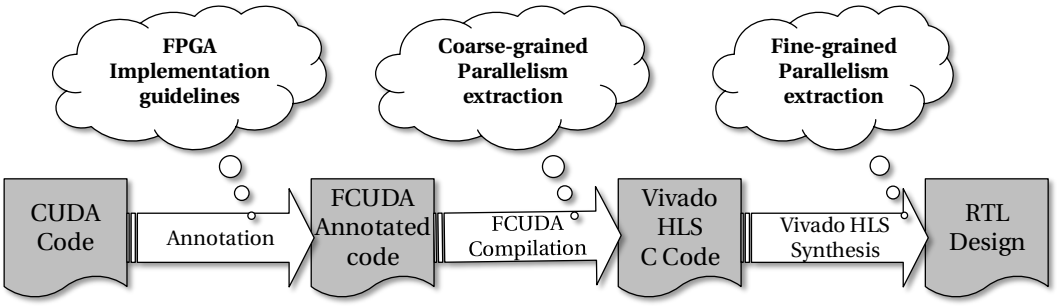


Figure 2.9 – FCUDA: CUDA to FPGA Flow.

CUDA and OpenCL general purpose GPU development C-like languages have expanded their capabilities for hardware synthesis. FCUDA [63] adapts the CUDA programming model into an FPGA design flow that maps the coarse and fine grained parallelism exposed in CUDA onto the reconfigurable fabric. The primary goal of the FCUDA is to convert thread blocks into C functions, and them use a C-to-gates HLS for synthesis. SOpenCL [64] generates hardware circuits from OpenCL programs as FCUDA does. SOpenCL is based on a source-to-source code transformation step that coarsens the OpenCL fine-grained parallelism into a series of nested loops, and on a template-based hardware generation back-end that configures the accelerator based on the functionality and the application performance and area requirements. Altera's OpenCL SDK permits the developers to test their algorithms on a personal computers and then, their OpenCL compiler converts the OpenCL program into an FPGA bitstream.

Other programming Languages used for HLS

JHDL or Just Another hardware Description Language [65] is an HLS language that focuses on designing circuits through an object-oriented approach. JHDL synthesizes Java 1.1 without further language extensions. Sea Cucumber [66] is another Java HLS that permits developers

to describe the circuit coarse-level parallelism as concurrent threads. Kiwi [67] is a C# based HLS that accepts the intermediate language output from either .NET or the open source Mono C# compiler and generates Verilog. Pebble [68] is a language for parameterized and reconfigurable hardware design with a simple block-structured syntax. The objective of Pebble is to support development of designs that can be reconfigured in run-time. Esterel [69] is a synchronous programming language for developing systems that react continuously to their environment.

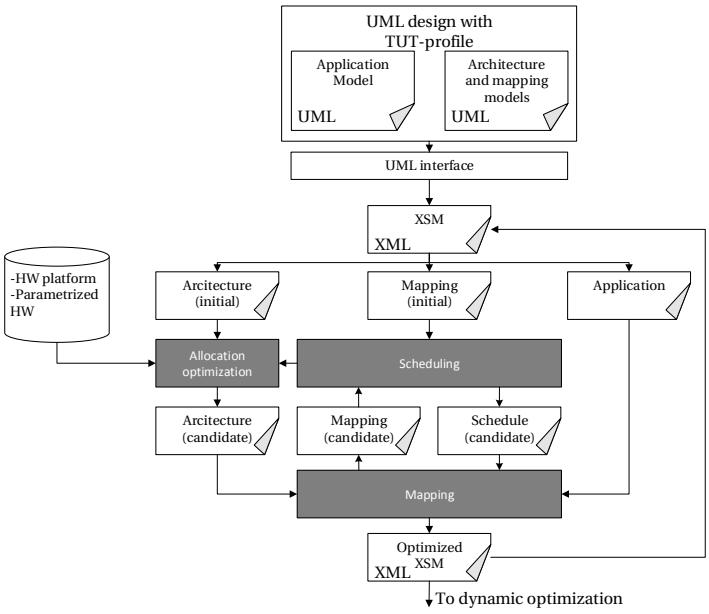


Figure 2.10 – Koski a UML based Design Flow for HW-SW prototyping.

Unified Modeling Language (UML) is used in software engineering for designing large software programs. A complete design flow using UML for system modeling is achieved by Kukkala et al. and is called Koski, represented in Figure 2.10. The target of the Koski design flow [70] 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 an application specific implementation. Hailpern et al. [71] highlighted that graphical languages are not well accepted because it is slower to use than writing code.

2.4 Scheduling of Operators, Operators Pipelining and Power Optimization in HLS

2.4.1 Scheduling of Operators

Scheduling is the process of splitting the IR into states and control steps. With the purpose to do a temporal mapping of the given representation. Any behavioral description and its intermediate representation consist of a sequence of operators to be performed by the synthesized hardware. The task of scheduling consists on partitioning into time steps such that each or set operations is executed in one-time step.

The most popular way of modeling the operator's scheduling is by using Finite State Machine with Datapath (FSMD). As a matter of fact, FSMDs are used to describe digital systems at the register transfer lever. An FSMD consist of an FSM called control unit and a datapath. The datapath expresses the storage and functional unit necessary for the system. The FSM consists of a set of states that corresponds to time/control step of the scheduling, a set of transitions between the states, and a set of actions involving the datapath that is associated with each transition.

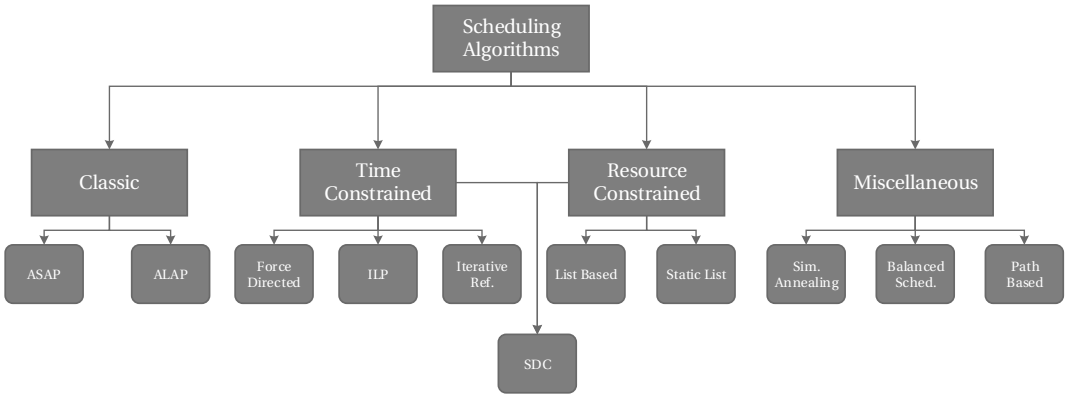


Figure 2.11 – Classification of the most known scheduling algorithms.

In the following the most used scheduling algorithms are described. Scheduling algorithms are separated in three categories: basic algorithms such as ASAP and ALAP, time constrained and resource constrained. Algorithms that combines them all are also described. Time constrained is essential for designing applications in real-time systems such as DSPs where the main objective is to minimize the cost of the hardware. Resource constrained scheduling has as a goal to produce schedules that give the best possible performance but still meet the given resource constraints.

The most basic scheduling algorithms are the As Soon As Possible (ASAP) and As Late As Possible (ALAP). The ASAP algorithm starts with the highest nodes in the CDFG and assigns time steps in increasing order as it proceeds downwards. ASAP considers that a successor node

can execute only after its parents has executed. The algorithm schedules in the least number of control steps, but it does not take on account the resource constraints. Two examples of using ASAP scheduling are Facet [72] from CMU/Bell Laboratories and CATREE [73].

ALAP in contrary to ASAP, starts at the bottom of the CDFG and proceeds upwards. The algorithm gives the slowest possible schedule that takes the maximum number of control steps. This approach is a refinement of the ASAP scheduling with conditional postponement of operations. ALAP is used in MIMOLA [74] system for postponing the concurrency of operators when there are not sufficient functional units.

Integer Linear Programming (ILP) [75] tries to find an optimal schedule using branch and bound algorithm. It also involves backtracking, for example, decisions that were made earlier are changed afterward. The ILP formulation increases rapidly with the number of control steps. For unit increase in the number of control steps, we will have n additional x variables. Therefore, the time of execution of the algorithm also increases rapidly. In practice, the ILP approach is applicable only to a limited set of applications. Heuristics methods such as scheduling one operation at time-based criterion can eliminate the ILP backtracking, thus saving a considerable time of computation.

Force-Directed Scheduling (FDS) [35] is popular for time constraint scheduling. The main goal of the scheduling algorithm is to reduce the total number of functional units used in the implementation of the design. The algorithm achieves its objective by uniformly distributing operations of the same type (for example a multiplication) into all available states. This uniform distribution ensures that functional units allocated to perform operations in one control step are used efficiently in all other control steps, which leads to a high unit utilization. Like ILP, the FDS algorithm uses both ASAP and ALAP to determine the range of the control steps for every operation. The FDS scheduling algorithm it never backtracks on its decision and hence is classified as *constructive* algorithm.

Iterative Rescheduling [76] was developed due to the lack of a look-ahead scheme of the FDS algorithm, which might provide a sub-optimal solution. The idea is to take an initial scheduling given by a scheduling algorithm and tries to reschedule one operation at a time. An operation can be rescheduled into an earlier or a later step, as long as it does not violate the data dependence constraints. The essential issue on rescheduling is the choice of a candidate for rescheduling, the rescheduling procedure and the control of the improvement method. Iterative Rescheduling has is based on the paradigm originally proposed for the graph bisection problem by Kernigham and Lin [77].

List based scheduling [20] is the generalization of the ASAP algorithm. A list based algorithm maintains a priority list of nodes whose predecessors have already been scheduled. A priority function, that resolves any resource contentions, sorts all the operations of the priority list. For each iteration, operations with higher priority are scheduled first and lower priority operations are scheduled to later control steps. If an operator is scheduled to a control step it may make some other non-ready operations ready. Thus, these operations are inserted into the list

according to the priority function. The QoR of the list based scheduling depends on its priority function.

2.4.2 Operators Pipelining

In computing, a pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one [78, 79]. The elements of a pipeline are executed in time-sliced fashion; in that case, pipeline registers are inserted in between elements (pipeline stages). The pipeline stage time has to be larger than the longest delay between pipeline stages. A pipelined system consumes more resources than one that executes one batch at a time, because its stages cannot reuse the resources of a previous stage.

Numerous languages and intermediate representations have been created for describing pipelines. Thus, the programming language C is widely used as behavioral input of pipelining tools [80, 81, 82]. Pipelines can be represented as a data flow graph (DFG) [83], signal flow graph [84, 85] , transactional specification [86] and other notations [87, 88, 89]. They can also be synthesized from binaries [81]. The concurrent algorithmic language, CAL has been developed for representing pipelined networks of actors [90, 91].

A pipeline system is determined by several parameters such as clock cycle time, stage cycle time, number of pipeline stages, latency, data initiation interval, turnaround time, and throughput. A pipeline synthesis problem can be constrained either by resource or time or a combination of both. Given the available hardware, the objective of a scheduler is to find a pipeline schedule with maximum performance. Given the constraint on the throughput, the goal of a scheduler is to find a pipeline schedule consuming minimum hardware.

An important concept in circuit pipelining is re-timing, which exploits the ability to move registers in the circuit in order to decrease the length of the longest path while preserving its functional behavior [92, 93, 94]. The aim of constrained min-area retiming is to minimize the number of registers for a target clock period, under the assumption that all registers have the same area. In the retiming problem, the objective function and constraints are linear, so linear programming techniques are used to solve this problem. Retiming assume that the degree of functional pipelining has already been fixed and consider only the problem of adding pipeline buffers to improve performance of a circuit.

Sehwa [33] can be considered as the first pipeline synthesis program. For a given constraint on resources, it generates a pipelined data path with minimum latency. Sehwa minimizes the time delay using a modified list scheduling algorithm with a resource allocation table. The force-directed scheduling that is proposed in [35] and modified in [85, 95] performs a time-constrained functional pipelining. ATOMICS [96] performs loop optimization starting with estimating a latency and inter-iteration precedence. The pipelined DSP data-path synthesis system SODAS [84] takes a signal flow graph as input and generates a trade-off in pipeline designs by changing the synthesis parameters such as data initiation interval, clock cycle

time and number of pipeline stages. In [97] an adaptation of ASAP list scheduling and an iterative modulo scheduling is used for design space exploration based on slow, but area efficient modules and fast but area consuming modules. Speculative loop pipelining from binaries is proposed in [81]. It speculatively generates a pipeline netlist at compile time and modifies it according to the result of runtime analysis. The automatic pipelining in [86] takes user-specified pipeline-stage boundaries and synthesizes a pipeline that allows concurrent execution of multiple overlapped transactions in different stages.

Integer Linear Programming is a very popular formulation of pipeline optimization problems, although pipeline parameters can often be precisely described only by nonlinear functions. Spaid [98] finds a maximally parallel pattern using ILP. In [99] an ILP formulation and a reduction of it are used for rapid pipeline design space exploration. An ILP formulation of the minimization problem, when delay buffers may need to be introduced for synchronizing the data paths, is proposed in [99].

Pipelining is an effective method for optimizing the execution of a loop with or without loop-carried dependencies. Highly concurrent implementations can be obtained by overlapping the execution of consecutive iterations. Forward and backward scheduling is iteratively used to minimize the delay in order to have more silicon area for allocating additional resources that in turn will increase throughput. The loop winding method was proposed in Elf [100]. The percolation based scheduling [101] deals with the loop winding by starting with an optimal schedule [102] that is obtained without considering resource constraints. PLS pipelining is an effective method [72] to optimize the execution of a loop especially for DSP. The rotation scheduling of loop pipelining by means of retiming processing nodes is introduced in [83]. In [80] a pipeline vectorization method is proposed that pipelines the innermost loops in a loop nest based on removing vector dependencies. Known transformation techniques such as loop unrolling, tiling, merging, distribution and interchange are adapted to pipeline vectorization.

2.4.3 Power Optimization

Due to the rapid growth of personal wireless communication, power optimization has attracted a lot of attention. Most research in power estimation and optimization of VLSI circuits has concentrated on the logic and lower levels of the design hierarchy [103, 104, 105]. Yet, several research publications [106, 107, 108] have show that most power saving in power consumption is often obtained at the higher level of design hierarchy.

First efforts on architectural power optimization was presented by Chandrakasan and al. in [106] and [107]. In [106], the authors uses an architectural parallelism by the means of data path replication and pipelining. So that is possible to enable supply voltage scaling for power reduction. Another way of reducing power consumption using compiler transformations was introduced in [107]. In [109], authors analyze activity metrics at high level for adders and multipliers and derive architectural transformations for synthesizing low power circuits. With

the goal to identify data flow graph transformations that reduce overall circuit activity rather than an accurate prediction of power consumption.

Optimizing memory-dominated computations for power consumption was addressed in [110, 111, 112]. In [110] and [112] the crucial impact of memory related power consumption on the global system power budget, in particular for systems with intensive memory access patterns and multi-dimensional signal processing subsystems. In [111] the importance of reducing computational complexity in algorithmic and architectural level has a high impact on power reduction.

Methods for performing data path allocation and assignment with the aim of minimizing the switched capacitance in the data path were in given in [113, 114, 115, 116]. In [113] an allocation method for low power is investigated by optimizing the controller to reduce data path power dissipation. Hardware sharing effects on power dissipation on the switched capacitance and transition activity is presented in [114]. Authors in [115], formulates a minimum cost clique for minimum power consumption for the switching activity of registers that shares different data values. A level design technique to reduce the energy dissipated in switching of the buses is proposed in [116]. A technique based on reducing the activity of functional units during high-level synthesis was proposed in [117]. The use of limited-weight codes to minimize power consumption in buses and I/O circuitry was described in [118].

Another method for reducing power consumption is by using multiple supply voltages, which is well researched and several studies have appeared in the literature. Authors in [119] applies resource and latency constrained scheduling algorithms to minimize power/energy consumption when the resources operate at multiple voltages. A set datapath scheduling algorithms for simultaneous minimization of peak and average power are proposed in [120, 121]. In [122], an algorithm called MOVER (Multiple Operating Voltage Energy Reduction) to minimize datapath energy dissipation through use of multiple supply voltages. An algorithm named MuVoF is proposed in [123] to perform multivoltage multifrequency low-energy high-level synthesis for functionally pipelined datapath under resource and throughput constraints. A dynamic programming technique for solving the multiple supply voltage scheduling problem in both nonpipelined and functionally pipelined data-paths is presented on [124]. An ILP model in [125] offers variable scheduling techniques that consider in turn timing constraints alone, resource constraints alone, and timing and resource constraints together for design space exploration.

FPGAs compared to ASIC chips are perceived as not power efficient because of their large amount of transistors and to their architecture to provide the reconfigurability. To this consequence, FPGAs has been restrained for low power applications. As FPGAs have millions of gates, the increase of design complexity and the need to reduce design time for early time-to-market, there is a need to estimate power consumption at higher level.

Jha and al. [126] present a HLS approach for synthesizing power-optimized as well as area optimized circuits from hierarchical data flow graphs under throughput constraints. Authors

in [127] explores the accuracy of applying Rent's rule [128] for wire length estimation during high-level synthesis for FPGA architectures. Then, due to the importance of switching activity for power estimation, they adopt a fast switching calculation algorithm [129]. After that, they built a simulated annealing engine with a cost function the power estimation. During the annealing process resource selection, function unit binding, scheduling, register binding, and data path generation simultaneously are carried out. Finally, they apply a MUX optimization algorithm to further reduce the power consumption of the design. This approach does not consider multiple-clock design nor clock gating.

Globally Asynchronous Locally Synchronous (GALS) based systems consist of several locally synchronous components that communicate with each other asynchronously. Works on GALS can be divided into three categories; partitioning, communication devices, and dedicated architectures. Dataflow design modeling, exploration and optimization for GALS-based designs has been studied previously by several authors. For example Hemani et al. [130] proposed a GALS design partitioning method for high performance and very large VLSI systems. The system is partitioned into an optimal configuration of synchronous blocks by exploring relationships between power consumption and the number of synchronous blocks which define the granularity of this approach. In this case, the main limitation is that the synchronous blocks have fixed sizes that cannot be changed during the optimization process. Moreover, this approach does not take into account system performance in the optimization process. Shen et al. [131] proposed a design and evaluation framework for modeling application-specific GALS-based dataflow architectures for cyclo-static applications, where system performance, e.g. throughput, is taken into account during optimization. Similarly, Wuu et al. [132] and Ghavami et al. [133] proposed a method for automatic synthesis of asynchronous digital systems. These two approaches are developed for fine-grained dataflow graphs, where actors are primitives or combinational functions.

2.5 Dataflow Design Flows for HW and SW Co-Design

Hardware/Software Co-Design can be defined as the simultaneous design of both the hardware and software to implement in a desired application. It investigates the concurrent design of hardware and software processing elements of complex electronic systems. It tries to exploit the synergy of hardware and software with the goal to optimize and satisfy design constraints such as cost, throughput, and power of the final product. At the same time, it targets to reduce the time-to-market frame considerably. Several works have been conducted on HW-SW co-Design Flows that uses as behavioral description dataflow programming languages or models.

CodeSign [134] framework uses Object Oriented Time Petri Nets (OOTPN) as the modeling language. This language is quite powerful to model real-time systems, and it is possible to analyze it mathematically. The notion of time allows performance evaluation at the early stages of the design. It produced the Moses Tool Suite, a tool for modeling and simulating and

evaluating heterogeneous systems using Petri nets.

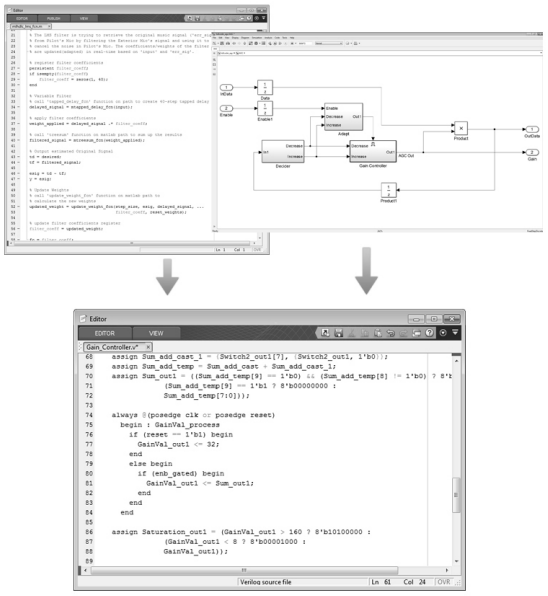


Figure 2.12 – Matlab HDLCoder HLS tool.

The Trotter design flow [135] enables rapid prototyping and design space exploration of applications specified using an internal graph representation of the application. In the very early steps of the design flow, the framework provides useful metrics allowing the designer to evaluate the impact of algorithmic choices on resource requirements in terms of processing, control, memory bandwidth and potential parallelism at different levels of granularity. The goal of their work is to perform automatically and rapidly the algorithmic exploration for the functions called from 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 [136] is a graphical and interactive software implementing the Algorithm Architecture Adequation methodology (AAA). Within this environment, the designer defines an algorithm graph, an architecture graph, and system constraints. Syndex is a Computer-Aided-Design software aiming at mapping an algorithm onto an architecture. The architecture taken into account is only composed of several processors and hardware logic like FPGA cannot be taken into consideration in this flow. The design space exploration is done according one unique criterion, throughput. Another framework based on the AAA principles is Preesm [137], compared to Syndex it offers schedulability analysis and it ensures deadlock freeness in the generated code.

Finally, schematics based tools such as LabView and Matlab recently enabled high-level synthesis on their design flow. National Instruments (NI) LabView FPGA hardware modules [138] permits the LabView graphical development tool to target FPGAs on specific hardware modules developed by NI. MatLab's HDLCoder [139] generates Verilog or VHDL from Simulink and Stateflow designs.

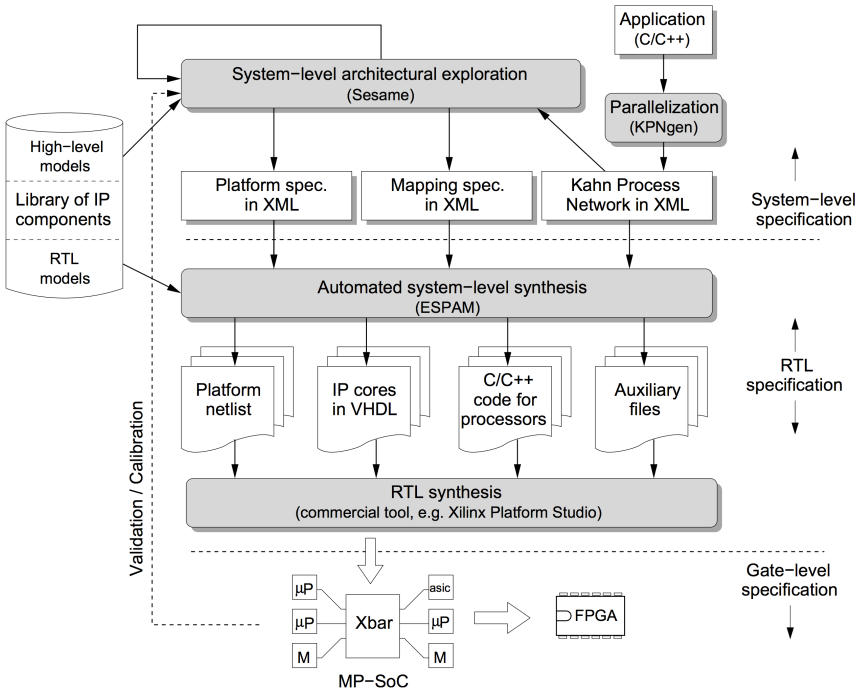


Figure 2.13 – Daedalus Design Flow a unified environment for rapid system-level architectural exploration.

Daedalus [140], depicted in Figure 2.13 provides a unified environment for rapid system-level architectural exploration, high-level synthesis, programming and prototyping of multimedia MP-SoC architectures. The Daedalus framework is an automatic design flow from Kahn Process Networks or directly from C/C++ specifications. Khan Process Network is well suited for signal processing systems. The modeling of interrupts is complicated because of the nature of Kahn Process Network model. Thus, it makes the study of time-dependent systems limited.

Metropolis [141] is a framework allowing the description and refinement of a design at different levels of abstraction and integrates modeling, simulation, synthesis, and verification tools. The function of a system, such as the application, is modeled as a set of processes that communicate through media. Architecture building blocks are represented by performance models where events are annotated with the costs of interest. A mapping between functional and architecture models is determined by a third network that correlates the two models by synchronizing events (using constraints) between them.

Mescal project [142] from University of California at Berkeley aims at designing heterogeneous, application specific, programmable (multi) processors. The goal is to allow the programmer to describe the application in any combination of models of computation that is natural for the application domain. The goal is also to find a disciplined and correct by construction abstraction path from the underlying micro-architecture to an efficient mapping between application and architecture.

The PeaCE Environment [143] specifies a system level design with a heterogeneous composition of three 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 [144]. When dealing with C/C++ specifications, the Peace approach, however, does not propose an automatic procedure to transform this specification into dataflow graphs.

SystemCoDesigner [145] is an actor-oriented approach using a high-level language named SysteMoC, which is built on top of SystemC. It generates HW-SW SoC with automatic design space exploration techniques. The model is translated into behavioral SystemC model as a starting point for HW and SW synthesis. The HW synthesis is delegated to a commercial tool, viz. Forte's Cynthesizer, which generates RTL code from their SystemC intermediate model.

Hardware/Software Co-Design based on RVC-CAL programming language has been studied in all steps of this thesis design flow. A first approach on CAL and RVC-CAL simulation and hardware code generation was provided by the OpenDF [146, 147] framework developed by Xilinx. An alternative to OpenDF for software synthesis called Open RVC-CAL Compiler has been effectuated in [148] and it is the compiler infrastructure used in this thesis. A third experimental compiler infrastructure for CAL developed by Ericsson is called Caltoopia and described in [149]. Hardware and C++ software code generation for Orcc were firstly introduced in [2, 3] and further developed in this thesis. Moreover, Interface synthesis for heterogeneous platforms is presented in [150]. Furthermore, a design space exploration for RVC-CAL dataflow programs is discussed in [10, 7]. Finally, a complete Co-Design environment is presented in [9, 19].

Actor and Dataflow Machines [151, 6] is a machine model for dataflow actors that focuses on minimizing the overhead of action selection for efficient implementations of static and dynamic dataflow programs. Actor machines are used to reduce the runtime testing of conditions, also actor machines can be composed to eliminate testing of port conditions. For this thesis Actor Machines were not used due to the late arrival of a compiler infrastructure supporting them. Finally, in the future works chapter is described that Dataflow Machines is going to replace the Orcc intermediate representation for a better software and hardware code generation that contains fewer tests in the action selection.

2.6 Conclusion

In this chapter, the state-of-the-art of high-level synthesis tools and design flows for heterogeneous platforms were presented. It was shown that the behavioral description of the third-generation HLS tools is mainly C or C-like programming languages. In fact, C languages have an important limitation; they do not express parallelism. As discussed, to circumvent these obstacle vendors and academics have either modified the C language structures or they provide specialized library, or add pragmas that recognized only by a single tool. Alternative languages for HLS either offer a block-structure syntax or the possibility to design systems

that react continuously to their environment.

Three building blocks found in almost all HLS are also introduced and discussed. Those are scheduling of operators, pipelining, and power optimization. The list of the most used scheduling algorithm is given and two types of scheduling problems were considered: time constrained and resource-constrained scheduling. On one hand, the ILP approach solves the time-constrained problem but it has long time executions. On the contrary, FDS finds a solution quickly, but the optimality is not guaranteed. The Iterative Rescheduling improves an initial schedule generated by one the previous schedulers. Moreover, list-scheduling solved the problem of resource scheduling. As discussed, pipelining optimization is a time and resource constrained scheduling problem with the purpose to increase the frequency of the overall system. Power saving methodologies are related to the technology that is being used. Most of the introduced methodologies and strategies are for ASICs, but there is a growing interest for power saving techniques for FPGAs. In addition, there is no strategy that helps to reduce the dynamic power dissipation for dynamic dataflow programs. Thus, power saving techniques are applied or to static dataflow programs or synchronous or the power reduction is effectuated statically during synthesis. A solution for reducing the dynamic power dissipation caused by flip flop switching activities and extends the state-of-the-art in clock-gating is given in Chapter 6.

Furthermore, design flow for hardware and software Co-Design were presented. Each design-flow uses a high-level behavioral description for representing a design, mainly a derivative of C language. In contrast, LabView's behavioral description is a schematic based one. Matlab, Syndex, Deadalus and others provides also a graphical representation of the dataflow dependencies between the process. In addition, part of tools are using a single dataflow model of computations such as the Kahn Process Network or in Mescal a combination of models of computation is used. Finally, an RVC-CAL based flow were also introduced. In contrast, with other design flows the RVC-CAL design flow, which is described and provided with this thesis, permits the description of dynamic systems from a single representation. Moreover, it offers support for both hardware and software processing elements, and a design space exploration that permits the performance estimation and refactoring directions that can be applied for accelerating the system latency (Chapter 5).

3 CAL Dataflow Programming Language

3.1 Introduction

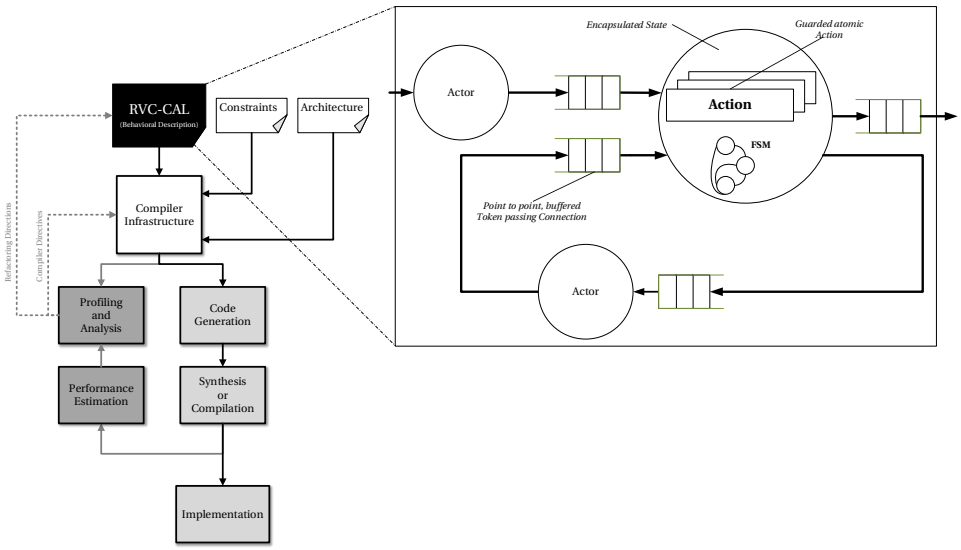


Figure 3.1 – RVC-CAL as the Behavioral Description in the Design Flow.

The emergence of parallel processing elements such as many-cores/multi-cores, FPGAs, GPGPUs demands to rethink the way of programming them. It is widely recognized that programming parallel platforms is difficult and tedious. In addition, heterogeneous platforms consisting on parallel processing elements is becoming a standard on personal computers, a combination of multi-core processors and massively parallel GPUs, and also the introduction of MPSoCs with programmable logic in the industry demands higher level of abstraction. A key to the heterogeneous system level design is the notion of models of computation (MoC) [152]. A MoC is the semantics of the interactions between modules. Moreover, it is the model or the specification principles of a design. Furthermore, MoCs relate strongly to the design style but is not necessary to refer to the implementation technology. Classes of MoCs include:

Imperative, Finite State Machine, Discrete Event, Synchronous Languages, and Dataflow.

The imperative MoC executes the modules sequentially to accomplish a task. In Finite State Machines MoC, an enumeration of set of states specifies the steps to achieve a task. In the Discrete Event MoC, modules react to event that occurs at a given time instant and produces other events at the same time instant or at some future time instant. In Synchronous Languages MoC, modules simultaneously react to a set of input events and instantaneously produce output events.

A Dataflow MoC, is conceptually represented as a direct graph where nodes, called actors, represent computational units, while edges describes communication channels 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. Using Dataflow MoC in such application domains often leads to behavioral descriptions that are much closer to the original conception of the algorithms than if an imperative MoC was used. Dataflow models also date back to the early 1970s, starting with seminal work by Dennis [153] and Kahn [88]. Several execution models that define the behavior of a dataflow program have been introduced in literature [88, 152]. A Dataflow MoC may constrain the behavior of an actor, how actors are executed relatively to each other, and aspects of their interaction with one another. As a result, different MoCs offer different degrees of analyzability and compile-time schedulability of dataflow programs written in them, and permit different guarantees (such as absence of deadlocks or boundedness of buffers) to be inferred from them.

The first two MoC are fundamentally sequential and the last three are concurrent. In fact it is possible to use the first two on parallel processing elements and the last three on sequential machines. Thus, there is a distinction between MoCs and the way that they are implemented. As a consequence, the efficiency might take a hit. In heterogeneous platforms there should be a separation of tasks depending properties of a design. Modules that are sequential should preferably execute on sequential platforms and parallel ones should perform on concurrent machines. In effect, for system level design either should be a mix of different MoCs or the properties and semantics of a single MoC should be rich enough to support heterogeneous designs.

A potential candidate for heterogeneous system level design is RVC-CAL. RVC-CAL is dataflow programming language that is based on the Dataflow MoC and it has the property to express applications as network processes. In fact, it offers parallelism scalability, modularity, scheduling by finite state machines, portability, and adaptivity properties that are necessary to unify the system level design for heterogeneous platforms. The MoC underlying the dataflow networks that are expressed using the CAL formal language is based on the *dataflow process networks* (DPN) model [152]. In addition to the properties of dataflow mentioned above, each DPN actor executes a sequence of discrete computational steps, called *firings*. In each step, an actor may (a) consume a finite number of input tokens, (b) produce a finite number of output tokens, and (c) modify its internal state, if it has any. In an actor language such as CAL [154]

and its subset RVC-CAL this behavior is specified as one or more *actions*. Each action describes the conditions under which it may be fired (which may include the availability and values of input tokens, and the actor's state), and also what happens when it fires, i.e. how many tokens are consumed and produced at each port, the values of the output tokens, and how the actor state is modified. The execution of such an actor consists of two alternating phases: the determination of an action who firing conditions are fulfilled (including a choice if there is more than at some point), and the execution of that action itself.

Table 3.1 – System-Level Requirements and Coverage. With ●supported, ◗partially supported, and ❍not supported.

	C	C++	OpenCL	Java	VHDL	Verilog	HardwareC	SpecC	RVC-CAL
Behavioral hierarchy	❍	❍	◗	❍	❍	❍	❍	●	●
Structural hierarchy	❍	❍	❍	❍	●	●	●	●	●
Concurrency	❍	◗	●	◗	●	●	●	●	●
Synchronization	❍	❍	●	◗	●	●	●	●	●
Exception handling	◗	●	●	●	●	●	❍	●	❍
Timing	❍	❍	❍	❍	●	●	◗	●	❍
State transitions	❍	❍	◗	❍	❍	❍	❍	●	●
Composition data types	●	●	●	●	◗	◗	❍	●	◗
Heterogeneous & CoDesign	❍	❍	●	❍	❍	❍	❍	❍	●
Fine-Grain Profiling	❍	❍	◗	❍	❍	❍	❍	❍	●

Table 3.1 compares traditional languages against a set of language requirements. Partial values are retrieved from [48]. RVC-CAL supports the following behavioral hierarchies: sequential execution inside actions, FSM by the finite state machine of the actor, concurrent and pipelined execution by the MoC. In addition, RVC-CAL supports structural hierarchy by actor composition. An actor composition may contain another actor composition. Furthermore, Synchronization is provided by the FIFO queues that the actors are interconnected. Exception handling is not currently supported. RVC-CAL is high-level language that makes a total abstraction of time. Moreover, RVC-CAL support list types and future version will also support composite types. Finally, this thesis demonstrates that RVC-CAL is a potential candidate heterogeneous system level design and that it supports fine-grain profiling for hardware and software processing elements.

Before describing the CAL programming language and its features, a brief introduction to the Process Networks and the Model of Computations that CAL uses is given in the next section.

3.2 Process Networks

In this section, the Kahn Process Network and Dataflow Process Network MoCs are described. CAL inherits the properties of both models and extends them with the Actor Transition System. A Model of Computation or MoC is a formal representation of the operational semantics of a network of functional blocks describing a computation [155]. Moreover, it allows to specify the algorithm and the cost (i.e. time) of the operations.

3.2.1 KPN

The Kahn Process Networks [88] is a formal model of concurrent computation first introduced by French computer scientist Gilles Kahn in 1974. He introduced a language with simple semantics with the goal of applying mathematical approaches to programming languages and system design. Kahn's model can naturally describe signal processing systems in which infinite streams of data samples are incrementally transformed by a collection of processes executing either in sequence or parallel.

In Kahn's model, a network of processes communicates with each other via unbounded FIFO queues. Kahn describes its model as a set of Turing machines [156] connected via one-way tapes. Each process shares data with each other only through the input and output queues. Each queue may contain a possible infinite sequence of tokens. By using the same notation as in [152], each sequence or a stream is denoted with $X = [x_1, x_2, x_3, \ldots]$ where each x_i is a token drawn from a set. A token is an atomic data object that is *written* (produced) exactly once, and *read* (consumed) exactly once. *Writes* to the queues are *non-blocking*, in the sense that they always succeed immediately, but *reads* from queues are *blocking*, in the sense that if a process attempts to read a token from a queue and no data is available, then it stalls (i.e. wait) until the queue has sufficient tokens to satisfy the read. In other words, it is not possible to test the presence of input tokens.

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

$$F : S^p \rightarrow S^q \tag{3.1}$$

The KPN process F has *event semantics* instead of state semantics such as continuous time which is used in some other domains. Furthermore, the only technical restriction is the need of F a continuous mapping function.

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

The increasing chain (possibly infinite) of sequences is defined as $\chi = \{X_0, X_1, \ldots\}$ where $X_1 \sqsubseteq X_2 \sqsubseteq \ldots$. Such an increasing chain of sequences has one or more upper bounds Y, where $X_i \sqsubseteq Y$ for all $X_i \in \chi$. The least upper bound (LUB) $\sqcup\chi$ is an upper bound such that for any other upper bound Y, $\sqcup\chi \sqsubseteq Y$. The LUB may be an infinite sequence.

Let consider a functional process F and an increasing chain of sets of sequences χ. As defined in the Equation (3.1), F will map χ into another set of sequences Ψ that may or may not be an increasing chain. Let $\sqcup\chi$ denote the LUB of the increasing chain χ. Then F is said to be **continuous** if for all such chains χ, $\sqcup F(\chi)$ exists and:

$$F(\sqcup\chi) = \sqcup F(\chi) \tag{3.2}$$

Networks of continuous processes have a more intuitive property called monotonicity. A process F is said to be **monotonic** if:

$$X \sqsubseteq Y \Rightarrow F(X) \sqsubseteq F(Y) \tag{3.3}$$

A continuous process is monotonic. However, a monotonic process might be not continue. A key consequence of these properties is that a process can be computed iteratively [157]. This means that given a prefix of the final input sequences, it may be possible to compute part of the output sequences. In other words, a monotonic process is non-strict (its inputs need not be complete before it can begin computation). In addition, a continuous process will not wait forever before producing an output (i.e. it will not wait for completion of an infinite input sequence). Networks of monotonic processes are *determinate*. The KPN monotonicity prove is given in [152].

3.2.2 Dataflow Process Network

Dataflow Process Networks (DPN) [152] formally establish a particular case of KPN, where the computational blocks are called *actors*. As for the KPN process, actors can communicate only through unidirectional and unbounded queues that can carry possible infinite sequences of tokens. As for KPN, writes to queues are *non-blocking*. Contrarily, reading from queues is *blocking* because an actor can test the presence of input tokens. If there are not enough input tokens, then the read returns immediately and the actor needs not be suspended as it cannot read. This could introduce *non-determinism*, without requiring the actor to be nondeterminate.

DPN networks naturally extend the KPN embracing the notion of actor firing [153]. Actor firing can be defined as an indivisible (atomic) quantum of computation. The firings themselves can be described as functions, and the invocation of them is controlled by firing rules. Sequences of firings define a continuous Kahn process as the least fixed point of an appropriately constructed function and are therefore formally establishing DPN as a particular case of KPN [158].

An actor with m inputs and n output is defined as a pair $\{f, R\}$, where:

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

The Kahn process F defined in Equation (3.1) based on the actor $\{f, R\}$ has to be interpreted as the least-fixed-point function of the functional $\phi : (S^m \rightarrow S^n) \rightarrow (S^n \rightarrow S^m)$ defined such as:

$$(\phi(F))(s) = \begin{cases} f(r) \bullet F(s') & \text{if there exist } s \in R \text{ such that } s = r \bullet s' \text{ and } s \sqsubseteq s' \\ \Lambda & \text{otherwise.} \end{cases} \tag{3.4}$$

where $\bullet$ represents the concatenation operator, Λ the tuple of empty sequences and $(S^m \rightarrow S^n)$ the set of function mapping S^m to S^n. It is possible to demonstrate that ϕ is both a continuous and monotonic function. The firing function f does not need to be continuous. In fact, it might not be even monotonic. It merely needs to be a function, and its value must be finite for each of the firing rules [158].

3.2.3 Actor Transition System and Composition

Actor transition systems (ATS) [151] describe actors in terms of labeled transition systems. The ATS extends the notion of actor with firings by introducing the concepts of internal state, atomic step and priority. In an ATS, a step makes a transition from one state to another. An actor maintains and updates its internal variables: those are not sequences of tokens, but simple internal values that can not be shared among actors. Moreover, the notion of priority allows actors to ascertain and react to the *absence* of tokens. On the other hand, however, it can also make them harder to analyze, and it may introduce unwanted non-determinism into a dataflow application.

Let Σ denote the non-empty actor state space, $\mathscr{U}$ the universe of tokens that can be exchanged between actors and S^n a finite and partially ordered sequence of n tokens over $\mathscr{U}$. A n-to-m **actor** is a labeled transition system $\langle \sigma_0, \tau, \succ \rangle$ where:

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

Any $(\sigma, s, s', \sigma') \in \tau$ is called a **transition**, where $\sigma \in \Sigma$ is its source state, $s \in S^n$ its input tuple, $\sigma' \in \Sigma$ its destination state and $s' \in S^m$ its output tuple. It must be noted that $\succ$ is a non-reflexive, anti-symmetric and transitive partial order relation on τ, also called **priority** relation. An equivalent and more compact notation for the transition (σ, s, s', σ') is $\sigma \xrightarrow{s \to s'} \sigma'$.

Enabled transition and step of an actor

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

- the required input tokens must be present
- there must not be another transition that has priority

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

$$\begin{cases} v \sqsubseteq s \\ \nexists \sigma \xrightarrow{r \to r'} \sigma'' \in \tau \ : \ r \sqsubseteq v \wedge \sigma \xrightarrow{s \to s'} \sigma' \succ \sigma \xrightarrow{r \to r'} \sigma'' \end{cases} \tag{3.5}$$

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

Actors composition

For any transition relation τ its set of *input ports* P_τ^I and its set of *output ports* P_τ^O are defined as the ports where at least one transition consumes input from or produce output to:

$$\begin{cases} P_\tau^I = \{p \in P \mid \exists \sigma \xrightarrow{s \to s'} \sigma' \in \tau \ : \ \sigma(p) \neq \bot\} \\ P_\tau^O = \{p \in P \mid \exists \sigma \xrightarrow{s \to s'} \sigma' \in \tau \ : \ \sigma'(p) \neq \bot\} \end{cases} \tag{3.6}$$

where P is the set of input and output ports names. It is assumed that an input port with name p and an output port of the same name are in no way related. In order to express complex functionality, actors are composed into a **dataflow network** as the one depicted in Figure 3.2. The structure of a network can be represented by a partial function from (input) ports to (output) ports, mapping each input port in its domain to the output port that connects to it. Note that this implies the absence of fan-in (as every input port is connected to at most one output port), and it permits unconnected (open) input (and output) ports.

3.3 CAL Actor Language

The Cal Actor Language (CAL) [90] is a domain-specific language that provides useful abstractions for dataflow programming with actors. The language directly captures the features of DPN [159] MoC by adding the notion of atomic action firings, also called *steps*.

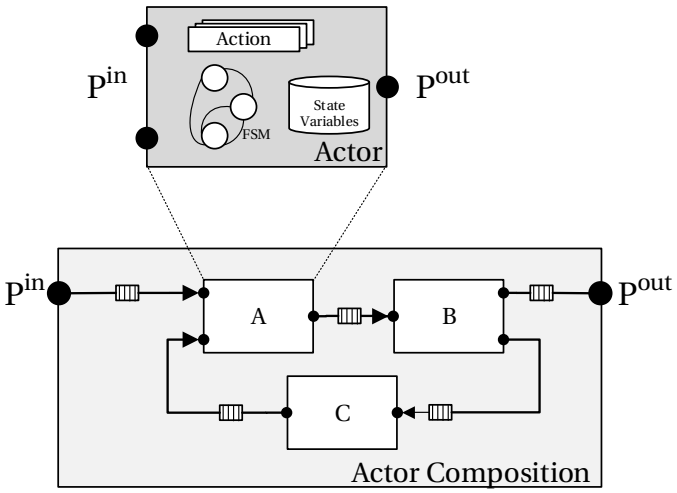


Figure 3.2 – Actor Composition and Actor Structure.

Figure 3.2 illustrates the basic concepts of a CAL program. It represents a dataflow network composed by a set of **actors** and a set of first-in first-out (FIFO) **queues**. Each CAL actor is defined by a set of **input ports**, a set of **output ports**, a set of **actions**, and a set of **internal variables**. CAL also includes the possibility of defining an explicit **finite state machine** (FSM). This FSM captures the actor state's behavior and drives the **action selection** according to its particular state to the presence of input tokens and to the value of the tokens evaluated by other language operators called **guard functions**. Each action may capture only a part of the firing rule of the actor together with the part of the firing function that pertains to the input/state combinations enabled by that partial rule defined by the FSM. An action is **enabled** according to its *input patterns* and *guards expressions*. While patterns are determined by the amount of data that is required for the input sequences, guards are boolean expressions on the current state and/or on input sequences that need to be satisfied for enabling the execution of an action.

