Răzvan Nane

Automatic Hardware Generation for Reconfigurable Architectures

Automatic Hardware Generation for Reconfigurable Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op

donderdag **17 april 2014** om *10:00* uur

door

Răzvan NANE

Master of Science in Computer Engineering
Delft University of Technology
geboren te Boekarest, Roemenië

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. K.L.M. Bertels

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. K.L.M. Bertels	Technische Universiteit Delft, promotor
Prof. dr. E. Visser	Technische Universiteit Delft
Prof. dr. W.A. Najjar	University of California Riverside
Prof. dr.-ing. M. Hübner	Ruhr-Universität Bochum
Dr. H.P. Hofstee	IBM Austin Research Laboratory
Dr. ir. A.C.J. Kienhuis	Universiteit van Leiden
Dr. ir. J.S.S.M Wong	Technische Universiteit Delft
Prof. dr. ir. Geert Leus	Technische Universiteit Delft, reservelid

Automatic Hardware Generation for Reconfigurable Architectures
Dissertation at Delft University of Technology

ISBN 978-94-6186-271-6

Printed by CPI Koninklijke Wöhrmann, Zutphen, The Netherlands

To my family

Automatic Hardware Generation for Reconfigurable Architectures

Răzvan Nane

Abstract

RECONFIGURABLE Architectures (RA) have been gaining popularity rapidly in the last decade for two reasons. First, processor clock frequencies reached threshold values past which power dissipation becomes a very difficult problem to solve. As a consequence, alternatives were sought to keep improving the system performance. Second, because Field-Programmable Gate Array (FPGA) technology substantially improved (e.g., increase in transistors per mm^2), system designers were able to use them for an increasing number of (complex) applications. However, the adoption of reconfigurable devices brought with itself a number of related problems, of which the complexity of programming can be considered an important one. One approach to program an FPGA is to implement an automatically generated Hardware Description Language (HDL) code from a High-Level Language (HLL) specification. This is called High-Level Synthesis (HLS). The availability of powerful HLS tools is critical to managing the ever-increasing complexity of emerging RA systems to leverage their tremendous performance potential. However, current hardware compilers are not able to generate designs that are comparable in terms of performance with manually written designs. Therefore, to reduce this performance gap, research on how to generate hardware modules efficiently is imperative. In this dissertation, we address the tool design, integration, and optimization of the DWARV 3.0 HLS compiler.

Dissimilar to previous HLS compilers, DWARV 3.0 is based on the CoSy compiler framework. As a result, this allowed us to build a highly modular and extendible compiler in which standard or custom optimizations can be easily integrated. The compiler is designed to accept a large subset of C-code as input and to generate synthesizable VHDL code for unrestricted application domains. To enable DWARV 3.0 third-party tool-chain integration, we propose several IP-XACT (i.e., a XML-based standard used for tool-interoperability) extensions such that hardware-dependent software can be generated and integrated automatically. Furthermore, we propose two new algorithms to optimize the performance for different input area constraints, respectively, to leverage the benefits of both jump and predication schemes from conventional processors adapted for hardware execution. Finally, we performed an evaluation against state-of-the-art HLS tools. Results show that application execution time wise, DWARV 3.0 performs, on average, the best among the academic compilers.

Acknowledgments

It is a great pleasure to write this (last) part of my dissertation. The period spent on working towards this goal has not always been easy, and, at times, finalizing the thesis did not even seem possible. Fortunately, I am lucky to have a very supporting family and warmhearted friends alongside, and to have met very helpful, understanding and skilful people that made the task of completing the work both realizable and enjoyable. I am confronted now with words that cannot express my deepest gratitude I have for all these family members, friends and colleagues. For all the people who I forget at the time of writing, please accept my apology.

First of all, I want to thank my supervisor, prof. dr. Koen Bertels, for giving me the opportunity, research freedom and self-confidence to complete a Ph.D. study. I am also grateful for including me in different European projects that allowed me to travel to project meetings, as well as to various international conferences. This allowed me not only to extend my professional network by meeting, working and collaborating with well-known people in the field, but also to discover different parts and cultures of the world. Thank you!