In the following, a basic overview of the main concepts concerning the syntax and semantics of CAL language is presented.

3.3.1 CAL Program

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

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

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

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

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

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

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

3.3.2 Execution Model

In this thesis it is assumed that the firing of an action is performed following the serial execution of the four stages illustrated in Figure 3.3. Those stages are respectively:

- **Action selection** where the internal actor scheduler selects the next schedulable action (i.e. that satisfies all firing conditions). It should be noted that the action selection must wait until all the input tokens are available on the respective input queues (i.e. block-reading).
- **Read from input queues** where all the input tokens required by the algorithmic part of the action are read from the respective input queues.

- **Action execution** when the algorithmic part of the action is executed.
- **Write to output queues** when all the output tokens produced during the action execution are written to the respective output queues. It is to mention that when an actor is implemented either in software or hardware processing elements, the action can only start writing on the output queues when enough space for accommodating all output tokens is available.

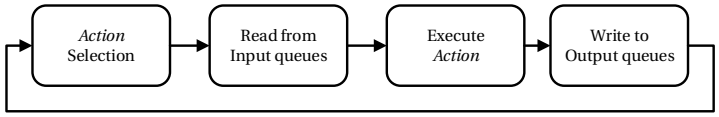


Figure 3.3 – Actor Execution Model.

3.3.3 CAL Syntax and Semantics

Lexical tokens

Lexical tokens help the user to understand the functionality provided by any language. They are a string of indivisible characters known as lexemes. The CAL lexical tokens, also summarized in Table 3.2, are described in the following:

- **Keywords** Keywords are a special type of identifiers. They are already reserved by default in the programming language. These keywords can never be used as identifiers in the code. Some of them are `action`, `actor`,`procedure`, `function`, `begin`, `if` `else`, `end`, `foreach`, `while`, `do`, `procedure`,`in`,`list`, `int`, `uint`, `float`, `bool`, `true` and `false`.
- **Operators** Operators usually represent mathematical, logical or algebraic operations. Operators are written as any string of characters `!`, `%`, `^`, `&`, `*`, `/`, `+`, `-`, `=`, `<`, `>`, `?`, `~` and `|`.
- **Delimiters** Delimiters are used to indicate the start or the end of this syntactical element in the CAL. Following elements are used as delimiters: `(`, `)`, `[`, `]`, `{` and `}`.
- **Comments** Comments in CAL language are the same as in Java, C and C++. Single line comments start with `//` and multiple line comments start with `/*` and end with `*/`.

Actions

The simplest actor that can be described using CAL is the `Passthrough` actor defined in Listing 3.1. This actor copies a token from its input port and places it into its output port.

Table 3.2 – CAL lexical tokens

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

The actor header is defined in the first line, which contains the actor name, followed by a list of parameters provided inside the `()` construct (empty, in this case), and the declaration of the input and output ports. The input ports are those in front of the `==>` construct and the output ports are those after it. In this case the input and output ports set are defined as $P^{in}_{\texttt{Passthrough}} = \{\texttt{I}\}$ and $P^{out}_{\texttt{Passthrough}} = \{\texttt{O}\}$ respectively. For each parameter and port, the data type is specified before the name (all defined with an `int` data type, in this case). This actor contains only one *action*, labeled as `pass` as defined in second line. In this case, the actions set is defined as $\lambda_{\texttt{Passthrough}} = \{\texttt{pass}\}$. Action `pass` demonstrates how to specify token consumption and production. The part in front of the `==>`, which defines the *input patterns*, it specifies how many tokens to consume from which ports and what to call those tokens in the rest of the action. In this case, there is one input pattern: `I:[v]`. This pattern indicates that one token is to be read (i.e. consumed) from the input port `I`, and that the token is to be called `v` in the rest of the action. Such an input pattern also defines a condition that must be satisfied for this action to fire: if the required token is not present, this action will not be executed. Therefore, input patterns do the following:

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

The *output patterns* of an action are those defined after the `==>` construct. They simply define the number and values of the output tokens that will be produced on each output port by each firing of the action. In this case, the output pattern `O:[v]` says that exactly one token will be generated at output port `O`, and its value is `v`.

Listing 3.1 – Passthrought.cal

```
actor Passthrought() int I ==> int O :
    pass: action I:[v] ==> O:[v]
    end
end
```

Action *Guard*

So far, the only firing condition for actions was that there be enough tokens to consume, as specified in their input patterns. However, in many cases it is possible to specify additional criteria that need to be satisfied for an action to fire. Conditions, for instance, that depend on the values of the tokens, or the state of the actor, or both. These conditions can be specified using *guards*, as for example in the `Split` actor, defined in Listing 3.2. This actor defines one input port `I`, two output ports `O1` and `O2`, and two actions `A` and `B`. Those actions require the availability of one token in `I`. However their selection is guarded by the value of the input token `val` read from `I`. In this example, if `val >= 0` then the action `A` is selected, otherwise action `B`.

Listing 3.2 – Split.cal

```
actor Split() int I ==> int O1, int O2 :

    A: action I:[val] ==> O1:[val]
    guard
        val >= 0
    end
    B: action I:[val] ==> O2:[val]
    guard
        val <  0
    end

end
```

In the `PingPongMerge` actor, reported in Listing 3.3, a finite state machine schedule is used to sequence the two actions `A` and `B`. The schedule statement introduces two states `s1` and `s2`. Contrarily, in the `BiasedMerge` actor, reported in Listing 3.4, the selection of which action to fire is not only determined by the availability of tokens, but also depends on the priority statement.

Actors composition

In CAL, it is possible to define a composition of actors or a network of interconnected actors as the one illustrated in Figure 3.2. It is composed by three actors `A`, `B` and `C`, and by five queues b_1 and b_2. Two different representations approach are supported: the first one is called Functional unit Network Language (FNL) (see Listing 3.5), the second one is based

Listing 3.3 – PingPongMerge.cal

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

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

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

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

end
```

Listing 3.4 – BiasedMerge.cal

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

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

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

    priority
        A > B
    end

end
```

on eXtensible Markup Language (XML) (Listing 3.6) known as XML Dataflow Format (XDF), which in most of the case is edited by visual editor.

Listing 3.5 – BasicNetwork.nl

```
network BasicNetwork () int I ==> int O :

entities
    A = ActorA(maxValue = 3);
    B = ActorB();
    B = ActorC();

structure
    I    --> A.I1;
    A.O  --> B.I;
    B.O1 --> O;
    B.O2 --> C.I;
    C.O  --> A.A2;
end
```

3.4 Standardization

A subset of the CAL programming language has been standardized by the ISO MPEG comity and is called RVC-CAL. The MPEG Reconfigurable Video Coding ISO/IEC 23001-4 has as a purpose to offer more flexible use and faster path to innovation of MPEG standards in a way that is competitive in the current dynamic and evolving environment. Thus, MPEG standards give an edge over its competitors by substantially reducing the time for which technology is developed and the time the standard is available for market applications. The RVC initiative is based on the concept of reusing commonalities among different MPEG standards and provide possible extensions by using appropriate higher level specification formalisms. Thus, the objective of the RVC standard is to describe current and future codecs in a way that makes such commonalities explicit, reducing the implementation burden by providing a specification that its starting point is closer to the final implementation. To achieve this objective, RVC provides the specification of new codecs by composing existing components and possibly new coding tools described in modular form.

The MPEG-B standard defines the language that is used to build the MPEG RVC framework. The RVC-CAL dataflow programming language is the core of the system; it is used to describe the behavior description of each module called Functional Unit (FU). With the specification of the FU network topology, the functional behavior of a video decoder is specified. An FU network topology is also called abstract decoder module. The term abstract refers to the fact that FUs are only characterized by the I/O behavior and the firing rules embedded in the RVC-CAL language. Thus, the interaction of each FU other FUs is fully specified by abstracting the time and by only defining dependencies of data production and consumption. The MPEG-C standard defines the Video Tool Library, a library of video coding tools. Figure 3.4 illustrates the concept that any abstract decoder model, which is constituted by an FU network description,

Listing 3.6 – BasicNetwork.xdf

```
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="Composition">
 <Port kind="Input" name="I">
  <Type name="int">
    <Entry kind="Expr" name="size">
     <Expr kind="Literal" literal-kind="Integer" value="32"/>
    </Entry>
  </Type>
 </Port>
 <Port kind="Output" name="O">
  <Type name="int">
    <Entry kind="Expr" name="size">
     <Expr kind="Literal" literal-kind="Integer" value="32"/>
    </Entry>
  </Type>
 </Port>
 <Instance id="A">
  <Class name="ActorA"/>
   <Parameter name="maxValue">
    <expr kind="literal" literal-kind="integer" value="42"/>
   </Parameter>
  </Instance>
  <Instance id="B">
    <Class name="ActorB"/>
  </Instance>
  <Instance id="C">
    <Class name="ActorCr"/>
  </Instance>

  <Connection src="" src-port="I" dst="A"   dst-port="I1"/>
  <Connection src="A"   src-port="O" dst="B" dst-port="I"/>
  <Connection src="B"   src-port="O1" dst="" dst-port="O"/>
  <Connection src="B"   src-port="O2" dst="C" dst-port="I"/>
  <Connection src="C"   src-port="O" dst="A" dst-port="I2"/>
</XDF>
```

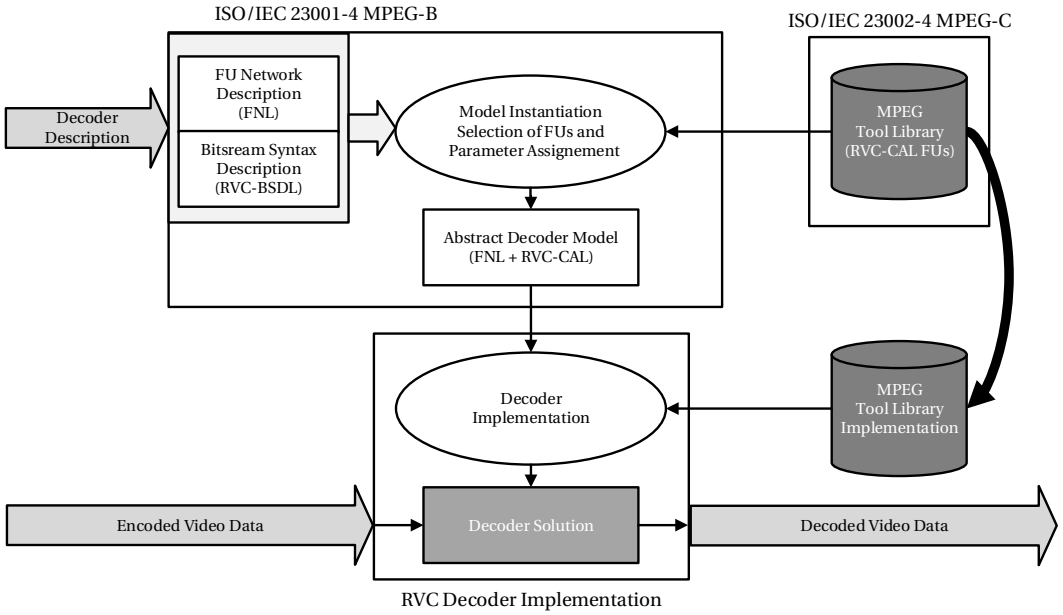


Figure 3.4 – Reconfigurable Video Coding.

can be implemented either in hardware or software.

3.5 RVC-CAL Compiler Infrastructure

A compiler supporting compilation of programs written using the standard RVC-CAL language is called Orcc [160]. Orcc stands for Open RVC-CAL compiler, and it is collaboration work between INSA of Rennes and EPFL. Recalling the RVC-CAL Design Flow in Figure 1.2. Orcc provides the necessary tools for designing, simulating and generating source code for different targets.

Figure 3.5 represents the Orcc building blocks of the Orcc compiler Infrastructure. The compilation flow primarily translated the RVC-CAL into source code or into intermediate representation (e.g. such as the LLVM), instead of generating machine code like traditional compilers (e.g. GCC) do.

Orcc uses extensively Model-Driven Engineering (MDE) by representing the IR with meta-models. Also, it uses the same MDE technologies that are employed in Eclipse IDE. Those MDEs are the Eclipse Modeling Framework (EMF), Xtext, and Xtend. The use of meta-models and MDE speeds up the development by automating time-consuming and most important error-prone tasks. Meta-modeling offers maintainability of the source code by having a global homogeneity of a unique model for different meta-tools (in Orcc's case Xtext and Xtend uses the same EMF meta-model for the Orcc's IR). Also, a meta-model is a source of documenting

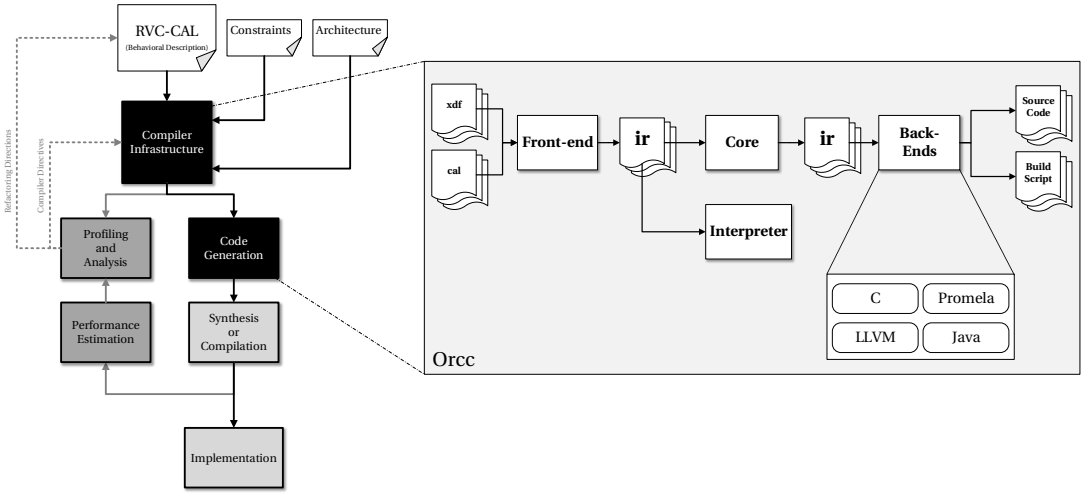


Figure 3.5 – Open RVC-CAL Compiler Infrastructure.

the code equivalent to UML.

The Orcc's compiler infrastructure is the following:

- **Front-end**: RVC-CAL is parsed and translated into an Abstract Syntax Tree. The parsing is implemented using Xtext [161], a framework dedicated to the development of DSL that automatically generates a parser, a linker and an editor from the behavioral description grammar. The AST then is transformed into an Intermediate Representation. During this step the front-end, by extending classes in Xtext framework, performs semantic validation, type inference, and expression evaluation.

- **Core**: Defines the IR and the visitors for optimizing it. The IR is modeled using the Eclipse Modeling Framework [162, 163]. This framework offers many methods for manipulating the data structure, one of them is the containment relationship between objects. Furthermore, it provides the automatic serialization of the meta-model (IR), allowing incremental compilation.

- **Simulation**: Orcc offers a simulation of an RVC-CAL program by interpreting its IR. The simulation is type accurate, and it permits verifying the correct functionality of the RVC-CAL program before implementation.

- **Back-end**: Is the final block in the compiler flow. It applies target particular optimization (IR to IR transformations) before the code is generated. Orcc's back-ends translate the RVC-CAL program into a general purpose programming language to benefit from the optimizations that those compilers offer. Whenever these optimizations are not enough, additional IR optimization passes are performed by Orcc's back-ends to meet the demands. To generate the code for a particular target, the Xtend [164] framework is

used. This framework provides a template based code generation that is flexible and easy to use. It is meta-language based on Java and is fully integrated into Eclipse IDE.

3.6 Orcc Intermediate Representation

The Orcc's IR has two parts, one representing the MoC of RVC-CAL and the other being a procedural IR that represents the computational parts of actions.

3.6.1 Dataflow IR

Most of the compilers convert between different representations, one of those is to have a linear code and then represent its execution using a Control Flow Graph and then back again to a linear code. In dataflow programming it is necessary to describe the application as a graph, that represents actors as nodes and connections as edges. Before describing the Dataflow IR meta-model, it is important to define the Graph metal-model in order to represent dataflow relations

In the graph metal-model the following classes can be defined:

- **Graph**: Is an object that contains a list of vertices and a list of edges. The hierarchy of the graph that contains the sub-graph is naturally inherited from the Vertex class.
- **Vertex**: Is a *Graph* object. It contains two references, one for incoming and one for outgoing edges which enable to deduct the successors and predecessors of the *Vertex*.
- **Edge**: Is a directed edge that defines the source and target reference of vertices.

The Dataflow IR meta-model contains the following aspects of the RVC-CAL MoC: `Network`, `Actor`, `Port`, `Connection` and two helper classes Entity and Instance.

- **`Network`** (N): As introduced earlier in this chapter, actors can be composed. The Dataflow IR represents the composition of actors with the `Network` class. A `Network` has a *name* and contains two sets of *inputs* and *output* ports. Finally, a network includes a set of connection and a set of vertices representing `Network` or `Instances`.
- **`Actor`** (A): Represents the basic component of the RVC-CAL MoC. It communicates with two sets of *input* (P^{in}) and *output* (P^{out}) ports. An actor contains a set of *procedures* (see Section 3.6.2) and includes a set of `Actions` that are ordered according to their priorities. Also, RVC-CAL `Procedures` and `Functions` are expressed as *procedures*. The actor contains state variables called *stateVars* ($\mathcal{V}$) and actor *parameters* as constant variables. Finally, an actor may contains an `FSM`, which is used to schedule the actions.

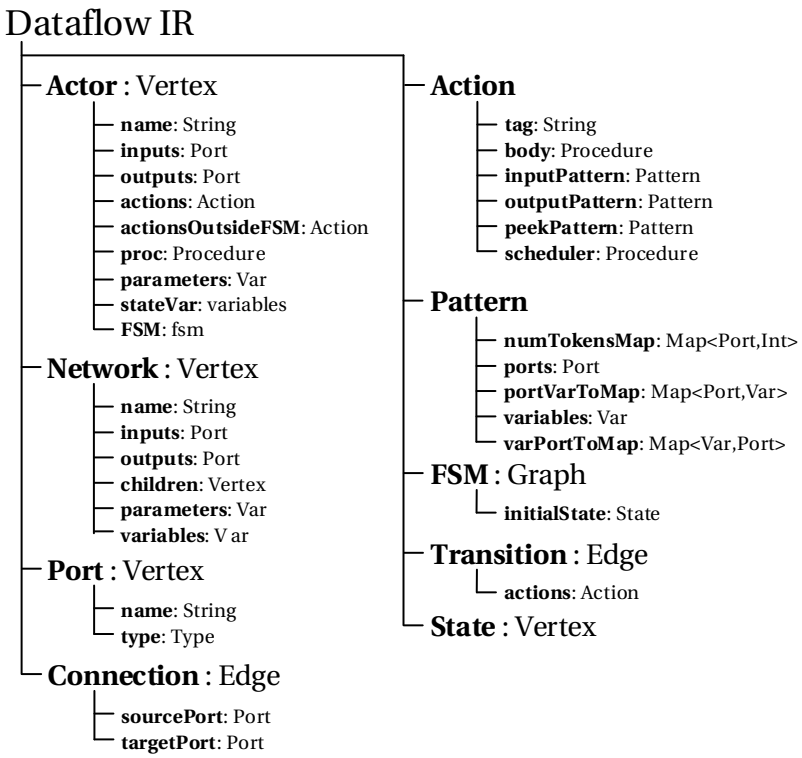


Figure 3.6 – Class tree for Blocks, Instruction and Expression classes of the Procedural IR.

- **`Port`**: Implements the external interface for `Network` and `Actor` classes. It has a *name* and a *type*.
- **`Connection`** (B): Is either an edge between `Ports` of a `Network` or an `Actor` that models the queue. A connection has a *source* and a *target* port, a *type*, and a *size*.
- **`Entity` and `Instance`**: Is the superclass of `Actor`, `Network` and `Instance`. `Instance` is can reference a single `Network` or `Actor` several times in one description without duplicating it.
- **`Action`** (Λ): Implements the firing function. An `Action` defines two procedures. The first one, contains the *body* of the action, i.e. the firing function. The second one is called *isSchedulable*. It expresses the guard condition, i.e. firing condition. An `Action` contains three patterns: *input pattern* that specifies the number of tokens to be consumed by the action's input port, the *output pattern* which determines the number of token productions from the action output patterns and the *peek pattern* that corresponds to the values of the tokens that need to be validated later on by the actor scheduler.
- **`Pattern`**: Describes a mapping between the input/output `Port` and the local variables

of the action procedural variables. It also defines the number of tokens that are going to be consumed/produced and the amount of input tokens that should be checked before firing an `Action`.

- **`FSM`** (*FSM*): Is a graph that implements the finite state machine of an `Actor`. It represents states by vertices and the transitions, i.e. actions, by edges.

3.6.2 Procedural IR

As mentioned in the previous section, an action is composed of two `procedures`. In addition, RVC-CAL expresses `Procedures` and `Functions` as `procedures`. The computation part in Orcc IR is called Procedural IR and describes the computational step of the imperative language paradigm which is very close to general programming languages. The Procedural IR is composed of the following classes:

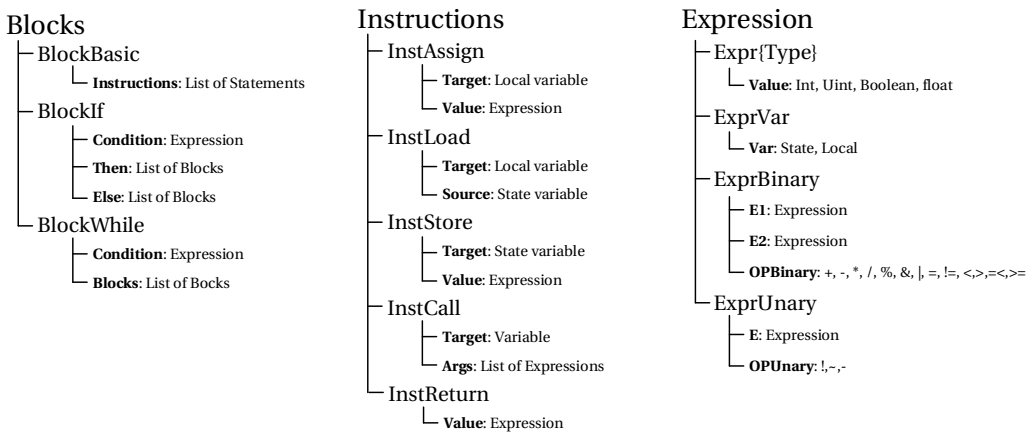


Figure 3.7 – Class tree for Blocks, Instruction and Expression classes of the Procedural IR.

- **`Procedure`**: Is the top class that contains a sequence of sequential statements. A procedure has a name, a set of sequenced `Blocks`, a set of local `Variables`, and a set of `Parameters`.
- **`Variable`**: Implements the concept of a variable. A variable has a `name`, a `type` and property of being `assignable`, a `define` and `use` chain.
- **`Type`**: Describes the type kind of a `Variable`, a `Parameter` or a `Port`. `Type` is a superclass and its subclasses are:
  - **`TypeInt`**: An Integer `Type`.
  - **`TypeInt`**: An Unsigned Integer `Type`.
  - **`TypeBool`**: A Boolean `Type`.

  - **TypeFloat**: A Floating-point `Type`.
  - **TypeString**: A String `Type` for a sequence of characters.
  - **TypeVoid**: A Void `Type` is used for procedures that does return a value.

- **Parameter/Argument**: Is used to parameterize procedures, actors and networks. The mechanism of parameterization is a standard construct that is used in many programming languages in order to increase the code re-utilization.

- **use/def**: The use and define chains model the utilization and definitions of variables. Once an `Instruction` inserts a value, a `def` is attributed to a variable. A `use` on the other side is created whenever an instruction is requiring a value from a variable. The Use/Def chain is very useful for compiler's dataflow analysis like the live variable analysis one.

- **Block**: Describes a portion of a sequence code and it is the super class for `BlockBasic`, `BlockIf` and `BlockWhile`. BlocBasic is a `Block` that contains a sequence of instructions. `BlockIf` is a branch `Block` that contains a true and a false branch blocks. The decision on which a branch should be taken is given by its condition, which is always a boolean expression. Finally, `BlockWhile` is a loop block that contains a set of ordered blocks, it also has a boolean expression condition that defines whether the loop should continue looping or exit.

- **Instruction**: Is a statement that is performed within a `BlockBasic`. `Instruction` is a superclass and its subclasses are:

  - **InstAssign**: Describes the assignment of an expression into a local `Variable`.
  - **InstLoad**: Describes a `Read` access from a state `Variable` to a local `Variable`.
  - **InstStore**: Describes a *Write* access to a `Variable` from an `Expression`.
  - **InstCall**: Describes the call of a `Procedure` given to a set of *Arguments*.
  - **InstReturn**: Describes the return of an `Expression` when a `Procedure` returns a value.
  - **InstPhi**: Is a special instruction used only when the Procedural IR is described in SSA form. It is used to resolve the conditional assignment of the exit as a `BlockIf` and `BlockWhile`.

- **Expression**: Is a combination of explicit values, constants, variables and operators. `Expression` is a superclass and its subclasses are:

  - **ExprVar**: Models the evaluation of a variable.
  - **ExprInt,ExprUint,ExprFloat,ExprFloat,ExprList**: an `Expression` that contain a constant value of its `Type`.
  - **ExprList**: an `Expression` that contains a List Value of a given `Type`

  - **ExprCall**: an Expression that evaluates the call of Procedure.
  - **ExprUnary**: a unary Expression that evaluates a unary Operator given a single expression.
  - **ExprBinary**: a binary Expression that evaluates a binary Operator given two expressions.

- **Operator**: An enumeration that defines the mathematical operation used in an ExprUnary and in an ExprBinary.
  - **OpUnary**: A unary operation such as: !, not, ~,-.
  - **OpBinary**: A binary operation such as: +, -, /, %, and, or, <, >, ≤, ≥ and others.

The Procedural IR is very close to other general purpose IR such as the Low Level Virtual Machine. The IR is feature rich and it supports all the standardized RVC-CAL computational constructs.

3.6.3 Visitors for Dataflow and Procedural IR and IR Interpreter

The *Gang Of Four* in Design Pattern [165] defines a visitor as: *"Represent an operation to be performed on elements of an object structure. A visitor lets you define a new operation without changing the classes of the elements on which it operates"*. In Orcc visitors, patterns are used to traverse the Dataflow and Procedural IR for code analysis, IR to IR transformations, IR optimizations passes, and code generation.

The Orcc Interpreter is implemented as a Visitor that executes for each Object in the Dataflow and Procedural IR a set of functions that represent the interpretation and execution of the IR Object.

3.7 Conclusion

In this chapter, the notion of dataflow parallel programming has been illustrated. A formal behavioral description called CAL has then been introduced and presented through a collection of CAL source code examples. Concepts like actors, actor composition, actions, guards, priorities, finite state machine have been illustrated. As presented, the CAL and its Dataflow MoC has the property to express an application as network processes. Beside that fact that is inherently concurrent and modular, CAL offers parallelism scalability, no shared memory between the process, communication with queues, state encapsulation, sequential execution inside each process provided by finite state machines and bitwise types. All previous properties lead to the portability of CAL which makes it a potential candidate for unifying the system level design for heterogeneous platforms.

A standardized subset of CAL called RVC-CAL and RVC standard and its compiler were also described. Moreover, an in-depth illustration of a compiler infrastructure and its Dataflow

and Procedural IR called Open RVC-CAL Compiler. Despite the fact that Dataflow IR makes it useful to describe most of the dataflow MoCs, limitations due to early decisions made by its original developers are perceived. First of all, the mandatory ordering of the actions makes indeterministic actors deterministic. Secondly, the absences of an action selection class in the IR makes the construction of an action sequencer for each platform implementation tedious. To construct a new scheduler, a back-end developer should visit all the patterns of actions and the FSM class of the actor and should extract the guard part of the actions. Thirdly, the guard of an action is an expression whose transformation into a procedure makes the analysis of the firing condition more difficult. In the Conclusion chapter, a better IR structure will be discussed. Finally, this chapter described all needed information on the Orcc's Dataflow and Procedural IR that is used in the following chapter. The following chapter is about the RVC-CAL high-level synthesis tool called Xronos. This tool extends Orcc IRs and implements missing compilation techniques that are necessary for high-level synthesis.

4 High-Level Synthesis of Dataflow Programs: Xronos

Njeriu në jetë ka nevoj për një filxhan shkencë, një shishe kujdes, dhe një oqean durim,
"Man in life needs a cup of science, a bottle of care, and an ocean of patience"
— Ismail Kadare

4.1 Introduction

The first implementation of a direct path to HW generation, called CAL2HDL, from CAL dataflow program has been reported by Janneck and al. in [146]. CAL2HDL was a part of the OpenDF [147] framework for developing CAL programs. At first it transforms a CAL actor to an intermediate representation called XML Language Independent Model (XLIM). Secondly, OpenForge is used to generate a Verilog description of the XLIM. OpenForge was initially developed by Xilinx under the name Forge until it became available as open source in 2008 and renamed as OpenForge. CAL2HDL supports only a subset of CAL (for instance synthesis of unsigned integer types, procedures with arguments and action input/output patterns with multi-token lists are not supported). Thus, the unification of software and hardware development is limited. Rewriting an elegant and compact piece of code into a version that could be synthesized, for very complex applications, is especially resource consuming and an error prone task (i.e. a parser of a video decoder).

As described in Section 3.4, a subset of CAL was standardized by ISO MPEG in ISO/IEC 23001-4 with the purpose to become the reference software code for video coding specifications. There was a need to support the full specification of the RVC-CAL standard to facilitate the creation of video codec circuits. An early attempt was made in [166]. Yet, it lacked loop support and was therefore never finalized. In [2] an XLIM backend for Orcc was developed for unifying the software and hardware code generation. However, support of multi-tokens on input and output port was missing. In addition, the code generation process when dealing with complex design such as AVC/H.264 video decoder was very slow. Then Jerbi and al proposed an interesting solution for the multi-token support in [167]. This approach was based on transforming the Orcc's IR of actors by modifying the actor's finite state machine and by

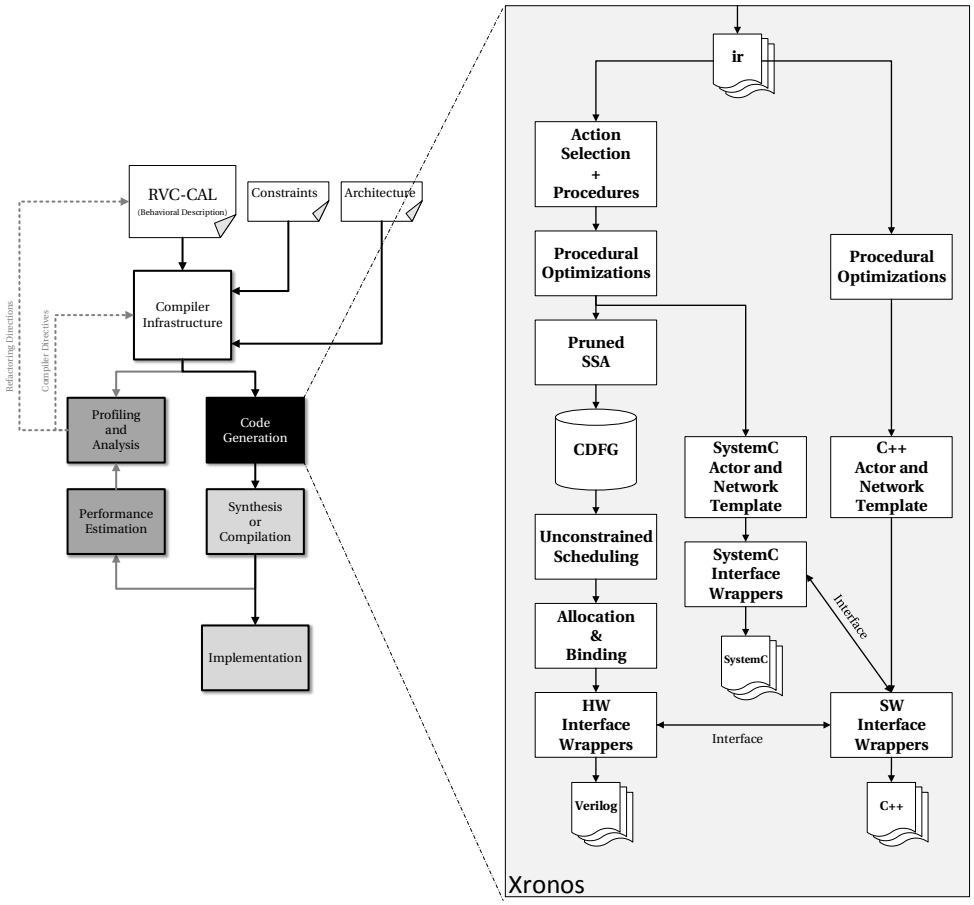


Figure 4.1 – Xronos in the Design Flow.

introducing to the IR the equivalent of mono-token actions. Although this represented a solution to the problem, it resulted into an excessive sequentialization of equivalent actor states. As a result, overall latency was increased, the overall performance was reduced, and also the resources were increased.

Xronos has been coded from scratch but it can be seen as an evolution of the CAL2HDL and the work done in [2]. Xronos was created with the purpose to resolve all the above stated problems and to accelerate the code generation. It uses Orcc as the RVC-CAL frontend and OpenForge as a backend for generating synthesizable Verilog code for each actor. To be precise, Xronos is the middle-end between these tools. Firstly, it transforms the IR of ORCC with a set of transformations/optimizations. Secondly, it turns the IR to a Control-and-Data Flow Graph so that OpenForge can generate a Verilog representation of the actor. After that, Xronos produces C++ source code for a general purpose and embedded processing platforms. Finally, it generates code for the hardware and software interfaces for heterogeneous platforms.

Table 4.1 summarizes the features and contributions of this thesis. All features in bold have

Table 4.1 – Xronos features versus the state of the art. The contributions of this thesis is related to the high-level synthesis of dataflow programs which are highlighted in **bold**.

Features	**Xronos**	CAL2HDL	Orcc HLS + Vivado HLS
CAL	✓	✓	✓
Bit accurate	✓	✓	✗
Unsigned Integer Support	✓	✗	✓
Procedures with Arguments	✓	✗	✓
Generators statement	✓	✗	✓
Foreach statement	✓	✗	✓
Repeat Construct	✓	✗	✓
Parallel I/O Read/Write	✓	✗	✗
Pipelining	✓	✗	✗
Static Resource Analysis	✓	✗	✗
Dynamic Profiling	✓	✗	✗
Testbench Generation	✓	✗	✓
TURNUS Integration	✓	✗	✗
Synthesizable SystemC	✓	✗	✗
HW & SW CoDesign	✓	✗	✗
Coarse Grain Clock Gating	✓	✗	✗
Open source	✓	✓	Only HLS backend

been incorporated by the author. A part from the TURNUS Integration(Chapter 5) and the Coarse Grain Clock Gating(Chapter 6), all other features are outlined in this Chapter.

Furthermore, this chapter describes in depth all advances that are included in Xronos in comparison to the Orcc's compiler infrastructure. After that, it explains in detail the Forge intermediate representation called LIM, it's scheduling and the hardware code generation. It then clarifies how the Control Data-Flow Graph of an Orcc `Procedure`, Actors, and Actor composition are mapped into LIM. Moreover, it presents the development of a synthesizable SystemC and a C++ for embedded software processing elements. Afterwards mapping, interface synthesis between hardware and software processing components, and profiling of heterogeneous platforms are explained. Finally, experimental results proves the capabilities of Xronos in hardware synthesis and in co-Design for heterogeneous platforms.

4.2 Advances on the Orcc compiler infrastructure for Hardware Synthesis

Figure 4.2 depicts the Xronos compiler infrastructure. Xronos hardware code generation is based on OpenForge and its intermediate representation is a CDFG based one. To facilitate the translation of the Orcc IR to a CDFG for hardware synthesis, additional IR classes are inserted in the Procedural IR and Dataflow IR. Furthermore, a number of classic compiler dataflow

analysis algorithms have been implemented within Xronos for optimizing the code, such as Control Flow Graph, Dominance Graph, Reaching Definitions and Live Variable Analysis. To minimize the local variables, the Procedure IR form is transformed into a pruned SSA one. Thus, fewer wires and registers in the final RTL generation are needed.

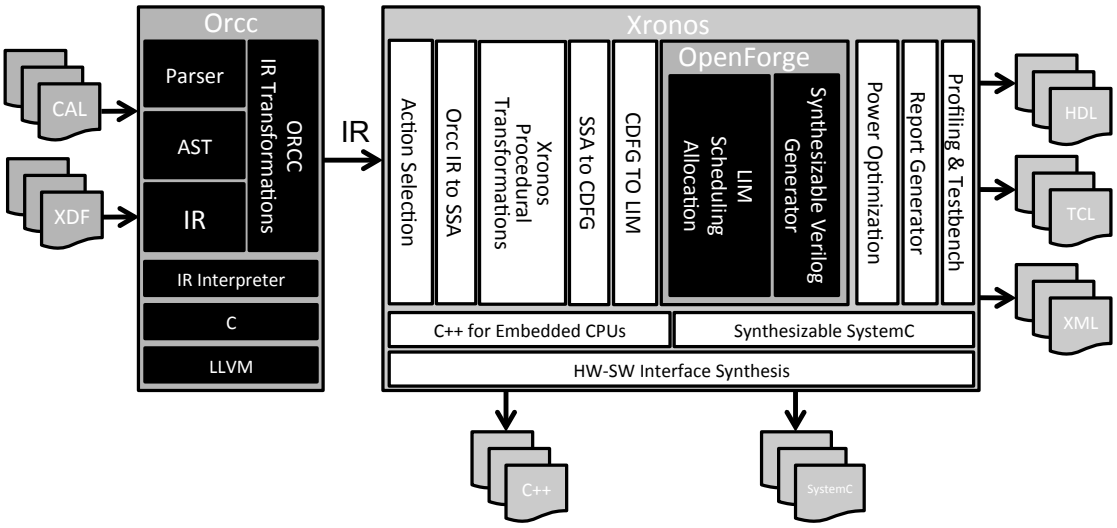


Figure 4.2 – Detailed Xronos Compiler Infrastructure, white boxes indicates personal contributions.

After SSA transformation, a set of IR optimization passes is applied on the Procedure. This furthers minimizes the BlockBasic instructions and correctly handles the casting of local variables bit size. In addition, operations such as division and modulo are transformed into synthesizable operators and other BlockBasic IR passes are carried out.

On the Procedural IR meta-model the following classes are added:

- **Instructions**: A set of Instructions related to the port of the actors and a Casting Instruction.
  - **InstCast**: Is a casting Instruction, which cast a local variable to a given Type with a different bit size.
  - **InstPortRead**: Defines a single token reading from an input Port.
  - **InstPortWrite**: Defines a single token writing to an output *Port*.
  - **InstPortPeek**: Defines a peek (reading the value but not consuming it) from an input *Port*.
  - **InstPortStatus**: Defines whether there is a single token on an input or output *Port*
- **BlockMutex**: Defines a kind of Block that contains Blocks that are mutually exclusive (no dependencies) and can execute in parallel.

- **CFG**: Defines the Control Flow Graph that has as vertexes the *Blocks* of a *Procedure* and as edges the order of the *Blocks*.

- **CDFG**: Control-Data Flow Graph, vertex are operations or CDFG nodes, two kinds of edges: control and data. Compared to CFG, it contains all data and control dependencies even inside Blocks.

4.2.1 Control Flow Graph Construction

A Control Flow Graph (CFG) models the flow of the control between Basic Blocks in a program. A CFG is represented as a directed graph $G = (N, E)$, with each node $n \in N$ corresponding to a Basic Block (BlockBasic in Procedural IR) and each edge $e = (n_i, n_j) \in E$ corresponding to a possible path of control from block n_i to block n_j.

For each *Procedure* in the Procedural IR, the CFG member is added. The CFG is *Graph* and has two essential features: It identifies the beginning and the end of each basic block and it connects the resulting blocks with "control" edges that describe the directed control transfer of blocks. A CFG may contain multiple starts and exits but in RVC-CAL only one single beginning and one single end of a program is possible which makes the construction of the CFG easier.

The Algorithm 1 starts by creating an empty CFG graph. After that, it adds an empty BlockBasic as entry node of the CFG. It then visits the Blocks of the Procedure. If a Block is a *BlockBasic*, it creates a new CFG node and makes it the last (the visit function set this node as the last). If a Block is a *BlockIf*, it will first visit the true branch in order to create all CFG nodes for it and will then create all nodes for the false branch. For resolving the exit of the *BlockIf*, an empty join node is added as well as two incoming edges that are connected to it, one from the last node of the true branch and one from the last node of the false branch. In the case of a *BlockWhile*, it adds an edge from the last node to the node of the *BlockWhile* and stocks this node in the memory. After that, it creates all the nodes from the Blocks of the *BlockWhile* and attributes to them a true flag. Once being finished with those Blocks, it returns to the node of the *BlockWhile* as it is the last one. It should be mentioned that the previous operation inherits the false flag. Once all blocks have been processed, an empty *BlockBasic* is added to the graph which is then connected to an edge from the last node.

4.2.2 Dominance Graph

A dominance graph is an iterative data-flow analysis that is used by many optimization techniques. In a CFG with entry node b_0, node b_i dominates node b_j, written $b_i \; dom \; b_j$, if and only if b_i lies on every path from b_0 to b_j. By definition b_i dominates itself, $b_i \; dom \; b_i$. Xronos calculates the dominance as follows:

$$Dom(n) = \{n\} \cup \left(\bigcap_{m \in preds(n)} Dom(m) \right) \tag{4.1}$$

Algorithm 1: Control Flow Graph Construction, visitor pseudo.

```
1  class CfgConstruction(Procedure procedure):
       Variables: Cfg cfg
       Variables: Vertex last
       Variables: Boolean flag
2      def caseProcedure(Procedure procedure):
3          cfg := CreateCfg();
4          procedure.setCfg(cfg);
5          Vertex entry := createVertex();
6          cfg.setEntry(entry);
7          last := entry;
8          last := visit(procedure.getBlocks());
9          Vertex exit = createVertex();
10         cfg.setExit(exit);
11         addEdge(exit);
12     def caseBlockBasic(BlockBasic block):
13         Vertex node := addNode(block);
14         if last != null then
15             addEdge(node);
16         return node;
17     def caseBlockWhile(BlockWhile block):
18         Vertex node := addNode(block);
19         if last != null then
20             addEdge(node);
21         last := node;
22         flag := true;
23         last := visit(block.getBlocks());
24         flag := true;
25         addEdge(node);
26         last := node;
27         return node;
28     def caseBlockIf(BlockIf block):
29         Vertex node = addNode(block);
30         if last != null then
31             addEdge(node);
32         Vertex join = addNode(block.getJoinBlockl());
33         join.setLabel("join");
34         last := node;
35         flag := true;
36         last := visit(block.getThenBlocks());
37         flag := false;
38         addEdge(join);
39         last := node;
40         last := visit(block.getElseBlocks());
41         addEdge(join);
42         last := join;
43         return join;
44     def visit(List<Block> blocks):
45         for block in blocks do
46             last := visit(block);
47         return last;
48     def addEdge(CfgNode node):
49         Edge edge := cfg.add(last, node);
50         if flag then
51             edge.label := true;
52             flag = false;
53     def addNode(Block block):
54         Vertex node = createVertex(block);
55         cfg.add(node);
56         return node;
```

with the initial condition $Dom(n_0) = \{n_0\}$, and $\forall n \neq n_0$, $Dom(n) = N$, where N is the set of all nodes in the CFG. $Dom(n)$ is computed as a function of n's predecessors($preds(n_i)$). As a result, Dominance is a forward data-flow problem.

4.2.3 Reaching Definition

Given for each statement $t \leftarrow ...$ with a target t there is a definition with a label d_l, saying that d_l reached a statement d_n in the program if there is some path of control-flow edges from d_l to d_n without an intervening assignment that modifies that target of t. Reaching definitions can be solved as a forward data-flow problem:

$$ReachIn(n) = \bigcap_{p \in pred[p]} ReachOut[p] \tag{4.2}$$

$$ReachOut(n) = Gen(n) \cup (ReachIn(n) - Kill(n)) \tag{4.3}$$

$Gen[d : y \leftarrow f(x_1, \ldots, x_n)] = \{d\}$ and $Kill[d : y \leftarrow f(x_1, \ldots, x_n)] = Defs[y] - \{d\}$, where $Defs[y]$ is the set of all definitions that assign to the variable y.

$Gen(m)$ contains those variables that are used in m before any redefinition in$m$. $Kill(m)$ contains all the variables that are defined in m.

First to solve the equation Gen and $Kill$ sets should be filled. In Xronos, reaching definitions is implemented as a Procedural IR visitor. This visitor, visits the instructions of all blocks in the Procedure as described with the pseudo visitor in Algorithm 2. All sets are stored inside the attributes for each *BlockBasic*.

4.2.4 Live Variable Analysis

Live Variable Analysis or liveness is a compiler backward dataflow analysis that calculates for each "program point" (it can be a statement or a Block) the variables that may be potentially read before their next write. It should be noted that the compiler dataflow analysis is an entirely different concept from dataflow programming. A variable is live if it holds a value that may be needed in the future. A useful property of liveness is that it finds which variables are used and defined for each *BlockBasic*. Also, liveness enables to find the executed code that has no overall effect on the program and uninitialized variables.

A variable v is live at point p if and only if there exists a path in the CFG from p to a use of v, without being redefined. Live information is computed for each *Block b* in the procedure. $Gen(b)$ is defined as a set that contains all variables that are live at exit from b. Coming back to the first property of the LVA, each variable on $Gen(n_0)$ has a potential uninitialized variable.

Algorithm 2: Creating Gen and $Kill$ sets.

```
1  class Gen_Kill(Procedure procedure):
       Variables:Set gen
       Variables:Set kill
2      def caseBlockBasic(BlockBasic block):
3          gen := create a new empty set kill := create a new empty set
4          for instruction in block.getInstructions() do
5              visit(instruction);
6          block.setAttribute("Gen",gen);
7          block.setAttribute("Kill",kill);
8      def caseAssign(InstAssign assign):
9          visit(assign.getValueExpression());
10         Var target := assign.getTargetVariable();
11         kill.add(target);
12     def caseLoad(InstLoad load):
13         for expr : load.getIndexesExpressions() do
14             visit(expr);
15         Var target := load.getTargetVariable();
16         kill.add(target);
17     def caseStore(InstStore store):
18         visit(store.getValueExpression());
19         for expr : store.getIndexesExpressions() do
20             visit(expr);
21         Var target = store.getTargetVariable();
22         kill.add(target);
23     def caseStore(InstStore store):
24         for expr : store.getArgumentsExpression do
25             visit(expr);
26         if call.geTarget != null then
27             Var target = call.getTargetVariable();
28             kill.add(target);
29     def caseExprVar(ExprVar exprVar):
30         Var var := exprVar.getVariable(); gen.add(var);
```

The computation of the $LiveOut$ is the following: For each node n in the *Procedure* CFG, a set $Gen(n)$ contains all the variables that are live at exit from the block of the node n. $LiveOut(n)$ is defined by an equation that uses the $LiveOut$ sets of the n successors in the CFG, as well as two sets $Gen(n)$ and $Kill(n)$.

The dataflow equation is defined as follows:

$$LiveOut(n) = \bigcup_{m \in succ(n)} Gen(m) \cup (LiveOut(m) - Kill(m)) \tag{4.4}$$

$Gen(m)$ and $Kill(m)$ sets are calculating according to Algorithm 2. The $LiveOut(n)$ is just the union of those variables that are live at the head of some block m that immediately follows n.

The *GetReversePostOrder1* function on Algorithm 3 gives the reverse postorder of the CFG graph of the Procedure. A postorder traversal visits as many of a node's n children as possible before visiting n. However, a reverse postorder traversal is the opposite. It visits as many of node's n predecessors as possible before visiting n. Two helper sets *lineInsP* and *liveOutsP* are used for testing if the liveIns and liveOuts have been changed after a traversal of all vertex in the CFG. If *liveInsP* is equal to *liveIns* and if *liveOutsP* is equal to *liveOuts* then liveness analysis has terminated. Lines 19 to 23 calculate the most right part of *LiveOut* ($LiveOut(m) - Kill(m)$) and lines 25 to 28 calculate the *LiveOut* of a vertex v.

4.2.5 Single Static Assignment, Pruned Form

Single Static Assignment (SSA) is an intermediate representation in which each variable has only one definition in the program text [168]. The SSA form represents both the data flow and control flow in the IR. Thus, it serves as a basis for a large set of transformations. Any Operation $x \leftarrow ...$ is a definition of x and any operation $... \leftarrow x$ is a use of x. A procedure is in SSA form if the following constraints are applied: (1) every variable is defined only once and (2) every use of the variable corresponds to a single definition.

To make sure that only one unique definition exists a variable Global Value Numbering (GVN) is applied to the procedure. To transform Procedure IR to SSA form, Xronos inserts ϕ-functions (InstPhi on the Procedural IR) at points where different control flow paths merge, and renames variables as defined in constraints. Xronos constructs a *pruned*-SSA version which means that all assignments by the ϕ-functions to a variable that is not *live* are eliminated. The live information of the variable is provided by the live variable analysis. Thus, less register are being used for loops, branches because pruned SSA minimizes data communication between basic blocks. It is to say that the construction of a pruned-SSA is beyond the scope of this thesis, for more information the reader may refer to [168].

Algorithm 3: Live Variable Analysis, Procedure IR pseudo visitor.

```
1  class Liveness(Procedure procedure):
2      def caseProcedure(Procedure procedure):
3          Boolean changed := true;
4          Ordering rpo := GetReversePostOrder(cfg);
5          List vertices := rpo.getVertices();
6          Map liveIns := new Map(Vertex,Set(Variables));
7          Map liveOuts := new Map(Vertex,Set(Variables));
8          Map liveInsP := new Map(Vertex,Set(Variables));
9          Map liveOutsP := new Map(Vertex,Set(Variables));
10         while changed do
11             changed := false;
12             for vertex in vertices do
13                 liveInsP.put(vertex, liveIns.get(vertex));
14                 liveIOutsP.put(vertex, liveOuts.get(vertex));
15                 Set gen = vertex.getBlock().getMap("gen");
16                 Set kill := vertex.getBlock().getMap("kill");
17                 Set liveOut := vertex.getBlock().getMap("kill");
18                 /* Get Live In                                          */
19                 Set calcIn := Set.copy(gen);
20                 Set calcRemove := Set.copy(liveOut);
21                 calcRemove.remove(kill);
22                 calcIn.addAll(calcRemove);
23                 liveIns.put(vertex, calcIn);
24                 /* Get Live Out                                         */
25                 Set calcOut = new Set(Var);
26                 for s in vertex.getSuccessors() do
27                     Set in = liveIns.get(s);
28                     calcOut.addAll(in);
29                 liveOuts.put(vertex, calcOut);
30             if !liveIns.equals(liveInsP) and liveOuts.equals(!liveOutsP) then
31                 changed := true;
32         /* Store liveIns and liveOuts for each vertex                   */
33         for vertex in vertices do
34             Block block := vertex.getBlock();
35             block.setAttribute("LiveIn", liveIns.get(vertex));
36             block.setAttribute("LiveOut", liveOuts.get(vertex));
```

4.3 Procedural IR Transformations

To prepare the IR for hardware synthesis some additional transformation should be applied before converting the Procedural IR of each action to the Openforge LIM IR. Thanks to the SSA representation of a Procedure, the constant propagation algorithm, and the dead code elimination are easy to implement. For the hardware synthesis the variables and the operation should have the correct bit size for reducing the resource footprint. Operations such as division or modulo are often not synthesizable for either ASICs or FPGAs. Thus, Xronos is transforming these operations into synthesizable components.

4.3.1 Expression Evaluator/Simplification

Expression evaluator/simplification is a helper visitor that finds algebraic identities to simplify the expression. For instance, a constant folding transformation as described in Section 4.3, uses the Expression Evaluator for streamlining the operations of constants.

Table 4.2 shows some identities that can be handled by the expression simplification if the constant values of the expressions are unknown.

Table 4.2 – Algebraic identities for Expression Simplificator. With $\bigwedge$ and logic and operator, and $\bigvee$ or logic or operator.

$a+0=a$	$a-0=a$	$a-a=0$	$a*n^2=a<<n$
$a*x1=a$	$a*0=0$	$a\div 1=a$	$a/n^2=a>>n$
$a>>0=a$	$a>>0=a$	$a\bigwedge a=a$	$a\bigvee a=a$

The Expression Evaluator checks if the value of *ExprUnary* or *ExprBin* is known. If it is, then knowing the operation, the calculation is effectuated and the *ExprUnary* or *ExprBin* is replaced by constant Expressions such as *ExprInt*, *ExprUint*, etc..

4.3.2 Single Read and Write Register Optimization

Multiple readings and writings to the same memory are expensive operations. Each of them requires at least one clock cycle for either retrieving or writing a variable value. Furthermore, multiple read/write operations increase the latency of the overall execution of the procedure. To minimize this latency, Xronos traverses the CFG of a procedure and stores all definitions and uses of the scalar state variables to a set called `UsedVars`. At the entry of the `BlockBasic` CFG node, Xronos inserts for each v in `UsedVars` a Load instruction with a target $temp_v$. After this operation it propagates for all the uses of v the variable $temp_v$ and replaces load/store instructions with assignments. Finally, at the exit of the `BlockBasic` CFG node, it inserts store instructions for each v with a value of $temp_v$. Here it should be mentioned that this optimization is effectuated at the Procedural IR level and may increase the hardware's critical path length of an action.

```
actor Actor()
    int IN ==> int OUT:

    int a := 0; int b := 0; int c := 0;

    action IN:[token] ==> OUT:[c]
    do
        b := token;
        b := a + b;
        c := a + b;
    end
end
```

(a) Orignial Code

```
actor Actor()
    int IN ==> int OUT:

    int a := 0; int b := 0; int c := 0;

    action IN:[token] ==> OUT:[c]
    var
        int temp_a, int temp_b, int temp_c
    do
        temp_a := a;
        temp_b := b;

        temp_b := token;
        temp_b := temp_a + temp_b;
        temp_c := temp_a + temp_b;

        a := temp_a;
        b := temp_b;
        c := temp_c;
    end
end
```

(b) Optimized Register Use.

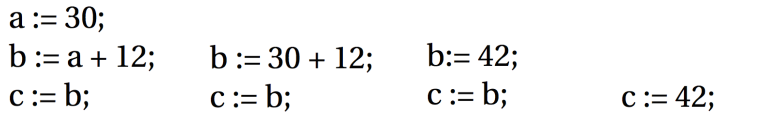
Figure 4.3 – Single Read and Write Register Optimization. Only a single read and a single write for a, b, and c state variables.

4.3.3 Uninitialized Variables

Listing 4.1 – An actor with unitialized local variable

```
actor Uninitialized()
        ==> int O:

    Act0:action ==> O:[token]
    var
        int a,
        int token,
    do
        token := 5 + a;
    end
end
```

Uninitialized variables are trivial to find once the CFG and liveliness are computed. It is sufficient to retrieve the entry *Basic Block* n_0 of the CFG and its liveness. For every variable v that is not defined in n_0 but found on the liveness set of n_0, v is an uninitialized variable. By default, Xronos initializes those variables to zero or false (depending the type of the variable) and emits a warning to the programmer. In the Listing 4.1, Xronos will emit a warning that a local variable "a" in action Act0 is uninitialized.



(a) Code (b) CP Pass 1 (c) CF (d) CP Pass 1

Figure 4.4 – Constant Propagation (CP) and Constant Folding (CF).

4.3.4 Constant Folding/Propagation

Constant propagation searches for a constant value expression c that is assigned to a variable $d_l : t \leftarrow c$, and another statement d_n that uses t. $d_n : x \leftarrow t\ BinOp\ y$. Thanks to the reaching definition analysis pass it is known that t is constant in d_n if d_l reaches d_n, and no other definitions of t reach d_n. Thus, after constant propagation, $d_n : x \leftarrow c\ BinOp\ y$. C

Constant folding begins once the constant propagation has terminated. This transformation will search for a binary or unary expression that contains only value expressions and depending on the operator will calculate the new expression value, and will assign to the target this new expression.

Figure 4.4 represents the constant propagation and folding in action. For each procedure, the constant propagation is relaunched up until no further modification is possible.

4.3.5 Dead Code Elimination

Dead code elimination acts on Instructions, Blocks, and Actions. If for an instruction with a target variable t and $t \leftarrow \ldots$ contained in *BlockBasic b* such that t is not $LiveOut(b)$ then this instruction can be removed. This also can be resolved by using the use/def chain within the SSA representation. If the variable t has no use, then it can be removed.

Constant propagation also acts on the `BlockIf` *condition*. Thus, if the *condition* has a boolean expression value of true, then all *Then Blocks are copied* to the container of the `BlockIf` and the *BlockIf* is removed. Consequently, if the condition is false then the *Else Blocks* are copied to the container Block and the `BlockIf` is removed and if a condition on a While Block is false, then the `BlockWhile` is removed.

Dead Code Elimination is also applied on the Dataflow IR. If an actor is parametrized, it is possible that some actions contain a guard condition that includes the actor parameter value. Hence, if a guard on an *Action* is false then this action is eliminated from the dataflow model, and if this action is contained in a transition of the *FSM* of the actor then this transition is also removed.

4.3.6 Type Casting

Type casting is a necessity for having a bit accurate execution of a program. Is applied to the parameters of a function and to assignments. Each parameter of a function should have the same type of the function's corresponding argument. If this is not the case, then a cast operation is necessary. The same applies when a variable with a different type or the same type with different bit size is assigned (stored or loaded) to a left-hand variable in an instruction such as $t \leftarrow \ldots$.

For casting an expression, the Least Upper Bound should be defined. LUB represents the minimal type to which both *Expressions a* and *b* can be assigned. Xronos defines its LUB rules as defined in Table4.3. Type casting is applied as a visitor to the Procedural IR. All Block Instruction and Expression classes are iteratively visited up until all LUB rules have been employed.

Table 4.3 – Lest Upper Bound on Types

		BitSize b			
		int	unit	float	string
BitSize a	int	int max(a,b)	$int \begin{cases} max(a,b) & \text{if } a > b \\ max(a,b+1) & \text{if } a < b \end{cases}$	float	X
	uint	$int \begin{cases} max(a+1,b) & \text{if } a > b \\ max(a,b) & \text{if } a < b \end{cases}$	uint max(a,b)	flaot	X
	float	float	float	float	X
	string	X	X	X	string

4.3.7 Division and Modulo Implementation

Division and Modulo operations are supported by HDL simulators but not by logic synthesizers. In Xronos, those operations are replaced by synthesizable ones. The Type of the numerator and denominator plays an important role on how the division is effectuated. If either the numerator or denominator type is signed, then the unsigned one should be casted as integer. If the bit size of the unsigned one is greater than the integer one, then both of them should be casted as integers and the bit size of the signed integer Type should be incremented by one.

As it can be seen on Algorithm 4 the division is bit accurate and its output is an unsigned integer with a bit $size = maxSize(num, den)$. The algorithm takes $size$ number of clock cycles to finish the operation. This algorithm is implemented in Xronos as a Procedural IR visitor. If an *ExprBin* has an operator of division or modulo, then *num* and *den* are extracted from the expression and two new variables are added to the procedure. The Algorithm 4 is constructed programmatically and added to the instructions that contain the Division/Modulo expression. Algorithm 5 represents the integer version of the algorithm. To avoid overcharging

Algorithm 4: Unsigned Integer Divisions and Modulo replacement

```
1  def DivisionModulo(isDivsion, num, den):
2      size = maxSize(num,den);
3      uint(size) result := 0;
4      uint(size) remainder := num;
5      uint(size) mask := 1 « (size - 1);
6      for i = 0 to size do
7          uint(size) numer := remainder » (size - i);
8          if numer >= den then
9              result := result or mask;
10             remainder := remainder - (den « (size - i));
11         mask := mask » 1;
12     if not isDivision then
13         result := remainder;
14     return result;
```

Algorithm 5: Integer Divisions and Modulo replacement

```
1  def DivisionModulo(isDivsion, num, den):
2      size = maxSize(num,den);
3      int(size) result := 0;
4      int(size) remainder := num;
5      int(size) mask := 1 « (size - 1);
6      int(size) denom;
7      int(size) numer;
8      int flipResult := 0;
9      if num < 0 then
10         num := - num;
11         flipResult = flipResult xor 1;
12     if den < 0 then
13         den := - den;
14         flipResult = flipResult xor 1;
15     denom := den and (1 « size) ;
16     for i = 0 to size do
17         uint(size) numer := remainder » (size - i);
18         if numer >= denom then
19             result := result or mask;
20             remainder := remainder - (den « (size - i));
21         mask := (mask » 1) and ((1 « size) - 1);
22     if flipResult != 0 then
23         result = -result;
24     if not isDivision then
25         result := remainder;
26     return result;
```

the algorithms with more source code lines, the statement that contains the *result* variable is deleted in both algorithms.

4.4 Pipelining

A CAL network of actors can be seen as a pipeline structure. The design frequency of the pipeline is determined by a critical action with the longest combinatorial critical path. In general a design goal is to increase the operating frequency so that the overall data throughput increases as well. If the critical action is not in the critical path of the whole CAL program, then the goal can be achieved by dividing the action into smaller actions within the same actor without pipelining. Otherwise, the critical action can be pipelined by extracting it from the original actor and by partitioning it into two or more additional smaller actors that implement the pipeline stages. To find out the critical action the CAL program is first synthesized to HDL, and then to RTL, where the information on the combinatorial critical path can be obtained.

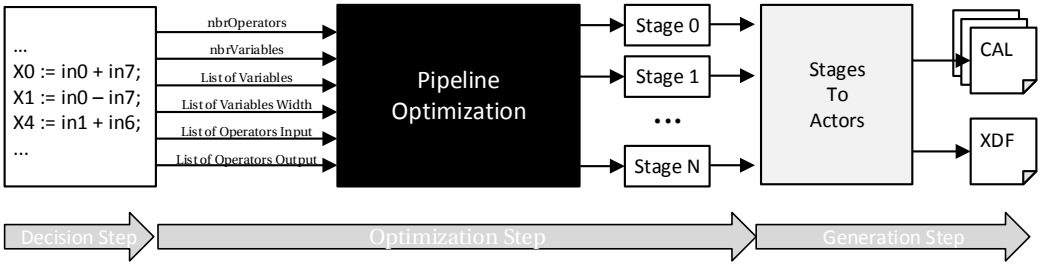


Figure 4.5 – Pipelining Optimization, from decision to generation.

The technique to extract an action from an actor that contains other actions into its own actor can be described as follows. First, the action has to be analyzed for its main input and output ports. Along with its original input port, the state variables read/used by the action have to be also received via input ports. If the action modifies a state variable, then the value has to be sent as feedback via an output port to the other actions that require this variable. Whenever the input port of the critical action is also used by other actions in the original actor, the consumption of the token from the port has to be accurately controlled by guard conditions so that the two actions do not consume the tokens at the same time. As for the output port, in the case when other actions in the original actor are also using the same port as in the critical action, the port has to be multiplexed correctly so that only a single output token from the port is taken at a time.

The pipeline technique implemented in Xronos is described in [78]. First, initial as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules are effectuated, and the corresponding mobility of operators is extracted from the CDFG graph (see Section 4.6). From this, an operator coloring technique is used on conflict and nonconflict directed graphs using

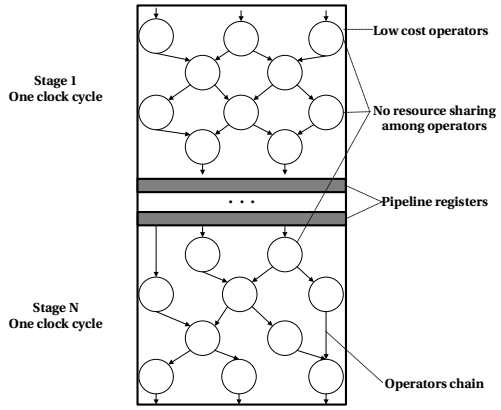


Figure 4.6 – One clock per stage pipeline scheduling with chaining and without sharing resources.

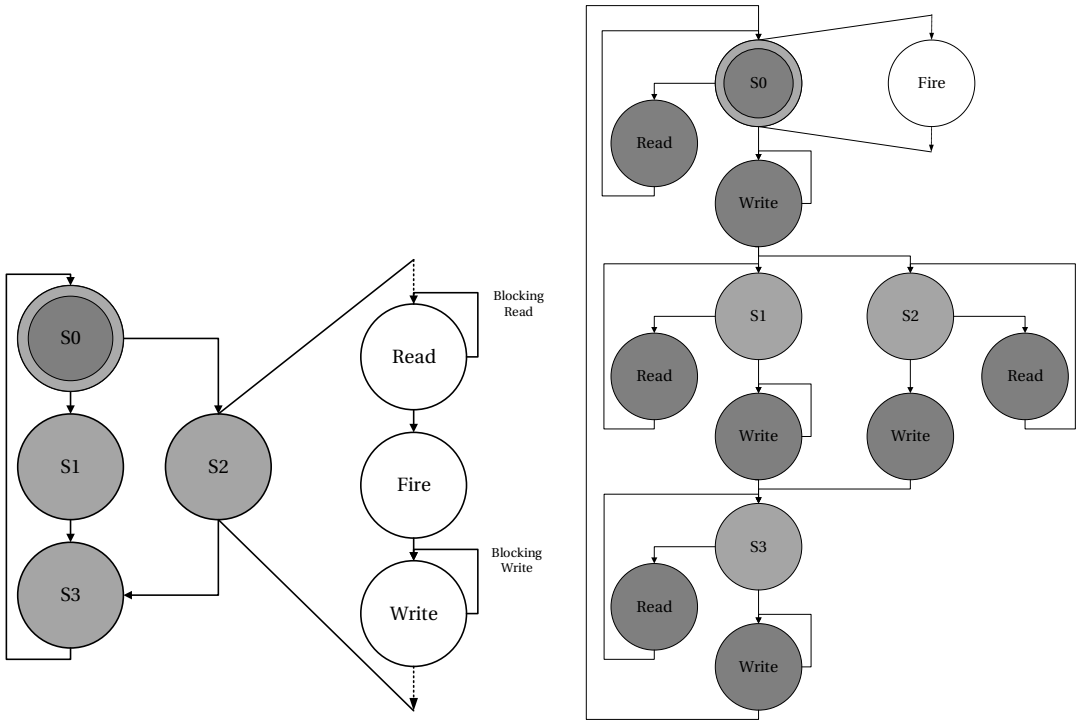
recursive functions and explicit stack mechanisms. For each feasible number of pipeline stages, a pipeline schedule with minimum total register width is taken as an optimal coloring, which is then automatically transformed into an RVC-CAL description. An RVC-CAL developer can add a *pipeline* attribute to a selected action for it to be pipelined.

4.5 Actor's *Action Selection* Procedure

An actor has a set of actions that each of them can read, write and change state variables as discussed in the previous chapter. If an actor has many actions, then either an FSM is defined or the control is effectuated by the firing conditions (guards and tokens availability) of the actions. When implementing an actor in either hardware or software, an action selection procedure should be defined. This procedure handles the execution of an actor as mentioned in Section 3.3.2 and it is implemented as a finite state machine. In comparison with actor machines [151], here the action selection calculates all conditions in parallel and let the *extracted* actor state machine to choose which action should be executed.

Figure 4.8 represents the action's selection finite state-machine of an actor with four states. For clarity, the image does not contain multiple transitions (actions) for each state. Figure 4.7a depicts a general software oriented action selection FSM that each state has a transition with three steps. First the transition has a blocking read on the input ports, then once the firing rules are satisfied it fires the action and finally has a blocking write on the transition output ports.

An action might read and write a list of tokens, but in hardware an FIFO queue implementation supports a single value in and a single value out which results in an implementation problem. However, a list of tokens might be implemented in four approaches: as a bus, as a concatenation of all list data as a single datum of a larger bit size, as a dual port memory that



(a) Software Action Selection

(b) Xronos Action Selection

Figure 4.7 – Finite State Machine of Action Selection.

acts as a queue or as a sequential reading and writing single values from the input and output ports. The first two approaches can not be implemented in hardware for two reasons. Firstly, in the general case when the actor's output port is connected to a successor actor, in which the data is consumed at a different rate than it is produced by the predecessor, then a part of data is lost. Secondly, the guard of an actor might do a peek in an index that is not in the head of the queue, which means that it does not satisfy the firing condition. In other words, all values should be available before the firing condition is resolved. The third approach, which is elegant and its implementation is given in [169], has the disadvantage of multiple latencies when both actors are writing and reading at the same time. Also, as mentioned in the same paper [169] for the same RVC-CAL application, Xronos, which implements the fourth option, has a two times higher throughput.

The fourth approach, reading and writing single tokens one by one, is easier to implement by adding two more states on the actors FSM: the *READ* and the *WRITE* state. However, it is necessary to have a local list for each input and output port with the maximum element size of the largest *repeat* found on the actor's actions. This might be considered as a disadvantaged but actually it is not, because by having small local memories the FIFO queue size can be decreased. In addition, another significant advantage is that actions with multiple read/writes on different ports can perform a parallel Read on those input ports in the READ state and a

parallel write on the output ports. This is possible because there are no dependencies between those operations.

Let consider the following example in Listing 4.2:

Listing 4.2 – Foo Actor

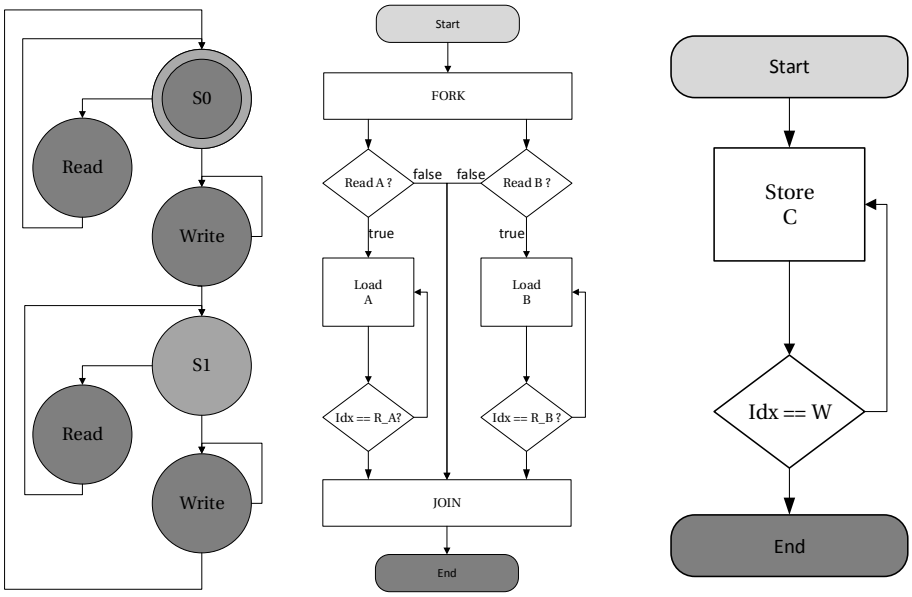
```
actor Foo()
    int A,
    int B
        ==> int C:

    Act0:action A:[a] repeat 64, B:[b] repeat 32 ==> C:[a[0] + b[0]]
    end

    Act1: action A:[a] repeat 32 ==> C:[a] repeat 32
    end

    schedule fsm s0:
        s0 (Act0) --> s1;
        s1 (Act1) --> s0;
    end
end
```

Actor Foo has two actions Act0 and Act1. Both of them uses the CAL keyword *repeat*. Act0 reads a list of 64 tokens from port A and 32 tokens from port B and outputs a single token. Act1 reads a list of 32 elements and writes the input list to its output C. Xronos modifies its FSM as depicted in Figure 4.8a by adding the READ and WRITE states. The READ state in Figure 4.8b starts by forking both reading paths in the input ports if the previous state was $s0$, only port A is read if $s1$. WRITE state in Figure 4.8c, depending on the previous state writes 1 token for $s0$ and 32 for $s1$.

4.5.1 Construction of the *Action Selection* Procedure

Contrarily to other Orcc code generators, Xronos does not pretty prints the code, but transforms the Dataflow and Procedural IR to a close to hardware intermediate representation and then it applies a scheduling on it. Moreover, Orcc backends do not create a *Procedure* for the *Action Selection*. They just prints it by visiting the actors FSM and the actions inputPatterns and outputPatterns. In Xronos, the choice was to first create an Orcc *Procedure* which then it is transforms into an object called *Task* in LIM IR, and effectuated by the Xronos Procedural IR to LIM mapping process, which is discussed in Section 4.8. Thus, it homogenizes and facilitates the overall transformation process of Procedures in the Procedural IR. Xronos applies the following steps/transformations for creating the *Action Selection* procedure:

1. **ActionSelection Procedure**: A new empty procedure with the name *ActionSelection* is created and added to the actor.
2. **AddFSM**: This transformation will add an FSM to the actor if it does not have one.



(a) Action Selection FSM.

(b) State READ.

(c) State WRITE, W=1 if act0, W=32 if act1.

Figure 4.8 – Foo actor *Action Selection*.

Also, if an actor has an FSM but contains actions that are outside of the FSM it will create a transition with the highest priority for each state and will add it to the state. In addition, some actors may contain special actions called initialize. This action is used for initializing variables as well as for any other initialization purpose. AddFSM will create a *INIT* state with a set of transitions, defined by the number of initialized actions in the code and will then add the INIT state and set it as the initial state. This transformation is applied in order to homogenize the generation of a general actor.

3. **WhileBlocks**: This step creates a list of Blocks that will accommodate the Blocks created for the following 7 steps.

4. **EnumerateStates**: This step creates a *BlockBasic* which assigns an index to each state of the actor's FSM;

5. **IOLists**: This transformation adds five state variables for each input and output port: *tokenIndex* represents the number of tokens that read/written, *MaxTokenIndex*. *RequestTokens* is a state variable that indicates the number of the input tokens at a given state that an action needs so that it can fire. *SentTokens* signifies the number of the output tokens that should be written on the output port. finally, *PortEnable* acts on the READ and WRITE states if a parallel reading/writing should be effectuated on those states. In addition, it adds a list of state variable with a name *portList* for each port with the maximum number of elements specified by the maximum *repeat* CAL construct

in each action port. Eventually, it modifies all the InstLoads that target the associate with an input port with the *portList*. It also does the same modification on InstStore instructions associated with an output port. It should be mentioned that the association is given by the inputPattern and outputPattern of the actions.

6. **LoadCurrentStateBlock**: This step creates a BlockBasic with a single InstLoad instruction that loads the actor's initial state index to the FSM's state variable called *fsmState*.

7. **IsSchedulableBlocks**: This step visits all the actions isSchedulable Procedure. As described in Section 3.6.1, the isSchedulable procedure returns a boolean value that signifies that the firing condition is satisfied. IsSchedulableBlocks will copy all the isSchedulable procedures body *Blocks* of the actors in a list of *Blocks* and will then replace the InstReturn instruction by an InstAssign one, that assigns the return value to *partialFiring* local variable.

8. **PortStatusBlock**: This step will create a BlockBasic and add for all ports of the actor the instruction InstPortStatus. Thus, it is possible to check if there is a token available in an input queue and if there is available space in the output queue.

9. **HasTokensBlock**: This step creates a BlockBasic with only InstAssigns instructions that will assign a boolean expression $tokenIndex == RequestTokens$ that signifies the partial firing condition on inputs. The *RequestToken* is changed when a state demands the number of tokens to be read by an input.

10. **StateBlocks**: For each state a *BlockIf* is constructed to accommodate the state transitions. If a state has a single transition, then: the BlockIf condition is an ExprVar that references the value of the variable created by the *IsSchedulableBlocks*, the BlockIf's *thenBlocks* contains the call of the action's procedure and a change of the state to following state on the FSM, and the BlockIf's else block contains an BlockIf for checking if there is a necessity to read from input port of the action and the changes to the READ state. If a state has many transitions, then a nested BlockIf is added on *elseBlocks* of the transition that has the highest priority. Figure 4.9 depicts the flow graph of a state with two transitions. The first transition handles the call for action *A* and the second one the call for action **B**. Both of them are reading from an input *PI* and writing to an output *PO*. If *Guard A* and *Guard B* are false, then it means that both actions require input tokens from PI. If both actions have different consumption rates, then $PI\_reqestTokens = max(repeat(A,PI), repeat(B,PI))$. The subtraction $PI\_tokenIndex\_read = PI\_requestTokens - PI\_tokenIndex$ helps to read only the necessary tokens for the next state.

11. **InifiteBlockWhile**: an infinite *BlockWhile* is constructed and added to the *Action Selection* procedure. The condition of the *BlockWhile* is an *ExprBool* with a boolean of true. All the blocks of the WhileBlocks are added on the *BlockWhile* blocks.

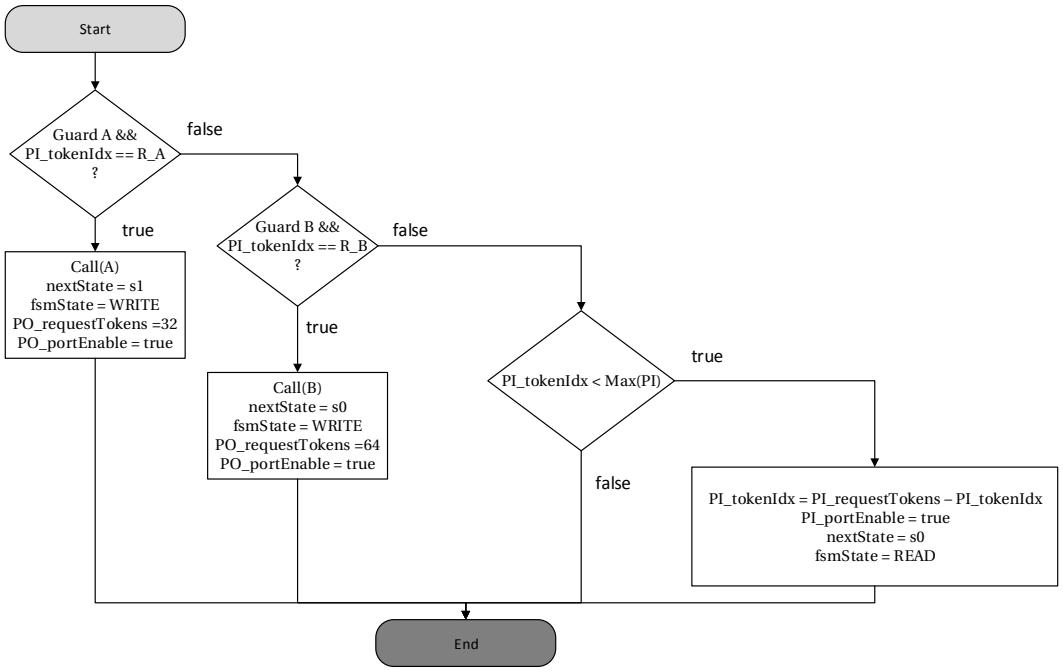


Figure 4.9 – A State that contains two transitions.

Once the Action Selection construction has finished, then two Procedural IR optimizations are applied on it. The first one is called BlockCombine and it combines all the different `BlockBasic` to a single one. The second one is called *RedundantLoadElimination* that eliminates all the redundant `InstLoads` from the state variables. Reading from a state variable has a latency of minimum one clock cycle from a register or a memory. Thus, all guards using the same state variable for the firing condition are combined to a single `InstLoad`, and all following `InstLoad` instructions are replaced by `InstAssign` instructions to a local variable of the procedure.

4.6 CDFG Representation of a Procedure

Before explaining each mapping process, the Control and Dataflow Graph (CDFG) should be defined. A CDFG is a graph G(V,E) with V nodes being themselves a CDFG graph (three node kinds: Block CDFG, Branch CDFG or Loop CDFG), special nodes: Entry, Init Entry, FeedBack(FB) Entry and Exit or an operation node. The CDFG has two types of edges $E = \{E_d, E_c\}$, with E_d a data dependency and E_c a control dependency. A data dependency signifies that a data value is passed from a node v_i to a node v_d. A control dependency however, signifies the flow of the control from node v_i to a node v_j; i.e v_i has finished its process and gives the hand to v_j. In literature CDFG nodes that contain subgraphs are also called Hierarchical Task Graphs (HTG) [170].

Compared to CFG, a CDFG contains more information on how `Blocks` operations are interconnected. The construction of the CDFG takes as input a CFG graph, and populates its nodes and edges by visiting the Instructions.

Let consider the following example in Listing 4.3:

Listing 4.3 – A Simple Procedure.

```
procedure toto(int b)
var
    int a
    int c
begin
    a := b;
    while a < 10 do
        if b > 5 then
            c := 2;
        else
            c := 1;
        end
        a := a - c;
    end
    ...
end
```

A partial CDFG of Listing 4.3 is given in Figure 4.10

4.7 Language Independent Model (LIM)

The OpenForge IR is called Language Independent Model (LIM). The LIM IR expresses operations in terms of graphical objects: nodes are components and edges are the control or data dependencies (as in a CDFG). Components are included into Modules and Modules into Tasks. Tasks can be executed sequentially or Parallel and included a Design, which is the Top Level of the program to be synthesized.

4.7.1 Component

Nodes in LIM are *Components*, represented in Figure 4.11. A component is the base class of all components in the model. It is a node in a data flow graph of operations that describes a module, an arithmetic or memory operation.

A component receive its input data from an **Entry** and output all processed data from its *Exit*. The **Entry** specifies the input data *Ports* and the **Exit** specifies the output data *Buses*.

- **Entry**: Specifies one possible set of inputs for a Component's Ports. For each Port, the inputs are specified as a collection of one or more Dependencies. When scheduling, the Dependencies of a given Entry must be scheduled so that they are all fulfilled

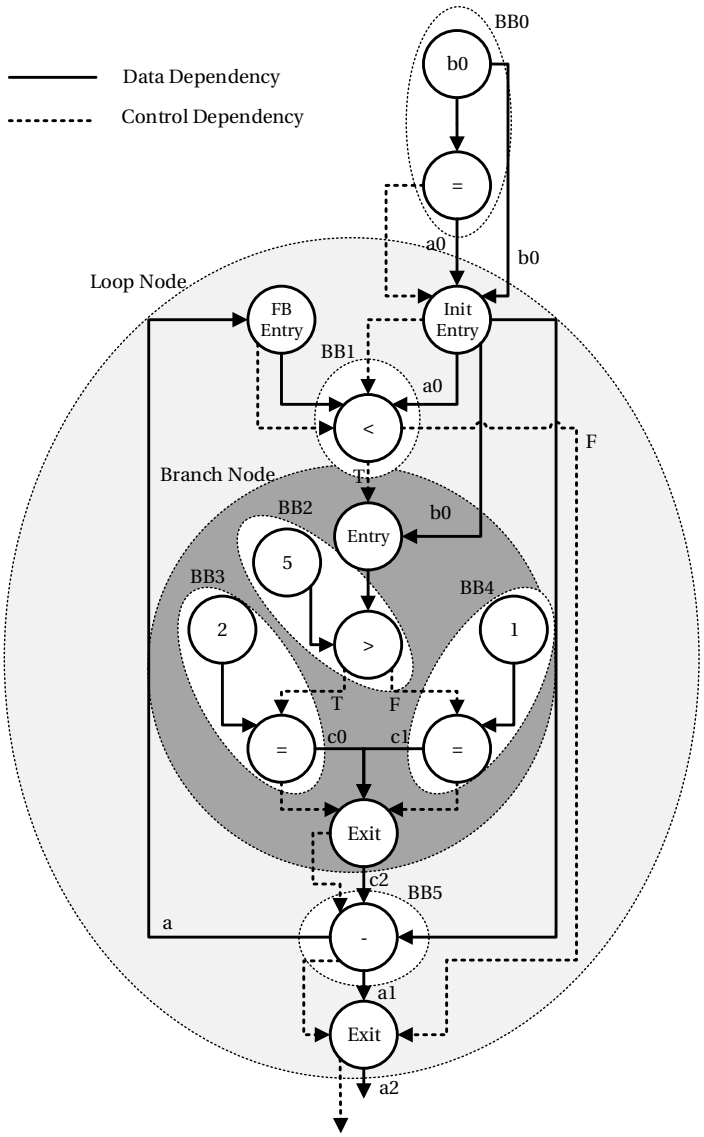


Figure 4.10 – Partial CDFG of the Listing 4.3.

simultaneously. However, they may be scheduled without regard to the Dependencies of other Entry objects. Multiple Entries for the same Component are logically muxed.

- **Exit**: An Exit is a group of Buses that represent an exit condition from an Operation. In addition, an Exit includes the control signal as well as all the data values that are output at that point. A given Component may have multiple Exits.
- **Port**: Represents the receiving side of a data connection between components. At most one source buses may be connected to a Port. A port can be owned only by a single component and graphically is an anchor.

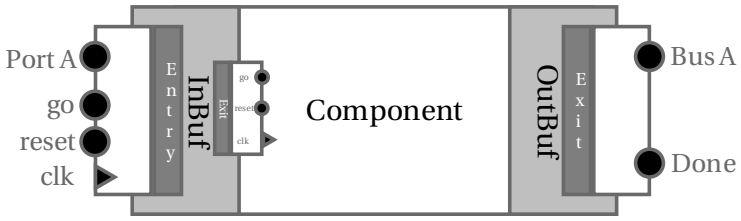


Figure 4.11 – LIM Component.

- **Bus**: Represents the source side of a data connection between components. One or more destination ports may be connected to the Bus; these represent the receivers of the Bus's data. Each Bus is owned by a single component. In addition to Port connections, a Bus may have multiple logical dependent Ports, represented by dependencies. It is from these that the final Port connections are ultimately derived. Each Bus has an underlying vector of Bits that comprise it. The number of bits in the vector is determined when the width of the Bus is established. Each of these Bits also designates the Bus as its owner; they can also be referenced to any value, but their number and identity within the Bus can never be changed once they are created. The Bus also has a value. An initial value is created along with the bit vector; this initial value just contains the bits of the vector. Other value may be set on the Bus containing Bits that are owned by other Buses.

- **InBuf**: An InBuf is used to bring a structural flow to the inside of a Module from its outside. A single InBuf is created automatically for each Module, and adding a data Port to the Module also adds a corresponding Bus to the InBuf. The InBuf itself has no Ports and no Entries. Logically, the Ports of the Module are the continued internally by the InBuf's Buses. The InBuf's single Exit is created with the usual done Bus and at least two data Buses. The done Bus is the continuation of the Module's go Port; that is why it is also known as the "go Bus". Clock and reset signals of the owner Module are represented by the first two data Buses; this is because this is the only case in which a clock or a reset will exit a Component.

- **OutBuf**: An OutBuf is used to bring a structural flow to the outside of a component's owner (Module) from its inside. An OutBuf is created automatically for each Exit of the Module, and adding a data Bus to the Exit also adds a corresponding Port to the OutBuf. The OutBuf itself has no Exit and no Buses. Logically, the Buses of the Exit are the continuation of the OutBuf's input Ports that are called *peer*.

- **Dependency**: A Dependency describes the relationship between a dependent Port and the Bus on which it depends. There are multiple types of dependency, and the type of dependency affects how it is handled during scheduling. Data Dependencies indicate that the port will receive a connection to the logical bus that is depended on, though it may be delayed or latched. Control Dependencies will cause the scheduler to find the

control signal that qualifies the dependent bus and will use that controlling signal to resolve the dependency.

- **Control Ports and Buses**: Each component has a *Clock*, *Reset*, and a *Go* Port. Also, each component has an Exit *Done* which includes the Done Bus. The Clock port indicates the clocking signal; the Reset Port indicated the reset signal and the Go Port indicates that the component is ready to be launched. The done bus indicated that this component has finished the process.

4.7.2 Primitives

A Primitive is a low-level component that performs a simple logic function. It has a single *Exit*, and the clock, reset, go, and done *Buses* are unused. The Primitive classes are the following:

- **And**: An And primitive accepts multiple 1-bit signals that make a **AND** operation on all of them together to generate a 1-bit result.
- **Mux**: accepts paired GO/Data signals, using each GO to select its related Data to be provided on the Result Bus.
- **Not**: A Not primitive accepts multiple 1-bit signals, that makes a **NOT** operation on all of them together to generate a 1-bit result.
- **Or**: An Or primitive accepts multiple 1-bit signals, that makes a **OR** operation on all of them together to generate a 1-bit result.
- **Reg**: Reg models a register and can be constructed to match any of the configurations which are possible in a Xilinx FPGA. Specific accessor methods are available to retrieve the data, enable, set, reset, clear, and preset ports. It's exit specifies that the Latency of this component is *One*. A Reg object always has ports for sync Set, sync Reset, Enable async Preset, async Clear.

4.7.3 Operation

An Operation is an executable Component. Operations are assembled into Modules. Operations are separated into three categories: ValeuOp, UnaryOp, and BinaryOp.

- **ValueOp**: An operation that generates a single value (a constant).
- **UnaryOp**: Base class of all unary operations, which require only one operand and generate one result. A type-casting operation that takes two parameters size and signed. Size signifies the bit size to be casted and sign if the cast should be signed. Other unary operations are ComplementOp, MinusOp, PlusOp, and ReductionOP.

- **BinaryOp**: Base class of all binary operations, which require two operands and generate one result. OpenForge's binary operations are: AddOp, AndOp, ConditionalAndOp, ConditionalOp, ConditionalOrOp, DivideOp, ModuloOp, MultiplyOp, OrOp, ShiftOp, SubstractOp, and XorOp.

4.7.4 Memory

LogicalMemory is a symbolic representation of a memory space in a Design. The two primary attributes of a LogicalMemory are its allocations and accesses.

- **Allocation**: Is a region of memory that is defined by the program via a variable declaration. Each allocation has associated with it a size (the number of bytes needed to represent the variable's type) and an initial value. Allocations may be added to and deleted from a *LogicalMemory*, as well as queried. Together the allocations represent the memory space that is legally usable by the Design.
- **Location**: Designates a region of memory. It is a symbolic representation, and so only records the size of the region as a number of addressable units. The actual address of the Location is given by the **LogicalMemory** to which the **Location** belongs.
- **LogicalValue**: An instance of LogicalValue is the initial value of a given location in a LogicalMemory. This initial value is specified when allocating a new region of the memory. A LogicalValue can report its size in addressable units (stride) as well as the content (numerical value) of those units.
- **Access**: Is a read or write of *LogicalMemory*. Accesses are denoted by *LValues* (left hand values), each of which is either a read or a write. The *LogicalMemory* can record all the *Locations* that are referenced by a given *LValue*, and determine whether a given Location refers to all, part, or none of an existing allocation.
- **MemoryAccess**: It factors out functionality that is common among all accesses to memory such as address port, done bus and methods that identify whether the node uses go or done. It is the base class of *MemoryRead* and *MemoryWrite*.
- **AbsoluteMemoryRead**: Is a fixed access to a LogicalMemory, in which the Location being accessed is fully specified at compile time and does not depend on a base or offset address. This module is populated with a MemoryRead and two constants. The first identifies the address being read and is a DeferredConstant based on the particular Allocation being accessed. The second is a fixed constant, indicating the number of bytes being accessed. It is used to read stored values from an array.
- **AbsoluteMemoryWrite**: Is a fixed access to a LogicalMemory, in which the Location being accessed is fully specified at compile time and does not depend on a base or offset address. This module is populated with a MemoryWrite and two constants. The first

identifies the address being written and is a DeferredConstant based on the particular Allocation being accessed. The second is a fixed constant, indicating the number of bytes being accessed. It is used to store values into an array.