I want to thank my office colleagues who provided me with valuable information that aided me in the various tasks performed along the years. First, I want to specially thank Vlad-Mihai Sima for all discussions both work and non-work related as well as for his immediate help with diverse Linux related tool issues. Furthermore, I am very thankful for the time taken to read the draft version of the thesis and for providing insightful comments and improvement suggestions. Second, I want to thank Yana Yankova for helping me in the beginning of the study and for creating the first version of the DWARV compiler. Third, I thank Giacomo Machiori for providing me insights into various hardware processes and for helping me solve some of the backend tool issues.

I thank all people involved in the European projects with whom I had the immense pleasure of working. I want to thank Bryan Olivier from ACE, who helped me kick-start my CoSy experience, as well as to Hans van Someren also from ACE for the optimization related discussions. Furthermore, I am grateful

for the collaborations with Pedro Diniz, João Cardoso, Zlatko Petrov, Michael Hübner, Georgi Kuzmanov in the Framework Programme 7 REFLECT project, as well as with Bart Kienhuis, Sven van Haastregt and Todor Stefanov in the MEDEA+ SoftSoc project.

I consider myself very fortunate to have worked in an international department that allowed me to meet people from all over the world. I want to thank Computer Engineering (CE) colleagues Cuong, Gustavo, Seyab, Roel, Changlin, Shanshan and many others for broadening my knowledge about other cultures. I thank also to CE colleagues Berna and Joost for helping me translate in Dutch the propositions and the abstract. At the same time, I am grateful to fellow Romanian colleagues Bogdan, Cătălin, George, Marius, Mihai, Nicoleta, Anca for the interesting back home related discussions. I am thankful to the always friendly and helpful staff members Lidwina Tromp, Eef Hartman and Erik de Vries who made administrative and technical support issues go unnoticeable.

A Latin expression says *'mens sana incorpore sana'*. I am very thankful that the CE department has a healthy attitude and encourages both sport and social activities. I am therefore very grateful to Said Hamdioui for organizing the CE weekly football games, and to the many enthusiast colleagues, Joost, Motta, Faisal, Imran, Lei, Adib and Innocent to name just a few who participate in this activity. CE social events give lab members the chance to interact outside work hours and have fun together. This made the work environment to feel more than just a work place and for this I specially thank to Koen Bertels, who always encouraged such activities. I am also grateful to the organizers of the diverse social outings, Mafalda, Mahroo, Kamana and Mihai.

I need to thank also to my Monday football team members Bogdan, Wouter, Christian, Manuel, Sebastian, Ilian, Mehdi, Robert and Fernando, with whom I enjoyed many outdoor matches in the TU Delft 'kunstgrass' competition. For the pool competition fun, I thank my fellow pool team members over the years Cristi, Pieter, Stein, Frank, Niels, Xavier, Michiel and Maurice. I want to thank our very close friends Dana and Alin, Anca and Dan, Mafalda and Vlad, Giacomo not only for great moments together, but also for their help and advice regarding technical details on how to raise a child. Furthermore, I am grateful to Mariana for her help during the intense months after the birth of my twin daughters, period in which I finalized this thesis.

I am extremely grateful for the big family I have and for all the support they have given me over the years. I have no words to describe the appreciation I have for all my parents and grandparents for creating the environment in which I was raised, for introducing me to mathematics since an early age or for en-

couraging and supporting me to study abroad. I am grateful to my parents and parents-in-law for their help and support in organizing precious family events and for their invaluable help with the children. I thank my sister Ana for looking at life from a non-technical perspective and for fully committing to what she believes in. I thank my brother Robert for all the great moments spent together and for the many more to come. Flori and Ionuţ, you are the best siblings-in-law I could wish for and I am very happy to have you in my life. I also thank our (wedding) godparents Remus and Dana for their support, help and for the many happy memories we have together. *Nasu'*, you are the first who suggested to me that doing a PhD is "not that bad", and I am immensely grateful for those talks in which I started to reconsider my future plans. Furthermore, I am very thankful to my extended family, Luci and Natalia, Andrei and Olesea, Mircea and Anca, Petre and Gina, who make each visit back home one to remember and each family excursion to the mountains a dream.