4.7.5 Module

A *Module* is a component that may contain other Components. Is the abstract class that is extending by Block, Decision, Branch, Loop, LoopBody, Kicker, and Referee.

Block

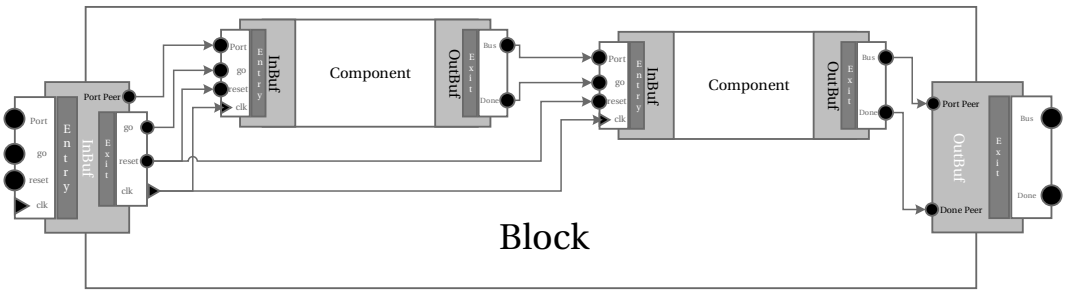


Figure 4.12 – A *Block* with two components that they are execute one after the other.

Block is a Module that contains a sequence of Components that are logically executed one after the other. It represents a portion of code that executes statements sequentially like basic blocks in a general compiler. All component's data and control dependencies are resolved during the construction of the block. Finally, a Block contains an *Entry* with a set of input ports and an Exit with a set of output ports. As with each *Component*, a *Block* has a go, reset and clock signal input ports and an output done signal output bus.

Latch

Latch is not a subclass of *Reg*. Rather, latch is the composition of a `Reg` with an enable and reset input ports, and a 2 to 1 Mux.

Decision

Decision is a refinement of `Module` for components that produce a truth value. It defines two Exits, one whose done bus signals *true* and one whose done bus signals *false*. On a true condition, the true bus is asserted, and the false bus is deasserted; on a false condition, the true bus is deasserted, and the false bus is asserted. The Decision is constructed with a `Component` representing the boolean value to be tested. A Decision produces a single data

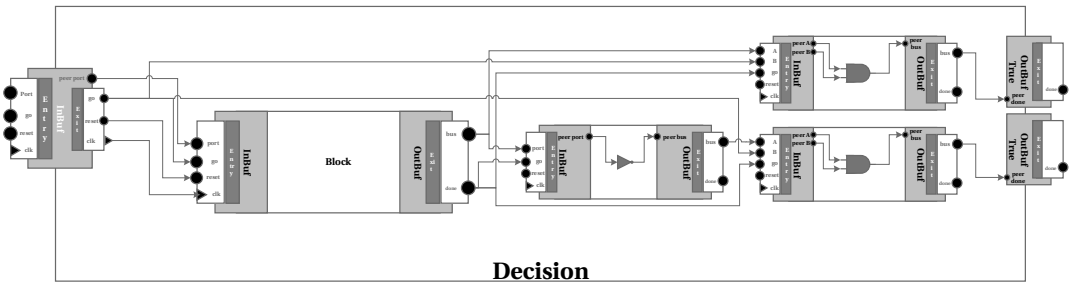


Figure 4.13 – A *Decision* Module .

value.

Branch

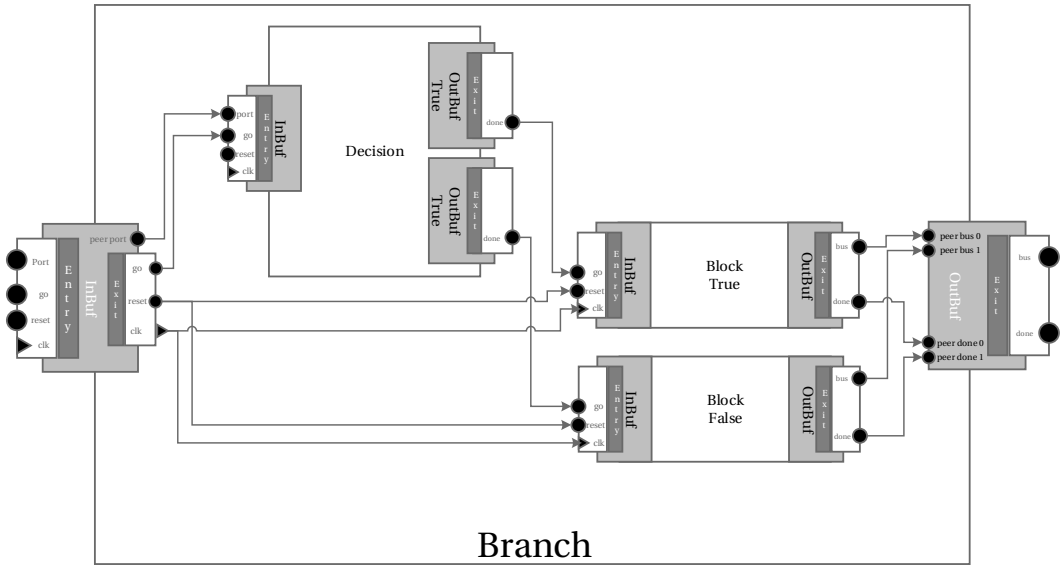


Figure 4.14 – A *Branch* with a true and a false part that both of them does not have an input port. The decision input is connected to a data dependency from the peer bus of the Branch input port to its input port. The Branch has an exit with two control dependencies, 0 from the Exit's Done of the true Block and 1 from the false Block.

A branch represents a choice of execution between two Components, one representing a path for a true condition, the other a path for a false condition. The choice is based on the output value of a *Decision*.

Loop and LoopBody

A Loop is a generic representation of a loop control structure. Its purpose is to execute the contents of a LoopBody iteratively. Internally, the Loop also creates data Registers and Latches to manage feedback data.

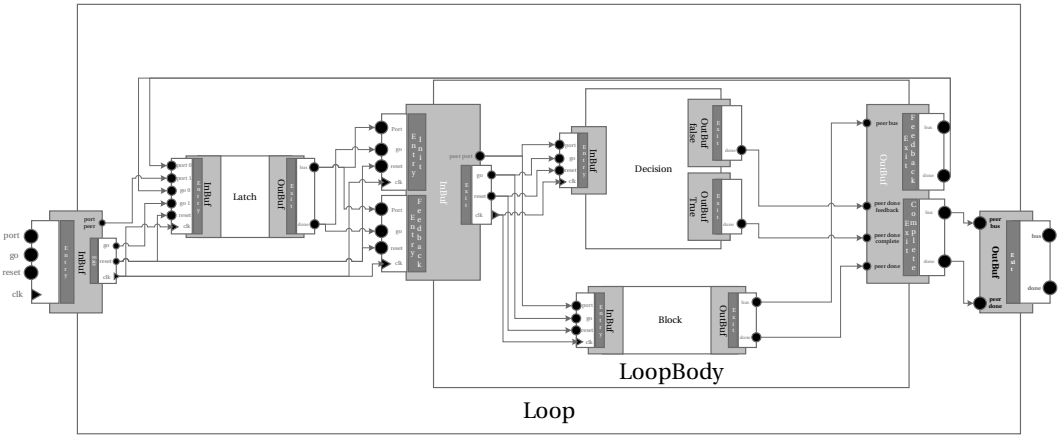


Figure 4.15 – A *Loop* Module, that contain a WhileBody.

A LoopBody characterizes the body of a Loop. A LoopBody designates one of its Exits as a "feedback exit." When this exit is asserted, it indicates that another iteration of the loop should begin. The Buses of this exit are connected to the feedback inputs of data and control *Registers*. If the body will never iterate (for example if it ends with a break), then this exit may be null. The Decision may also be null if it is known that it will never be reached (for instance, a do-loop that ends with a break). In addition, a LoopBody identifies the input Port that provides the initial value for each feedback data flow. Even thought, in OpenForge there are three kinds of LoopBodies: ForBody, WhileBody, UntilBody. In Xronos, only the WhileBody is used because the Procedural IR contains only one kind of Loops the `BlockWhile`.

4.7.6 Design

Design is the top level representation of an implementation in hardware. It defines a set of input and output ports, internal logical memories, clock domains and its computational part the Tasks.

Input/Output Interface

Designs communicate with each other via an interface protocol as depicted in Figure 4.16. This interface protocol is compatible with standard FIFO interfaces such as First Word Fall Through (FWFT). The FIFO size for each *Design* interconnection can be determined according to the user or from various strategies. A strategy for minimal FIFO size is illustrated in Section 5.4.3.

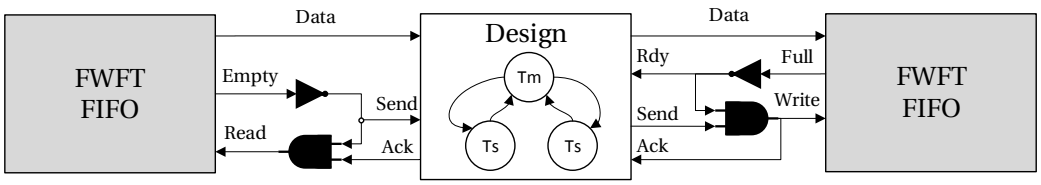


Figure 4.16 – Design I/O Fifo Interface.

The interface has four signals: *DATA*, *SEND*, *ACK* and *RDY*. *DATA*, is an N-bit data signal containing the token value. *SEND*, a 1-bit signal indicating that the value on *DATA* is valid. *ACK*, is a 1-bit acknowledge signal indicating that the token on *DATA* is being consumed. The token transaction occurs on the rising edge of the system clock when both *SEND* and *ACK* are asserted. Finally, *RDY*, is a 1-bit ready signal, which is incorporated only on output interfaces. If the ready is asserted, the consumer is indicating that an *ACK* will be provided immediately (combinatorially) in response to the assertion of *SEND*. Xronos also provides asynchronous FIFOs with two clock domains a write and a read one.

Logical Memories

See Section 4.7.4 for the logical Memories that the Design supports.

4.7.7 Clock Domains

Each `Design` by default has a single clock domain `CLK`. If a `Network` has been partitioned in different clock domains by the user, then this information is passed to a `Design` and the name of the clock domain is modified. The transition between two Designs belonging to different clock domains is effectuated by instantiating asynchronous FIFOs.

Task

A Task is a thread of execution within a *Design*. It exists two type of Task: *masters* and *slaves*. A *master* can *Call* slaves but a *slave* can not call neither a *master* nor *slave*. Also, a *master* can not call itself. The executable contents of a Task are expressed as a Call to a Procedure (to not be confused with the Procedural IR's Procedure). A task starts execution once its *go* signal has been activated and terminates by sending *done* signal. If a task can be executed combinatorially, then the done signal is connected directly to go, which means that it took zero clock cycles to execute the task.

- **Call**: Is a Reference to a *Procedure*. It represents a call by name, and it is expected that the HDL compilation will resolve the reference. A Call contains maps of Call's to referent

procedure body for ports, exits and buses.

- **Procedure**: Is a Component wrapper that allows the Component to be invoked via the execution of a Call. A Procedure represents the definition of a call-by-name. It is expected that the Procedure will be defined once in the output HDL and that the HDL compiler will link each call to that definition. Procedures are not in themselves shared; each Call represents a new instantiation of the hardware defined by the Procedure. A Procedure has a label and a *Block*.

4.7.8 Scheduling

LIM objects are scheduled by As Soon As Possible unconstrained scheduling. In general, Xronos operates as there is an unlimited amount of available resources in an FPGA. The reason for this is that resource sharing (a single hardware unit to implement multiple operations locally in a block or a super-block) requires adding multiplexers to the input ports of a shared hardware unit. As a matter of fact, multiplexers are costly to implement in FPGAs [171].

Let's consider a Block that has N components. The data control-flow graph of the block is a directed acyclic graph $G(V,E)$, with V the list of the components of the Block and E the data dependencies. Each $v_i \in V$ represents an operation o_i in the behavioral description. A direct edge e_{ij} from $v_i \in V$ to $v_j \in V$ exists in E if the data produced by operation o_i is consumed by o_j. In this case v_i is an immediate predecessor of v_j which is denoted by $Pred_{v_i}$. The immediate predecessor in LIM is achieved by asking the entry of v_i of its port dependency. In the same way v_j is an immediate successor of v_i denoted by $Succ_{v_i}$ and in LIM the successor of the component is given by the Exit data buses

Each component in LIM can be executed in D_i control steps that are given by the control dependencies of the *Block* components. Each control step signifies a latency of zero or more clock cycles. Xronos assumes that each combinatorial operation has a D_i of a latency ZERO. That means that a set of operation connected to each other is chained in the same state. Modules, such as Blocks, Branches and Loops, schedule first their internal components. Their Latency is the accumulation of the Latencies of their internal components. Thus, it exists a scheduling function for each *Component* and each class of *Module* (Block, Branch, Loop, LoopBody).

Considering the information given by the previous paragraphs, Xronos applies ASAP on a Block as defined in [20] or [24]. It should be noted that the LIM scheduler properly handles instructions with multi-cycle latencies, such as pipelined multipliers or memory accesses and loops.

4.8 Mapping of Dataflow and Procedural IR to LIM

The mapping of the Dataflow and Procedural IR to LIM is the "heart" of Xronos. The Xronos mapping process is implemented as a set of visitors that creates for each Network, Actor State Var, Action, Procedure, Operation, Instruction, Expression, BlockBasic, BlockIf and BlockWhile an equivalent LIM object. For each `Actor` a `Design` object is generated which represents the hardware implementation of an `Actor`.

4.8.1 Network construction and Actor to Design

The source code of a `Network` of Dataflow IR is generated using an Xtend VHDL template, as explained in Section4.8.9. For each `Actor` in the `Network` a LIM `Design` is created and for each state variable `Var` a memory allocation is generated in `Design`. After that, all `Actions` are visited and a LIM *slave* `Task` is created and added to the `Design`. Finally, the `Action Selection` is mapped into a LIM *master* Task. Once the Design is complete, scheduling, allocation, binding is effectuated and a Verilog RTL file for each `Actor` is created. Algorithm 6 represents the mapping from an `Actor` to a `Design`.

Algorithm 6: Var to LogicalMemory

Input :Actor: A Prcedural IR Actor
Result :Design: A LIM Design

```
1  design := create an empty design ;
2  // Create the Block and the components
3  ;
4  VarToLogicalMemory(Actor get State Variables, Design);
5  // Create Task from the sctors actions
6  for a_i ∈ Actor Actions do
7      task := ActionToTask(a_i.body);
8      Add task to Design;
9  master_task := ActorToTask(get Action Selection Procedure from Actor) set attribute master to master_task;
10 Add master_task to Design;
11 return Design;
```

4.8.2 State variable to Memory Allocation

For each state variable (`Var`) in an actor, a memory allocation (`LogicalMemory`) is created in a Design. Tasks are not allowed to have list memories, thus for every *Action* that contains list local variables, a visitor will visit each action procedure body and will: 1) create a state variable with the name of the local variable and the name of the action, 2) add this state variable to the actor,3) visit all `InstLoad` and `InstStore` instructions and replace their source and target variable respectively with the variable previously created, and 4) delete the local variable of the `Procedure`.

All state variables if they are not initialized yet, will be initialized with 0 or with $false$ by a visitor depending on their type.

Algorithm 7: VarToLogicalMemory

```
   Input : SV: state varibales set
   Input : Design: A design
   Input : collocate: A boolean that signifies if memories should be collocated
   Data  : M : a map with entries as LogicalMemory and values as strides
1  for v_i ∈ SV do
2      lv_i := create LogicalValue(v_i)
3      stride := lv_i.getStride();
4      m_i := M.get(stide);
5      if m_i = ∅ then
6          m_i := create LogicalMemory; m_i.name := var.name; design.add(m_i);
7          if collocate then
8              M.put(stride,m_i);
9      mem.allocate(lv_i);
```

The Algorithm 7 starts by creating a LogicalValue for v_i by taking as stride the number of bits of the *Type* of v_i and as value the initial one of the v_i. Then, it tests whether a memory with the same stride exists and if not it creates one and adds it to the design. If a designer has chosen that the memories should be collocated for memory consistency, then Xronos puts this LogicalMemory to the Memories map.

4.8.3 Action to Task

Each Dataflow IR `Action` is mapped to a LIM `Task` and then added to the `Design` of the actor. The Algorithm ActorToTask, which is not represented here for its simplicity, takes as input a `Procedure`, and returns as output a `Task`. It visits all `Blocks` of the Action body `Procedure` and creates a LIM `Block`. Finally, it adds the `Block` to the `Task`.

4.8.4 Operation to Node

The Procedural IR operation included in the *Expressions* is represented as a single node v_{op_i} that is not a subgraph. v_{op_i} has a set of inputs $P^{in}_{v_{op_i}}$ and as an output a set of a single entry $P^{out}_{v_{op_i}}$. In addition, it has a member that defines the kind of operator $v_{op_i}.op$, and $v_{op_i}.component$ member signifies the LIM component associated with v_{op_i}.

4.8.5 Expression to CDFG

An Expression is mapped to a Block. This is done because an Expression may contain another expression. Thus, an expression on the CDFG is represented as a node v_{e_i} that contains another CDFG. A v_{e_i} has a set of inputs $P^{in}_{v_{e_i}}$ and as output a set of a single entry $P^{out}_{v_{e_i}}$.

The Algorithm 8 represents the process from a Procedural IR to a CDFG. For each Expression, a CDFG graph is constructed. Procedure `PROPAGATE_NODE_INPUTS` takes all inputs of a v_{e_0}, creates a Port on the parent graph and adds an edge between the Port and the v_{e_0} input

Algorithm 8: Expression To CDFG

Input : E: Expression
Result : G: CDFG Graph

if *E is ExprBinary* **then**
- E_0 = E.getExpressionE0();
- v_{e_0} = ExpressionToCDFG(E_0);
- G.addNode(v_{e_0});
- E_1 = E.getExpressionE1();
- v_{e_1} = ExpressionToCDFG(E_1);
- G.addNode(v_{e_1});
- v_{op_0}.op = BinaryOperation(E.OpBinary);
- G.addNode(v_{op_0});
- `// Create data and control edges`
- PROPAGATE_NODE_INPUTS(v_{e_0}, G);
- G.addEdge("data", $P^{out}_{v_{e_0}}(0)$, $P^{in}_{v_{op_0}}(0)$);
- G.addEdge("data", $P^{out}_{v_{e_1}}(0)$, $P^{in}_{v_{op_0}}(1)$);
- G.addEdge("control",v_{e_0},v_{op_0});
- G.addEdge("control",v_{e_1},v_{op_0});
- PROPAGATE_CONTROL(v_{op_0}, G);
- return G;

else if *E is ExprUnary* **then**
- `// Create a CDFG from Expression, and add` v_e `to G`
- E_0 = E.getExpression();
- v_{e_0} = ExpressionToCDFG(E_0);
- G.addNode(v_{e_0});
- `// Create an operation node and add it to G`
- v_{op_0}.op = UnaryOperation(E.OpUnary);
- G.addNode(v_{op_0});
- `// Create data and control edges`
- PROPAGATE_NODE_INPUTS(v_{e_0}, G);
- G.addEdge("data", $P^{out}_{v_{e_0}}(0)$, $P^{in}_{v_{op_0}}(0)$);
- G.addEdge("control",v_{e_0},v_{op_0});
- G.type = $Block$;
- PROPAGATE_CONTROL(v_{op_0}, G);
- return G;

else if *E is ExprVar* **then**
- v_{op_0}.op = NoOperation(E.Var);
- $P^{in}_{v_{op_0}}$ = E.Var;
- G.addNode(v_{op_0}) ;
- PROPAGATE_NODE_INPUTS(v_{op_0}, G);
- PROPAGATE_CONTROL(v_{op_0}, G);
- return G;

else if *E is (ExprInt or ExprUint or ExprBool)* **then**
- v_{op_0}.op = Constant(E.Value, E.Type);
- $P^{in}_{v_{op_0}}$ = $\emptyset$;
- G.addNode(v_{op_0});
- PROPAGATE_CONTROL(v_{op_0}, G);
- return G;

port. Procedure `PROPAGATE_CONTROL` produces a *control* edge from a CDFG node to the CDFG. Internal *data* and *control* edges are resolved for each class Expression.

4.8.6 BlockBasic to Block

A Procedural IR `BlockBasic` is mapped to a LIM `Block` through the `BlockBasicToBlock` Algorithm 9. All `BlockBasic`'s Instructions are mapped to a CDFG node. The list of all instructions creates a CDFG graph that represents all data and control dependencies. Finally, the CDFG is mapped to the LIM Block by the CDFGToBlock Algorithm 10. For simplicity only the CDFG node of the `InstAssign` is illustrated in Algorithm 9. The input and output Ports for each CDFG nodes are given by the Liveness analysis illustrated in Section 4.2.4. The procedure `PROPAGATE_NODE_INPUTS(source,target)` will create a CDFG node, and will add a data dependency from the source to target.

Algorithm 9: BlockBasicToBlock

```
Input  : BlockBasic: A Prcedural IR BlockBasic
Result : Block: A LIM Block
Data   : G: A CDFG
Data   : M_V: a Map of defined Var and CDFG Ports
Data   : last_node: Last CDFG node
1  for inst_i ∈ BlockBasic Instructions do
2      if inst_i is InstAssign then
3          E = inst_i.Value;
4          v_e = ExpressionToCDFG(E);
5          G.addNode(v_e);
6          for p_in_j in P_ve^in do
7              var = p_in_j get variable;
8              if var ∈ LiveIn(BlockBasic) then
9                  PROPAGATE_NODE_INPUTS(p_in_j, G);
10             else
11                 G.addEdge("data", M_V.get(var), p_in_j);
12         var = inst_i.Target;
13         if var ∈ LiveOut(BlockBasic) then
14             PROPAGATE_NODE_OUTPUTS(P_ve^out(0), G);
15         else
16             M_V.add(var,P_ve^out(0));
17         PROPAGATE_CONTROL(last_node, v_e);
18         G.kind = "Block";
19         last_node := v_e;
20     else
21         ...
22 Block = CDFGToBlock(G);
23 return Block;
```

4.8.7 BlockIf to Branch and BlockWhile to Loop

A visual mapping of the BlockIf and BlockWhile CDFG is illustrated in Figure 4.10. For BlockIf, the condition is first inserted into BlockBasic (BB2) and is extracted from the CDFG. The

Then and *Else* Blocks CDFG is extracted as well. Afterwards, the CDFG graph of the BlockIf is constructed as follows: 1) an Entry node is inserted and control dependency is added to the Entry and to the condition CDFG. If any data dependencies are required by both the *Then* and the *Else* Blocks, then the data dependencies are inserted from the Entry to the *Then* and *Else* Blocks by the `LiveIn` set of the Blocks. In addition, an Exit node is inserted into the CDFG and all control and data dependencies are propagated from the *Then* and *Else* Blocks to the Exit node. Finally, the BlockIF CDFG is mapped to Branch by the CDFGToBlock Algorithm.

The BlockWhile CDFG has two Entry nodes, the Init Entry and the FeedBack (FB) Entry. The Init entry propagates all the input data dependencies to the BlockWhile CDFG, whereas the Feedback Entry propagates all feedback data dependencies to the *Body* CDFG node and to the BlockBasic CDFG node. The condition CDFG node has two control dependencies. A **T** control dependency which is connected to the condition BlockBasic node that signifies that the loop continues, and a **F** data dependency that is attached to the Exit node that signifies that the loop has completed.

4.8.8 CDFG to Block

The Algorithm 10 represents the transformation from a CDFG graph to a LIM Block. First all components of the Block are created and then the data and control dependencies are being resolved.

4.8.9 Behavioral HDL Code Generation

The `Network` of an RVC-CAL program is generated as a VHDL file. The entity has the same name as the `Network`'s name and for each input and output port additional VHDL ports are added to the entity as outlined in Section 4.7.6. The architecture of the `Network` declares all `Designs` (Actors) as components. Moreover, fanouts and queues are instantiated directly because of their fanout and their queue size which are specified as parameters. Finally, all internal signals are declared in advance and all Actors are then instantiated with correct port maps between fanout and queues. An example of a Network dataflow architecture is depicted in Figure 4.17.

OpenForge backend generates a Verilog Module file for each `Design` that represents an `Actor`. To clarify the code generation the CAL source code of the Actor *Acc* in Listing 4.4 is given. Figure 4.18 represents the Actor *Acc* Module, the internal signals and the inner Modules. As described previously an `Actor Design` has a master Task which is the Action Selection, and a set of slave tasks. In the Actor `Acc` example, it contains only one, and it is called *Action*. Each memory allocation (a state variable in Procedural IR) is also a Module (StateVar *acc*). Each Design has two additional Modules: *Global Reset* and *Kicker*. Global Reset is used to synchronize the reset with the actor clock. The Kicker generates a pulse based on the synchronized reset signal and is used to activate the Action Selection Module.

Algorithm 10: CDFG to Block

```
Input  : G: CDFG
Result : Block: a LIM Block
Data   : C: a list of components
Data   : Entry: entry of the Block
Data   : Exit: exit of the component
Data   : M: a Map of nodes and components
// Create the components
for v_i ∈ G vertices do
    if v_i.type is Block then
        b_{v_i} = CDFGtoBlock(v_i);
        C.add(b_{v_i}); M.put(v_i, b_{v_i});
    else
        c_{v_i} = v_i.component;
        C.add(c_{v_i});
        M.put(v_i, c_{v_i});
// Create the Block and the components
Block = create a Block with C as the list of its components ;
// Resolve data dependencies
for d_i ∈ G data edges do
    P_s = d_i.source;
    P_t = d_i.target;
    if P_s and P_t not a Graph Input or Output Port then
        Bus bus = P_s get Bus;
        Port port = P_t get Port;
        ADD_DATA_DEPENDENCY(bus,port);
    else if P_s is a Graph Input Port then
        P^{in}_{Block} = create a Block port;
        Bus bus = P^{in}_{Block} get peer Bus;
        Port port = GET_COMPONENT_PORT(P_t);
        ADD_DATA_DEPENDENCY(bus,port);
    else if P_t is a Graph Output Port then
        P^{out}_{Block} = create a Block bus;
        Bus bus = P_s get Bus;
        Port port = P^{out}_{Block} get peer Port;
        ADD_DATA_DEPENDENCY(bus,port);
// Resolve control dependencies
for c_i ∈ G control edges do
    P_s = c_i.source;
    P_t = c_i.target;
    if P_s and P_t not a Graph Input or Output Port then
        v_s = source node;
        v_t = target node;
        Bus bus = P_s get done Bus of v_s.component;
        Port port = P_t get done Port of v_t.component;
        ADD_CONTROL_DEPENDENCY(bus,port);
    else if P_s is a Graph Input Port then
        v_t = target node;
        P^{in}_{Block} = get Block go Port;
        Bus bus = P^{in}_{Block} get peer Bus;
        Port port = P_t get done Port of v_t.component;
        ADD_CONTROL_DEPENDENCY(bus,port);
    else if P_t is a Graph Output Port then
        v_s = source node;
        P^{out}_{Block} = get Block done Bus;
        Bus bus = P_s get done Bus of v_s.component;
        Port port = P^{out}_{Block} get peer Port;
        ADD_CONTROL_DEPENDENCY(bus,port);
return Block;
```

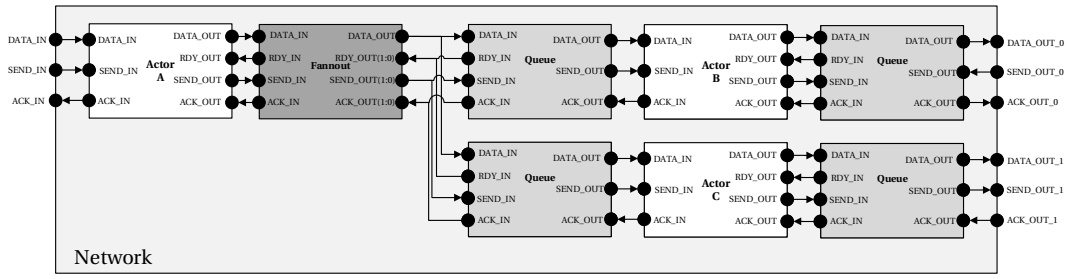


Figure 4.17 – Network representation of three Actors. The Actor A's output port is connected to the input of Actor B and Actor C. It is woth mentioning that if an output port is connected to more than one input port a fanout is added. As depicted, each connection has its proper queue.

In a case of multiple actios that read and write from the same state variable, a memory referee is added to the Actor's Module. RAMs and ROMs are also Modules, openForge applies a binding algorithm that instantiates correctly Xilinx RAMs (LUT RAM and Dual/Sigle Port BRAM). If chosen by the user, Xronos can also instantiate generic memories.

Listing 4.4 – An accumulator actor

```
actor Acc()
    int IN
        ==> int OUT:

    int acc := 0;

    action IN:[token] ==> OUT:[acc]
    do
        acc := acc + token;
    end
end
```

4.9 Xronos SystemC Code Generation

SystemC is a modeling and simulation language based on standard C++ classes expressly conceived for systems modeling and design. SystemC provides an event-driven simulation interface that enables the simulation of concurrent processes. Other relevant features of SystemC are the support for bit accurate datatypes, timing primitives that can be specified by the user and communication patterns among processes and modules. Such features make SystemC a preferable starting design point than standard C or standard C++ for high-level hardware synthesis for the more natural and expressive way of specifying concurrence.IEEE SystemC standard was developed initially for the simulation and verification of complex Intellectual Property (IP) blocks, but later has been used as an High-Level Synthesis (HLS) approach alternative to C and C++. The Open SystemC initiation group is in the process of

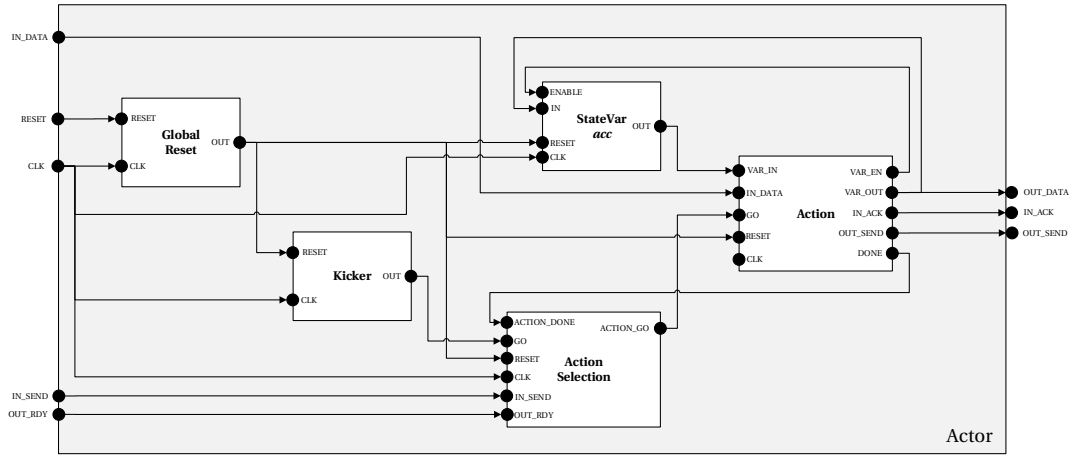


Figure 4.18 – Internal Modules Representation of the Actor Acc of Listing 4.4.

standardizing a subset of Synthesizable SystemC (SCC) [172]. The current draft consists of the definition of a synthesizable subset of the SystemC language and provides the coding guidelines and the specification of which SystemC syntactic elements could be synthesized by HLS tools.

4.9.1 SystemC Actor Template

The SystemC code generator generates hierarchically C++ header and source files for the network and each actor. The network of actors is a header file that instantiate the queues and the actors. Each actor is a SystemC module. The SystemC module defines the actor's ports, the methods declaration of the Procedural IR procedures and the constructor which register the `Action Selection` method as a SystemC CThread process. The source file contains the state variables that are declared as static, the implementation methods for each IR procedure and the implementation of the `Action Selection`. Figure 4.20 represents the SystemC module declaration of the inverse quantification CAL actor in Figure 4.5.

The action selection is a process that is sensitive to the positive clock edge, and it can be reseted. It is an infinite process that waits for the start signal to start executing. This process acts on each input and output port and calculates all the action guard conditions (the boolean return value of each guard expressed by the isSchedulable procedure) a priori before selecting which action should be executed.

Actions could read automatically from FIFO ports but in some cases a guard can be dependent not only on the head of the queue but even further. From the SSC draft only primitive *sc_fifo* types are supported which does not provide the peek support on queues. Those two conditions requires a supplementary *READ* state. This state reads the tokens from the input ports and stores it in a register or an array with size of the maximum number of readable tokens of all

Listing 4.5 – Dequantizer.cal

```
actor Dequantizer()
    int(size=24) Block, uint(size=8) QT, int(size=16) SOI ==> int(size=13) Out:

    List(type:List(type:uint(size=8), size=64), size=2) quant;

    int count := 0;
    int comp;

    receive_QT:action QT:[q] repeat 130, SOI:[w, h] ==>
    guard
        count = 0
    do
        foreach int i in 0 .. 63 do
            quant[0][i] := q[i + 1];
            quant[1][i] := q[i + 66];
        end
        comp := 0;
        count := 6 * (w) * (h);
    end

    // Dequant and unzigzag.
    receive_block:action Block:[b] repeat 64 ==> Out:[out] repeat 64
    guard
        count != 0
    var
        int compType,
        int(size=24) out[64]
    do
        compType := comp >> 2;
        foreach int i in 0 .. 63 do
            out[inv_zigzag[i]] := b[i] * quant[compType][i];
        end

        count := count - 1;

        comp := (comp + 1) mod 6;
    end

    priority
        receive_QT > receive_block;
    end
end
```

actions that reads from this particular port. A positive side of having a state for reading is that it can multiple port reading at the same time because the reading is independent. To have this advantage also for writing on outputs ports a *WRITE* state is inserted.

```
#include "systemc.h"
SC_MODULE(dequant){
 // -- Control Ports
 sc_in<bool> clk;
 sc_in<bool> reset;
 sc_in<bool> start;
 // -- Queue Ports
 sc_fifo_in<sc_int<24> > Block;
 sc_fifo_in<sc_uint<8> > QT;
 sc_fifo_in<sc_int<16> > Block;
 sc_fifo_out< sc_int<13> > Out;

 // -- Action Body Declaration
 void r_QT();
 void r_block();
 void isSchedulable_r_QT();
 void isSchedulable_r_block();

 // -- Action Selection
 void action_selection();
 SC_CTOR(dequant)
 :clk("clk")
 ,reset("reset")
 ,start("start")
 {
  // -- Actions Selection Registration
  SC_CTHREAD(action_selection, clk.pos());
  reset_signal_is(reset, true);
 }
};
```

Figure 4.19 – Header file of the SystemC inverse quantification actor.

4.9.2 SystemC Actor Composition Template

The Actor Composition SystemC Template is also a `SC_MODULE`. It defines the input and output port as `sc_fifo` and the constructor `SC_CTOR` instantiates all actors, internal queues and finally makes the interconnection between I/O ports, queues and actors.

```
void dequant::action_selection(){
 // -- Ports indexes
 sc_uint<32> p_Block_index = 0;
 sc_uint<32> p_Block_index_read = 0;
 bool p_Block_consume = false;
 ...
 bool p_Out_produce = false;
 // -- Action guards
 bool guard_receive_QT;
 bool guard_receive_block;

 wait();
 state = s_dequant;
 old_state = s_dequant;
 // -- Wait For Start singal
 do { wait(); } while ( !start.read() );
 while(true){
  // -- Calculate all guards
  guard_receive_QT = isSchedulable_receive_QT();
  guard_receive_block = isSchedulable_receive_block();

 switch (state){
   case (s_READ):
    if(p_Block_consume && ( Block.num_available() > 0 ) ){
     for(int i = 0; i < p_Block_index_read; i++){
      p_Block[i] = Block.read();
      p_Block_index++;
      p_Block_consume = false;
     }
    }
    if(p_QT_consume && ( QT.num_available() > 0 ) ){
    ...
    break;
  case (s_WRITE):
   if(p_Out_produce){
    for(int i = 0; i < p_Out_index_write; i++){
     Out.write(p_Out[i]);
     p_Out_index++;
    }
    p_Out_produce = false;
   }
   state = old_state;
   break;
   case(s_dequant):
    if(guard_receive_QT && p_SOI_index == 2
       && p_QT_index == 130){
     receive_QT();
     p_SOI_index = 0;
     p_QT_index = 0;
     state = s_dequant;
    } else if(guard_receive_block && p_Block_index == 64){
     receive_block();
     p_Block_index = 0;
     p_Out_index_write = 64;
     p_Out_produce = true;
     old_state = s_dequant;
     state = s_WRITE;
    } else {
     if( p_SOI_index < 2 ){
      p_SOI_index_read = 2 - p_SOI_index;
      p_SOI_consume = true;
    }
    if( p_Block_index < 64 ){
    ...
    old_state = s_dequant;
    state = s_READ;
   }
   break;
   default:
    state = s_dequant;
   break;
  }
 wait();
}
```

Figure 4.20 – Action Selection process of the inverse quantification actor.

4.10 Xronos C++ Code Generation for Embedded Platforms

Xronos generates C++ source code for software processing elements. A Xtend class extends the Dataflow and Procedural IR and visits all Actors in the Dataflow IR with purpose to create a single header C++ class file that represents the Actor model.

The structure of the C++ Actor is represented in Listing 4.6. To facilitate the understanding of how the C++ code is generated, the same CAL Dequantizer of Listing 4.5 is used here as well.

Listing 4.6 – Structure of C++ Actor

```
#ifndef __<NAME_OF_ACTOR>_H__
#define __<NAME_OF_ACTOR>_H__

#include <iostream>
#include "actor.h"
#include "fifo.h"

class <name of actor>: public Actor{
public:
    <name of actor>(){
        ..
    }
    /* FIFO Declaration */
    Fifo<<port type>, <number of fanout>>* port_<port name>;
    ...

    /* Action Declaration, isScedulable and Body */

    bool isSchedulable_<action name>(){
        ...
    }

    void <action name> (){
        ...
    }

    /* Action Selection */
    void action_selection(EStatus& status){
        ...
    }

    /* Actor State Variables */
private:
    <type of variable> <name of variabel>;
    ...
};
#endif __<NAME_OF_ACTOR>_H__
```

Actor I/O

The Actor's Input and Output port are represented by the $FIFO$ Class developed specifically for the Xronos C++ Code Generation. Each $FIFO$ is defined by its type and by the number of readers: $FIFO < type, nbReaders >$. The $FIFO$ is implemented as a circular buffer with

two pointers read and write. Furthermore, the *FIFO* Class provides methods for retrieving the pointers *read_address* and $write_address$, by updating read and write pointers *read_advance* and *write_advance*, available tokens *count*, and available space *rooms*. Listing 4.7 represents the three input ports and one port of the actor Dequantizer.

Listing 4.7 – Input and Output Queues

```
public:
    Fifo<int, 1>* port_Block;
    Fifo<unsigned char, 1>* port_QT;
    Fifo<short, 3>* port_SOI;

    Fifo<short, 1>* port_Out;
```

State Variables

Actor state variables are C++ private members and are initialized by the class constructor.

Listing 4.8 – State Variabes

```
...
private:
    unsigned char inv_zigzag[64];
    unsigned char quant[2][64];
    int count;
    int comp;
...
```

Actions

`Action` *Body* and *isSchedulable* procedures are translated into C++ function members. As illustrated in Listing 4.9, if an action access data from input and output ports it retrieves the pointers of the FIFOs by the *read_address* and *write_address* methods. At the end of the function it updates FIFO indexes and its local token counters with *read_advance* and *write_advance* methods. Contrarily to the hardware code generation, the CAL repeat statement is supported by reading and writing directly from the FIFO pointers. Thus, there is no need for an individual state in the action selection for reading and writing.

Listing 4.9 – Action Body and isSchedulable Functions

```
private:
    bool isSchedulable_receive_block (){
        bool result;
        int local_count;
        local_count = count;
        result = (local_count != 0);
        return result;
    }
    void receive_block (){
        int* Block = port_Block->read_address(0, 64);
        short* Out = port_Out->write_address();
        int compType;
```

```
        int local_comp;
        int i;
        unsigned char tmp_inv_zigzag;
        int tmp_Block;
        unsigned char tmp_quant;
        int local_count;
        local_comp = comp;
        compType = (local_comp >> 2);
        i = 0;
        while((i <= 63)){
            tmp_inv_zigzag = jpeg_decoder_Dequantizer_inv_zigzag[i];
            tmp_Block = Block[i];
            tmp_quant = quant[compType][i];
            Out[tmp_inv_zigzag] = (tmp_Block * tmp_quant);
            i = (i + 1);
        }
        local_count = count;
        count = (local_count - 1);
        local_comp = comp;
        comp = ((local_comp + 1) % 6);
        port_Block->read_advance(0, 64);
        status_Block_ -= 64;
        port_Out->write_advance(64);
        status_Out_ -= 64;
    }
```

Action Selection

Compared to Xronos' hardware Action Selection procedure, the construction for the C++ code generation is done directly in the Xtend template. The action selection starts by getting the information of available tokens on input queues and available space of output queues. Then the firing conditions of the actor are being tested inside a `while` statement. The condition of the `while` is given by the Boolean variable "res" that indicates if an actor can still execute. For example if no input tokens are available at a particular time, then "res" is false, and the action selection terminates its execution.

The firing condition is a combination of the available input tokens and the `isSchedulable` function return value. These conditions are provided as an if condition. If the condition is true, then the actor fires the action by calling the "action body" function and by updating the status to *hasExecuted*.

Listing 4.10 – Action Selection

```
public:
    void action_selection(EStatus& status){
        status_Block_=port_Block->count(0);
        status_QT_=port_QT->count(0);
        status_SOI_=port_SOI->count(1);
        status_Out_=port_Out->rooms();

        bool res = true;
        while (res) {
            res = false;
            if(status_QT_ >= 130 && status_SOI_ >= 2 && isSchedulable_receive_QT()){
```

```
                receive_QT();
                res = true;
                status = hasExecuted;
            }
            else if(status_Block_ >= 64 && isSchedulable_receive_block()){
                if(status_Out_ >= 64) {
                    receive_block();
                    res = true;
                    status = hasExecuted;
                }
            }
        }
    }
```

Actor Composition

The actor composition is a C++ header file where the FIFO queues and the actors are declared. If a processing element has not as many cores as actors, then a round-robin scheduling is applied. Hence, if a mapping configuration is given with n processing cores, then n threads, with a partition of actors, are being created. Listing 4.11 illustrates the actor composition of Figure 3.2. The `EStatus` is an enumeration that defines whether an actor has executed. The instantiation and the connection of actors and FIFOs are effectuated in the constructor. Moreover, the actor composition class is instantiated by a `main` function that is manually written by the developer. This is due to the differences in each platform and development board. The developer needs to make all system calls and interface interconnections before calling the run function of the actor composition class.

Listing 4.11 – C++ header of an Actor Composition of Figure 3.2

```
#include <map>
#include <string>
#include "fifo.h"
#include "actor.h"

#include "A.h"
#include "B.h"
#include "C.h"

class Composition{
private:
    A *inst_A;
    B *inst_B;
    C *inst_C;
    /* FIFO Instanciation */
    Fifo<int, 1> fifo_0;
    Fifo<int, 1> fifo_1;
    Fifo<int, 1> fifo_2;

public:
    Composition(){
        inst_A = new A();
        inst_B = new B();
        inst_C = new C();
        fifo_0 = new Fifo<int, 1>;
```

```
        fifo_1 = new Fifo<int, 1>;
        fifo_2 = new Fifo<int, 1>;
        inst_A->port_I1 = I;
        inst_A->port_I2 = fifo_2;
        inst_A->port_O = fifo_0;
        inst_B->port_I = fifo_0;
        inst_B->port_O1 = O;
        inst_B->port_O2 = fifo_1;
        inst_C->port_I = fifo_1;
        inst_C->port_O = fifo_2;
    }
    ~Composition(){
        delete inst_A;
        ...
    }

    Fifo<int, 1> *I;
    Fifo<int, 1> *O;
    void run(){
        EStatus status = None;
        do{
            status = None;
            inst_A->action_selection(status);
            inst_B->action_selection(status);
            inst_C->action_selection(status);
        }while(status != None);
    }
}
```

4.11 Mapping HW-SW and Interface Synthesis

Different interfaces have been implemented and demonstrated in [9]. The HW-SW mapping results in partitioning the `Network` graph with actors belonging to a processing element (i.e. FPGA or CPU). The process mainly consists of transforming the Network by inserting additional vertices that represent communications between partitions, using the appropriate media between processing elements from the architecture (see Section 5.3). Such transformation introduces special vertices in the `Network`, which will encapsulate at a later stage the (de)serialization of data and the inclusion of the corresponding interfaces between partitions. This step is illustrated in Figure 4.1 (HW-SW Interface Wrappers) where (de)serialization (resp., Ser. and Des.) and interface vertices are inserted. An example of partitioning and the interface between partitioning is depicted in 4.21. Deserialization and Serialization for CAL dataflow programs were formalized in [150].

The serialization aims to schedule the communication between actors that are allocated on different partitions at runtime. Whenever several edges from the dataflow graph are associated with a single medium of the architecture, the data needs to be interlaced in order to be able to share the same underlying medium.