Last, but most important, I cannot even begin to sketch how much I appreciate all the love and support I have received during the Ph.D. study from my wife. I am forever grateful that she made me understand that home means much more than the place where you were born, that she is extremely understanding especially in not my best moments, for the many great memories we have together, but by far the most important, for being the best mother I could wish for the three beautiful and healthy daughters we have together. *Ti...ccm!*

I want to express my gratitude towards all the above once again as the words cannot express enough. I consider this dissertation the product of a truly wonderful journey that encompassed much more than the current text. It has been a journey filled with professional, personal and cultural realizations, with plenty of ups and downs, period in which I did plenty of mistakes but from which I learned a lot. It is the the outcome of an extraordinary period in my life. I now look happily towards the future and for the new challenges it will bring. However, until then, *'Carpe diem!'*

Răzvan Nane Delft, The Netherlands, April 2014

Table of contents

List of Tables

List of Figures

List of Listings

List of Acronyms and Symbols

ASIC Application-Specific Integrated Circuit

CCU Custom Computing Unit

CDFG Control Data Flow Graph

CSP Communication Sequential Processes

DDG Data Dependency Graph

CPU Central Processing Unit

CSE Common Subexpression Elimination

DSE Design Space Exploration

DSL Domain-Specific Language

DSP Digital Signal Processor

DWARV Delft Workbench Automated Reconfigurable VHDL Generator

DWB Delft Workbench

ELF Executable and Linkable Format

FF Flip Flop

FMax Maximum Frequency

FP Floating-Point

FPGA Field-Programmable Gate Array

FSM Finite State Machine

gcc GNU Compiler Collection

GPL General-Purpose Language

GPP General-Purpose Processor

GPU Graphics Processing Unit

GUI Graphical User Interface

HDL Hardware Description Language

HdS Hardware-dependent Software

HLL High-Level Language

HLS High-Level Synthesis

HW/SW Hardware/Software

ILP Instruction Level Parallelism

IT Information Technology

IR Intermediate Representation

IP Intellectual Property

ISA Instruction Set Architecture

LLVM Low Level Virtual Machine

LUT Lookup Table

RA Reconfigurable Architecture

RC Reconfigurable Computing

RTL Register Transfer Level

SaPA Speculative and Predicative Algorithm

SoC System on Chip

SSA Static Single Assignment

VLIW Very Long Instruction Word

VHDL VHSIC Hardware Description Language

VHSIC Very-High-Speed Integrated Circuits

XREG eXchange Register

1 Introduction

COMPILERS are nowadays an indispensable software tool and are one of the enablers behind the exponential growth of the Information Technology (IT) domain in the last decades. A **compiler** can be defined as software that performs a translation of a code written in a high-level language (HLL) to a different, lower-level language, which is closer to the specific representation used by the target computer. The importance of having compilers was immediately clear after the appearance of the first computers on the market, i.e., the Atanasoff-Berry Computer (ABC) [87] and the ENIAC [67] in the 1940s, for which the low-level manual programming method (i.e., configuring the computers to perform useful work) was very time consuming besides being error-prone. As a consequence of the difficulty of writing programs in the lowest-level programming language, the idea of higher abstractions appeared. Subsequently, an automated process (the compiler) would translate from the abstract language to the low-level language.

Since Grace Hopper designed the first compiler for the A-0 System language in 1952, a substantial number of compilers have been implemented and released by the software community for an increasing number of high-level programming languages. From these compilers, it is worth mentioning the FORTRAN compiler designed by John Backus for the IBM-704 computer in 1954-57, the ALGOL58 compiler for the first general imperative language in 1958, and the first cross-compiler COBOL demonstrated on the UNIVAC II computer in 1960. This first generation of compilers influenced all subsequent compilers, such as Ada, C, Pascal, Simula. The increasing number of available compilers and high-level languages, coupled with the fast increasing processor frequencies, the decreasing price for hardware resources, and the invention of the internet are the main reasons for the wide-spread adoption of general-purpose computers in the late 1980s. The *under-the-hood general* Central Processing Unit (CPU)s of these computers is what made the IT domain one of

the biggest technological revolutions of the 20th century. Nevertheless, this progress would not have been possible without the availability of high-level abstraction languages and associated compilers that hid the complexity of programming these general-purpose machines and that allowed for the fast creation of general-purpose software by a wide range of engineers.

However, by the first decade of the 21st century, the frequency scaling problem of a CPU was becoming more evident as the size of the elemental unit of hardware, i.e., the transistor, was reaching its threshold value. At the same time, the demand for computational power was growing higher than ever before because every industry was adopting IT. Until recently, the increasing processing requirements were satisfied by increasing the frequency of the CPU. As this becomes increasingly difficult to achieve, new solutions to maintain the same performance increase per year ratio are investigated. One straightforward solution is to increase the number of processing units, i.e., homogeneous multi-core computing. Unfortunately, this approach does not always scale. For example, for single applications containing large parts of parallelizable code, increasing the number of cores beyond a small amount of cores (e.g., four cores) does not increase the application's performance further. The main reason for this performance wall is the communication overhead, which increases greatly as the number of cores increases and end up taking more time than the actual computations [23]. Furthermore, the fixed amount of computational resources on CPUs is also a limiting factor in the possible speedup that can be achieved on these multi-core platforms. These problems, coupled with the drastic decrease in price for the transistor, which led to the possibility of directly using hardware as a general-purpose platform, made heterogeneous computing an economically feasible alternative.

A heterogeneous computing system can be defined as an electronic system that is composed of different types of computational elements or cores, with each core being able to perform a different set of tasks than the others. What makes this approach more flexible and has the potential to increase the system performance beyond the wall that homogeneous systems hit, is that some of the cores used in a heterogeneous system do not have predefined, generic, execution pipeline stages that are needed to work for every scenario. Instead, these cores can be programmed on the fly for the specific functionality required and can allocate as many hardware resources as needed. This is particularly true for Reconfigurable Architecture (RA)s applications used mostly in the embedded systems domain. However, the programmability of these new systems that can reconfigure based on system requirements poses major challenges, similar to how software compilers had their own challenges when they first appeared

more than 50 years ago; and as the history had taught us, the success and rate of adoption of these heterogeneous systems depends greatly on the maturity of tools (i.e., compilers) that do allows us to program them easily. Therefore, in this thesis we will address some of the issues regarding hardware compilers for RA using applications from the embedded systems domain.

1.1 Problem Overview

Heterogeneous systems can be considered the next evolutionary step in the history of (high-performance) computers after homogeneous systems. Their advantage is the combination of general-purpose processors with *predefined* specific accelerators to perform the expensive (i.e., time-consuming) computations for a particular (set of) application(s), thus increasing the overall system performance by delegating the computationally intensive tasks to those specific accelerators (cores). However, designing *predefined* heterogeneous systems is not always enough to guarantee their success. One of the most widely known examples of a heterogeneous system is the IBM's Cell Broadband Engine processor [46]. Although the heterogeneous approach offers more flexibility and higher performance than the standard homogeneous multi-core computing, the lack of reconfigurability of these architectures is still restrictive when it comes to performing well for various classes of computations. Consequently, the adoption of such a system can be prohibited by its high application development cost which cannot be amortized. Furthermore, implementing new algorithms on a predefined architecture can be also a very time consuming task. In our opinion, even though the Cell processor was a success for the Playstation 3, because the Cell architecture did not include reconfigurable hardware to allow for a different utilization of resources, it could not be easily applied in other types of applications. Although, at that time, due to the lack of mature tools and languages to program reconfigurable devices, supporting reconfigurability wouldn't have had a different impact on the outcome of the Cell processor, which was abandoned in 2009, the story of the Cell architecture showed the advantages and the need to design reconfigurable devices.