In the case of serialization, on the sender side, "virtual" FIFOs are used to connect the `serializer` to incoming actors. Contrarily to conventional FIFOs, that store data and

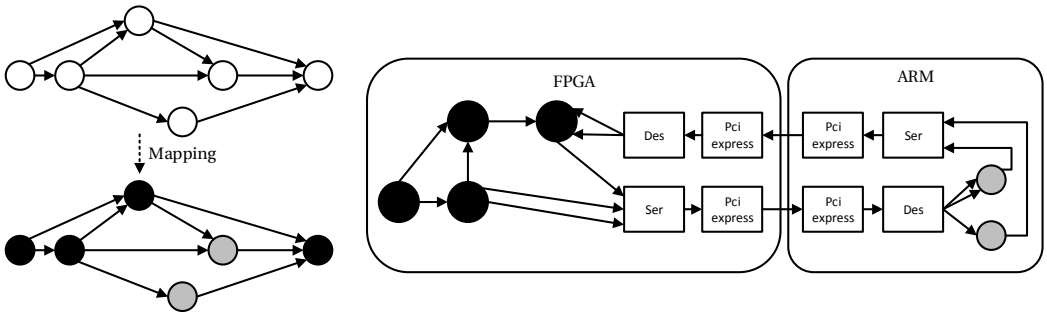


Figure 4.21 – Partitioning of a Design to FPGA and ARM CPU and Interface Synthesis.

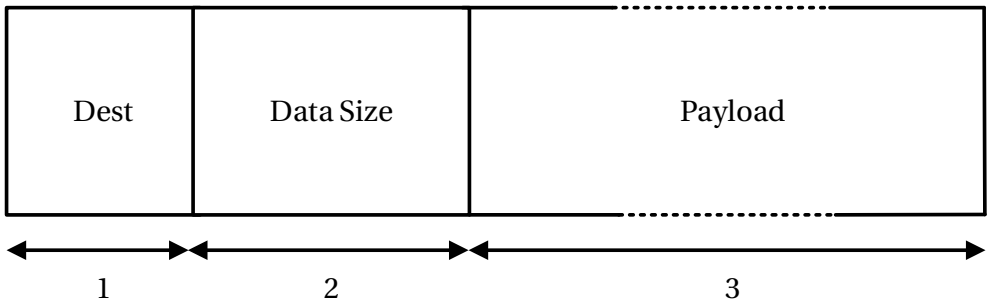


Figure 4.22 – Header and the payload of the stored data in the serialization FIFO.

maintain the state (read/write counters), the "virtual" FIFOs keeps the state while the data is directly stored into a single FIFO, shared by incoming actors. The idea behind such a procedure is to emulate the history of FIFOs (emptiness, fullness) in order to schedule reasonably the data in the serialization FIFO. Data is scheduled by actors themselves, without using any scheduling strategy in the `serializer`. In order to retrieve data on the receiver side, a header is placed at the beginning of each data that defines the destination FIFO and the size (in byte) of the payload. This simple header is illustrated in Figure 4.22. On the receiver side, conventional FIFOs are used to connect the `deserializer` to outgoing actors. The `deserializer` is responsible for the decoding of the header and puts the payload to the appropriate destination FIFO.

4.12 TestBench Generation and Profiling Data Extraction

Xronos creates for each actor and network a testbench that is used for functional verification and profiling. The input and output vectors of the testbench can be generated by the Orcc's bit-accurate RVC-CAL simulator or are given by a golden reference. Each testbench takes an input and output file (if any I/O has been defined for the actor or network) that represents

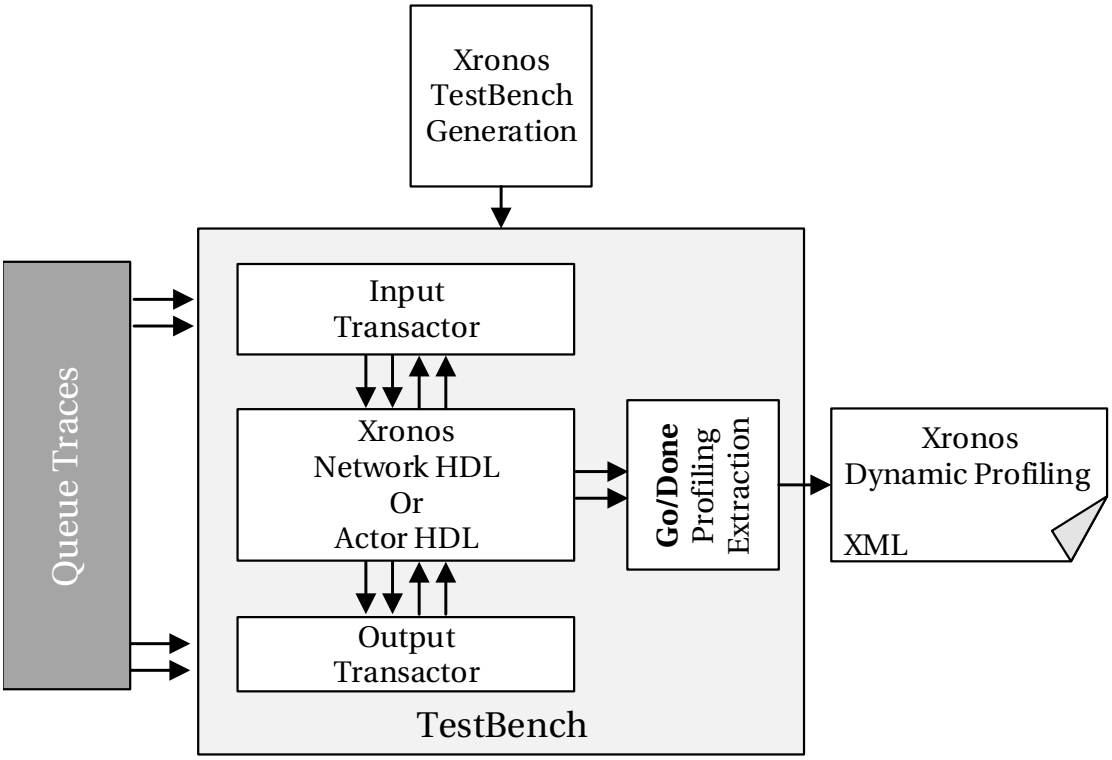


Figure 4.23 – Xronos TestBench and Profiling.

the input and output tokens. A driver module takes the input files and fills the input queues with the data file. Then a comparing module for each output file reads the golden references and the output of the unit that is under test and compares the results. If an error occurs, the simulation stops and indicates which output port and sequence has an incorrect value. The simulation finishes once all tokens of output ports have been compared with the golden references.

The Go/Done signals of each task are extracted from RTL simulation. The clock cycles difference between Done and Go indicates the latency for each Task execution. Thus, for each action firing is possible to extract the number of clock cycles required for an execution. The Go/Done signals values for each action at every clock cycle are dumped from the VCD waveform and are stored to file. After that, Xronos retrieves the data, calculates the Done/Go difference for each action and stores them to a map. Finally, an XML file is generated that contains for each actor the overall firings of actions and the elapsed clock cycles. In addition, the min, mean and max execution time for each action is included in the file. An example of profiling file is depicted in Listing 4.12.

Listing 4.12 – Example of Profiling Extraction File

```
<?xml version="1.0"?>
<actors>
    <actor name="encoder_huffman">
```

```
        <actions>
            <action name="dchuffman" min="3.0" mean="3.0" max="3.0" variance="0.0"/>
            <action name="sendstuffing" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
            <action name="stuffing" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
            <action name="getszh" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
            <action name="buildoutputbuffer" min="11.0" mean="16.95" max="45.0" variance="34.77/>
            <action name="achuffman" min="4.0" mean="6.89" max="34.0" variance="27.58"/>
            <action name="sendbits" min="2.0" mean="2.0" max="2.0" variance="0.0"/>
            <action name="donesend" min="3.0" mean="3.0" max="3.0" variance="0.0"/>
            <action name="generateht" min="44.0" mean="194.0" max="344.0" variance="25714.28"/>
            <action name="doneachuffman" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
            <action name="doneht" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
            <action name="getht" min="133.0" mean="133.0" max="133.0" variance="0.0"/>
            <action name="getblock" min="2.0" mean="2.0" max="2.0" variance="0.0"/>
            <action name="fillhtcodeslist" min="363.0" mean="363.0" max="363.0" variance="0.0"/>
            <action name="eoi" min="4.0" mean="4.0" max="4.0" variance="0.0"/>
            <action name="getszw" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
        </actions>
    </actor>
    <actor name="encoder_fdct_retranspose">
        <actions>
            <action name="read" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
            <action name="finish" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
            <action name="sent" min="1.0" mean="1.0" max="1.0" variance="0.0"/>
        </actions>
    </actor>
    ...
</actors>
```

For the Xronos C++ code generator, the same profiling information is extracted by using a profiling API such as the Performance API(PAPI) [173]. PAPI works on CPUs that contains hardware performance counters. Most new X86 and ARM CPUs are supported by PAPI on Linux Systems. A function call in PAPI is profiled by adding a PAPI system call at the entry of the function call. Thus, the hardware counters start. In addition, another PAPI system should be inserted at the exit of the function for stopping the counters. As a result, the clock cycles of the function are retrieved. As discussed in Section 4.10, every `Action` *Body* is mapped into a C++ function. Thus, the clock cycles for each firing are retrieved for every executed function thanks to the difference of the value of exit and entry counters. Finally, an XML file, as previously is extracted for profiling the generated C++ code.

4.13 Experimental Results

In this section, the experimental results are conducted. In the beginning, *StreamBench* a benchmark for RVC-CAL dataflow programs is assembled and synthesized with Xronos. This provides a benchmark for high-level synthesis of dataflow programs, not only in different application domains but also with distinct degrees of algorithmic complexity. In addition, StreamBench was also built for comparing the future advancement of Xronos or other alternative HLS for RVC-CAL. After that, Xronos is compared with another RVC-CAL HLS tool. This tool generates C code from a RVC-CAL program and calls afterwards Vivado HLS for the

synthesis. Moreover, an experiment on an embedded multi-core platform is effectuated for demonstrating the scalability of RVC-CAL when more processing power is available. Finally, a proof of concept experiment of hardware and software co-design on two heterogeneous platforms is conducted.

4.13.1 StreamBench: a benchmark suite for streaming applications

Benchmarking programs in StreamBench are brought from widely-used applications in the real world. Table 4.4 summarizes the brief description and the sources of the programs.

Table 4.4 – Brief description and source of the Streambench benchmark RVC-CAL programs.

Domain	Name	Description	Source
Filter	FIR	A 4-tap FIR filter	University of Oulu [174]
	IIR	A 1-tap IIR filter	University of Oulu [174]
	LMS	A adaptive LMS Filter	University of Oulu [174]
Arithmetic	DFADD	Double precision floating-point addition	SoftFloat [175]
	DFMUL	Double precision floating-point multiplication	SoftFloat [175]
Media	ADPCM	ADPCM Encoder and Decoder	SNU [176]
	GSM	Linear predictive coding analysis of GSM	MediaBench [177]
	JPEG	A JPEG Encoder	EPFL [2]
	MPEG4 SP	Serial MPEG-4 Simple Profile Decoder	Xilinx [146]
	RVC MPEG4 SP	Parallel MPEG-4 SP Simple Profile Decoder	MPEG

FIR: Is a Finite Impulse Response signal processing filter. The FIR implementation is a 4-tap which means that the order of the filter is 4 and therefore has 4 coefficients. It is an 11 actor RVC-CAL design developed by the University of Oulu [174].

IIR: Is an Infinite Impulse Response signal processing filter. It is implemented as a first order filter in RVC CAL, it has also been developed by the University of Oulu.

LMS: Is a Least Mean Square adaptive filter used to mimic a desired filter by finding its coefficients. It is also developed by Univesity of Oulu.

DFADD: Implements IEC/IEEE standard double-precision floating point addition using 64-bit integer numbers. It is implemented without loops. The original code is authored by SoftFloat [175]. The version used for this benchmark was implemented in RVC-CAL from the C description in [178].

DFMUL: Implements IEC/IEEE standard double-precision floating point multiplication using 64-bit integer numbers. DFMUL has several common functions which are also used in DFADD. The design was rewritten into RVC-CAL from the C description in [178].

ADPCM: Is an algorithm for Adaptive Differential Pulse Code Modulation. It implements the CCIT F.722 ADPCM algorithm for voice compression. The design includes two actors, one for encoding and one for decoding. It was rewritten into RVC-CAL from the C description in [178].

GSM: This design contains a part of Linear Predictive Coding analysis of the GSM communica-

tion protocol for mobile phones. Only the lossy sound compression of GSM is implemented. It was rewritten into RVC-CAL from the C description in [178].

JPEG: Is a RVC-CAL implementation [2] of YUV 4:2:0 JPEG encoder standard ISO/IEC 10918. The RVC-CAL JPEG encoder is composed by 10 actors. Firstly, the input image is transformed from a raster implementation to an YUV 4:2:0 macroblock by the raster to macroblock actor. Secondly a forward Discrete Cosine Transformation is applied by the FDCT actor which is composed by 6 actors. Then the quantization and zigzag scan algorithms are applied in the same actor. And finally, the variable-length encoding (Huffman) is applied to each block, and finally the bitstream organizer and writer generates a 4:2:0 JPEG bitsream.

MPEG-4 SP: Is a CAL implementation of full MPEG-4 4:2:0 Simple Profile decoder standard ISO/IEC 14496-2. The main functional blocks include a parser, a reconstruction block, a 2-D inverse discrete cosine transform (IDCT) block, and a motion compensator. All of these functional units are hierarchical compositions of actors in themselves. The entire description of the decoder is composed of 31 actors. In the first place, the parser analyzes the incoming bitstream and extracts the data from it. Then it feeds the data into the rest of the decoder depending on where it is need. It is to mention that the parser is a single actor that is composed by 71 actions. It is therefore the most complicated actor in the entire decoder. Thirdly, the reconstruction block performs the decoding that exploits the correlation of pixels in neighboring blocks. The IDCT is the most resource demanding actor as it performs most of the computation performed by the decoder. Finally, the motion compensator adds selectively the blocks by issuing from the IDCT the blocks taken from the previous frame. Consequently, the motion compensator needs to store the entire previous frame of video data, which it needs to address into with a certain degree of random access. In that case the entire frame is stored by the actor *ddr*. For implementation reasons and in order to make it possible to fit into an FPGA, the *ddr* memory actor can only fit a CIF (384x288 pixels) image. In a real implementation as well as performed in [146], this actor is replaced by a memory referee that communicates with an external memory.

RVC MPEG-4 SP: Is the RVC standarized MPEG-4 SP insipred by the previous decoder with the main difference being that Y, U, and V components have their own decoding units. Thus, the decoding is effectuated in parallel.

Table 4.5 and Table 4.6 summarize source-level characteristics such as the number of lines of RVC-CAL code, the number of actors, the number of actions and the number of Function/Procedures. In addition, Procedural IR characteristics such as Operators, Block Statements, and also Assign, Load and Store instructions are outlined. Furthermore, Table 4.6 contains the number of Assign, Load, and Store instructions.

Figure 4.24 illustrates the reduction on Load and Store Instructions after the Single Read and Write Register optimization. Designs such as FIR, IIR, and LMS have a higher rate of `Load` reduction. These designs have also the greatest Store Reduction whereas the DFADD has the second most significant Store reduction. For FIR, IIR, and LMS this reduction is due to

Table 4.5 – Program Characteristics - 1

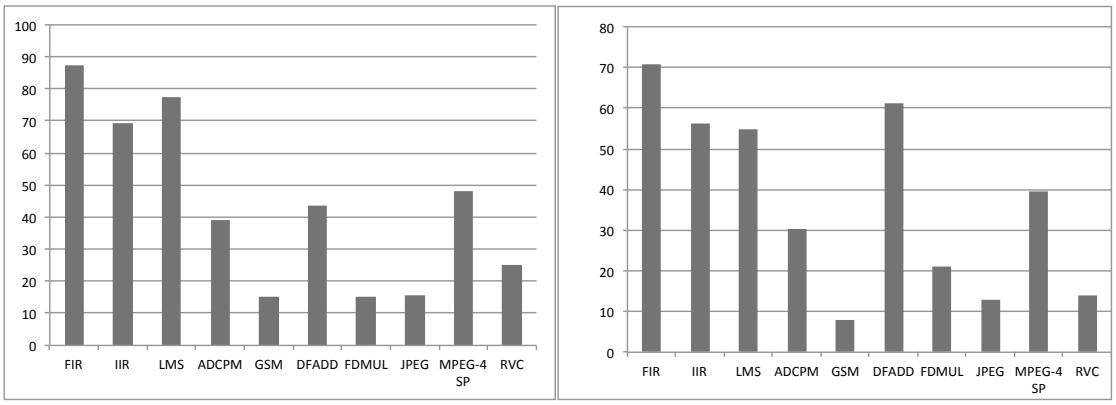
					Operators					
Name	**Lines**	**Actors**	**Actions**	**Procedures**	**+/-**	*****	**/**	**logic**	**shift**	**comp.**
FIR	48	11	11	0	6	8	0	2	2	0
IIR	48	5	5	0	4	4	0	0	2	2
LMS	75	36	43	0	28	28	0	57	4	0
ADCPM	692	2	2	18	152	12	0	7	51	47
GSM	482	4	4	17	83	1	0	29	18	85
DFADD	550	14	32	12	55	8	0	94	28	161
DFMUL	489	7	17	8	22	8	0	54	20	66
JPEG	1557	10	94	14	351	4	0	20	40	141
MPEG4-SP	4276	31	478	34	401	15	2	343	83	419
RVC	3899	43	380	83	2768	45	3	539	266	1050

Table 4.6 – Program Characteristics - 2

	Statements						
				Nominal		IR Optimization	
Name	**if**	**while**	**calls**	**load**	**store**	**load**	**store**
FIR	0	0	0	47	31	6	9
IIR	0	0	0	26	16	8	7
LMS	0	0	0	178	133	40	60
ADCPM	33	15	63	529	243	321	169
GSM	61	33	157	729	162	618	164
DFADD	165	0	37	343	327	193	127
FDMUL	75	0	31	230	180	195	142
JPEG	53	28	52	910	642	768	558
MPEG4-SP	404	6	433	2129	1217	1101	734
RVC	852	51	652	5074	2891	3798	2487

the small number of instruction for each action. In these designs each action contains just one single operation and only actors acting as delays have access to the registers. Thus, the majority of reduced load and store instructions are in the Action Selection procedure. It should be noted that the RVC design, which is the biggest design in terms of resources and source code lines, has a non-negligible Load reduction of 26 %.

Figure 4.25 illustrates the required hardware resources of the streambench designs when synthesized for a Xilinx Zynq 7045 (XC7Z045-2FFG900C). The graph is scaled from 0 % to 20 %, even thought the MPEG-SP BRAM utilization is 96% due to the required memory for the referenced images used by inter prediction. RVC is the second design that requires more BRAMs. However, that is normal considering the fact that each Y,U, and V component has its



(a) Load Reductions

(b) Store Reductions

Figure 4.24 – Load and Store Instruction Reduction after Single Read and Write Register Procedural IR Optimization.

Table 4.7 – Xronos HLS - Synthesis and Simulation Results.

Name	**Max Freq. (MHz)**	**Slice Registers**	**Slice LUTs**	**BRAM**	**DSP**	**Input Data**	**Output Data**	**Cycles**	**Throughput (Mbit/s)**
FIR	265	512	683	-	-	16340	16340	43581	3032
IIR	190	200	351	-	-	128	128	516	1438
LMS	119	1751	1241	1	42	16340	16340	228760	259
DFADD	86	3789	7820	2	-	46	46	600	107
DFMUL	117	3732	4115	15	16	20	20	368	313
ADPCM	52	6304	8785	1	156	100	100	4865	121
GSM	74	4270	8096	-	43	160	160	3642	61
JPEG	137	4867	9899	14	5	38016	3357	219821	16
MPEG4 SP	162	4416	8816	527	7	6748	190080	690260	340
RVC	113	15239	32087	57	27	6748	190080	989766	165

own framebuffer. In addition, the RVC design requires a considerable amount of LUTS(15 %). This is due to the replication of the same actors for decoding in parallel the Y, U and V components. ADPCM is the design that requires the most of DSP multipliers. In general terms all designs, except the two MPEG-4 SP decoders and ADPCM, require less then 6 % of the device. The BRAM of the decoders can be reduced significantly if the framebuffers are replaced by a direct access to DRAM.

Table 4.7 contains the synthesis and simulation results of the different dataflow designs. As illustrated, the synthesis frequency varies for each design. ADPCM has the lowest frequency because of it insufficient design modularity. It contains only two actors, each with one single action. As the number of lines indicates, each actor contains approximately 350 lines of sequential code. Consequently, this demonstrates the Achilles heel of the LIM scheduling. Considering a zero latency for each combinatorial operation it minimizes the total latency

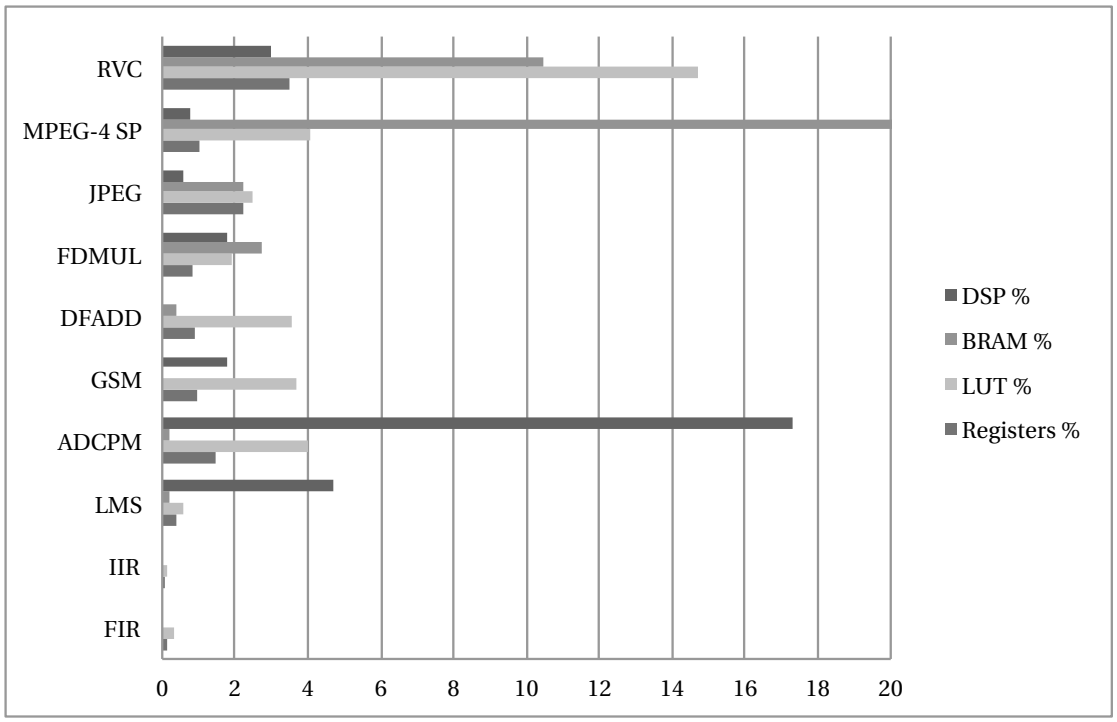


Figure 4.25 – Resource utilization on Xilinx Zynq 7045.

while increasing the critical path of the action. The GSM design on the other hand has exactly the same problem but its synthesis frequency is slightly higher. As design becomes more modular the synthesis frequency increases as can be observed for the rest of the designs. Finally, the FIR design has the highest frequency of them all. FIR is simple and every actor acts either as a single delay, as a multiplication, or as an addition. Due to this functional decomposition, each actor is connected to a queue and the critical path is defined by the combinatorial latency of the operator.

Table 4.8 – Xronos C++ Code generation Throughput results in Zynq 7045 ARM with a frequency of 999MHz.

Name	**Throughput (Mbit/s)**
FIR	87
IIR	6.2
LMS	3.9
DFADD	38.1
DFMUL	209.3
ADPCM	17.5
GSM	0.3
JPEG	1
MPEG4 SP	9.2
RVC	31.5

The throughput for each design is illustrated as Mbit/s in the last column of Table 4.7. It depends on the frequency of the circuit and on the elapsed cycles taken for generating the output data. The slowest design in stream-bench is the JPEG encoder becaus of the complexity of the huffman actor. The MPEG-4 SP decoder, however, has a higher throughput than RVC due to a better IDCT implementation and to improved handling of the inter frame images. Considering a video sequence of 1280x720 at a frame rate of 30 images per second (720p), the video decoder should have a throughput of 316Mbit/s (image width*image height*1.5*8*30, 4:2:0 format,8 bit for each Y,U,and V and 30 images per second). Consequently, the MPEG-4 SP decoder can decode 720p sequences, but only when the internal frame buffer is replaced by a DRAM. As a result, it is proved that with Xronos is possible to generate circuits that can be used in real applications domains such as video decoding and this is thanks to the modularity that the RVC-CAL programming language offers. Finally, due to their high frequency, the FIR and IIR filter have a very high throughput.

Table 4.8 illustrates the throughput of the StreamBench Xronos C++ generated designs. The generated code is compiled and run on a bare-metal ARM Cortex A9 core of the Zynq 7045 configured with a frequency of 999 MHz. The hardware throughput of Table 4.7 is higher for all designs. FDMUL SW throughput is only 1.5 slower than the hardware one but for the other designs the throughput is slower from 2.8 up to 233 times. This is due to the number of actors of the StreamBench designs and that for SW there is a Network scheduler (as described in Xronos C++ Actor Composition) that sequentializes the execution of actors, thus decreasing the performance. In addition, even thought ARM has a frequency of 999 MHz, memory and operation instructions takes from 1 to hundreds of cycles for the calculation of a statement. Finally, the ARM core in the Zynq can be used as a co-processor for executing sequential actors such as the entropy decoding of the RVC and MPEG-4 SP decoder or for actors that requires large memories that does not fit on an FPGA.

4.13.2 Xronos versus state-of-the-art RVC-CAL to hardware synthesis

In [3] the *Action Selection* procedure is compared with the Multi-to-Mono token transformation [167] on Virtex 4 FPGA. Table 4.9 illustrates synthesis and simulation results, which are retrieved from [3]. Synthesis results from the old Orc2HDl framework are also given. The values from the first column "Orc2HDL M2M" are retrieved from the paper of Jerbi and al. [167]. The second column "Xronos M2M" presents the evolution of the old framework Orc2HDL and Xronos with the M2M transformation being activated. As depicted Xronos has almost doubled the maximum synthesis frequency and has a speed-up of 3 times higher than Orc2HDL. Moreover, the slices are reduced by 39.9% and the LUTs by 25.6%. The low frequency on Orc2HDL is due to the division operator used in the Inverse AC prediction on the Texture block. The third column "Xronos" compares the Xronos support of multi-token, which is incorporated in the action selection procedure, with the M2M transformation. As with Xronos M2M, the results are better than the Orc2HDL M2M ones. Xronos gives a speed-up of 5, and uses almost less than 50% of hardware resources.

Table 4.9 – Three-way comparison of the same RVC Intra MPEG-4 SP decoder on a Virtex 4 FPGA, using the old Orc2HDL framework with the M2M source to source transformation, Xronos with the M2M and Xronos. (All results are post-place-and-route, using Xilinx XST.)

	Hardware Generators		
	Orc2HDL M2M	**Xronos M2M**	**Xronos**
Slices	45574	27388	19291
LUTs	68962	51295	35315
BRAMs	18	18	18
DSPs	48	48	48
Max Freq. (MHz)	26.4	51	65
Max fps	73.8	222	406
fps/Max Freq.	2.8	4.35	6.24
Slice Reduction%	-	39.9	**57.7**
LUT Reduction%	-	25.6	**48.8**
Speed Up	-	3	**5.5**

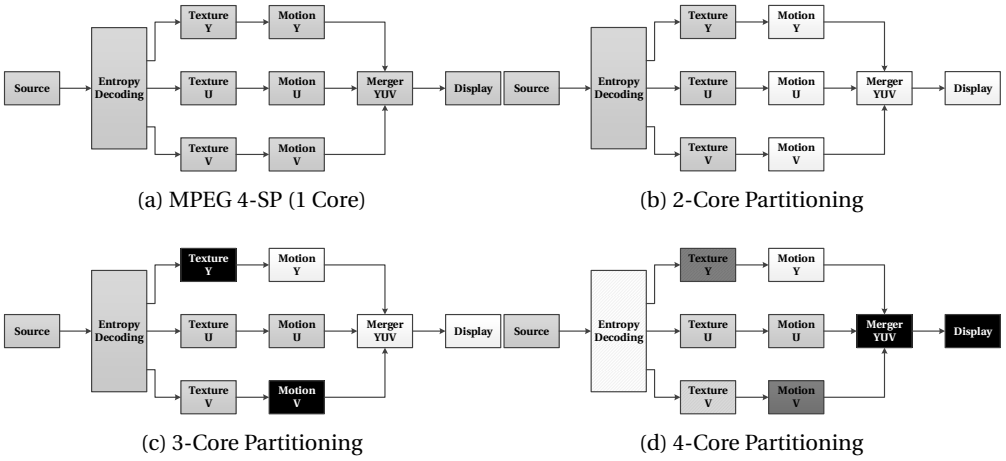
In [169] the authors present an Orcc backend that generates C code for Xilinx Vivado HLS high-level synthesis tools. The Table 4.10 is retrieved from [169] and illustrates the synthesis and simulation results of the RVC MPEG-4 SP video decoder presented on the previous section. As illustrated, Xronos has on the one hand a higher throughput, and on the other hand it uses much fewer resources. In addition, it is depicted that Xronos uses 77% of the BRAMs on the Virtex 4. This is due to the default buffers size (512) that writers of [169] have not modified. In general, this design requires buffers with smaller size than 512.Furthermore, Table 4.7 demonstrates that even if a different FPGA is used the resource requirements will not be that high. Moreover, Xronos has a speed up of 1.8 compared to the design synthesized by Vivado HLS. Here it should be noted that not all advantages of Vivado HLS have been exploited. Vivado HLS offers various optimizations such as pipelining, loop unrolling, memory partitioning and more advance Procedural optimizations than Xronos. However, they come at a cost. The developer needs to activate these optimizations by adding directives to the source code. However, over-optimizing a function does not guarantee that the overall design will have a higher throughput. In the next chapter, it is demonstrated that a design space exploration and a high-level synthesis tool which does not contain all optimizations that State-of-the-art C HLS have, makes it possible to achieve great results thanks to the model of computation of RVC-CAL.

Table 4.10 – Xronos versus Orcc C backend + Vivado HLS, synthesis and throughput results on Virtex 4 FPGA

	RVC-CAL Hardware Generators	
	Xronos	**Orcc HLS + Vivado HLS**
Slices	22823/67584(**42%**)	142302/67584(210%)
4 input LUTs	51898/135168(**38%**)	194583/135168(143%)
BRAMs	223/288(**77%**)	150/288(52%)
Throughput fps	**232**	125
Simulation Freq.	50 MHz	50 MHz

4.13.3 Multi-core performance on an embedded platform

The goal of this experiment is to validate the software synthesis for a multi-core embedded platform. In the literature, it is possible to find different implementations on the multicore using CAL [179, 180]. The experiment reported demonstrate the capabilities of the Xronos C++ backend in terms of speedup and throughput for an embedded platform. The execution is a Freescale P4080 platform, using a PowerPC e500 processor with 8 cores at 1.2GHz. Only a single executable is compiled and four mapping files, a partitioning from a single to four cores, are given. Foreman (QCIF, 300 frames, 200 kbps), crew (4CIF, 300 frames, 1 Mbps) and Stockholm (720p60, 604 frames, 20 Mbps) are the sequences used. Partitions are described on the Figure 4.26. The blocks represented by a striped background (Entropy Decoding, Texture Y, and Texture V) are distributed over different partitions.



(a) MPEG 4-SP (1 Core)

(b) 2-Core Partitioning

(c) 3-Core Partitioning

(d) 4-Core Partitioning

Figure 4.26 – The RVC MPEG 4 SP decoder and its partitioning from 1 to 4 cores.

Values of Table 4.11 are retrieved from [11] and they describe the frame-rate (in frame per second or fps) of the decoder from 1 to 4 cores and resulting speedup. Results reveals that it is possible to achieve a significant speedup when more cores are available.

Table 4.11 – Framerate of the RVC MPEG-4 SP decoder at QCIF, SD and HD resolutions.

platform	**resolution**	**framerate** (# of cores)			
		1	2	3	4
Freescale P4080	176x144	223	465	711	853
	704x576	15	30	43	52
	1280x720	5	9	13	18
Normalized Speed-Up		1	2.08	3.18	3.86

In term of speedup factor versus the single-core performance, results are of the same order of magnitude than the ones presented in [180]. In term of absolute throughput, this experiment provides a speedup of four compared to [180] for the P4080 when normalized at the same

frequency. It should be mentioned, that the partitioning is effectuated manually and the speed-up is almost linear. Although the P4080 has eight processing elements, using additional cores has a negative impact on the speed-up. Consequently, either there is a limitation on the communication between cores or the potential parallelism of the design is not higher than four.

4.13.4 Hardware and Software Co-Design on Heterogeneous platforms

The goal of this experimental part is to validate the portability of RVC-CAL programs to heterogeneous platforms. Therefore, additionally to the JPEG encoder described previously, a JPEG decoder was developed in RVC-CAL for creating a baseline profile JPEG codec represented in Figure 4.27. The codec is implemented on three different platforms made of FPGAs and embedded processors.

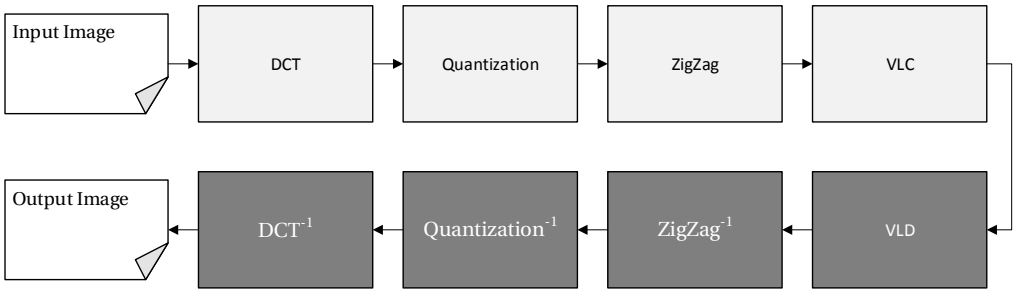


Figure 4.27 – JPEG codec functional units and the partitioning for the platforms.

The JPEG decoder is implemented on the software part whereas the JPEG encoder is implemented on the hardware part. As a software platform, the embedded low power board with a Freescale P4080 PowerPC CPU was chosen. The P4080 board contains an Ethernet port and a PCI-Express input connector. For the hardware part, two platforms were used. A proprietary Xilinx Spartan 3 FPGA with an Ethernet connection and a Xilinx University Program ML509 board that contains a Virtex-5 FPGA with an Ethernet port and a PCI-Express connector.

It should be noted that several partitioning configurations have been tested with success. That confirms that the portability objectives of the design flow are reached. In fact, a single actor can seamlessly be synthesized and mapped to general-purpose processors or FPGA. A partition of the RVC-CAL dataflow network can be swapped from a SW to an HW implementation and vice versa and from all yield functionally-equivalent implementations. To clarify, the results are given only for meaningful partitioning, separating encoding and the decoding processes. More precisely, the partitioning of the application consists of assigning the whole encoding process to the specialized processing element, while the host does the decoding process.

Results of the experiment are summarized in Table 4.12. Two media of communication have been tested. The results obtained using Virtex-5 FPGA, and the PCI-Express interface are

Table 4.12 – Framerate of the JPEG codec with a 512x512 video resolution on P4080 and two FPGA boards with 2 different interfaces.

	Ethernet	PCIe
Spartan3	4.3	N.A.
ML509 with Virtex5	4.6	10.2

comparable with [181] and [145]. On one hand, the encoder implemented on the Virtex-5 can encode around 4 Full HD frames per second at 80 MHz. On the other hand, the decoder on the P4080, can decode at 135 fps 512x512 frames. This result clearly indicates that either the interface bandwidth or the communication scheduling is a limit for the design performance.

4.14 Conclusion

This chapter presented Xronos, a high-level synthesis tool for dynamic dataflow programs expressed in RVC-CAL. As illustrated, Xronos is a set of different tools. It uses Orcc for parsing RVC-CAL programs and OpenForge for generating Verilog HDL. In addition, Xronos extends Orcc in order to act as a compiler. As discussed in Chapter 3, Orcc works as a pretty-printer which constructs from the RVC-CAL Abstract Syntax Tree a Procedural intermediate representation that fits the C code generation. Xronos, however, adds the missing compiler building blocks such as Control Flow Graph representation for procedures, compiler dataflow analysis such as Dominance Graph, Reaching Definitions, and Live Variable analysis. In addition, Xronos transforms the Procedural IR to a Pruned Single Static Assignment form. As a result, the Procedural IR is more compact, and all control and dataflow information for each Basic Block is available thanks to the Single Static Assignment and Live Variable analysis. To represent the actor execution model, Xronos constructs the *Action Selection* procedure.

Moreover, it was presented how Xronos produces an abstract Control-and-Dataflow Graph for each Procedure. This is because OpenForge Intermediate Representation represents Procedures as CDFG graphs with nodes as Components and edges as control and data dependencies. Additionally, the Language Intermediate Model (LIM), OpenForge Intermediate Representation, and the Components, the basic building block that represents an operation, were discussed. As presented, the LIM is rich and represents Modules, Branches, Loops and Memory access in an elegant way. Moreover, the Achilles heel of OpenForge and as a consequence Xronos one is the LIM operation scheduling was discussed. The missing information on operators combinatorial latencies produces ASAP schedules that might reduce the overall maximum synthesis frequency of the design. In fact, if an RVC-CAL design is modular the effect on the unconstrained scheduling disappears as it is observed by the Experimental results. Finally, it is presented how the Xronos CDFG is mapped onto the LIM one. Future works on OpenForge will consist of creating a library of FPGA families with their associated operator combinatorial latencies. Thus, new scheduling such as Scheduling of Different Constrained, as discussed in the Conclusion and Future Work Chapter, will reduce the current clock latencies and will

enable to apply constrained scheduling.

Furthermore, it was introduced how Xronos generates synthesizable SystemC code as an alternative to the OpenForge Verilog one. In fact, the available HLS for SystemC that was used, Vivado HLS from Xilinx, provided inconsistent results when Verilog or VHDL code was generated. Even though the generated Xronos SystemC code is functionally correct and the Vivado HLS post-synthesis SystemC code generation and simulation give the right output, the generated Verilog or VHDL code when simulated represented faulty results, and the output results were different for both of them. Future work should, therefore, consist in using different HLS such as Calypto Catapult or Cadence Cynthesizer.

Moreover, Xronos produces C++ code for embedded platforms. Even thought Orcc generates ANSI C code, prior work on interface synthesis was developed under C++. As a result, Xronos generic C++ code has been tested with x86 workstation machines and also embedded ARMs and PowerPC CPUs. The C++ generate code provides the necessary user space for driving interfaces such as Ethernet and PCI-Express. Experimental results have demonstrated that the current interface implementation lacks in efficiency.

Successively, experimental results presented a benchmark for streaming applications called Streambench. The behavioral synthesis was produced by Xronos and RTL synthesis results for a Xilinx FPGA were analyzed. As a matter of fact, Xronos generates quality code for modular RVC-CAL programs and synthesizes complex applications such as full MPEG video decoders. In addition, results and comparison with an alternative RVC-CAL to C for Vivado HLS backend were analyzed. As depicted, the results of Xronos compared to the synthesized generated C code by Vivado HLS are better in terms of throughput and resources. Moreover, a JPEG codec mapped to a heterogeneous platform proved that Xronos handles code generation for SW, HW and the interface synthesis between the processing elements. Results demonstrate that the interface bandwidth is not fully exploited, and the communication scheduling is limiting to the design performance. Hence, future work consists on providing a better implementation for the current supported interface but also to add support for AXI interface. All new SoCs from Xilinx and Altera handles the communication between the programmable logic and processing system (ARM) through AXI. Thus, is mandatory to provide such an interface with Xronos for a broader support of heterogeneous platforms. Finally, Xronos has been used for synthesizing even more complex video decoders such as the AVC and during the writing of this thesis for implementing an HEVC main profile video decoder description on a Xilinx Zynq 7045 SoC (double core ARM + FPGA).

5 Iterative Design Space Exploration for Xronos

5.1 Introduction

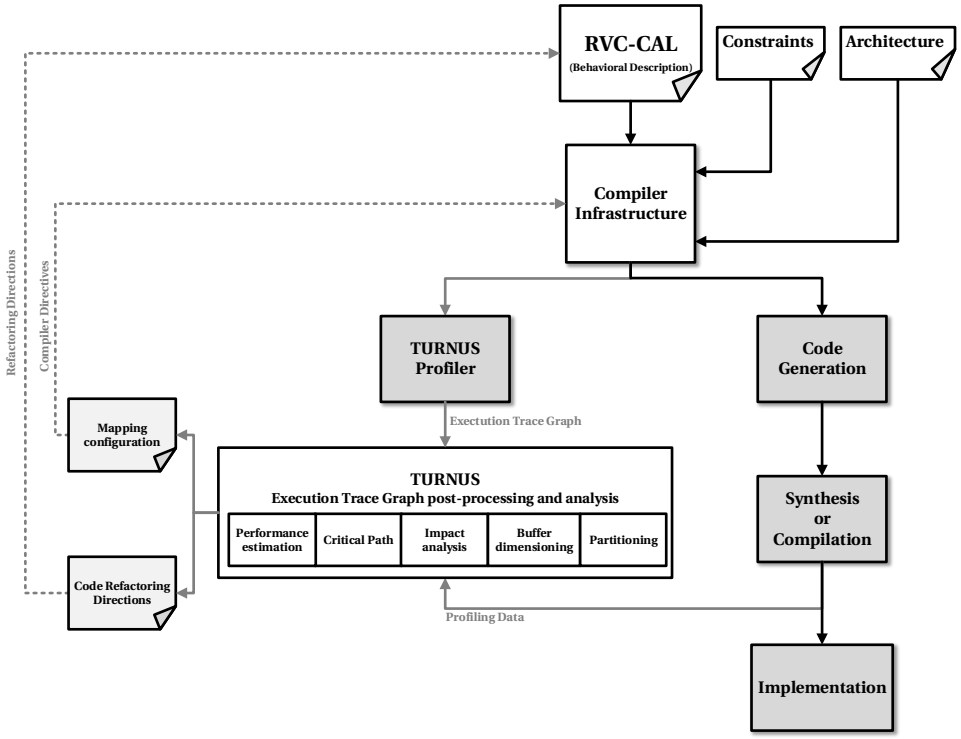


Figure 5.1 – Iterative Design Space Exploration on the RVC-CAL design flow by using TURNUS.

Complex software systems have many design points in terms of selection of software components and hardware architectures for implementation. These point choices create a large space of possible design solutions called the design space. The design process requires exploration of the design space in order to find valid design solutions before the actual implementation. The aim of Design Space Exploration (DSE) is to find design solutions that satisfy functional

performance *constraints* and/or *optimize* portions of the system (Figure 5.2). In addition, the heterogeneity of modern parallel architectures and the diverse requirements of target applications significantly complicates modern system designs. The development of efficient programs for this kind of platforms requires design methodologies that can deal with system complexity and flexibility. This has lead to the notion of *system level design*, where key roles are played by aspects such as high-level modeling and simulation, and separation of concerns. In this context, the exploration of the design space becomes an essential step for implementing applications to heterogeneous and parallel platforms. This is due to the combinatorial explosion of design options when dealing with multiple concurrent processing units. In order to ensure an efficient implementation and integration process, the design has to be sufficiently modular and portable, without the need of any or partially manual rewriting.

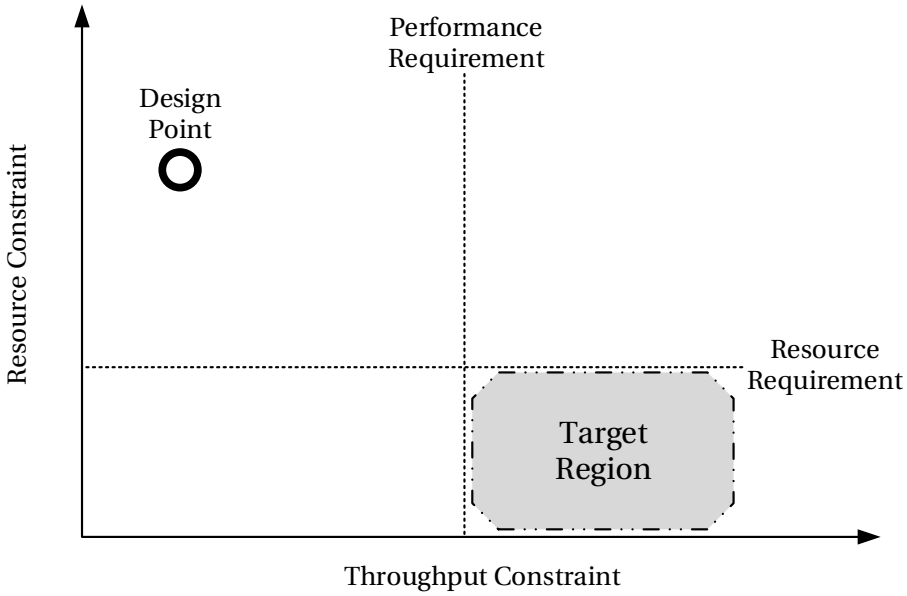


Figure 5.2 – Representation of the design space according to two constraints.

Most of the HLS tools, presented in the state-of-the-art Chapter 2, provides an estimation on clock cycles for a given function extracted from the operator scheduling. In a case that a parent function calls other inner functions, in the majority of the cases except if the developer chooses not to inline the inner functions, the estimated clock cycles is only given for the parent one. Thus, all information about the inner functions is lost. Even thought in Xronos inlining is also effectuated for each that calls CAL Procedures or Functions, but the hierarchy of Actions inside Actors is maintained and the clock cycles for execution those actions is estimated by the operator scheduler if possible or by RTL simulation.