Reconfigurable computing can be defined as a heterogeneous computer architecture with increased flexibility by allowing the specific hardware accelerator resources available on the system to be reconfigured. The concept of reconfigurable computing was introduced by computer scientist Gerald Estrin in the 1960s [25]. However, due to the lack of reconfigurable hardware available that could be used for general-purpose applications, research for this type

of computing platforms stagnated until the second part of the 1990s. With the appearance of FPGA devices that could be reconfigured and were not expensive for general-purpose usage, the stage was set for a renaissance in this area. One of the first reconfigurable systems to be designed was the Garp processor [17] from Berkeley University in 1997. The success of this research project marked the shift from homogeneous to heterogeneous reconfigurable systems, and in the first decade of the 21st century a number of academic reconfigurable processor architectures were proposed.

MOLEN Machine Organisation [24, 73] is a reconfigurable architecture developed at TU Delft and one of those first heterogeneous reconfigurable systems introduced in the 2000s. The MOLEN *programming paradigm* consists of a one-time extension of the Instruction Set Architecture (ISA) to implement arbitrary functionality. In this thesis, we employ MOLEN as the reconfigurable platform on which all experiments will be performed. This machine organization will be described in detail in Chapter 3. For the scope of this introduction, it is sufficient to understand that this architecture is essentially composed of a CPU tightly connected to an FPGA, exchanging data via a shared memory. Figure 1.1 depicts the **Delft Workbench** tool-chain showing a simplified high-level overview of the steps involved in programming both the software and the hardware parts of the MOLEN machine illustrated by the *Heterogeneous Hardware Platform* box on the bottom of the figure.

One of the Molen objectives is to improve the performance of legacy code. Starting from an application completely written in a HLL, i.e., C in this particular case denoted by **.c* box, the first step is to profile the application and identify the spots in the code that have high computational demands. In this work, unless stated otherwise, the application under discussion is written in C. The result of the *Profiling and Cost Estimation* execution will identify hotspots that are good candidates for acceleration when moving these parts to hardware. Based on a *Quantitative Model* that contains information about how to quantify the amount of computational resources required by a particular piece of code and how many resources it would allocate, coupled with particular *User Directives* that indicate how to interpret those quantifications in terms of what can be moved to hardware and what should not, the next step in the tool-chain restructures the application. This is denoted by the *C2C: application restructuring* box in the figure, which transforms the code in such a way that further tools down the tool-chain can process it. *MOLEN Compiler* is the tool that compiles the software part, outputting an assembly file **.s* instrumented with calls to hardware. These calls are set up according to a predefined *Architecture Description* that contains information regarding sizes of exchange registers

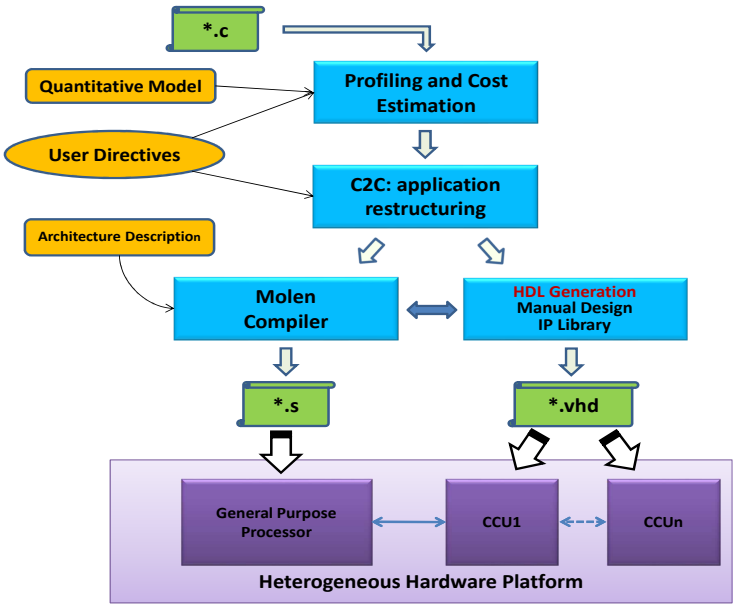


Figure 1.1: High-Level Overview of Tool-Chain Used to Program MOLEN.

(i.e., registers used to transfer function parameter values to/from hardware), memory data and address sizes and memory access times (i.e., the number of cycles required to access the memory).