Another performance estimation issue arises when multiple parallel C functions that communicate with each other are synthesized or even worse; some of them are executed in a different

processing element. How is possible to estimate which of these functions is the critical one? In one hand, for the hardware part current HLS tools will first synthesize these functions and will extract the Hardware critical path and either optimize or refactor the function with the lowest frequency. On the other hand, for the software part is possible to extract profiling information if the function is not nested, is a simple (i.e. a filter) and the interface of the function is fixed (i.e. arguments have fixed array size). Another important issue is the interface between hardware and software components. If the data traversing the interface are fixed and packed for optimizing the bandwidth, then the conditions are ideal, but it is not always the case. What happens when multiple connections are traversing the same interface? What should be the scheduling policy? And if a scheduling exist can it be measured efficiently?

All the above question are only partially answered by the state-of-the-art and up to the author's best knowledge, there is no tool that fully answers all the above questions. In this chapter an iterative design exploration tool, called TURNUS and integrated with Xronos, efficiently answers the first question. As for the other two questions, there are not yet answered due to the current interface limitations discussed in the previous Chapter. The main contribution of the author on TURNUS is the *performance estimation* described in Section 5.5.

TURNUS efficiently gives a solution to the first question because of the Dataflow MoC used in RVC-CAL. In fact, the provided hierarchy of RVC-CAL gives the possibility to extract fine-grain profiling information from actions and actors implemented in hardware components. The exploration methodology is the following. Once, the behavioral description has been verified by the compiler's (Orcc Interpreter) functional verification, the architecture target and the constraints on the design has been defined, then the compiling infrastructure generates the source code for the HW and SW parts. After that synthesis and/or compilation is effectuated. The first performance estimation is either retrieved by platform simulation or by run-time profiling of the design into a real target as explained in Section 4.12. If the implementation meets the design constraints, then the design goals have been achieved, else the design should be explored by TURNUS and refactored iteratively up until the constraints are satisfied.

The Iterative Design Exploration consists of: 1)the extraction of the Execution Trace Graph, 2) the low-level *Profiling data* extraction from the implemented design, 3) overall *Performance estimation*, 4) *Critical Path Evaluation*, 5) *Refactoring Directions* with Impact Analysis, 6) *Queue dimensioning* and 7) *Mapping* by post-processing if the target is a parallel platform or a heterogeneous one. In the following of this chapter each step is explained.

5.2 Profiling and Execution Trace Garph

The very first step of design space exploration is a functional high-level and platform independent profiled simulation [182, 7]. During this stage, an exhaustive analysis of the design under study is performed leading to the definition of its fundamental structure and complexity. This initial analysis enables multidimensional design space explorations and helps to find bottlenecks and potentially unexploited parallelism. In the literature, several methods have

been proposed to measure the complexity of an algorithm in terms of execution of its building blocks.

Two main axes are typically defined as: (a) the Computational Load (CL) and (b) the data transfer and storage load [183]. In this direction, the TURNUS profiler is used, whic extends the Orcc's Interpreter (see Section 3.6.3), evaluates each executed action for both (a) and (b). The "abstract" computational load is measured in terms of executed operators and control statements (i.e. comparison, logical, arithmetic and data movement instructions). Data transfers and storage loads are evaluated in terms of state variable, input/output port, queues utilization and tokens production/consumption. During a profiled simulation, all the executed actions with their dependencies are stored as a data set that adequately describes the program's behavior. The profiler will extract the following dependencies: (a) State Variable Read/Write and Write/Read, (b) Finite State Machine, (c) Guard Enable and Disable, (d) Port Read/Read and Write/Write, and (e) Token or FIFO dependencies. Detailed explanation of these definitions can be found in [7, 10].

As defined in [182, 7], the ETG is a multi directed acyclic graph $\mathbf{G}(\mathrm{V},\mathrm{E})$. Each single firing of an action $\tau \in \mathscr{T}$ is represented by a node $v_i \in \mathrm{V}$. Thus, the set $\mathrm{V}^\tau \subseteq \mathrm{V}$ contains all the firings of the action τ. Moreover, each single dependence between two fired actions is represented by a directed arc $e^n_{i,j} \equiv (v_i, v_j)_n \in \mathrm{E}$. The latter defines an execution order $v_i \prec v_j$, meaning that the execution of v_j depends on the execution of v_i. The previous considerations show that V can be considered as a partially ordered set of executed actions. Indeed, constructing consistent dependencies set E is fundamental in order to define constraints on the execution order between any couple of fired actions describing a platform-independent design behavior. The ETG needs to be built carefully to provide a solid basis that can produce quantified statements about a dataflow program execution. In fact, in the general case of a dynamic Dataflow network, such as the one expressed by the RVC-CAL dataflow language, the dependencies set can vary by changing the input stimulus. In other words, the explored states of a dynamic design can be dependent on the provided input. However, in the case of systems implementing several classes of signal processing applications (e.g. video or audio codecs, packet switching in communication networks), probabilistic approaches are a meaningful representation of the underlying processing model. Therefore, input sets that are sufficiently *large* with respect to the type of application can be used to generate statistically meaningful ETGs. Finally, the file format of the ETG is XML and it can be reused for further analysis.

5.3 Model of Architecture, Mapping and Constraints

The Model of Architecture or MoA is a formal representation of the operational semantics of networks of functional blocks describing architectures [184, 155]. Depending on the modeling perspective, MoAs can be classified as abstract or executable description [185]. Abstract models are used to represent performance symbolically. For example, they associate the required latency in clock cycles with each operation without executing any hardware description. On

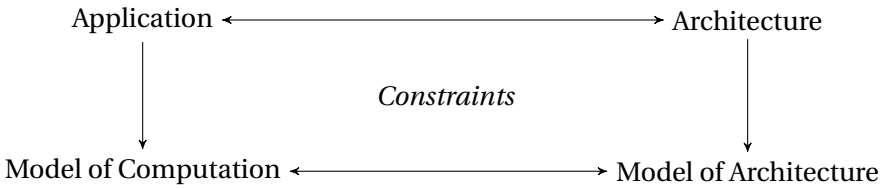


Figure 5.3 – Mapping from an application to an architecture. Constraints represent the feasible regions of the design space.

the other hand, executable specifications allows modeling precisely state-dependent behavior, such as the timing of caches and pipelines. In the context of this thesis, only abstract MoAs are analyzed as the ones proposed in [11, 186].

The mapping involves defining which part of the program is executed for a particular processing element, and which part of the communication structure is assigned to a particular medium. In the context of HW-SW co-Design, the problem to be solved is coordinating the design of the parts of the system that need to be implemented as SW and those as HW blocks [187]. The primary requirement is to avoid HW/SW integration problems that can arise when heterogeneous platforms are used. Hence, a set of **constraints** should be imposed and respected. Figure 5.3 depicts this process: the application is mapped onto a target architecture if the set of constraints is fully satisfied. Constraints can be defined in terms of data type [184, 155] (e.g. an application that makes use of floating points can be mapped only to an architecture that support this kind of operation) but also in terms of memory allocation, power consumption, and most important **latency**.

5.4 ETG Analysis

From the Execution Trace Graph of a dataflow program the Algorithmic Critical Path (CP), and the minimum queue size are estimated.

5.4.1 Critical Path Evaluation

Once the ETG has been post-processed, and the computation weights have been calculated as described in the previous steps, the calculation of CP length CP(x) using the linear time algorithm proposed in [7] is used. Once the CP is calculated the following informations are provided:

i A set of critical actors $\mathscr{T}_C$, actions $\mathscr{A}_C$ and queues $\mathscr{B}_C$ (i.e. that have at least one fired action or fifo dependencies along the CP)

ii The CP Participation of each action cpp_τ and actor cpp_a (defined as the overall computational load contribution to CP of each action and actor)

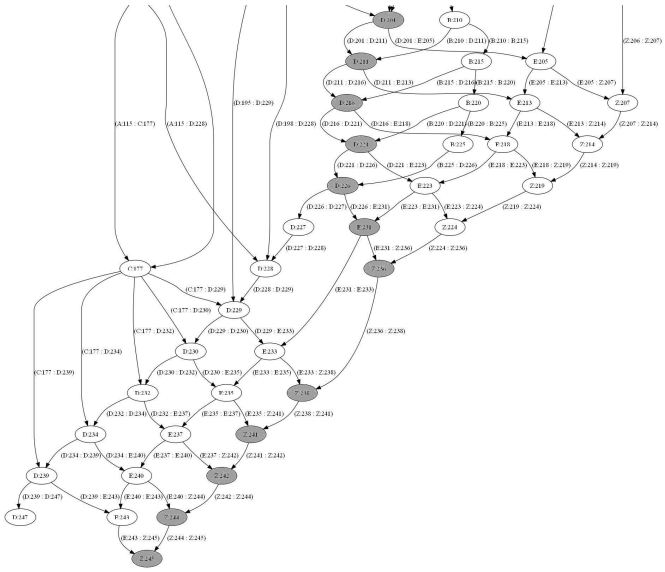


Figure 5.4 – Critical Path on partial execution trace graph.

iii The critical latency introduced by the scheduling or reading/writing tokens. All this information is used during the exploration stages by the underlying optimization algorithms such as the queue size optimization heuristic illustrated in Section 5.4.3

iv The maximum achievable design performances, from Equation5.4 with **unbouded** queue size configuration and when all actors executes in **parallel** i.e $\mathrm{cl}_{v_i}^{x_\sigma} = 0$ and $\mathrm{cl}_{v_i}^{x_\beta} = 0$. Moreover, the fact that all actors can run in parallel implies that no additional dependencies are added to the original ETG. The CP calculated in this way and represented as CP_∞. In the following, CP_∞ highlights the most serial and algorithmic-complex part of the design. Due to this, the corresponding weighted ETG contains only the minimal set of dependencies, and the weights that are only defined according to x_ρ (i.e. the action execution time).

5.4.2 Impact Analysis

In order to highlight the actions (or actors) that reduce the overall design performance, each node of the trace is assigned to a weight which corresponds to the CL required for executing the action. The CL is provided by Xronos and corresponds to the number of clock cycles needed for the action firing. The CP of the design is evaluated from this weighted ETG. A first metric provided by TURNUS is a ranked set of critical actions sorted by their CP Participation (CPP) value. The action that has the highest CPP value is said to be the *most critical action* [7], and is considered a target for optimization and refactoring. However, the reliability of this metric tends to decrease as the design becomes more parallel. In fact, for highly parallel designs reducing the CL of the most critical action does not necessarily cause a substantial reduction of the CP. As illustrated in [7], in this case more than one CP might exist. Thus, in order to obtain more robust guidance for refactoring the Impact Analysis is employed.

As explained in [7, 3, 188] the Impact Analysis exhibits the action λ which requires the less refactoring effort in order to maximally reduce the CP and consequently improve the overall throughput of the design.

5.4.3 Queue Size Minimization

The required memory size for a dataflow application consists of the sum of the actor's state variables, the local List of the actions, the program size and the queues that interconnect actors. Empirically, the size of the state variables and the local lists in actions can be reduced by applying Procedural IR optimization such as the one discussed in Section 4.2. However, in dynamic dataflow applications the queues can be minimized by having a finite overall set of **possible** input vectors (i.e reference sequence video streams that a video decoder should support). In a possible input vector set of dynamic dataflow program, it is possible to guarantee that the overall execution is deadlock free.

Thus, minimizing the total queue size can be a critical optimization objective in order to reduce the resource costs on modern FPGAs and many-core platforms that have limited memory. In the domain of SDF, CSDF designs, which are typically implemented on memory constrained hardware platforms, the queue minimization problem is a NP-complete scheduling problem [189]. Consequently, in a DPN application the design space exploration requires the use of heuristic algorithms when dimensioning the design queue size configuration. As reference in [7] the exploration process is split in two steps: (a) it evaluates a close to minimal deadlock-free queue size configuration; (b) successively it explores different queue size configurations.

If b_β is the queue size of $\beta \in \mathscr{B}$, the objective of the queue minimization problem is to find a deadlock-free solution for:

$$\begin{aligned} &\underset{\forall \beta \in \mathscr{B}}{\text{minimize}} \quad \sum \mathrm{b}_\beta \\ &\text{subject to} \quad \mathrm{b}_\beta \leq \mathrm{b}_\beta^{max}, \forall \beta \in \mathscr{B} \end{aligned} \tag{5.1}$$

where b_β^{max} defines the maximum queue size configuration imposed by the architecture. TURNUS computes a close-to-minimal configuration x_β^{min} using the execution trace-walk based algorithm proposed in [12] (i.e. suitable for DPN designs).

5.5 ETG Post-Processor

The Execution Trace Graph Post-Processor is an event-based simulator of the ETG. The Post-Processor permits the performance estimation and the queue size dimensioning for Dataflow program.

5.5.1 An event-based trace simulator

The ETG Post-processor is based on Adevs [190]. Adevs is a simulator for models described in terms of the Discrete Event System Specification (DEVS) [191]. The key feature of models described in DEVS is that their dynamic behavior is defined by events. An event is any change that is significant within the context of the model being developed.

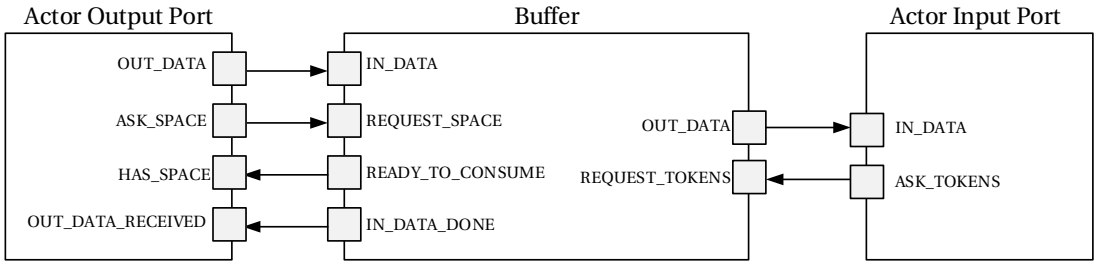


Figure 5.5 – Representation of Post-Processor Actor I/O and Buffer Model I/O events.

An Atomic DEVS model, as define in [191], is defined as a tuple $M = (X, Y, S, t_a, \delta_{ext}, \delta_{int}, \lambda)$

- X is the set of input events
- Y is the set of output events
- S is the set of sequential states (or also called the set of partial states)
- $t_a : S \rightarrow \mathbb{T}^{\infty}$ is the time advance function which is used to determine the lifespan of a state
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function which defines how an input event changes a state of the system
- $\delta_{int} : S \rightarrow S$ is the internal transition function which defines how a state of the system changes internally (i.e. when the elapsed time reaches the lifetime of the state)
- $\lambda : S \rightarrow Y^{\phi}$ is the output function where $Y^{\phi} = Y \cup \phi$ and $\phi \notin Y$ are a silent or an unobserved event. This function defines how a state of the system generates an output event (when the elapsed time reaches the lifetime of the state)

where $Q = \{(s, t_e) : s \in S, t_e \in (\mathbb{T} \cap [0, ta(s)])\}$ is the set of total states, t_e is the elapsed time since the last event, $\mathbb{T}^{\infty} = [0, \infty]$ defines the extended time base that is the set of non-negative real number plus infinity [191].

The DEVS model used in Post-Processor has three AtomicModel Objects. The first one `AtomicActor` represents the Actor DEVS Model, the second one `AtomicFifo` represents the FIFO DEVS Model, and the third one `AtomicPartition` represents the Mapping DEVS Model. To complete the DEVS model, an additional object called `PortValue`, that represents the value of an input or output event, is added.

AtomicActor

Each actor is modeled as an `AtomicActor` which describes a DEVS atomic model. As illustrated in Figure 5.8, each actor output port $p_i^{out} \in P_a^{out}$ is modeled with the following four `PortValue` elements:

- **`OUT_DATA`**: used for sending tokens to the connecting queue
- **`ASK_SPACE`**: used for sending the number of tokens that should be sent to the connecting queue
- **`HAS_SPACE`**: used for receiving an acknowledgment from the queue that the requested number of tokens is available on the connecting queue
- **`OUT_DATA_RECEIVED`**: used for receiving an acknowledgment from the connecting queue when it receives and successfully stores on its internal memory all the sent tokens

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

- **`IN_DATA`**: used for receiving the input tokens sent from the connecting queue
- **`ASK_TOKENS`**: used for sending the number of tokens that need to be consumed from the connecting queue

It must be noted that an input event is associated for each input port. Similarly, an output event is associated for each output port. The state transition system of this atomic model is depicted in Figure 5.6. This is composed by the following set of sequential states:

- **`BEGIN`**: the actor has been selected by the partition scheduler and select its next firing from the ETG
- **`IDLE`**: the actor has finished the post processing of its last firing and has not already been selected by the partition scheduler
- **`WAIT_READ`**: the actor is waiting the availability of input tokens from its input queues
- **`READ`**: the actor is consuming tokens from its input queues
- **`PROCESS`**: the actor is executing its algorithmic part
- **`READY_TO_SEND`**: the actor has finished the processing of its algorithmic part and is waiting that the output queues can accommodate the produced tokens
- **`SEND`**: the actor is sending the tokens to its output queues

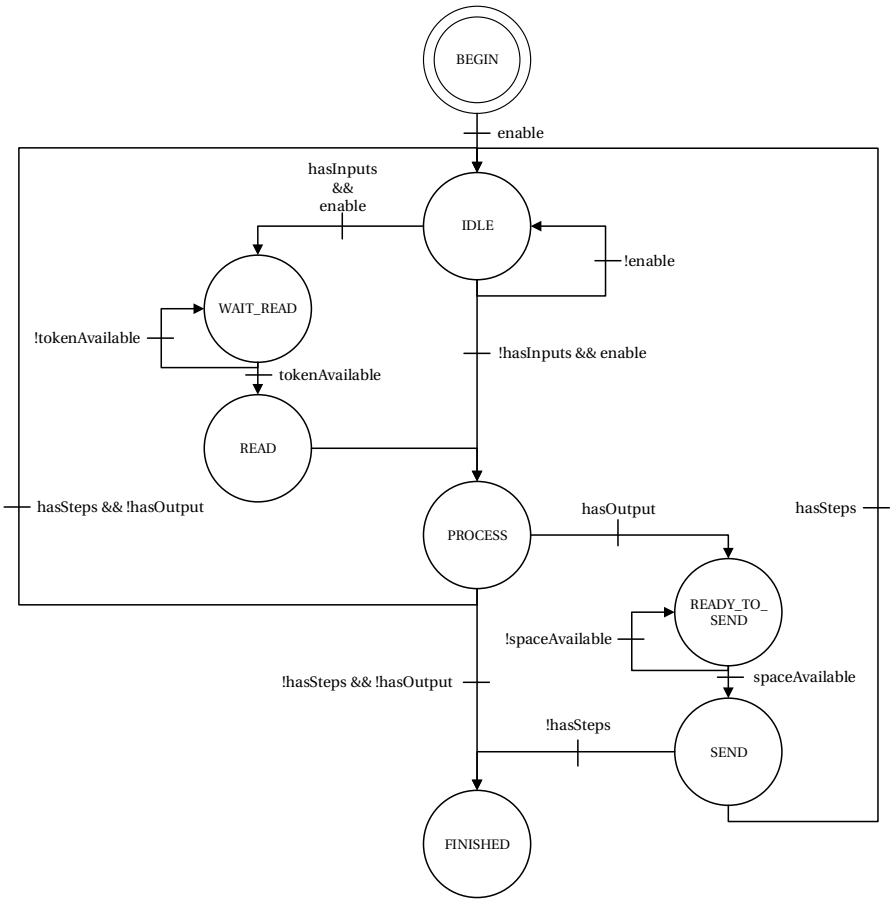


Figure 5.6 – Atomic Actor FSM.

- **`FINISHED`**: there are no more firings of this actor that need to be post processed

where the transition conditions are the following:

- **`enable`** : If the instantiated actor is enabled by an `AtomicPartition`.
- **`hasInputs`**: If the current step needs to read a `PortValue` from an input event.
- **`hasOutputs`**: If the current step needs to write a `PortValue` from an output event.
- **`tokenAvailable`**: If a `PortValue` is available from an input event.
- **`spaceAvailable`**: If there is enough space to sent a `PortValue`.
- **`hasSteps`**: If there are any steps V left that have not yet been simulated.

The post-processing of the `AtomicActor` begins only when an external event `enable` is processed by δ_{ext} function. It retrieves then the first step from the V list of the instantiated

actor. If a step contains an input port dependency it goes to the state $WAIT\_READ$, otherwise it goes directly to the $PROCESS$ state. Once the state $WAIT\_READ$ is reached, then the `AtomicActor` sends an output event $ASK\_TOKENS_{P_{in}}$ and simultaneously waits until it receives an input event $IN\_DATA_{P_{in}}$ (i.e. tokenAvailable is true). If tokenAvailable is true, it goes to state $READ$ with the `PortValue` being read and then the state changes to $PROCESS$. If the step has an output port dependency, the state changes to $READY\_TO\_SEND$, otherwise it changes either to $IDLE$ if there are still steps to be simulated or to $FINISHED$ as the simulation of this `AtomicActor` is terminated. However, if it goes to $READY\_TO\_SEND$ then the `AtomicActor` will first send an event to $ASK\_SPACE_{P_{out}}$ and it will then wait in the same state until it receives a $HAS\_SPACE_{P_{out}}$ (i.e. spaceAvailable is true). In the case that the spaceAvailable is true, it will either change to state $IDLE$ if there are still steps to be simulated or to $FINISHED$ as the simulation of this actor is terminated.

AtomicFifo

Each Fifo is modeled as an `AtomicBuffer` which describes a DEVS atomic model. As illustrated in Figure 5.8, each queue $b \in B$ is modeled with six `PortValue` elements. The following four `PortValue` elements models the connections with the source actor:

- **`IN_DATA`**: used to receive the tokens produced by the connecting source actor
- **`IN_REQUEST_SPACE`**: used to receive the number of tokens value that the connecting source actor want to store on the queue
- **`READY_TO_CONSUME`**: used to send the space availability acknowledgment to the connecting source actor that asked for storing the tokens
- **`IN_DATA_DONE`**: used to send an acknowledgment to the connecting source actor when all the received tokens have been processed and stored in the internal memory

Similarly, the following two `PortValue` elements models the connections with the target actor:

- **`OUT_DATA`**: used to send the tokens that need to be consumed by the connecting target actor
- **`REQUEST_TOKENS`**: used to receive the number of tokens value that the connecting target actor want to consume

It must be noted that an input event is associated for each input port. Similarly, an output event is associated for each output port. The state transition system of this atomic model is depicted in Figure 5.7. It can be seen how two independent transition systems are defines: one for receiving tokens (i.e. *Rx*), and one for transmitting tokens (i.e. *Tx*). The set of the *Rx* sequential states:

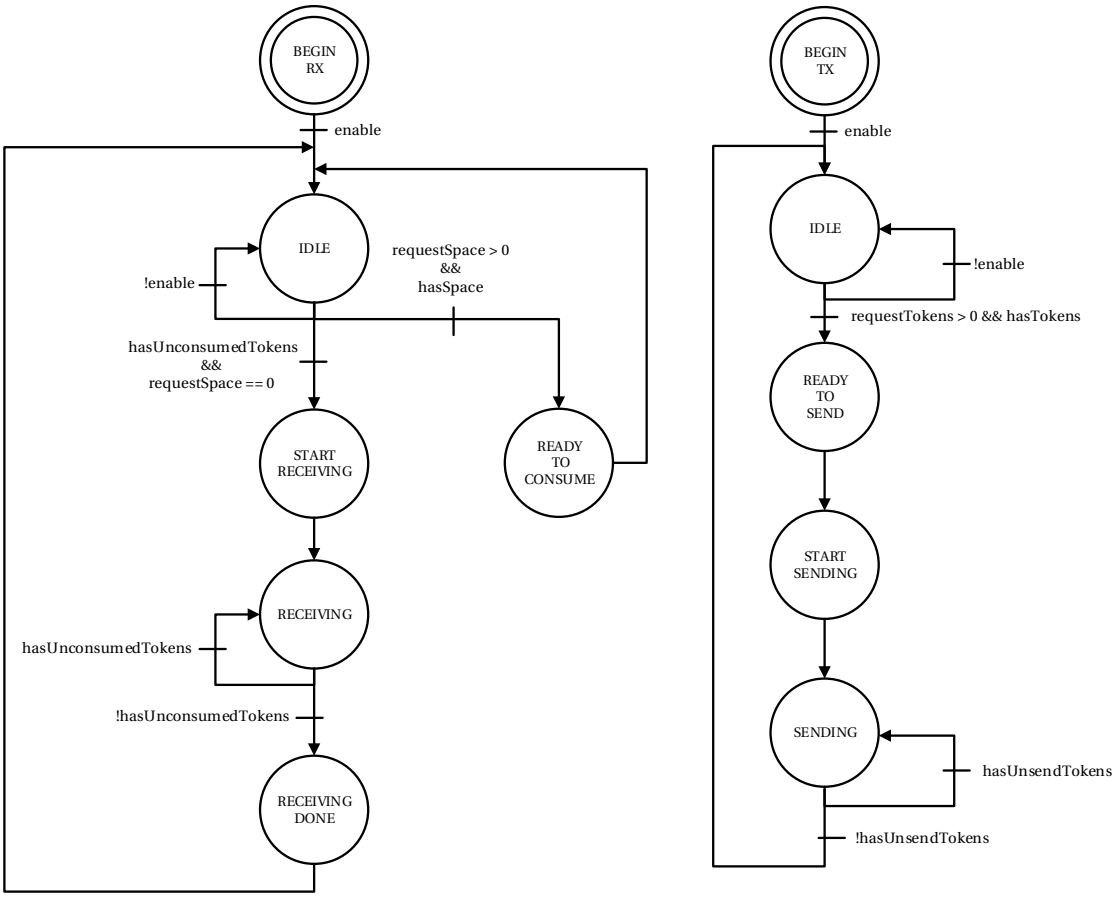


Figure 5.7 – Atomic Actor FSM.

- **`IDLE`**: the queue is waiting for some input tokens
- **`READY_TO_CONSUME`**: the queue is ready to receive the tokens produced by the actor
- **`START_RECEIVING`**: the queue starts receiving the tokens produced by the actor
- **`RECEIVING`**: the queue is receiving the tokens produced by the actor
- **`RECEIVING_DONE`**: the queue has just finished receiving tokens produced by the actor

where the transition conditions and variables are the following:

- **`enable`**: if the receiving part of the queue has been enabled by the `AtomicPartition` where it is mapped
- **`requestSpace`**: it contains the number of tokens places that the actor producer asked to accommodate on the queue
- **`hasSpace`**: if the queue can accommodate the tokens produced by the actor

- **hasUnconsumedTokens**: if there are tokens that the actor has send to the queue but that are not yet stored on the internal memory of the queue

Similarly, The set of the *Tx* sequential states:

- **IDLE**: the queue is waiting for a request to send tokens
- **READY_TO_SEND**: the queue is ready for sending tokens to the actor
- **START_SENDING**: the queue starts sending tokens to the actor
- **SENDING**: the queue is sending tokens to the actor

where the transition conditions and variables are the following:

- **enable**: if the transmission part of the queue has been enabled by the `AtomicPartition` where it is mapped
- **requestTokens**: it contains the number of tokens that the actor consumer asked to the queue
- **hasTokens**: if the queue has the number of tokens required by the actor
- **hasUnsendTokens**: if there are still some tokens that should be send to the actor

Mapping model

Each partition is modeled as an `AtomicPartition` which extends a DEVS atomic model. For each actor and queue partition, the scheduling policy is modeled by activating the corresponding object. For each actor and queue atomic models an **ENABLE** `PortValue` element is defined. Each actor has an `ENABLE` port, which is used by the partition scheduler to select the executable actor(s). Each queue has two `ENABLE` input ports: one for the input and one for the output. Those ports can be used asynchronously. Thus, it makes possible to model, for example, queues that are on the boundary of two actor partitions or queues that are used in a multi-clock domain architecture. As an example, Figure 5.8 illustrate the post-processing model of a simple network with three actors. In this case the `Producer` and `Filter` actor are partitioned on the same partition *PartitionA*, and the `Consumer` actor is partitioned on *PartitionB*. Each of those partitions has an actor and a queue scheduler. It must be noted how b_1 is modeled as a synchronous queue (i.e. input and output are activated at the same time), and contrarily b_2 is modeled as an asynchronous queue (i.e. the activation of the input and the output is decoupled).

5.5.2 Performance Estimation

For a design space configuration x, defined in Section 5.5.3, an associated design performance can be defined as a non-linear function:

$$\mathrm{T} = f(\mathrm{x}) \tag{5.2}$$

In order to minimize the exploration time and effort, the performance must be estimated using the design information provided by the application and architecture models (Section 5.3). In order to obtain reliable exploration results the following condition must be satisfied:

$$||\mathrm{T} - \widehat{\mathrm{T}}|| = ||f(\mathrm{x}) - \widehat{f}(\mathrm{x},\mathrm{p})|| \approx 0 \tag{5.3}$$

where $\widehat{\mathrm{T}}$ represents the estimated performance. Performances are evaluated using a platform model $\widehat{f}(\mathrm{x},\mathrm{p})$ that can be refined by enhanced profiling information p obtained as illustrated in Section 4.12. Given the event-driven Post-Processor simulation run, the execution time required for each fired action contained in the ETG is estimated according to the mapping configuration x. Thus, the performance is estimated at an x point. To this end, a weight w_{v_i} is assigned to each fired action $v_i \in \mathrm{V}$ of the ETG. This weight is the computational load of each fired action as defined in [7]:

$$\mathrm{cl}(\mathrm{x},\mathrm{p})_{v_i} = \mathrm{cl}_{v_i}^{x_\sigma} + \mathrm{cl}_{v_i}^{x_\rho} + \mathrm{cl}_{v_i}^{x_\beta} \tag{5.4}$$

Where

- $\mathrm{cl}_{v_i}^{x_\sigma}$: represents the time overhead introduced by the scheduler x_σ when scheduling v_i (i.e. *action selection*)
- $\mathrm{cl}_{v_i}^{x_\rho}$ represents the action computational time obtained with the mapping configuration x_ρ (i.e. *action execution*);
- $\mathrm{cl}_{v_i}^{x_\beta}$ represents the time overhead introduced by both queue reading and writing (i.e. *read/write delay*) which also contains additional latency due to a full output queue. In other words, this represents the time elapsed between the moment v_i becomes *schedulable* and the one when all its output queues can receive the produced tokens.

Queue Size Dimensioning

With a minimal queue size configuration, the latencies introduced by the queues (i.e. $\mathrm{cl}_{v_i}^{x_\beta}$ of Equation 5.4) are maximized. Thence, for a given mapping configuration $x_\rho^*, x_\sigma^* >$ the following relation arise:

$$\mathrm{CP}_\infty \leq \mathrm{CP}(x_\rho^*, x_\sigma^*, x_\beta) \leq \mathrm{CP}(x_\rho^*, x_\sigma^*, x_\beta^{min}) \tag{5.5}$$

The design space needs then to be explored in order to find a queue configuration x^*_β that both meets performance requirements and respects resources utilization constraints. The queue size dimensioning can be defined as a multi-objective minimization problem:

$$\begin{aligned} \underset{\forall \beta \in \mathscr{B}}{\text{minimize}} \quad & \begin{cases} \mathrm{CP}(x^*_\rho, x^*_\sigma, x_\beta) \\ \mathrm{B} = \sum \mathrm{b}_\beta \end{cases} \\ \text{subject to} \quad & \mathrm{B} \leq \mathrm{B}^{max} \\ & \mathrm{b}^{min}_\beta \leq \mathrm{b}_\beta \leq \mathrm{b}^{max}_\beta, \forall \beta \in \mathscr{B} \end{aligned} \tag{5.6}$$

The solution calculated in the TURNUS environment is obtained using the heuristic illustrated in [12]. The ETG is iteratively post-processed by trying different queue configurations. These are evaluated starting from x^{min}_β and at each iteration step the most critical queue is calculated as illustrated in Section 5.2. Moreover, its size is incremented by the maximum number of blocked tokens.

5.5.3 Mapping

The design mapping links each CAL application component (i.e. actors, actions, ports, and queues) to the corresponding architectural components (i.e. processing element, the communication element, interface).

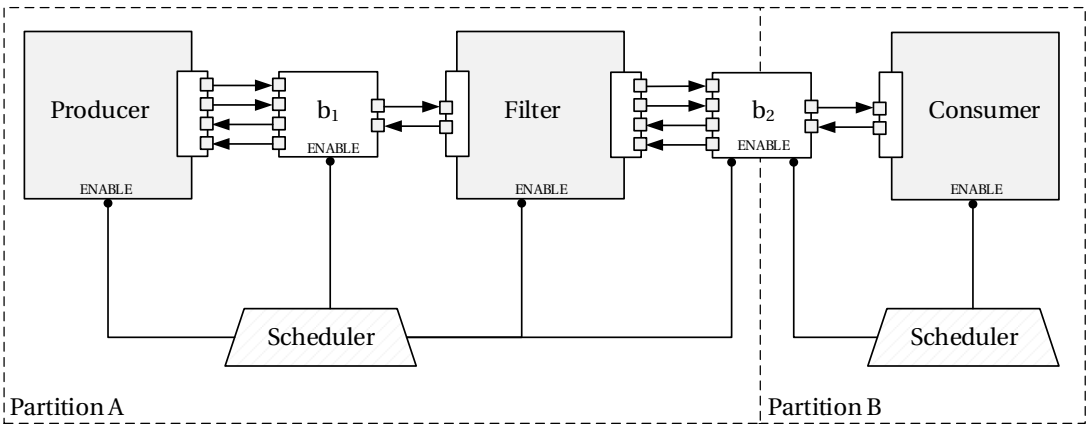


Figure 5.8 – Post-Processor Mapping of a heterogeneous platform.

The mapping is a file that is passed on the *Compiler Infrastructure* for defining which Actors and queues are mapped into a processing element defined in the architecture. In addition, further information such as queues and scheduling policies for each "software" processing element can be attributed into the corresponding object in the file. The mapping configuration defines then a *configuration* design space point that is provided into IDES for another iteration of exploration.

A mapping configuration, as defined in [7], can be represented as a 3-tuple:

$$\mathrm{x} =< x_\rho, x_\sigma, x_\beta > \tag{5.7}$$

where x_ρ, x_σ and x_β respectively define a particular partitioning, scheduling and queue configuration. Thence, the objective on the post-processing of the ETG is to find a configuration x^* that satisfies the input constraints.

5.6 Optimization by Design Refactoring in IDSE

The starting point of the exploration process is a simulation of the behavioral description. Once the ETG is constructed and the Critical Path and Impact Analysis are effectuated, a list of the most critical actions is provided and the first action that should be modified is given by the impact analysis.

5.6.1 Levels of parallelism

Parallelism is a form of computation in which many calculations are executed simultaneously. The parallelism, depending on the architecture, can be expressed in various types of which some can be applied automatically without the need of the developer's input. In the following the most used kinds of parallelism are presented:

- **Task-Level**: refers to the execution of a given task in a concurrent manner, where the task is partitioned across several parallel subtasks. In order for subtasks to execute in parallel, no precedence relations are permitted. If there are dependencies between the subtasks, it is called pipeline parallelism.
- **Data-Level**: refers to the execution of several similar tasks in a concurrent manner, where the task is replicated several times. In this case, input data is partitioned accordingly and sent to each of the replicated tasks.
- **Loop and Instruction-Level**: refers to the amount of parallelism available among instructions that can exploit parallelism among iterations of a loop. It is a form of data parallelism that is performed inside a procedure.

5.6.2 Complexity and issues of automating refactoring optimizations

The refactoring techniques for parallelism and memory optimizations involve significant modifications of the design architecture. For RVC-CAL programs, the flexibility provided by the model makes the task of automatically generating a parallel or a memory optimized architecture very difficult. Especially, when considering the number of design points that can be implemented. As a consequence, there is a high number of design styles, parameters, and

controls that could be applied in a Dataflow program. Some specific issues are summarized below:

- **Data and Task Parallelism**: One challenging part is to create automatically an actor for a partitioned/replicated action while considering all possible state variable dependencies. Moreover, it includes generating the relevant interfaces and control that will resolve the state variable dependencies. Finding a generic automatic technique for splitting actors for either data or task parallelism is considered a very difficult task.
- **Loop and Instruction parallelism**: are well-known parallelisms that compilers can handle with ease in certain cases. The difficulty in applying such optimizations is to find in which part of the program those optimizations should be effectuated to minimize the CL. In addition, if this portion of code is not in the algorithmic CP then the impact on throughput by optimizing this loop can be unimportant or even null.

The problem of automatically finding a program structure can be related to the Kolmogorov complexity [192] or descriptive complexity theory. The main problem is to find a minimum description for a given string of output values, given by $C_f(x) = min\{|p| : f(p) = x\}$ which aims to find the smallest $|p|$ in order to represent the output string x using a computable function f. Programs with an input specification, have an extension to a conditional Kolmogorov complexity with a Universal Turing Machine U [156]. So the conditional Kolmogorov complexity for U is given by $C_u(x/y) = min\{|p| : U(p,y) = x\}$, where the objective includes an input y. So if x can be described as a function, then the minimum p can be found. However, if x has the notion of being random or incompressible, x can not be expressed as a function anymore, and p can be proven to be incomputable [192]. Thus, the program structure p cannot be refactored automatically, and the developer should modify it manually.

5.6.3 A refactoring strategy using impact analysis

The starting point of the co-exploration process is a simulation of the design similar to the one used throughout the development process for the functional validation. Here, however, it is used to obtain a scheduler-independent execution trace [182], which will be used extensively by TURNUS during the analysis and optimization process. As discussed in 5.4.2, the Impact Analysis highlights the actions where the refactoring effort should be concentrated in order to reduce the overall CP length.

Figure 5.9 illustrates the iterative design space exploration methodology. At first, the developer provides the RVC-CAL program, the initial mapping and the throughput constraints. After that, a clock accurate simulation with large queue size (the queues should never be full) is performed, and the Execution Trace Graph is created. Then, the CP analysis is effectuated. At that stage, the CP analysis and the profiling data are handled to the Post-Processor. Thus, the first simulation of the ETG is carried out. From there on, the Impact Analysis is calculated,

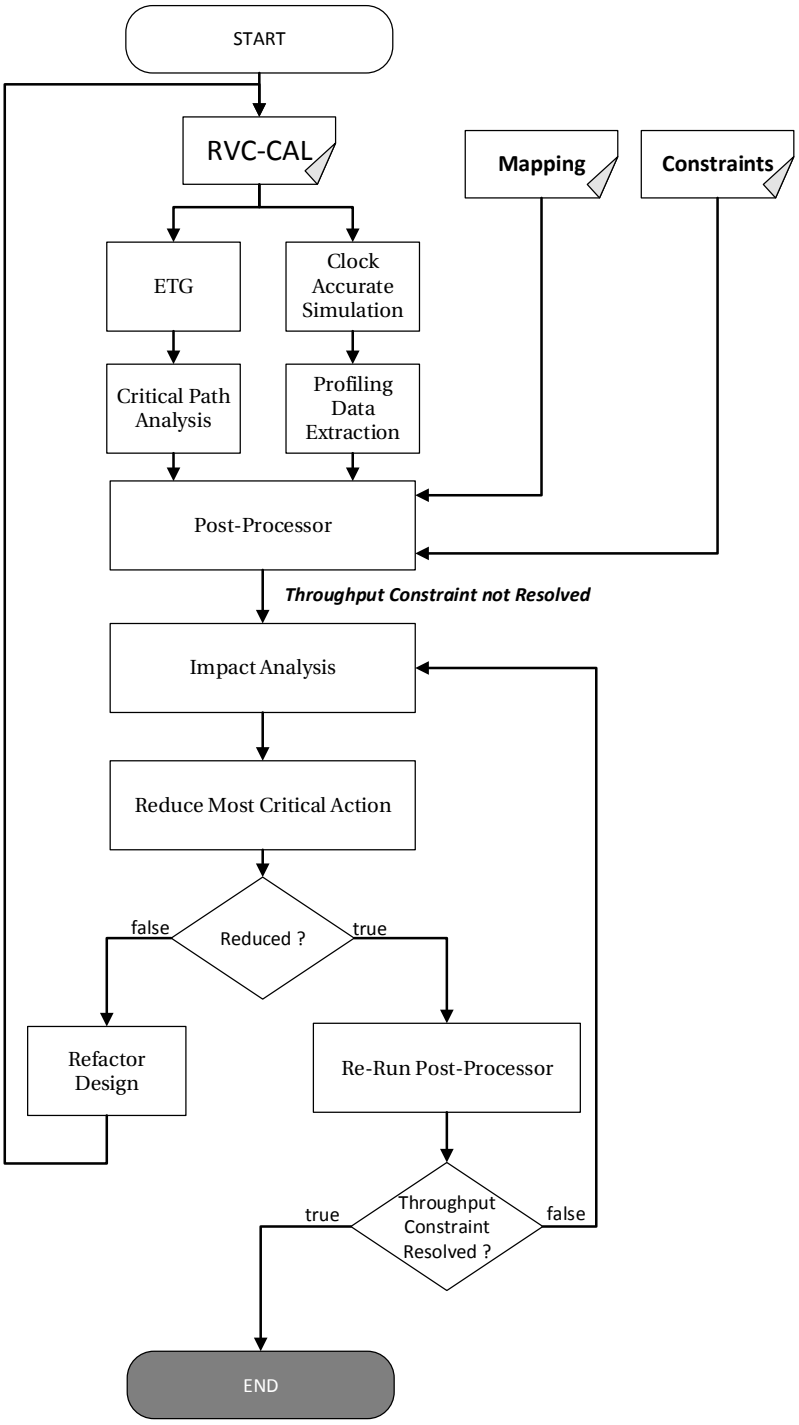


Figure 5.9 – Iterative Design Space Exploration methodology.

and the most critical action λ is highlighted. Therefore, the developer needs to reduce the CL of that action λ. If it is not possible to reduce it, then the design should be refactored, and the

process starts again. If action λ has been reduced, then only the Impact Analysis is relaunched n times until the throughput constraints are met. Finally, the Post-Processor is launched once more for retrieving the queue size dimensioning (Section 5.5.2).

5.7 Experimental Results

In this section, the analysis and implementation steps of a full RVC-CAL MPEG 4 SP decoder (ISO/IEC JTC1/SC29/WG11) are illustrated. The same RVC decoder is used as presented in Section 4.13.1. As the first step of the optimization process, the initial RVC-CAL reference code has been profiled with the TURNUS profiler. The purpose of this experiment was to optimize the decoder as much as possible during a single week.

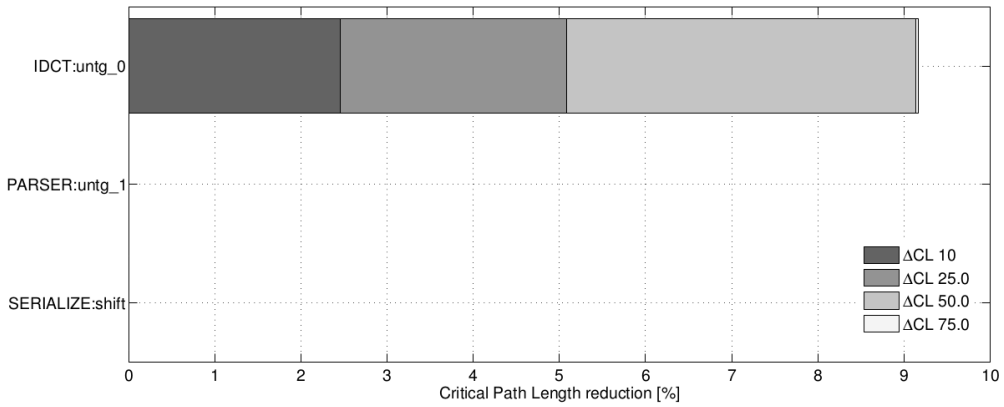
Table 5.1 – Initial Critical Actions Ranking. E%: number of executions of the action as a percentage of the total number of steps in the profiled run, CL%: computational load as a share of the total load, CPE%: the number of executions of that action on the critical path as a share of its length, CPP%: the share of the computational load of those executions on the critical path relative to its total load.

Actor	**Action**	**E$_{\%}$**	**CL$_{\%}$**	**CPE$_{\%}$**	**CPP$_{\%}$**
IDCT (TEX Y)	untg_0	0.17	20.75	78.99	99.92
PARSER (P)	untg_1	4.45	1.58	7.03	0.03
SERIALIZE (P)	shift	5.43	0.96	6.75	0.01
IQ (TEX Y)	ac	0.17	5.63	0.03	0.01
SPLITTER_420 (P)	splity	0.17	3.83	0.03	0.01
IS (TEX Y)	read_write	7.71	2.74	2.20	0.01
FBUFF (MOT Y)	untg_0	0.29	4.21	0.03	0.01

At the beginning, the ETG is extracted by the profiler and the weights for each action has been retrieved using the profiling information provided by Xronos. After that, the CP is calculated and the initial critical actions ranking list is shown in Table 5.1. The most critical action is the IDCT:untg_0: CPP is 99.92%. However, since this is a high parallel design, reducing only the CL of this action does not necessarily guarantee an improvement of the overall performance of the design.

According to Figure 5.10, where the results of the Impact Analysis are depicted, among all initial critical actions only the IDCT:untg_0 needs to be initially optimized. TURNUS estimates that by improving the performance of this action the overall CP length can be reduced by 9.5%. It is therefore clear that the following actions of the IDCT:untg_0, SPLITTER420:splity, MERGER420:read_y, FBUFF:untg_0, and DC_SPLIT:untg_0 should be refactored to increase the overall system throughput. Here must be mentioned that the action MERGER420:read_y does not appear in the initial critical actions ranking list. This is due to the fact that the design contains other parallel critical paths. From these results, the highlighted actions have been refactored by modifying the source code with the purpose to reduce their CL. The synthesis and simulation information obtained during the different refactoring stages are summarized in Table 5.2.

Figure 5.10 – TURNUS analysis results.



The Impact Analysis indicates that the IDCT actor is the first one to be modified. The original IDCT actor has a single action with an input port that read 64 tokens and produced 64 tokens in its output whose weight or latency is 634 clock cycles. To reduced the action's weight latency the pipeline optimization in Section 4.4 was used to decrease the latency by 40 %. An eight stage pipelining was generated by the optimizations with a latency of 235 clock cycles for a block of 64 tokens. After the design refactoring, the system was first synthesized with the new IDCT actors and then simulated. The first column on *Modifications* depicts that the number of LUTs was increased, which is normal due to additional queues and actors, and that the Speed Up was increased as expected by the Impact Analysis.

Table 5.2 – The modifications steps by most critical actor on the MPEG4 SP decoder. Synthesis results for Xilinx Virtex 4 FPGA.

		Modifications						
	Original	**IDCT**	**IQ**	**SPLITTER_420**	**MERGER_420**	**FBUFF**	**DC_SPLIT**	**IAP**
Slices	27658	31062	30143	29966	26756	26024	25389	24835
LUTs	50661	53394	51663	51329	45535	43994	42791	41779
Max Frequency (MHz)	49.9	49.9	49.9	49.9	50	50	50	85
Max fps	232	257	257	26	304	378	404	692
fps / Max Frequency	2.33	2.58	2.58	2.62	3.04	3.77	4.04	4.05
Speed Up	-	1.11	1.11	1.12	1.31	1.63	1.74	**2.98**

After the first modification, the ETG was extracted from the refactored design and the Impact Analysis was re-run. The next actor to be modified is the Merger 420. It should be noted that if the actions of SPLITTER_420 or IQ are refactored, because as indicated in Table 5.1 are the next with the higher CL, will not have an impact on the system throughput. To prove that the modifying action outside the scope of the Impact Analysis does not increase the system throughput, both actions were modified and as illustrated in the fourth and fifth column the same throughput was maintained. As expected, the optimization of those actions did not lead to a performance improvement. For both actors the input and output ports were not

changed. The only difference was that previous actions were consuming a set of 64 tokens and producing 64 tokens and were thus having a latency of 64 clock cycles for reading, 64 for processing and producing the tokens. As a result, the total latency was 128 clock cycles. The actors were modified so that they consume and produce a token at each firing with a latency of one clock cycle.

As indicated by the Impact Analysis, the MERGER_420 was the correct actor to modify. This actor reads serially four luminance blocks (256 pixels) and it restructures the pixels in a raster form. The read input latency for this actor is 256 clock cycles, after that the pixels are processed with nested loops for 256 clock cycles and 256 for producing the final raster form of the macroblock. This leads to a total latency of 768 clock cycles. The actor was optimized by adding more actions with the purpose to consume and produce a token as soon as possible. Six actions were created: 1) an action that read and produce directly a token, 2) an action that stores to a list of 56 elements, 3) an action that produces a token from the list of 56 elements, 4) an action that produces a token from the list of 56 elements and stocks the new block 56 elements 5) an action with a guard counter on 8 elements, and 6)an action that stops a counter on 56 elements. To summarize, at the beginning 8 elements are read and sent directly, and then 56 elements are stored, after that 8 lines again are read and sent directly, and finally 56 elements are stored. Hence, the first of the 128 elements are consumed. The next action to be fired is the one that produces a token from the memory and stocks a new one from the input. In this case the total weight for producing and consuming the 256 luminance pixels takes 368 clock cycles. Even though it is not the most optimal solution, the computation load for the actor is drastically reduced. As a consequence, the Speed-Up has increased by 31%. After that, the impact analysis is applied once again and the computation load of the untag_0 action of the FBUFF actor was reduced by rearranging the structure of the actor. As a matter of fact, that action was completely removed, and parts of it were ported to other actions. Thence it is providing an additional Speed-Up improvement of 32%.

Once again the structure of the decoder was changed. The ETG was once more regenerated and new profiling data information on the new actions was given by Xronos. The Impact Analysis indicated that the DC_SPILT untag_1 should be modified. As before, the latency was high due to the repeat CAL statement. The actor structure was modified, and the Speed-Up was increased by additional 11%. Finally, a last iteration on the IDES was conducted, and the impact analysis indicated that the CL action *copy* of actor IAP needed to be reduced. This action contains a division operator that limits the frequency of the decoder (50MHz the hardware critical path indicated by XST synthesis tool was in the copy `Task` Module of the actor DC_SPLIT). Thus, the division operation was replaced by a set of actions in which the finite state machine replaced the While statement of the division (see Section 4.3.7). The decoder had therefore a higher synthesis frequency of 85 MHz, which led to a final speed of 298%.

Finally, the buffers for the decoder were given by the methodology introduced in [12]. The decoder can be even further improved if a specific constraint on output is given, for example

giving a constraint on throuhput of decoding pictures at 720p30 as the MPEG-4 SP decoder handles in Section 4.13.1. This experiment was effectuated only for a week and a speed-up of **3** was achieved.

5.8 Conclusion

In this chapter, an iterative design space exploration methodology based on TURNUS and Xronos was presented. It was illustrated how TURNUS fits in the RVC-CAL design flow and which input information is needed by the tool. Most of C HLS tools, presented in the state-of-the-art Chapter 2, provide an estimation on clock cycles for a given function. However if this function calls other inner functions, the clock cycles estimation is given only for the parent one. Another performance estimation problem arises when multiple parallel C functions that communicate with each other are synthesized. How is it possible to estimate which of those C functions is the critical one? Current HLS will first synthesize these functions and will then extract the hardware's critical path. It will either optimize or re-factor the function with the slowest frequency. In Dataflow MoC however each actor executes a sequence of discrete computational steps called firings. Xronos can extract the dynamic profiling information for each firing in terms of clock cycles during RTL simulation or software execution for both hardware and software, as described in Section 4.12.

It has been proven that TURNUS answers efficiently the question raised on estimating the critical functions on a parallel programs. For doing so, TURNUS first extracts the Execution Trace Graph by simulating the RVC-CAL program with an input stimuli. Then, each step in the Execution Trace Graph is weighted by the profiling information provided by Xronos. Furthermore, a post-processing ETG scheduler provides an event-driven simulation of the execution trace graph. This event-driven simulator permits the developer to accelerate the refactoring of dataflow program when only the computation load, i.e. the reduction of the needed clock cycles for task, of an action has been changed. But also, it allows to estimate the overall performance of the design and to dimension properly the queues with a size that does not affect the overall throughput. Based on the Critical Path analysis of the Execution Trace Graph a refactoring strategy called Impact Analysis is used for reducing the refactoring iterations by illustrating the developer which action should be modified for achieving the design goals. Experimental results have demonstrated the usefulness of the iterative design space exploration by optimizing three times the throughput of an MPEG-4 SP video decoder only in a single week.

6 Power Optimization

6.1 Introduction

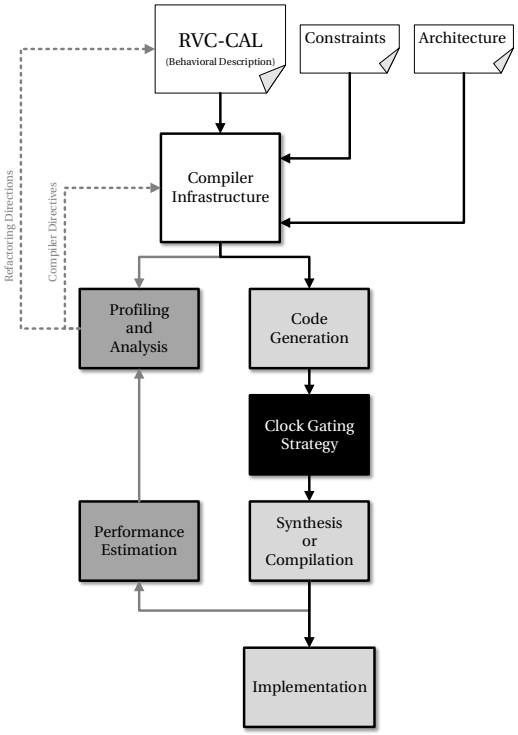
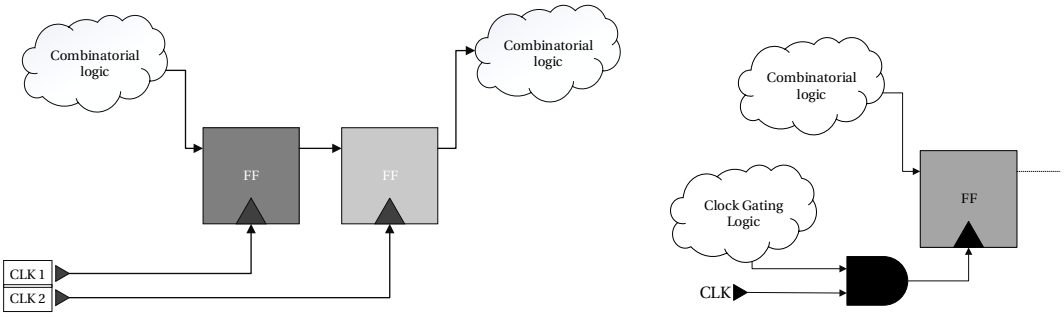


Figure 6.1 – Clock-Gating Strategy applied in the Design Flow.

Power dissipation is currently the major limitation of silicon computing devices. Reduced power consumption implies a lower need for cooling, greater longevity, longer battery life in the case of mobile devices and, of course, lower power costs. For these reasons power also frequently affects the choice of the computing platform right at the outset. For example, Field-Programmable Gate Arrays (FPGAs) incur a higher power consumption per logic unit

compared to an equivalent Application-Specific Integrated Circuit (ASIC) but often compare favorably to a conventional processor used for the same task.

For any silicon device, power dissipation can be broken down into two components: static and dynamic. Static power dissipation, also referred to quiescent or standby power consumption, is the result of the leakage current of the transistors, also affected by the ambient temperature. By contrast, dynamic power dissipation is caused by transistors being switched and charges being moved along wires. The power dissipation is roughly increasing linearly with circuit frequency since the most significant source of dynamic power consumption in CMOS circuits is the charge and discharge of transistors capacitance. To counteract this, ASIC designers have been using *clock gating* (CG) techniques in the last twenty years [193, 194, 195].



(a) Multi Clock Domain (b) Clock Gating

Figure 6.2 – Power Reduction Strategies.

Modern FPGAs have clock enable pins on the flip-flops in logic blocks. However, the use of such pins is not to provide switching activity reduction on the clock signal, and to yield power savings on the clock network [196]. FPGAs can provide gating at the top-level of the clock tree [197]. However, the top-level gating shuts down the entire clock network and, therefore, cannot be applied when some clock loads which are not used in gated form and others in gated form, or when there exist multiple enabled domains for a single clock signal. In addition, clock gating can be applied to large IP blocks such as DSP blocks and multipliers.

Different strategies for optimizing power consumption on ASICs and FPGAs were discussed in the state-of-the-art in Chapter 2.4.3. All of them describe the impact of a chosen technology or a given architecture, but they do not describe how to reduce power at the level of design abstraction. Adding power controllers at the behavioral description might reduce the portability of the code if a platform is changed during the development process. Moreover, it is difficult for HLSs that uses Imperative MoCs to apply power optimization that are extracted from the behavioral description. In contrary, dynamic Dataflow MoC such as the one used with RVC-CAL has two interesting properties that can be exploited for reducing the power consumption without modifying the behavioral description at all. Firstly, every actor is concurrent and has an input blocking read, and secondly every actor communicates with lossless FIFO queues. As a result, an actor may be stopped for a certain period if it is idle or its output

FIFO is full without impacting the overall throughput of the design. In addition, the more dynamic a dataflow program is, compared to synchronous dataflow, the more efficient the power reduction can be. This is due to the fact that synchronous dataflow programming always consumes and produce a fixed amount of data, and the queue size are fixed. Thus, synchronous dataflow design always consumes the same amount of power.

As a result, to reduce the power consumption of streaming applications, a coarse-grain clock-gating strategy was developed. This strategy consists of clock-gating an actor when the output queues of each actor are becoming full. The idea here is to select which local clock can be stopped when the circuit is in an idle state or when no output transition is effectuated. The clock-gating strategy aims at significantly reducing power consumption of streaming application designs based on asynchronous queued blocks. The approach, which is an elaboration of the ideas in [198], is based on controlling the top-level clock of the FPGA by using its clock buffers. Gating is coarse-grained because clocks are switched off for relatively *large* portions of the design. A similar effort targeting ASICs has been reported in [199]. Finally, the clock-gating strategy does not affect at all the Design Flow (see Figure 6.1), because it modifies just the actor composition (Network) by adding modules that control the clock for each actor.

6.2 Clock buffers on Xilinx FPGA's

In an ASIC design, the clock tree that distributes the clock to all clocked elements is individually routed for each different design. Therefore, there is the freedom of realizing clock trees with any logic circuits in the tree so that it is possible to gate only particular clocks or groups of clocks. In doing so, the clock tree is built to handle all delays related to the routing of the logic elements. FPGAs, on the other hand, have dedicated clock trees that are represented as nets and buffers that distribute the clock signals to all logic components of the chip. There is dedicated support for allowing a designer to divide a design into clock regions, to control the distribution of clock signals to these regions and their frequency, and also to shut them off completely.

For Xilinx FPGAs, the root of the clock tree is called global buffer. All global clocks are driven by the Xilinx primitive *BUFGCTRL*, those global clocks are located in the middle of the die. Each *BUFGCTRL* drives a vertical spine in the center of the die that runs from the bottom to the top of the die. This can been seen in figure 6.3. To use those clocks buffers the user needs to use the Xilinx primitive *BUFGCTRL* or its derivatives such as the *BUFGCE*. This primitive can be found on all latest Xilinx architectures. It represents a global clock buffer with a clock enable port and is used in the coarse-grain clock gating strategy. Finally, It allows the clock to turn on and off dynamically.

As a result, dynamic clock gating is enabled for power savings and the generation of decimated clocks by using the BUFGCE primitive. Furthermore, the "clock enable" signal is generated by the user's design. Moreover, the *BUFGCE* primitive can do a glitch and runt free clock gating if the clock enable is generated synchronously. In the 7 family, most of the FPGAs have 32 global

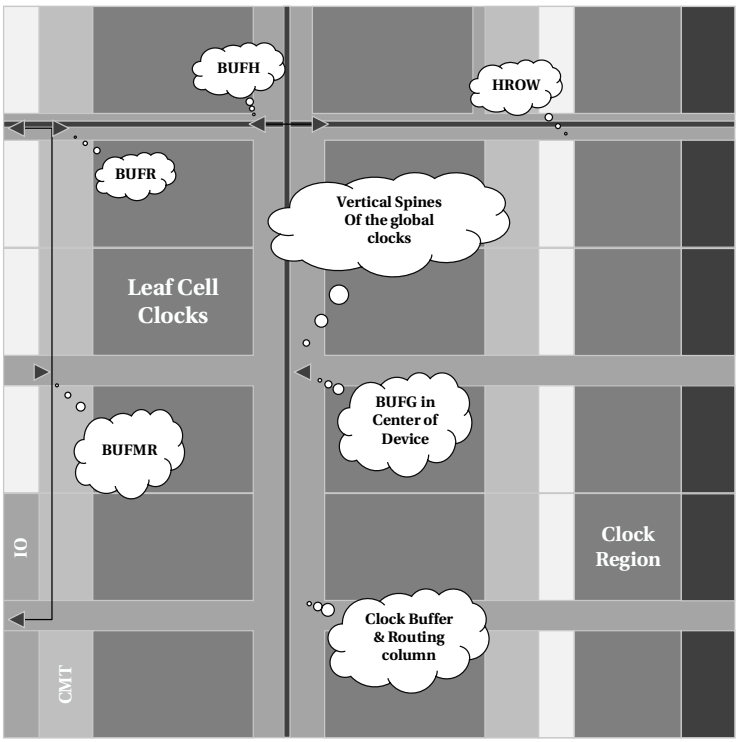


Figure 6.3 – View of an FPGA die, clocking trees and different clocking buffers found on a Xilinx 7 family.

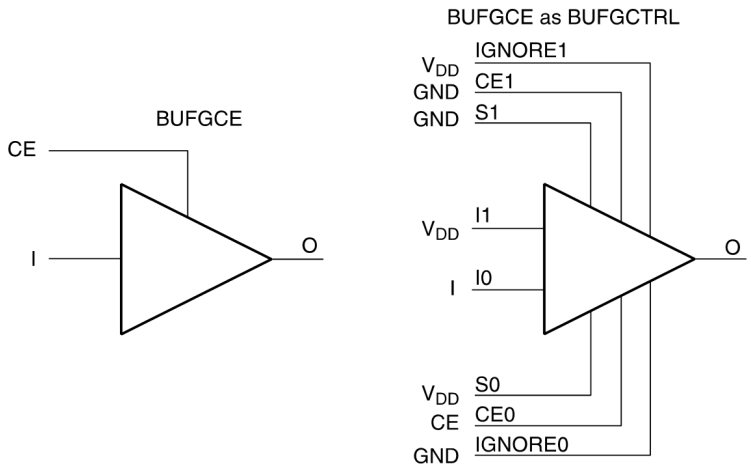


Figure 6.4 – Xilinx BUFGCE primitive for user clock gating.

clock networks, except of the smallest Artix-7 that has 16 and the largest Virtex-7 having 128 clock buffers.

In addition, there is also the *BUFH* clock buffer. Those horizontal cells are the connection

points between the vertical spines of the clock buffers and the horizontal clock nets that enter each clock region. They can be also driven with a clock enable primitive *BUFHCE*. These cells can gate the clock entering the clock region. There are 12 of these per clock region, but they have a strong restriction. All logic that uses such a gated clock must fit in one clock region. The clock buffers are not used by depicted strategies due to the previously described restriction. Furthermore, the tools are not capable enough to cut and fit portions of the design to certain clock regions. However, gating using any other resource, for example gating a clock with an LUT, the clock needs to leave the clock network and to be routed using general routing to an LUT and then routed back to another clock buffer such as BUFG/BUFH. As a consequence, this clock will arrive later than the other clocks on the clock tree.

In other words this routing will add an extra amount of delay which will make the design slow and is not recommended for FPGA design. Here should be mentioned that each flip-flop within the Xilinx FPGA has a clock enable pin. In some cases, it is possible to turn off the updating of some flip-flops when their value is not needed which can be done automatically by the tool if the "intelligent clock gating" is activated. More information on the intelligent clock gating can be found in [200] and for clock buffers on Xilinx 7 family in [197].

6.3 Coarse-Grain Clock Gating Strategy

This clock gating strategy works thanks to the Dataflow MoC. When the output buffers of the actors are full, the clock of those actors should be turned off. In any case, when the buffer are full the actor is idle. Thus, switching off its clock will not have an impact on throughput of the design. Even thought RVC-CAL is used as the behavioral description, this clock gating strategy is more general and can be applied to systems that represent the execution of a process that communicates with asynchronous FIFO buffers. The queues should be asynchronous for a lossless communication when an actor is clock gated, and when a design has different input clock domains.

The strategy consists in adding a *Clock Enabler* circuit for activating the clock of the Actors. This circuit contains a controller for each output port queue of the actor, a combinatorial logic for the configuration of the output ports and a clock buffer that enables the clock. A representation of an Actor with a single output port being clock gated is illustrated in Figure 6.5. As depicted, queues are asynchronous. Queues have two input clocks, one for consuming and one for producing tokens, and they have two output port **AF** for almost full and **F** for full. The actors input clock is connected to the output of the *Clock Enabler* circuit. As described in the previous Section 6.2, the clock buffer BUFGCE input clock should be connected with a Flip-Flop for having a glitch-free clock gating.

***Note**: The Flip-Flop will introduce a one-clock latency when the clock is switched off, but this additional clock cycle will not have an impact on actors that are on the critical path. Those actors are not being clock gated because the TURNUS dimensioning of the FIFO queues is based on the critical path analysis. Thus, not impacting the overall performance.*

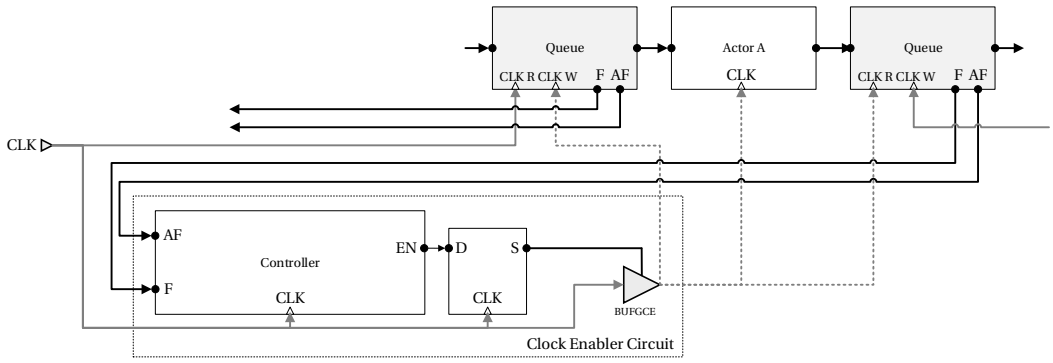


Figure 6.5 – Clock gating methodology strategy for Actor A with one output port. The `Clock Enabler` has as inputs the *Almost Full* and *Full* signal of each queue and a clock from a clock domain, and as a result it is going to activate or deactivate the clock of Actor A depending the FSM state of the controller.

6.3.1 Clock enabling controller

The clock enabling controller is represented in Figure 6.6. The controller is implemented as a finite state machine that has two inputs, *F* for full, *AF* for almost full and an output *EN* for enable. It is a finite state machine (FSM) with 5 states $S = \{INIT, SPACE, AFULL\_DISABLE, FULL, AFULL\_ENABLE\}$. The controller starts with the $INIT$ state and it maintains the *EN* output port at active high up until *F* and *AF* are at active low. The active high *EN* is maintained during the *SPACE* state, but once a queue is almost full then the state changes to $AFULL\_DISABLE$. In this state, the *EN* output passes to an active low. A conservative approach is taken in this state, this is due to the fact the *BUFGCE* is disabling the output clock on the high-to-low and that the clock enable entering the *BUFGCE* should be synchronous of the input clock. Once the queue becomes full, the controller maintains the *EN* at active low. When a token is consumed from the queue, the controller passes to the $AFULL\_ENABLE$ state, and it activates the clock. Then, depending whether the buffer becomes full, either full or almost full, it moves to the $FULL$ or $SPACE$ state respectively.

6.3.2 Clock Enabler Circuit

The user can choose a mapping configuration that indicates which actor should be clock gated. To do so, an attribute is given for each actor. If an actor has been selected to be clock gated then all of its output FIFO queues `A` and `AF` are connected to a clock enabler controller. Output queues can be connected through a fanout or directly to a queue. For the first case, the result of the controllers is connected to an AND logic port. This is a safe approach because as said previously, if one of the queues in the fanout is full then the fanout should give the command to the actor to not produce any token. For the second case, if an actor's output is connected directly to a queue without a fanout, the result should be connected to an OR logic port. This is due to the possibility that the next actor may need to consume a certain

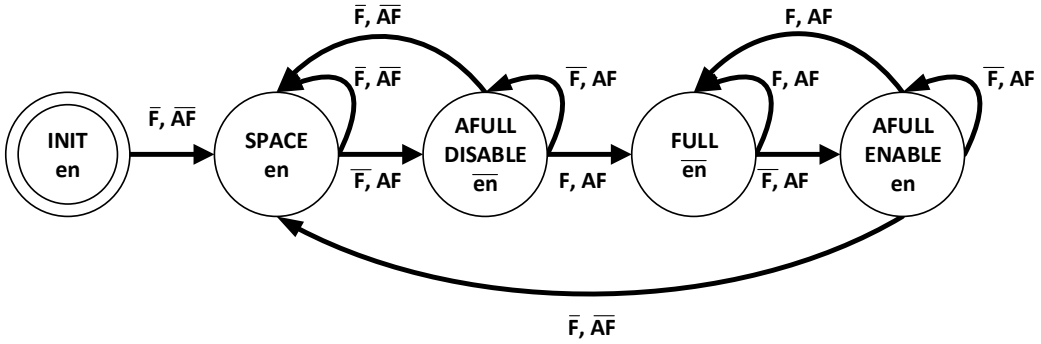
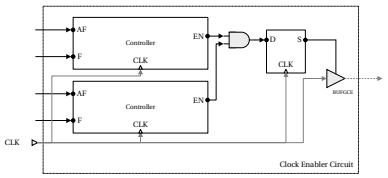


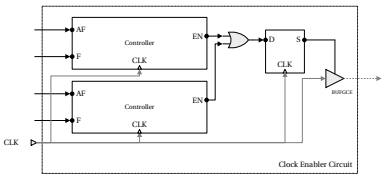
Figure 6.6 – State machine of the clock enabling controller. The controller has two inputs, **F** for full, **AF** for almost full and one output **en** as the enable signal.

number of tokens to output a token. As a consequence, it may lead the system to lock due to the unavailability of data. In the third case, if there is a combination of outputs with or without a fanout, then an n-input OR logic port is inserted. Figure 6.7 depicts the previous configurations.

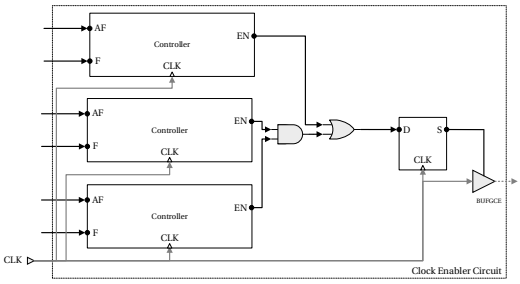
***Note**: AF is active high when there is only one space left on the FIFO queue.*



(a) Single Output Port with a Fanout



(b) Two Different Output Ports



(c) A single Output Port with a Fanout and another Ouput port

Figure 6.7 – Clock Enabler Circuit in three different configurations.

These different cases are being recognized automatically by Xronos, and the Verilog modules are being generated at the same time as the actors Verilog code. Eventually, a flip-flop is

Algorithm 11: Clock Enabler circuit module creation

Input : actor
Input : enable
Input : clk_in
Input : reset
Input : $\forall P^{out}_{almost\_full}$
Input : $\forall P^{out}_{full}$
Output : clk_out

```
1  module clock_enabler
2  for p in ∀P^out do
3      wire ["sizeof(p.fanout)":0] "nameof(p)"_enable;
4  reg clock_enable;
5  wire buf_enable;
6  for p in ∀P^out do
7      for idx in sizeof(p.fanout) do
8          controller c_"nameof(p)"_"idx"(
9          .almost_full("nameof(p)"_almost_full["idx"])
10         .full("port.name"_full["idx"]),
11         .enable("port.name"_enable["idx"]),
12         .clk(clk),
13         .reset(reset));
14 always @(posedge clk) being
15     clock_enable <= for p ∀P^out SEPARATOR "|" do
16         if sizeof(p.fanout) > 1 then
17             for idx in sizeof(p.fanout) SEPARATOR "&" do
18                 nameof(p)_enable["idx"]
19         else
20             nameof(p)_enable
21 assign buf_enable = en ? clock_enable : 1;
22 BUFGCE clock_enabling (.I(clk), .CE(buf_enable), .O(clk_out));
23 endmodule
```

connected between the BUFGCE and the final `OR` or `AND` port to have glitches and runt free clock enabling. The last output of the clock gating is a new clock that is connected to the actors, its fanouts, and its queues write clock (CLK B). The Verilog template of the clock Enabler circuit is given in Algorith 11.

6.4 Experimental Study

In this section, the clock gating strategy is evaluated by applying it to three designs. In [201] the reader might find a large variety of RVC-CAL applications that can be synthesized with Xronos and can test the proposed clock gating methodology. For all designs, the minimum buffer size was to obtain by using the minimization technique as presented in Section s:buffers. This information (an XML file containing all queue sizes) is given to Xronos so that it can generate the top HDL module of the designs with the estimated queue sizes. The design used for the following experiments are: JPEG encoder introduced on [2], Serial MP4 and MPEG4 SP decoder used as a case study on [146] and finally RVC Intra MPEG 4 SP decoder was used as a case study on [3].

For the experimental results, a Virtex 7 XC7VX485T-2 FPGA (VC707 Evaluation Kit) has been used. First the HDL code of the decoder has been generated by Xronos and synthesized with the Vivado synthesizer. After the synthesis, the place and route are applied to produce the final netlist. This netlist is then simulated with Modelsim so that it is possible to extract the switching activity information (SAIF file) of the design. Vivado Power analyzer tool is used to obtain the power results. This power analyzer takes as an input the design netlist, the design constraints and the simulation activity SAIF. It is to mention that all given results have a *high confidence level*, which means that at least 97% of the design nets are found in the SAIF file. Afterward, the activation rate for each clock-gated clock are extracted from the power analyzer as an XML file followed by Xronos generating a new top HDL module for the design.It should be mentioned that multiple XML files have been given to Xronos, for simulating the designs with a set of video sequences that have different image resolutions.

Table 6.1 reports the synthesis and the power consumption of the three designs. As it can be observed the designs are pretty small, all of them are less than 50k Slices. Compared to other designs, the JPEG encoder consumes more BRAMs than the video decoders because of its FIFO sizes. The RVC intra is the most demanding design on the number of slices. This is due to the parallel processing of luminance and chrominance, as the actors for the U and V chrominance are identical but slightly different for Y luminance. All designs have been given a clock constraint of 10ns clock period. However, the Serial MP4 decoder can handle a clock with higher frequency constraint [146].

Two configurations are compared for each design, `non CG` clock gating deactivated and `CG` clock gating activated one. Even though Table 6.1 does not represent the synthesis results for the `non CG` design, the difference in Slices is less than 2k slices for each `non CG` design and only one clock buffer is being used. In total every design consumes from 290 mW up to 400

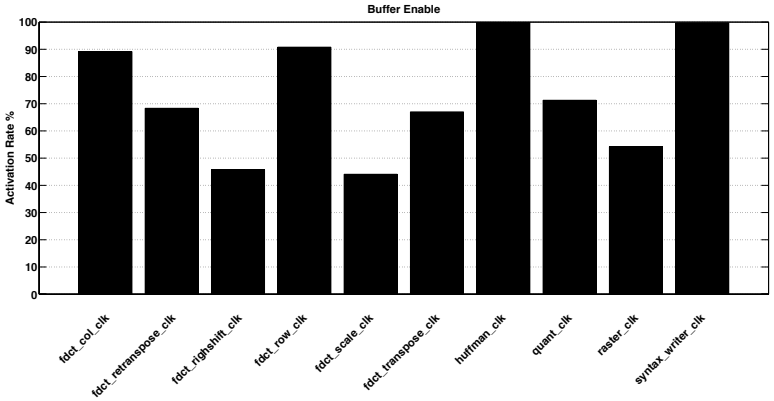
Table 6.1 – Synthesis and power results of three designs, a JPEG encoder, the RVC Intra MPEG 4 SP decoder and a full serial mpeg 4 sp decoder. The dynamic power reduction is given as the clock gated design over the non clock gated one.

Logic	**JPEG Encoder**		**Serial MP4**		**RVC Intra**	
Slices	19800		25497		42006	
BRAMs	35		7		18	
DSPs	5		7		18	
BUFGs	11		30		31	
Freq. MHz	100		100		100	
	Power (mW)					
	NON CG	CG	NON CG	CG	NON CG	CG
Clocks	53	46	108	97	159	123
Signals	10	10	22	14	17	15
Logic	8	7	15	10	14	12
BRAM	24	19	< 0.1	< 0.1	< 0.1	< 0.1
DSP	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1
I/O	1	1	4	4	2	2
Dynamic	96	83	149	125	192	152
Static	207	207	207	207	207	207
Total	303	290	357	334	399	357
	Dynamic Power Reduction %					
	13.54%		16.1%		20.8%	

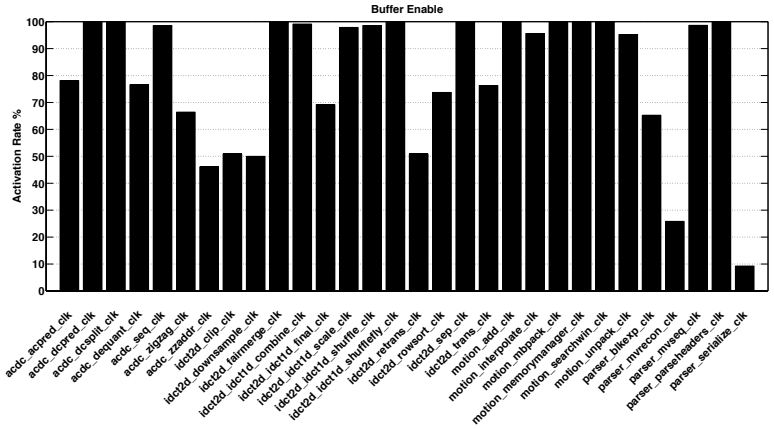
mW. The static power dissipation is the same for every design as all simulations were given the same configuration (e.g. temperature, clock frequency). The decoders BRAMs consume the same amount of power as the one found on the JPEG Encoder. Analyzing the results, the BRAMs power consumption in the JPEG encoder is almost one-third of the total dynamic power dissipation. A possible refactoring on the design to decrease the number of BRAMs will reduce significantly the JPEG encoder dynamic power.

Comparing the non-CG with the CG one, the highest power reduction comes from the clocks power consumption. The reduction in clocks goes as follows: for the JPEG encoder 13.2%, for the serial MP4 10% and the RVC Intra 22%. By comparing this values to the overall dynamic power consumption of Table 6.1 there are some differences. For the JPEG encoder and the RVC Intra decoder, the differences are slight. But for the serial MP4 there is a missing 6% that comes from the power consumed by *signals* and *logic*. This difference is due to the nature of the serial MP4 decoder that is composed of small actors that contain only one action executing in one clock cycle. As a result, it makes the design more parallel than the other two and more power hungry because those actors are always activated. Finally, the CG methodology reduces the power significantly on different streaming applications.

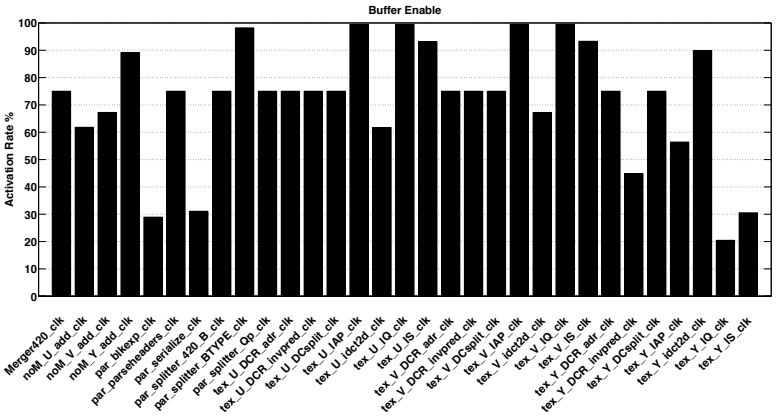
Figure 6.8 represents the activation rate of the CG clocks for the three designs. The image depicts that some clocks are always activated, 2 for the JPEG Encoder, 4 for the serial MP4 decoder and 10 for the RVC Intra. Those CG clocks should be removed from the designs as they consume BUFGCE clock buffers and more slices for the clock enabling controller. A future



(a) JPEG Encoder



(b) Serial MP4



(c) RVC Intra MPEG 4 SP

Figure 6.8 – Activation rates of each CG clock for each design with all their actors being clock gated. Average values retrieved from different QCIF input stimuli for all designs.

work of this methodology is to find the minimum activation rate of CG clocks that should be kept. The minimum activation rate for JPEG Encoder can be 90% or 75%, for the Serial

MP4 decoder 80%, and for Intra RVC decoder 75%. Achieving this minimum activation rate, it allows to save 4, 17 (17 clock buffers represent more than 50% of available clock buffer of a modern Xilinx FPGA), and 9 BUFGCEs respectively for each design.

6.5 Conclusion

This chapter presented a clock-gating strategy applied to dataflow designs, i.e. designs that consist of networks of computational kernels connected by FIFO queues. Even though the techniques are discussed in the context of dataflow, using a dataflow application to validate and evaluate it, it can be applied to processes that communicate with queues. Hence, it might conceivably be applied more broadly.

The results obtained are encouraging, the power saving strategy is achieved at the price of only small amounts of control logic without losing any throughput at all. Unsurprisingly, clock gating becomes particularly interesting in situations where the design is not used to full capacity, in which case it is a simple, automatic, and efficient way to recover power that would otherwise be lost in "idle" cycles. As a result, this technique is especially interesting in applications with dynamically varying performance requirements, where designing to a particular performance point is not an option, and where power is nonetheless at a premium.

Another strategy for reducing power dissipation is by using Multiple-Clock Domains. Authors in [202] presented an alternative solution for dynamic by using a design methodology to partition dynamic RVC-CAL dataflow applications on a multi-clock domain architecture without impacting the overall throughput performances. The results can not be directly compared with the proposed clock gating strategy because another FPGA was used, and the nominal frequency used is inferior to the one proposed in the previous results section. But, the total reduction percentage is higher with the presented coarse clock-gating strategy.

Future work in clock-gating consists in adding the condition of the idleness of an actor and the FIFO emptiness. The emptiness condition is not enough to clock gate an actor because it might be in a firing state. Thus, taking both conditions and the output fullness condition may further improve the overall power consumption. Another interesting direction is to combine the clock-gating strategy with multi-clock domains. Clocks that have a lower frequency than the nominal one might reduce the overall power consumption.

An important limitation of this strategy is the fixed clock tree of the FPGAs. Compared to ASICs the clock tree is fixed, and it provides a limited number of clock buffers. In addition, those clock buffers are separated in half for the upper and lower part of the FPGA. Thus, making it very difficult to create place and route constraints for an application that uses 16 clock buffers (most of the new Xilinx FPGAs have 32 clock buffers). Future FPGAs as the not yet available Xilinx Virtex UltraScale have more than 100 clock buffers, which makes the proposed clock gating strategy easier to implement.

7 Conclusion and Future Work

The research work described in this dissertation introduces significant contributions to the state-of-the-art of dataflow program high-level synthesis for hardware descriptions. Such contributions can be summarized as follows:

- The development of a design flow based on a compiler infrastructure called Xronos, in order to generate behavioral HDL code and C++ source code for embedded heterogeneous platforms.(Chapter 4)
- The development of the *Action Selection* Procedure that represents the actor execution model. This procedure not only reduces the resources but also increases the throughput of Actors for hardware synthesis compared to available alternative solutions in the state-of-the-art. In addition, by means of such procedure also advanced RVC-CAL statements such as "repeat" are now synthesizable.(Section 4.5).
- The development of a fine-grain profiling methodology at RTL level that create a direct correspondence between clock cycle accurate measures and abstract computation at dataflow program level. The structure of the generated RTL performing the profiling is achieved without impacting the throughput of the design (i.e. without side effects). In addition, the same methodology is applied to software code generation by using a library for retrieving the hardware counters and timers of different CPU families.(Section 4.12).
- The development of analysis and optimization strategies for dataflow programs hardware synthesis based on mapping clock cycle accurate measures onto structured (high level) dataflow computation representations. The methodology is called "Impact Analysis" and is integrated in the TURNUS design space exploration tool available as open source project.(Section 5.6)
- The development and implementation in the hardware synthesis stage of a clock-gating strategy applicable in principle to any dynamic dataflow program based design for reducing the circuit's dynamic power dissipation. The interest of the approach is that it is completely independent of the dataflow program semantics, thus can be applied

to whatever dataflow design and is also generalizable to designs not directly derived from dynamic dataflows computation models, but only based on asynchronous units communicating each others by means of queues. This strategy has a very low impact on resources and does not affect the throughput of the design.(Chapter 6)

7.1 Conclusion and Summary

The motivations that fostered this research work are driven by the typical problems that a system designer faces when develops applications implemented on heterogeneous platforms. The new dataflow design flow and associated tools were built with the purpose of supporting as much as possible automated tools assisted procedures along all the design steps, with the purpose of finding (close to) optimal design space points, clearly identify issues and objectives of manual designer intervention, thus with the purpose of reducing the design resources (time) for achieving quality implementation results on (massively) parallel heterogeneous platforms.

A fundamental issue when designing (complex) applications on heterogeneous platforms is the choice of the behavioral description. In Chapter 2 the state-of-the-art of high-level synthesis and design flows for heterogeneous platforms was critically revised. Most of the High-Level Synthesis tools uses C-like languages which presents several drawbacks. By taking such an imperative language, they encourage the designer to develop algorithms using a sequential paradigm and in general non analyzable operators and constructs, with the promise that the HLS tool provides all technologies that will automatically restructure the program and expose the potential parallelism of a sequential code. Unfortunately, such technologies are limited and the designer is confronted with the necessity to either refactor the code or to manually add code annotations (i.e. pragmas) or use additional ad-hoc constructs (threads, semaphores, etc.). In fact, concurrency must be in general derived by specifically considering the semantic of the program and is either supported by libraries or user inserted annotations. Thus, the resulted code is in general not portable or scalable to another design flows or even to other embedded SW platforms. Moreover, there is absolutely no guarantee that the code transformations/optimizations does not add undesired behaviors when executing concurrently (bugs!), that were not present in the formally correct sequential code.

In Chapter 3, a solution to the difficulties mentioned above is provided by the RVC-CAL programming language. As described in Chapter 3, the CAL and its Dataflow MoC have the property to express an application as network processes. Besides being inherently concurrent and modular, CAL offers parallelism scalability, no shared memory between processes, communication with queues, state encapsulation, execution of a sequence of steps cadenced by a finite state machine by each process and bitwise types. All previous properties lead to the portability of CAL which makes it a candidate for unifying the system's level design for heterogeneous platforms. It was also mentioned that a subset of CAL has been standardized by the MPEG committee with the purpose to provide the reference software of current and future video decoding standards. As a matter of fact, CAL or RVC-CAL applications are portable to

hardware and software processing elements as demonstrated in Section 4.13 and thanks to the work provided with this thesis.

To support the full subset of RVC-CAL for high-level synthesis, Xronos HLS was developed. In Chapter 4, Xronos is presented and analyzed. Prior work on HLS of CAL programs offered only a restricted synthesis of the subset of the language. Xronos on the other hand was built from scratch to circumvent these limitations. Previous toolchains used a set of XSLT transformations for transforming the Intermediate Representation of an Actor to a close to hardware one. Thus, the time of code generation was effectuated in tens of minutes or sometimes even in hours in case of complex actors. With the developments of this research work included in Xronos, the compiler operations and all the transformations are all executed using Java. As a result, the time for the behavioral synthesis stage of the actors that constitute a dataflow design has been reduced by a factor ranging between two or three order of magnitude. This make the synthesis and validation of manual refactoring practically interactive also for very complex designs. Thanks to the Procedural IR, procedures with arguments, generator statements or foreach statement are naturally supported by the Xronos RVC-CAL front-end called Orcc. Despite the advancement offered by the Orcc's Dataflow and the Procedural IR further actions were taken to optimize, reduce and transform these representations for hardware synthesis as discussed in the Chapter 4. One of the most important transformation is the Pruned Single Static Assignment form. This form makes an extraction of the Control and Dataflow Graph of a procedure possible and thanks to its pruned form the number of the used local variables and data dependencies between Basic Blocks is minimal. Thus, the number of wires and registers in behavioral synthesis is also minimized. Furthermore, it was discussed how Xronos supports bit accurate operations by casting instructions in the Procedural IR. Xronos has extended the state-of-the-art on HLS of RVC-CAL Programs by creating an Action Selection Procedure that defines the actor execution models for hardware synthesis. Xronos' Action Selection regroups all the firing conditions and optimizes the memory accesses in it. Thus, if an actor does not contain multiple guards on the same list state variable or a guard with a modulo or division operator, then the synthesized Action Selection Task guarantees that the following firing step is scheduled in the following clock cycle if the input rules are satisfied. In addition, the Action Selection permits the parallel reading and writing of list inputs and outputs of the "repeat" CAL statement and hence accelerates the consumption and production of tokens. Experimental results in Section 4.13.2 have demonstrated that the Action Selection not only reduces the resources but also increases the throughput compared to other methodologies.

For achieving better QoR for the generated behavioral HDL code more advanced and sophisticated Procedural IR optimizations should be applied. Commercial HLS tools, provides to some extent more advanced procedural optimizations than Xronos. Hence, those tools are the result of hundreds or even thousands man-years of engineering work compared to Xronos. Even though, such Procedural optimizations are not yet implemented in Xronos, because they would requires considerable engineering efforts beyond what available in this PhD work, the RVC-CAL model of computation abstraction offers several features which enable in some cases to outperform HLS results of typical C flows. This observation provides the motivation

for planning future work for introducing some fundamental procedural IR optimizations, as described in the next Section. For instance, a more sophisticated scheduling strategy could easily increase the RTL synthesis frequency of the design and will permit adding constraints on dataflow actions. As a result, the gap (or the advantage) in some performance metrics between the dataflow based design commercial HLSs and Xronos will decrease (or increase) even further.

In addition, Xronos also produces C++ code for embedded platforms and therefore unifies system level designs for heterogeneous platforms. To ensure the communication between hardware and software processing elements, Xronos provides the necessary user space drivers for driving interfaces such as Ethernet and PCI-Express. Experimental results have demonstrated that the current interface implementation lacks in efficiency. Future work, could provide a better performing implementation for the current supported interfaces and also of adding support for the AXI interface. All new SoCs from Xilinx, and Altera handle the communication between the programmable logic and processing system (ARM) through AXI. Thus, it is appropriate to provide such an interface with Xronos for a broader support of heterogeneous platforms.