To obtain the hardware design required to implement the custom hardware logic depicted in the figure by the Custom Computing Unit (CCU) boxes, three approaches can be used. The first is to use already available, i.e., off-the-shelf, possibly third-party IP cores from an existing *IP Library*. This is the easiest method offering a balanced trade-off between the core performance and the time spent to obtain the hardware solution. However, this method is not always available or satisfactory from a performance point of view. Therefore, a second option is to *manually design* the required functionality. This gives the best performance, and it is usually used for highly critical applications for which *automated* generation of the hardware is not satisfactory. However, this requires long design times that conflict with nowadays decreasingly smaller times-to-market. Therefore, this option is also gradually becoming unavailable. This fact leaves automation, i.e., *HDL Generation*, as the only viable solution to obtain hardware solutions for and from software programs.

However, currently there is a substantial gap between the performance obtained with manual implementation versus automated generation. The main

reason is that **"programming" the hardware** is not trivial. Several issues regarding the programmability of the hardware are addressed in this work, such as what optimizations are necessary, how to map software constructs to hardware logic elements, how to integrate hardware compilers in large tool-chains and others. The main challenges addressed in this work are summarized in the next section. Similar to the research performed in software compilers from more than five decades ago, likewise research and optimizations are necessary to close the gap between automatically generated hardware and manual designs as it is still the case today. Therefore, in this thesis we focus on the development, optimization, and integration of a hardware compiler.

1.1.1 Dissertation Scope and Challenges

The work performed in the scope of this dissertation was conducted within the seventh Framework Programme (FP7) REFLECT [70] and the Medea+ SoftSoc European Union (EU) projects. The first project focused on a holistic approach to integrate the concept of software "*aspects*" into the software/hardware co-design flow by developing, implementing, and evaluating a novel compilation and synthesis system approach for FPGA-based platforms. The REFLECTs approach intended to solve some of the problems that appear when efficiently mapping computations to FPGA-based systems. In particular, the use of aspects and strategies was proposed to allow developers to try different design patterns and to achieve solutions design-guided by non-functional requirements. In this respect, the need of a modular and easily extendable hardware compiler was essential to allow the run-time adaptation of the hardware generation process based on different aspect requirements that implied that different selections and orderings of compiler optimizations are possible. The second project, SoftSoC, aimed at solving the main System on Chip (SoC) productivity bottleneck by providing Hardware-dependent Software (HdS)[1] solutions to enable SoC designers to aggregate multiple HW IPs with their associated HdS into an efficient design. Concretely, a method was sought to allow a seamless integration of different party tools based on HdS and IP-XACT [1] descriptions. IP-XACT is a XML-based standard to describe hardware, i.e., Intellectual Property (IP) cores, to facilitate a seamless integration in third-party SoC. One particular case study investigated how to integrate two orthogonal computational models, namely DWARV2.0 respectively Compaan Design (described in Chapter 5), using the above mentioned descriptions. The computational models differ in the way they treat the memory, i.e., the former tool

[1] HdS is an IP *(software) driver*. These two definitions are used interchangeably in the text.

assumes a shared memory interface, whereas the latter assumes a distributed memory model.

Therefore, the challenges addressed in this thesis can be directly derived from a subset of goals of the above-mentioned projects and can be summarized as follows:

1. Analyze, design, and implement a highly modular hardware compiler that can be seamlessly extended with new or existing optimizations. Furthermore, the compiler should allow integration of external modules to facilitate an aspect-oriented design methodology.
2. Analyze, test, and propose a first set of IP-XACT extensions to support modeling of HdS in order to facilitate the automatic integration of generated hardware descriptions into large multi-vendor IPs SoC projects. Furthermore, the implications regarding the support available in a hardware compiler should be studied.
3. Analyze how area constraints are propagated through a hardware compiler. Concretely, investigate and devise an optimization model that supports the propagation of area constraints to the final generated HDL code output.
4. Analyze what well-known software optimizations can be applied to hardware generation. Look at classes of software optimizations and study if, how, and when these are beneficial in a hardware context. At the same time, consider individual optimizations and investigate how they should be changed given the new hardware context in which more resources became available.
5. Provide an overview and extensive comparison of different hardware compilers, both commercial and academic.

1.1.2 Contribution of the thesis

The main contributions of the work proposed in this dissertation are directly related to the described challenges. The following list briefly describes the contributions, where each numbered contribution corresponds exactly to the challenges with the same number from the previous list:

1. Design, implement, and evaluate a new research compiler based on the CoSy commercial compiler framework. This new version of DWARV

has a higher coverage of accepted C-language constructs. This is partially because the underlying compiler framework offers standard lowering (i.e., from high-level to low-level constructs mapping) transformations, which essentially allow the developer to implement just the important hardware primitives (e.g., goto state) from which all high-level constructs are composed. Furthermore, using CoSy, we obtain a highly robust and modular compiler that can be integrated in different tool-chains by extending it with custom compiler transformations to process third party information (e.g., coming from aspect oriented descriptions) and configure the process of hardware generation accordingly. We validate and demonstrate the performance of the DWARV2.0 compiler against another state-of-the-art research compiler. We show kernel-wise performance improvements up to 4.41x compared to LegUp 2.0 compiler [18].

2. Propose HdS based IP-XACT extensions and show how hardware kernels can be integrated into third party tool(-chains) automatically by using such descriptions. Therefore, we elaborate on the expressiveness of IP-XACT for describing HdS meta-data. Furthermore, we address the automation of HdS generation in the Reconfigurable Computing (RC) field, where IPs and their associated HdS are generated on the fly, and, therefore, are not fully predefined. We combine in this respect two proven technologies used in MPSoC design, namely IP-XACT and HdS, to integrate automatically different architectural templates used in RC systems. We investigate and propose a first set of three IP-XACT extensions to allow this automatic generation and integration of HdS in RC tool-chains.

3. Propose for streaming applications, i.e., loop-based, an optimization to control the unroll factor and the number of components, e.g., Floating-Point (FP) cores, when the area available for the kernel is limited. We assume thus that the hardware area for which a to be generated hardware accelerator is limited. In this respect, two important parameters have to be explored, namely the degree of parallelism (i.e., the loop unrolling factor) and the number of functional modules (e.g., FP operations) used to implement the source HLL code. Determining without any human intervention these parameters is a key factor in building efficient HLL-to-HDL compilers and implicitly any Design Space Exploration (DSE) tool. To solve this problem, we propose an optimization algorithm to compute the above parameters automatically. This optimization is added as an extension to the DWARV2.0 hardware compiler.

4. Propose for control based applications, i.e., executing path selection statements, a predication scheme suitable and generally applicable for hardware compilers called Speculative and Predicative Algorithm (SaPA). This technique takes into account the characteristics of a C-to-VHDL compiler and the features available on the target platform. Instruction predication is a well-known compiler optimization technique, however, current C-to-VHDL compilers do not take full advantage of the possibilities offered by this optimization. More specifically, we propose a method to increase performance in the case of unbalanced if-then-else branches. These types of branches are problematic because, when the jump instructions are removed for the predicated execution, if the shorter branch is taken, slowdowns occur because (useless) instructions from the longer branch still need to be executed. Based on both synthetic and real world applications we show that our algorithm does not substantially increase the resource usage while the execution time is reduced in all cases for which it is applied.

5. Provide an extensive evaluation of state-of-the-art hardware compilers against DWARV3.0. At the same time, a thorough retrospection of existing high-level tools has been performed. The comparison included a number of hardware compilers that comply with some predefined criteria in which DWARV can be included, as well. In particular, we looked at VivadoHLS, another ComercialCompiler, LegUp2.0 and 3.0, PandA 0.9.0 and 0.9.1, and two versions of DWARV, i.e. 2.0 and 3.0. The results obtained will show how all these compilers compare to Vivado HLS, which on average generated the most efficient hardware.