So far the design flow presented in this thesis proved that a RVC-CAL dataflow programming language is an attractive alternative for heterogeneous platforms. To complete the design flow an iterative design space exploration was added, and presented in Chapter 5 to circumvent the limitations on the performance estimation of HLSs. Most of C HLS tools, presented in the state-of-the-art Chapter 2, provide an estimation on clock cycles for a given function, but if this function calls other inner functions, the clock cycles estimation is given only for the parent one. Another performance estimation problem arises when multiple parallel C functions that communicate with each other are synthesized. How is it possible to estimate which of those C functions is the critical one? Current HLS will first synthesize these functions and will extract the hardware critical path and either optimize or re-factor the function with the slowest frequency. As introduced in Chapter 3, in Dataflow MoC each actor executes a sequence of discrete computational steps called firings. Xronos, for both the hardware and software can extract the dynamic profiling information for each firing in terms of clock cycles during RTL simulation or software execution, as described in Section 4.12. As presented in Chapter 5, the Execution Trace Graph is first extracted by simulating the RVC-CAL program with an input stimuli. Then, each step in the Execution Trace Graph is weighted by the profiling information provided by Xronos. A post-processing ETG scheduler provides afterwards an event-driven simulation of the execution trace graph. This event-driven simulator permits the developer to accelerate the refactoring of dataflow program when only the computation load, i.e. the needed clock cycles for task is reduced, of an action has been changed. But also, it allows estimating the overall performance of the design and dimensioning properly the queues with a size that does not affect the overall throughput. Moreover, based on the critical path analysis of the Execution Trace Graph a refactoring strategy called Impact Analysis is used for reducing the refactoring iterations by instructing the developer which action should be modified for achieving the design goals. Experimental results demonstrate the usefulness of the iterative

design space exploration by optimizing three times the throughput of an MPEG-4 SP video decoder only in a single week.

Finally, this thesis extends even more the state-of-the-art and the presented design flow by providing a clock-gating strategy for process networks that communicate with queues (see Chapter 6). Even though the strategy is discussed in the context of RVC-CAL, using a dataflow program to validate and evaluate it, there is really nothing about it that is particular to this kind of MoC, so it might conceivably be applied more broadly. The results obtained are encouraging, the power saving strategy is achieved at the price of only small amounts of control logic without losing any throughput. Unsurprisingly, clock gating becomes particularly interesting in situations where the design is not used to full capacity, in which case it is a simple, automatic, and efficient way to recover power that would otherwise be lost in "idle" cycles. As a result, this technique is especially interesting in applications with dynamically varying performance requirements, where designing to particular performance points is not an option, and where power is nonetheless at a premium. An important limitation of this strategy is the fixed clock tree of the FPGAs. Compared to ASICs the clock tree is fixed, and it provides a limited number of clock buffers. In addition, those clock buffers are separated in half for the upper and lower part of the FPGA. This makes it very difficult to create the "place and route" constraints for an application that uses 16 clock buffers (most of the new Xilinx FPGAs have 32 clock buffers). Future FPGA generations will offer more than 100 clock buffers, which makes the proposed clock gating strategy easier to implement and useful in power restricted implementations.

7.2 Future Work

The proposed design flow has demonstrated its efficiency for reducing the dynamic power dissipation in high-level synthesis, software code generation, design space exploration for heterogeneous platforms and clock gating strategy. Nevertheless, some problems have been identified and will need to be addressed in the future. As a result, future works will consist in exploiting the same Procedural optimizations that commercial HLSs and state-of-the-art compilers offer, but will also include methodologies and techniques that are not yet to be found in other tools (i.e. Multi-Actor hierarchical memory management).

7.2.1 Component Library Database

This part consists of exploring the FPGA architecture by retrieving the number of LUTS, I/O ports, Internal Memories size and types and the gates combinatorial delay. Each FPGA fabricator uses different names for their internal structure so a general library database should be created that has a one to one representation of the LIM structure. Apart from device hardware characteristics that are fixed (i.e. memory size in an FPGA), the gate delay varies depending on which place the gates are routed. Thus, the estimation of the gate propagation delay is close to the real delay.

This step is a necessity for better scheduling, allocation and binding. In addition, in the component library database characteristics of development boards should be given too. When the architecture model is described the information of off-chip memories (DDR, SDRAM, etc.) and interfaces is available to the developer.

The task by itself can be implemented as TCL scripts that run various benchmarks. For each operation, registers should be added before its input and after output port, which gives directly the gate delay of the operator. Once, the information is retrieved in can be added to Xronos' internal library that can be represented as an equivalent to a SQL database.

7.2.2 SDC Scheduling for LIM

Xronos Achilles heel is that it does not support constraints. Scheduling of Different Constraints proposed by Cong and Zhang in [203] performs a variety of optimizations under a unified mathematical programming framework. The advantage of using SDC is that it is possible to apply scheduling that supports constraints for resources, frequency, latency, relative timing and overall latency.

The idea is to formulate the scheduling problem as a linear program that can be solved with a standard LP solver. As described in [204], each operation is assigned to a variable that after having solved it, will hold the control step (clock cycle) in which the operation is scheduled. Let us consider two operations o_i and o_j and CC_{o_i} representing the clock cycle in which o_i is to be scheduled and CC_{o_j} for o_j. The value of CC_{o_i} is given by the library database described in the previous sub-section 7.2.1. Assuming that there is a control and data dependency from o_i to o_j, the following different constraints are added to the LP formulation $CS_{o_i} - CS_{o_j} \geq 0$. Furthermore, a clock period constraint can be incorporated. Let C_p be the target clock period and C_{hain} a chain of any N dependent combinational operations in a Block with a chain $C_{hain} = o_1 \longrightarrow o_2 \longrightarrow \ldots \longrightarrow o_n$. The total estimated combinatorial delay of operation is the sum $T_e = \sum_{i=1}^{n} CC_{o_i}$. The clock period constraint on the chain is $CC_{o_n} - CC_{o_1} \geq \lceil C_p / T_e \rceil - 1$. Such constraints control determine which operators can be chained together in a clock cycle. Chaining is only permitted if the target C_p is met. For more information on SDC refer to [203].

The advantage of having a flexible scheduler that constraints can be applied on *Actor* ports and *Actions* makes CAL programming language even more flexible for HLS. As described in the design space exploration chapter, the impact analysis gives the necessary clock cycles that an action should be reduced for not being in the algorithmically critical path. If the scheduler can resolve the latency constraint on the selected set of actors, TURNUS and Xronos can optimize a design automatically.

7.2.3 Integration of state of the art procedural optimizations

Commercial HLS such as Xilinx's Vivado HLS and Calypto's Catapult use C/C++/SystemC front-ends that are not developed by them. Vivado HLS for example uses the Clang the C

frontend of LLVM and it takes advantage of the open source community in a vast choice of procedural optimization. Xronos has a need for loop and if-hyper block optimizations. Loop unrolling, merging, unswitching and code invariant motion are necessary optimizations that should be implemented in Xronos.

7.2.4 Memory Partitioning

Memory partitioning is widely adopted to increase the memory bandwidth efficiently by using multiple memory banks and reducing data access conflict. Xronos does not support this feature and developers need to separate different memory banks manually by adding more actions and complexity into their existing code. An attractive solution that fits streaming application is proposed in [205] and [206]. Wang and al.[206] based in the methodology in [205] propose an automatic memory partitioning scheme for multidimensional arrays based on a linear transformation. With the purpose to provide high data throughput of on-chip memories for the loop pipelining in high-level synthesis.

7.2.5 Multi-Actor hierarchical memory management

In streaming video programs such as the MPEG decoders, there is a need to store reference images in large memories that do not fit in the internal FPGA memories. The concept is to create automatically, given a Component Library Database of supported board, an external memory referee that will manage large memories inside actors. Given the MoC of an actor, and especially the property that only one action can be executed, caching techniques should be investigated for pre-loading memory portions. In addition, this referee should manage read and write requests from other actors too.

7.2.6 Multiplexing and De-multiplexing queue channels for heterogeneous targets

As described in Section 4.13.4, the interface bandwidth or the communication scheduling had a huge impact on performance. In addition, in Section 5.6 the bandwidth presented a significant problem for finding a mapping between SW and HW processing elements. There is a need for a better interface implementation and for an analysis of the implemented communication scheduling. In addition, SoCs that contain an ARM processing element communicate nowadays through an AXI interface. AXI is an internal bus that is used to communicate with peripheral devices and memory and offers high bandwidth and data locality. Supporting the AXI interface immediately gives a broader support for constructor independent SoCs.

7.2.7 Clock Gating on input conditions and Multi-Clock Domains Partitioning

In the proposed clock-gating strategy, the idleness of an actor and the FIFO emptiness was not taken into account. The emptiness condition is not enough to clock gate an actor, because the actor might be in a firing state. Thus, taking both conditions and the output fullness condition may further improve the overall power consumption. A further interesting direction might be to combine the clock-gating strategy with multi-clock domains. Clocks that have a lower frequency than nominal ones might also reduce the overall power consumption as illustrated in [202].

7.2.8 Dataflow Machines: An alternative Intermediate Representation

The Orcc Dataflow IR and Procedural IR presents a set of limitation as discussed in Section 3.7. A new alternative intermediate representation called Dataflow Machines [6], based on Actor Machines [151], represents a better model for stream programs aimed at capturing their logical structure in a way that they are amenable to analysis, composition, and hardware/software code generation for parallel implementations. Replacing Orcc front-end in Xronos with Dataflow Machines will mainly consist of producing a CDFG graph from a Dataflow Machine model.

Bibliography

[1] S. Edwards, "The challenges of synthesizing hardware from c-like languages," *Design Test of Computers, IEEE*, vol. 23, pp. 375–386, May 2006.

[2] E. Bezati, H. Yviquel, M. Raulet, and M. Mattavelli, "A unified hardware/software co-synthesis solution for signal processing systems," in *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pp. 1 –6, nov. 2011.

[3] E. Bezati, S. Brunet, M. Mattavelli, and J. Janneck, "Synthesis and optimization of high-level stream programs," in *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pp. 1–6, May 2013.

[4] E. Bezati, M. Mattavelli, and J. Janneck, "High-level synthesis of dataflow programs for signal processing systems," in *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, pp. 750–754, Sept 2013.

[5] E. Bezati, G. Roquier, and M. Mattavelli, "Live demonstration: High level software and hardware synthesis of dataflow programs," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pp. 660–660, May 2013.

[6] J. Janneck, G. Cedersjo, E. Bezati, and S. Casale-Brunet, "Dataflow machines," in *Signals, Systems and Computers, 2014 Asilomar Conference on*, Nov 2014.

[7] S. Casale-Brunet, A. Elguindy, E. Bezati, R. Thavot, G. Roquier, M. Mattavelli, and J. W. Janneck, "Methods to explore design space for mpeg rmc codec specifications," *Signal Processing: Image Communication*, vol. 28, no. 10, pp. 1278–1294, 2013.

[8] G. Roquier, E. Bezati, R. Thavot, and M. Mattavelli, "Hardware/software co-design of dataflow programs for reconfigurable hardware and multi-core platforms," in *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pp. 1–7, Nov 2011.

[9] G. Roquier, E. Bezati, and M. Mattavelli, "Hardware and software synthesis of heterogeneous systems from dataflow programs," *Journal of Electrical and Computer Engineering*, vol. 2012, no. 484962, 2014.

[10] S. Casale Brunet, M. Mattavelli, and J. Janneck, "Profiling of dataflow programs using post mortem causation traces," in *Signal Processing Systems (SiPS), 2012 IEEE Workshop on*, oct. 2012.

[11] E. Bezati, R. Thavot, G. Roquier, and M. Mattavelli, "High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms," *Journal of Real-Time Image Processing*, vol. 9, no. 1, pp. 251–262, 2014.

[12] S. Brunet, M. Mattavelli, and J. Janneck, "Buffer optimization based on critical path analysis of a dataflow program design," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pp. 1384–1387, May 2013.

[13] S. Casale-Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. Janneck, "Multi-clock domain optimization for reconfigurable architectures in high-level dataflow applications," in *Signals, Systems and Computers, 2013 Asilomar Conference on*, pp. 1796–1800, Nov 2013.

[14] S. Casale-Brunet, E. Bezati, C. Alberti, G. Roquier, M. Mattavelli, J. Janneck, and J. Boutellier, "Design space exploration and implementation of rvc-cal applications using the turnus framework," in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pp. 341–342, Oct 2013.

[15] S. Casale-Brunet, E. Bezati, M. Mattavelli, M. Canale, and J. Janneck, "Execution trace graph analysis of dataflow programs: bounded buffer scheduling and deadlock recovery using model predictive control," in *Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2014.

[16] S. Casale-Brunet, M. Wiszniewska, E. Bezati, M. Mattavelli, J. Janneck, and M. Canale, "Turnus: an open-source design space exploration framework for dynamic stream programs," in *Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2014.

[17] M. Canale, S. Casale-Brunet, E. Bezati, M. Mattavelli, and J. Janneck, "Dataflow programs analysis and optimization using model predictive control techniques: An example of bounded buffer scheduling," in *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1–6, Oct 2014.

[18] C. Sau, L. Raffo, F. Palumbo, E. Bezati, S. Casale-Brunet, and M. Mattavelli, "Automated design flow for coarse-grained reconfigurable platforms: An rvc-cal multi-standard decoder use-case," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pp. 59–66, July 2014.

[19] E. Bezati, S. Brunet, M. Mattavelli, and J. Janneck, "Coarse grain clock gating of streaming applications in programmable logic implementations," in *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*, pp. 1–6, May 2014.

[20] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synbthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1st ed., 1992.

[21] D. Gajski and R. Kuhn, "Guest editors' introduction: New vlsi tools," *Computer*, vol. 16, pp. 11–14, Dec 1983.

[22] K. Wakabayashi, "C-based behavioral synthesis and verification analysis on industrial design examples," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, (Piscataway, NJ, USA), pp. 344–348, IEEE Press, 2004.

[23] M. J. S. Smith, *Application-specific Integrated Circuits*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[24] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st ed., 1994.

[25] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design Test of Computers, IEEE*, vol. 26, pp. 18–25, July 2009.

[26] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim, "The cmu design automation system - an example of automated data path design," in *Design Automation, 1979. 16th Conference on*, pp. 73–80, June 1979.

[27] A. Parker and S. Hayati, "Automating the vlsi design process using expert systems and silicon compilation," *Proceedings of the IEEE*, vol. 75, pp. 777–785, June 1987.

[28] M. Barbacci, D. of Computer Science, C.-M. U. Electrical Engineering, G. Barnes, R. Cattell, and D. Siewiorek, *The ISPS Computer Description Language: The Symbolic Manipulation of Computer Descriptions*. Carnegie-Mellon University, Computer Science Department, 1978.

[29] P. Marwedel, "The mimola design system: Detailed description of the software system," in *Design Automation, 1979. 16th Conference on*, pp. 59–63, June 1979.

[30] P. Marwedel, "The mimola design system: Tools for the design of digital processors," in *Design Automation, 1984. 21st Conference on*, pp. 587–593, June 1984.

[31] J. Granacki, D. Knapp, and A. Parker, "The adam advanced design automation system: Overview, planner and natural language interface," in *Design Automation, 1985. 22nd Conference on*, pp. 727–730, June 1985.

[32] R. Jain, K. Kucukcakar, M. Mlinar, and A. Parker, "Experience with the adam synthesis system," in *Design Automation, 1989. 26th Conference on*, pp. 56–61, June 1989.

[33] N. Park and A. Parker, "Sehwa: a software package for synthesis of pipelines from behavioral specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 7, pp. 356–370, Mar 1988.

[34] P. Paulin, J. Knight, and E. Girczyc, "Hal: A multi-paradigm approach to automatic data path synthesis," in *Design Automation, 1986. 23rd Conference on*, pp. 263–270, June 1986.

[35] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 8, pp. 661–679, Jun 1989.

[36] H. Trickey, "Flamel: A high-level hardware compiler," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 6, pp. 259–269, March 1987.

[37] G. De Micheli and D. Ku, "Hercules-a system for high-level synthesis," in *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pp. 483–488, June 1988.

[38] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The olympus synthesis system," *Design Test of Computers, IEEE*, vol. 7, pp. 37–53, Oct 1990.

[39] D. Ku and G. De Micheli, "Relative scheduling under timing constraints," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pp. 59–64, Jun 1990.

[40] A. Chandrakasan, M. Potkonjak, J. Rabaey, and R. Brodersen, "Hyper-lp: a system for power minimization using architectural transformations," in *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on*, pp. 300–303, Nov 1992.

[41] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. Brodersen, "Optimizing power using transformations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 14, pp. 12–31, Jan 1995.

[42] H. De Man, J. Rabaey, P. Six, and L. Claesen, "Cathedral-ii: A silicon compiler for digital signal processing," *Design Test of Computers, IEEE*, vol. 3, pp. 13–25, Dec 1986.

[43] R. A. Bergamaschi, R. A. O'Connor, L. Stok, M. Z. Moricz, S. Prakash, A. Kuehlmann, and D. S. Rao, "High-level synthesis in an industrial environment," *IBM J. Res. Dev.*, vol. 39, pp. 131–148, Feb. 1995.

[44] P. Lippens, J. Van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, and O. McArdle, "Phideo: a silicon compiler for high speed algorithms," in *Design Automation. EDAC., Proceedings of the European Conference on*, pp. 436–441, Feb 1991.

[45] K. Kucukcakar, C.-T. Chen, J. Gong, W. Philipsen, and T. Tkacik, "Matisse: an architectural design tool for commodity ics," *Design Test of Computers, IEEE*, vol. 15, pp. 22–33, Apr 1998.

[46] J. P. Elliott, *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[47] A. Hemani, B. Karlsson, M. Fredriksson, K. Nordqvist, and B. Fjellborg, "Application of high-level synthesis in an industrial project," in *VLSI Design, 1994., Proceedings of the Seventh International Conference on*, pp. 5–10, Jan 1994.

[48] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Normwell, Ma:Springer, 1st ed., 2000.

[49] "Impulse." http://www.impulseaccelerated.com/. Accessed: 12-2014.

[50] "Cadence Cynthesizer solution." http://www.cadence.com/products/sd/cynthesizer/pages/default.aspx?CMP=MOSS5/. Accessed: 12-2014.

[51] "Calypto Catapult." http://calypto.com/en/products/catapult/overview/. Accessed: 12-2014.

[52] "NEC Cyber WorkBench." http://www.nec.com/en/global/prod/cwb/. Accessed: 12-2014.

[53] "Synopsis synphonyc compiler." http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx. Accessed: 12-2014.

[54] "Xilinx Vivado High-level Synthesis." http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. Accessed: 12-2014.

[55] C. E. Stroud, R. R. Munoz, and D. A. Pierce, "Behavioral model synthesis with cones," *IEEE Des. Test*, vol. 5, pp. 22–30, May 1988.

[56] D. C. Ku and G. De Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[57] D. Galloway, "The transmogrifier c hardware description language and compiler for fpgas," in *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, FCCM '95, (Washington, DC, USA), pp. 136–, IEEE Computer Society, 1995.

[58] Y. Panchul, D. Soderman, and D. Coleman, "System for converting hardware designs in high-level programming languages to hardware implementations," Oct. 25 2001. US Patent App. 09/846,092.

[59] 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 (invited talk)," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, (New York, NY, USA), pp. 151–155, ACM, 2001.

[60] K. Wakabayashi and T. Okamoto, "C-based soc design flow and eda tools: An asic and system vendor perspective," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 19, pp. 1507–1522, Nov. 2006.

[61] P. Paulin, C. Pilkington, and E. Bensoudane, "Stepnp: a system-level exploration platform for network processors," *Design Test of Computers, IEEE*, vol. 19, pp. 17–26, Nov 2002.

[62] "Bluespec inc." http://www.bluespec.com/. Accessed: 12-2014.

[63] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "Fcuda: Enabling efficient compilation of cuda kernels onto fpgas," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pp. 35–42, July 2009.

[64] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of platform architectures from opencl programs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pp. 186–193, May 2011.

[65] P. Bellows and B. Hutchings, "Jhdl-an hdl for reconfigurable systems," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pp. 175–184, Apr 1998.

[66] J. L. Tripp, P. A. Jackson, and B. Hutchings, "Sea cucumber: A synthesizing compiler for fpgas," in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, FPL '02, (London, UK, UK), pp. 875–885, Springer-Verlag, 2002.

[67] D. Greaves and S. Singh, "Using c sharp attributes to describe hardware artefacts within kiwi," in *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pp. 239–240, Sept 2008.

[68] W. Luk and S. McKeever, "Pebble: A language for parametrised and reconfigurable hardware design," in *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm*, Springer Berlin Heidelberg, 1998.

[69] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, pp. 87–152, Nov. 1992.

[70] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "Uml-based multiprocessor soc design framework," *ACM Trans. Embed. Comput. Syst.*, vol. 5, pp. 281–320, May 2006.

[71] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Syst. J.*, vol. 45, pp. 451–461, July 2006.

[72] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin, "Pls: a scheduler for pipeline synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 12, pp. 1279–1286, Sep 1993.

[73] C. H. Gebotys and M. I. Elmasry, "Vlsi design synthesis with testability," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, (Los Alamitos, CA, USA), pp. 16–21, IEEE Computer Society Press, 1988.

[74] P. Marwedel, "A new synthesis algorithm for the mimola software system," in *Design Automation, 1986. 23rd Conference on*, pp. 271–277, June 1986.

[75] J.-H. Lee, Y.-C. Hsu, and Y.-L. Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis," in *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on*, pp. 20–23, Nov 1989.

[76] I.-C. Park and C.-M. Kyung, "Fast and near optimal scheduling in automatic data path synthesis," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, (New York, NY, USA), pp. 680–685, ACM, 1991.

[77] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[78] A.-H. Ab Rahman, A. Prihozhy, and M. Mattavelli, "Pipeline synthesis and optimization of fpga-based video processing applications with cal," *EURASIP Journal on Image and Video Processing*, vol. 2011, no. 1, 2011.

[79] L. Gao, D. Zaretsky, G. Mittal, D. Schonfeld, and P. Banerjee, "A software pipelining algorithm in high-level synthesis for fpga architectures," in *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*, pp. 297–302, March 2009.

[80] M. Weinhardt and W. Luk, "Pipeline vectorization," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 20, pp. 234–248, Nov. 2006.

[81] S. Oh, T. G. Kim, J. Cho, and E. Bozorgzadeh, "Speculative loop-pipelining in binary translation for hardware acceleration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, pp. 409–422, March 2008.

[82] G. De Micheli, "Hardware synthesis from c/c++ models," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pp. 382–383, 1999.

[83] L.-F. Chao, A. LaPaugh, and E.-M. Sha, "Rotation scheduling: a loop pipelining algorithm," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 16, pp. 229–239, Mar 1997.

[84] H.-S. Jun and S.-Y. Hwang, "Design of a pipelined datapath synthesis system for digital signal processing," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 292–303, Sept 1994.

[85] W. Verhaegh, P. Lippens, E. Aarts, J. Korst, J. Van Meerbergen, and A. van der Werf, "Improved force-directed scheduling in high-throughput digital signal processing," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 14, pp. 945–960, Aug 1995.

[86] E. Nurvitadhi, J. Hoe, T. Kam, and S. Lu, "Automatic pipelining from transactional datapath specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, pp. 441–454, March 2011.

[87] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, Sept 1987.

[88] G. Kahn, "The Semantics of Simple Language for Parallel Programming," in *IFIP Congress*, pp. 471–475, 1974.

[89] A. Prihozhy, "Net scheduling in high-level synthesis," *Design Test of Computers, IEEE*, vol. 13, pp. 26–35, Spring 1996.

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

[91] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard," *Signal Processing Magazine, IEEE*, vol. 27, pp. 159 –167, May 2010.

[92] C. Leiserson and J. B. Saxe, "Optimizing synchronous systems," in *Foundations of Computer Science, 1981. SFCS '81. 22nd Annual Symposium on*, pp. 23–36, Oct 1981.

[93] S. Malik, K. Singh, R. Brayton, and A. Sangiovanni-Vincentelli, "Performance optimization of pipelined logic circuits using peripheral retiming and resynthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 12, pp. 568–578, May 1993.

[94] N. Shenoy, "Retiming: Theory and practice," *Integr. VLSI J.*, vol. 22, pp. 1–21, Aug. 1997.

[95] K. Hwang, A. Casavant, C.-T. Chang, and M. d'Abreu, "Scheduling and hardware sharing in pipelined data paths," in *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on*, pp. 24–27, Nov 1989.

[96] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man, "An efficient microcode compiler for application specific dsp processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 9, pp. 925–937, Sep 1990.

[97] W. Sun, M. Wirthlin, and S. Neuendorffer, "Fpga pipeline synthesis design exploration using module selection and resource sharing," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, pp. 254–265, Feb 2007.

[98] B. Haroun and M. Elmasry, "Architectural synthesis for dsp silicon compilers," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 8, pp. 431–447, Apr 1989.

[99] H. Javaid, A. Ignjatovic, and S. Parameswaran, "Rapid design space exploration of application specific heterogeneous pipelined multiprocessor systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, pp. 1777–1789, Nov 2010.

[100] E. Girczyc, "Loop winding - a data flow approach to functional pipelining," in *ISCAS*, pp. 382–385, 1987.

[101] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation based synthesis," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pp. 444–449, Jun 1990.

[102] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, (New York, NY, USA), pp. 308–317, ACM, 1988.

[103] F. N. Najm, "Feedback, correlation, and delay concerns in the power estimation of vlsi circuits," in *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, (New York, NY, USA), pp. 612–617, ACM, 1995.

[104] M. Pedram, "Power minimization in ic design: Principles and applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, pp. 3–56, Jan. 1996.

[105] S. M. Srinivas Devadas, "A survey of optimization techniques targeting low power vlsi circuits," in *Design Automation, 1995. DAC '95. 32nd Conference on*, pp. 242–247, 1995.

[106] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power cmos digital design," *Solid-State Circuits, IEEE Journal of*, vol. 27, pp. 473–484, Apr 1992.

[107] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. Brodersen, "Optimizing power using transformations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 14, pp. 12–31, Jan 1995.

[108] R. Mehra and J. Rabaey, "Behavioral level power estimation and exploration," in *in Proc. Int. Wkshp. Low Power Design*, pp. 197–202, 1994.

[109] A. Chatterjee and R. Roy, "Synthesis of low power linear dsp circuits using activity metrics," in *VLSI Design, 1994., Proceedings of the Seventh International Conference on*, pp. 265–270, Jan 1994.

[110] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man, "Global communication and memory optimizing transformations for low power signal processing systems," in *VLSI Signal Processing, VII, 1994., [Workshop on]*, pp. 178–187, 1994.

[111] D. Lidsky and J. Rabaey, "Low-power design of memory intensive functions," in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pp. 16–17, Oct 1994.

[112] P. Panda and N. Dutt, "Reducing address bus transition for low power memory mapping," in *European Design and Test Conference, 1996. ED TC 96. Proceedings*, pp. 63–68, Mar 1996.

[113] A. Raghunathan and N. Jha, "Behavioral synthesis for low power," in *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, pp. 318–322, Oct 1994.

[114] A. Raghunathan and N. Jha, "An ilp formulation for low power based on minimizing switched capacitance during data path allocation," in *Circuits and Systems, 1995. ISCAS '95., 1995 IEEE International Symposium on*, vol. 2, pp. 1069–1073 vol.2, Apr 1995.

[115] M. P. Jui-Ming Chang, "Register allocation and binding for low power," in *Design Automation, 1995. DAC '95. 32nd Conference on*, pp. 29–35, 1995.

[116] A. Dasgupta and R. Karri, "Simultaneous scheduling and binding for power minimization during microarchitecture synthesis," in *Proceedings of the 1995 International Symposium on Low Power Design*, ISLPED '95, (New York, NY, USA), pp. 69–74, ACM, 1995.

[117] E. Musoll and J. Cortadella, "High-level synthesis techniques for reducing the activity of functional units," in *Proceedings of the 1995 International Symposium on Low Power Design*, ISLPED '95, (New York, NY, USA), pp. 99–104, ACM, 1995.

[118] M. Stan and W. Burleson, "Bus-invert coding for low-power i/o," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 3, pp. 49–58, March 1995.

[119] A. Manzak and C. Chakrabarti, "A low power scheduling scheme with resources operating at multiple voltages," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 10, pp. 6–14, Feb 2002.

[120] S. Mohanty and N. Ranganathan, "Energy efficient scheduling for datapath synthesis," in *VLSI Design, 2003. Proceedings. 16th International Conference on*, pp. 446–451, Jan 2003.

[121] S. Mohanty and N. Ranganathan, "Simultaneous peak and average power minimization during datapath scheduling," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 52, pp. 1157–1165, June 2005.

[122] M. C. Johnson and K. Roy, "Datapath scheduling with multiple supply voltages and level converters," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 2, pp. 227–248, July 1997.

[123] X. Xing and C.-C. Jong, "Multivoltage multifrequency low-energy synthesis for functionally pipelined datapath," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, pp. 1348–1352, Sept 2009.

[124] W.-T. Chang, S. Ha, and E. A. Lee, "Heterogeneous simulation—mixing discrete-event models with dataflow," *J. VLSI Signal Process. Syst.*, vol. 15, pp. 127–144, Jan. 1997.

[125] Y.-R. Lin, C.-T. Hwang, and A. C.-H. Wu, "Scheduling techniques for variable voltage low power designs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 2, pp. 81–97, Apr. 1997.

[126] G. Lakshminarayana and N. Jha, "High-level synthesis of power-optimized and area-optimized circuits from hierarchical data-flow intensive behaviors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 18, pp. 265–281, Mar 1999.

[127] D. Chen, J. Cong, and Y. Fan, "Low-power high-level synthesis for fpga architectures," in *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, pp. 134–139, Aug 2003.

[128] B. Landman and R. L. Russo, "On a pin versus block relationship for partitions of logic graphs," *Computers, IEEE Transactions on*, vol. C-20, pp. 1469–1479, Dec 1971.

[129] A. Bogliolo, L. Benini, B. Ricco, and G. De Micheli, "Efficient switching activity computation during high-level synthesis of control-dominated designs," in *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, pp. 127–132, Aug 1999.

[130] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D. Lundqvist, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style," in *Design Automation Conference, 1999. Proceedings. 36th*, pp. 873–878, 1999.

[131] S. Suhaib, D. Mathaikutty, and S. Shukla, "Dataflow architectures for GALS," *Electronic Notes in Theoretical Computer Science*, vol. 200, no. 1, pp. 33–50, 2008.

[132] T.-Y. Wuu and S. B. K. Vrudhula, "Synthesis of asynchronous systems from data flow specification," Research Report ISI/RR-93-366, University of Southern California, Information Sciences Institute, Dec 1993.

[133] B. Ghavami and H. Pedram, "High performance asynchronous design flow using a novel static performance analysis method," *Comput. Electr. Eng.*, vol. 35, pp. 920–941, Nov. 2009.

[134] M. Gerber and T. Gossi, "Parallel coprocessor architectures for molecular dynamics simulation: a case study in design space exploration," in *Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on*, vol. 6, pp. 155–158 vol.6, May 1998.

[135] Y. Le 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, no. 2, pp. 185–208, 2006.

[136] "The syndex project." http://www.syndex.org/. Accessed: 12-2014.

[137] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pp. 36–40, Sept 2014.

[138] "National instruments fpga." http://www.ni.com/fpga/. Accessed: 12-2014.

[139] "Matlab hdl coder." http://mathworks.com/products/hdl-coder/. Accessed: 12-2014.

[140] 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 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, (New York, NY, USA), pp. 9–14, ACM, 2007.

[141] 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 *DVCon 2007*, February 2007.

[142] M. P. Howarth, P. Flegkas, G. Pavlou, N. Wang, P. Trimintzios, D. Griffin, J. Griem, M. Boucadair, P. Morand, A. Asgari, and P. Georgatsos, "Provisioning for interdomain quality of service: The mescal approach," *Comm. Mag.*, vol. 43, pp. 129–137, June 2005.

[143] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "Peace: A hardware-software codesign environment for multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 24:1–24:25, May 2008.

[144] "The ptolemy project." http://ptolemy.eecs.berkeley.edu/. Accessed: 12-2014.

[145] J. Keinert, M. Streub&uhorbar;hr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "Systemcodesigner—an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, pp. 1:1–1:23, Jan. 2009.

[146] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing Hardware from Dataflow Programs: An MPEG-4 Simple Profile Decoder Case Study," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 241–249, 2009. 10.1007/s11265-009-0397-5.

[147] "Open dataflow." http://sourceforge.net/projects/opendf. Accessed: 12-2014.

[148] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia development made easy," in *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pp. 863–866, ACM, 2013.

[149] "Caltoopia." http://www.caltoopia.org/. Accessed: 12-2014.

[150] R. Thavot, *High-level dataflow programming for complex digital systems.* PhD thesis, STI, Lausanne, 2013.

[151] J. Janneck, "A machine model for dataflow actors and its applications," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pp. 756–760, Nov 2011.

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

[153] J. B. Dennis, “First version of a data flow procedure language,” in *Symposium on Programming*, pp. 362–376, 1974.

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

[155] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, “System-level design: Orthogonalization of concerns and platform-based design,” *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 19, pp. 1523–1543, nov 2006.

[156] A. Turing, “On computable numbers with an application to the "Entscheidungsproblem",” *Proceeding of the London Mathematical Society*, 1936.

[157] D. McAllester, P. Panangaden, and V. Shanbhogue, “Nonexpressibility of fairness and signaling,” *J. Comput. Syst. Sci.*, vol. 47, pp. 287–321, oct. 1993.

[158] E. Lee and A. Sangiovanni-Vincentelli, “Comparing models of computation,” in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pp. 234–241, IEEE Computer Society, 1997.

[159] E. Lee and T. Parks, “Dataflow process networks,” in *Proceedings of the IEEE*, pp. 773–799, 1995.

[160] “Open RVC-CAL compiler.” http://orcc.sourceforge.net/. Accessed: 12-2014.

[161] “Xtext: Language development made easy!.” https://eclipse.org/Xtext/. Accessed: 12-2014.

[162] “Eclipse modeling framework.” http://eclipse.org/modeling/emf/. Accessed: 12-2014.

[163] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd ed., 2009.

[164] “Xtend: Modernized java.” http://www.altera.com/products/software/opencl/opencl-index.html. Accessed: 12-2014.

[165] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[166] N. Siret, M. Wipliez, J.-F. Nezan, and A. Rhatay, “Hardware code generation from dataflow programs,” in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pp. 113–120, Oct 2010.

[167] K. Jerbi, M. Raulet, O. Deforges, and M. Abid, “Automatic generation of synthesizable hardware implementation from high level RVC-CAL description,” in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pp. 1597–1600, March 2012.

[168] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451–490, Oct. 1991.

[169] M. Abid, K. Jerbi, M. Raulet, O. Deforges, and M. Abid, "System level synthesis of dataflow programs: Hevc decoder case study," in *Electronic System Level Synthesis Conference (ESLsyn), 2013*, pp. 1–6, May 2013.

[170] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: a high-level synthesis framework for applying parallelizing compiler transformations," in *VLSI Design, 2003. Proceedings. 16th International Conference on*, pp. 461–466, Jan 2003.

[171] A. Inc., "Fpga performance benchmarking methodology."

[172] OSCI, "Systemc synthesizable subset draft 1.3," 2010.

[173] "performance api." http://icl.cs.utk.edu/papi/. Accessed: 12-2014.

[174] A. Ghazi, J. Boutellier, M. Abdelaziz, X. Lu, L. Anttila, J. Cavallaro, S. Bhattacharyya, M. Valkama, and M. Juntti, "Low power implementation of digital predistortion filter on a heterogeneous application specific multiprocessor," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 8336–8340, May 2014.

[175] "Berkeley softfloat." http://www.jhauser.us/arithmetic/SoftFloat.html. Accessed: 12-2014.

[176] "Snu real time benchmarks." http://www.cprover.org/goto-cc/examples/snu.html. Accessed: 12-2014.

[177] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pp. 330–335, Dec 1997.

[178] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[179] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J.-F. Nezan, and O. Déforges, "Reconfigurable video coding on multicore," *IEEE Signal Processing Magazine*, vol. 26, pp. 113 –123, november 2009.

[180] A. Carlsson, J. Eker, T. Olsson, and C. von Platen, "Scalable parallelism using dataflow programming," in *Ericson Review, On-line publishing www.ericsson.com*, 2011.

[181] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using kahn process networks: The compaan/laura approach," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, (Washington, DC, USA), pp. 10340–, IEEE Computer Society, 2004.

[182] 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.

[183] M. Ravasi, *An automatic C-code instrumentation framework for high level algorithmic complexity analysis and system design.* PhD thesis, STI, Lausanne, 2003.

[184] A. Kienhuis, *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools.* PhD thesis, Delft University of Technology, The Netherlands, January 1999.

[185] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integr. VLSI J.*, vol. 38, pp. 131–183, dec 2004.

[186] M. Pelcat, J. Nezan, J. Piat, J. Croizer, and S. Aridhi, "A System-Level Architecture Model for Rapid Prototyping of Heterogeneous Multicore Embedded Systems," in *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, (nice, France), p. 8 pages, Sep 2009.

[187] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli, "Formal models for embedded system design," *IEEE Design & Test of Computers*, vol. 17, no. 2, pp. 14–27, 2000.

[188] A. A. H. B. Ab Rahman, *Optimizing Dataflow Programs for Hardware Synthesis.* PhD thesis, STI, Lausanne, 2014.

[189] P. K. Murthy and S. S. Bhattacharyya, *Memory Management for Synthesis of DSP Software.* CRC Press, 2006.

[190] "A discrete event system simulator." http://web.ornl.gov/~1qn/adevs/. Accessed: 12-2014.

[191] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation.* Orlando, FL, USA: Academic Press, Inc., 2nd ed., 2000.

[192] J. Woodward, "Computable and incomputable functions and search algorithms," in *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, vol. 1, pp. 871–875, Nov 2009.

[193] M. Pedram, "Power minimization in ic design: Principles and applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, pp. 3–56, Jan. 1996.

[194] Q. Wu, M. Pedram, and X. Wu, "Clock-gating and its application to low power design of sequential circuits," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 47, pp. 415–420, Mar 2000.

[195] G. Tellez, A. Farrahi, and M. Sarrafzadeh, "Activity-driven clock design for low power circuits," in *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, pp. 62–65, Nov 1995.

[196] Xilinx, *Virtex-7 T and XT FPGAs Data Sheet*, November 2013. DS183.

[197] Xilinx, *7 Series FPGAs Clocking Resources*, August 2013. UG472.

[198] D. B. Parlour, "Power control in a data flow processing architecture." US Patent 7,437,582, October 2008.

[199] H. Prabhu, S. Thomas, J. Rodrigues, T. Olsson, and A. Carlsson, "A GALS ASIC implementation from a CAL dataflow description," in *NORCHIP, 2011-11-14/2011-11-15*, 2011.

[200] Xilinx, *Analysis of Power Savings from Intelligent Clock Gating*, August 2012. XAPP790.

[201] "Open RVC-CAL Applications," 2014. http://github.com/orcc/orc-apps, accessed 25-February-2014].

[202] S. Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. Janneck, "Partitioning and optimization of high level stream applications for multi clock domain architectures," in *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pp. 177–182, Oct 2013.

[203] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, (New York, NY, USA), pp. 433–438, ACM, 2006.

[204] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (New York, NY, USA), pp. 33–36, ACM, 2011.

[205] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pp. 697–704, Nov 2009.

[206] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, (New York, NY, USA), pp. 12:1–12:8, ACM, 2013.

Endri Bezati

36 Rue St.Martin
1005 Lausanne
(+41) 076.528.21.61
endri.bezati@epfl.ch

Education

2007 - 2010 **Engineer EII**, *INSA of Rennes (Institut National des Sciences Appliquées), France*
Master degree in Electronics and Computer Engineering,
With option in "Embedded Systems" and "Business Engineering"

2005 - 2007 **DUT GEII**, *University of Rennes 1, France*
Bachelor In Electronics and Computer Engineering

2004 - 2005 **STPI – 1er Cycle**, *INSA de Rennes (Institut National des Sciences Appliquées)*
First year of "Cycle Préparatoire" of "Engineer Student" in the EURINSA international section

2003 - 2004 **A-Level**, *3rd High School of Piraeus, Greece*
"APOLYTIRION" A-Level equivalent in Math's and Engineering, degree with Honors

Skills

Information Technology

Pro. Languages	C, C++, Java, CAL	**Hardware Languages**	Verilog, VHDL, SystemC
Script Languages	Bash, Javascript	**API**	.Net, MFC, GTK, QT,SWT
OS	Windows, Linux, Mac OS X	**Embedded OS**	Linux RT, Android
I.D.E	Visual Studio, Eclipse, Xcode	**Image/Video Standards**	JPEG,MPEG4, AVC/H.264, SVC, HEVC/H2.65

Electronics

Conception Tools	OrCAD, PSPICE, Modelsim, Quartus, Xillinx ISE/Vivado	**Signal Processing**	Matlab, LabView, Scilab, Octave
E.D.A.	Altium Designer, Protel, Eagle	**P.L.C.**	Siemens, Schneider

Professional Experiences

2010 **Master Thesis**, *EPFL SCI-STI-MM*, Lausanne, Switzerland
Full feature data flow Implementation of the entropy decoder CABAC (Context Adaptive Binary Arithmetic Coding) of the AVC/H.264 Main Profile decoder.

2009 **Engineer Internship**, *I.E.T.R.*, Rennes, France
Data flow modeling and programming of the MPEG-SVC (H.264) decoder for integrating it in the video tool library of the new MPEG-RVC (Reconfigurable Video Coding) standard

2007 **Bachelor Internship**, *Thomson Grass Valley (now Technicolor)*, Rennes, France
Hardware (schematic conception, component selection, routing) and Software (.Net GUI, C++) conception of an automatic digital signal cutter for testing robustness of the Thomson Grass Valley high definition video encoders

Foreign Languages

English	Advanced (TOEIC 920/990)	**French**	Perfect
Albanian	Mother tongue	**Greek**	Mother tongue

Center of Interest

IT	Linux Kernel, OpenCL, Android	**Films**	Science fiction, Action, Comedy
Books	Science fiction	**Sports**	Football, Body Building

Publications

2015 A. Prihozhy ,E. Bezati, A. Ab-Rahman , M.Mattavelli " Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , accepted for publication

2014 E. Bezati, R. Thavot, G. Roquier, and M. Mattavelli, "High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms," Journal of Real-Time Image Processing, vol. 9, no. 1, pp. 251–262, 2014.

E.Bezati, S.Brunet, M.Mattavelli, and J.Janneck, "Coarse grain clock gating of streaming applications in programmable logic implementations," in *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*, pp. 1–6, May 2014.

S. Casale-Brunet, M. Wiszniewska, E. Bezati, M. Mattavelli, J. Janneck, and M. Canale, "Turnus: an open- source design space exploration framework for dynamic stream programs," in Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP), 2014.

S. Casale-Brunet, E. Bezati, M. Mattavelli, M. Canale, and J. Janneck, "Execution trace graph analysis of dataflow programs: bounded buffer scheduling and deadlock recovery using model predictive control," in Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP), 2014.

S. Casale-Brunet, M. Wiszniewska, E. Bezati, M. Mattavelli, J. Janneck, and M. Canale, "Turnus: an open- source design space exploration framework for dynamic stream programs," in Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP), 2014.

M. Canale, S. Casale-Brunet, E. Bezati, M. Mattavelli, and J. Janneck, "Dataflow programs analysis and optimization using model predictive control techniques: An example of bounded buffer scheduling," in Signal Processing Systems (SiPS), 2014 IEEE Workshop on, pp. 1–6, Oct 2014.

C. Sau, L. Raffo, F. Palumbo, E. Bezati, S. Casale-Brunet, and M. Mattavelli, "Automated design flow for coarse-grained reconfigurable platforms: An RVC-Cal multi-standard decoder use-case," in Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on, pp. 59– 66, July 2014.

J. Janneck, G. Cedersjo, E. Bezati, and S. Casale-Brunet, "Dataflow machines," in Signals, Systems and Computers, 2014 Asilomar Conference on, Nov 2014.

2013 E. Bezati, S. Brunet, M. Mattavelli, and J. Janneck, "Synthesis and optimization of high-level stream programs," in Electronic System Level Synthesis Conference (ESLsyn), 2013, pp. 1–6, May 2013.

E. Bezati, G. Roquier, and M. Mattavelli, "Live demonstration: High level software and hardware synthesis of dataflow programs," in Circuits and Systems (ISCAS), 2013 IEEE International Symposium on, pp. 660– 660, May 2013.

E. Bezati, M. Mattavelli, and J. Janneck, "High-level synthesis of dataflow programs for signal processing systems," in Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on, pp. 750–754, Sept 2013.

S. Casale-Brunet, A. Elguindy, E. Bezati, R. Thavot, G. Roquier, M. Mattavelli, and J. W. Janneck, "Methods to explore design space for mpeg rmc codec specifications," Signal Processing: Image Communication, vol. 28, no. 10, pp. 1278–1294, 2013.

S. Casale-Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. Janneck, “Partitioning and optimization of high level stream applications for multi clock domain architectures,” in Signal Processing Systems (SiPS), 2013 IEEE Workshop on, pp. 177–182, Oct 2013.

S. Casale-Brunet, E. Bezati, C. Alberti, G. Roquier, M. Mattavelli, J. Janneck, and J. Boutellier, “Design space exploration and implementation of rvc-cal applications using the turnus framework,” in Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on, pp. 341–342, Oct 2013.

S. Casale-Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. Janneck, “Multi-clock domain optimization for reconfigurable architectures in high-level dataflow applications,” in Signals, Systems and Computers, 2013 Asilomar Conference on, pp. 1796–1800, Nov 2013.

2012 G. Roquier, E. Bezati, and M. Mattavelli, “Hardware and software synthesis of heterogeneous systems from dataflow programs,” Journal of Electrical and Computer Engineering, vol. 2012, no. 484962.

A. Ab-Rahman, R. Thavot, S. Casale-Brunet, E. Bezati, and M. Mattavelli, “Design space exploration strategies for fpga implementation of signal processing systems using cal dataflow program,” in Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on, pp. 1–8, Oct 2012.

2011 E. Bezati, H. Yviquel, M. Raulet, and M. Mattavelli, “A unified hardware/software co-synthesis solution for signal processing systems,” in Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on, pp. 1 –6, nov. 2011.

G. Roquier, E. Bezati, R. Thavot, and M. Mattavelli, “Hardware/software co-design of dataflow programs for reconfigurable hardware and multi-core platforms,” in Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on, pp. 1–7, Nov 2011.

2010 E. Bezati, M. Mattavelli, and M. Raulet, “RVC-Cal dataflow implementations of mpeg avc/h.264 cabac decoding,” in Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on, pp. 207–213, Oct 2010.

MPEG Contribution: Dataflow CABAC Implementation in RVC-CAL, June 2010