1.2 Dissertation Organization

The work described in this dissertation is organized in 10 chapters. Figure 1.2 highlights the chapters by relating them visually to the addressed challenges and the specific contributions made, while showing the chapter connections to the published papers and journals in the scope of this dissertation. The oval box represents a conference proceeding while the hexagon represents a journal publication. Furthermore, incoming chapter edges depict the fact that the source chapter was published in the target publication while the reverse represent the fact that the source publication was based on the target chapter. The dotted hexagon on the bottom of the figure represents the fact that the publication is submitted. The topic of each chapter is described below.

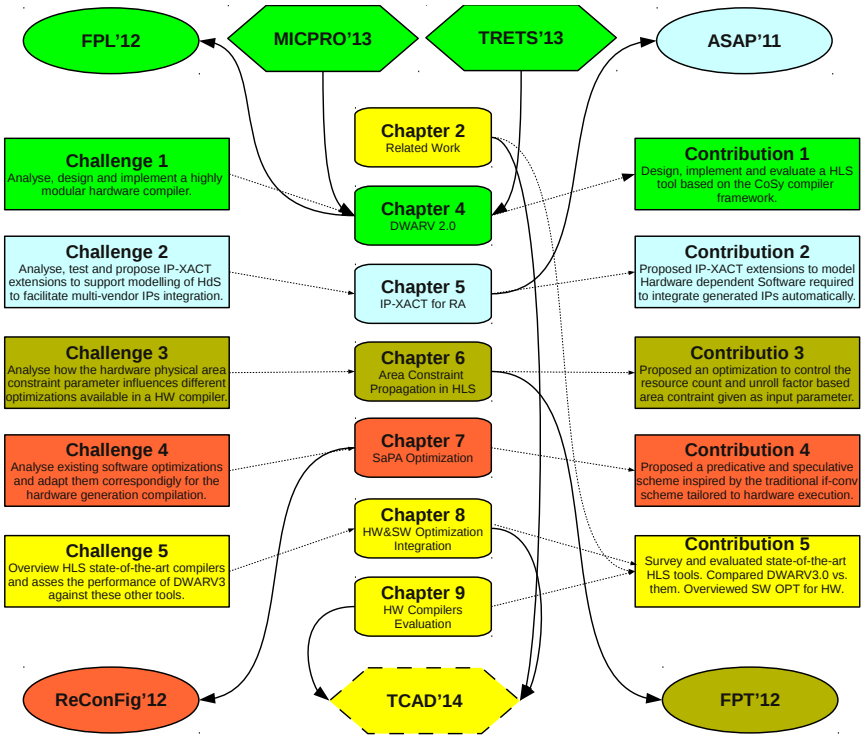


Figure 1.2: Overview of the Connections Between Challenges, Chapters, Contributions and Publications.

In Chapter 2, we present related work in which we describe past and present hardware compilers. The survey will be categorised based on the design (i.e., accepted) input language in domain-specific compilers respectively generic HLS tools. The particular tool description includes information such as for what application domain the tool can be used, what extensions are required, if the tool offers verification support, as well as under what type of license it is available (commercial or academic). Finally, we show for each tool the differences versus DWARV.

In Chapter 3, the previous (i.e., background) work is presented. We describe the Delft Workbench tool-chain, the Molen Machine Organization, and the simulation and synthesis flows used to validate and implement automatically generated hardware designs. Subsequently, we discuss important similarities and differences between software and hardware compilers, after which, we present the first version of the DWARV compiler that provided the inspiration for the current version. We also present the CoSy compiler framework used

to implement the new version of DWARV. Finally, we describe the complete C-to-FPGA tool-flow based on a simple example.

In Chapter 4, we describe DWARV2.0, the first DWARV version implemented in CoSy. The performance of the new version will be benchmarked by comparing and evaluating it against the LegUp 2.0 academic compiler.

Chapter 5 presents the HdS IP-XACT based extensions required when generating code for RC applications. These extensions are needed because the current IP-XACT standard supports only hardware modeling (i.e., IP related), but it does not allow to model software, that is, to model IP drivers that are required to integrate generated hardware automatically in SoC. The IP-XACT standard is used to facilitate the automatic integration of existing hardware components used by hardware designers in SoC design.

In Chapter 6, an optimization algorithm to generate hardware kernels subject to input area constraints is presented. These area constraints are highly important in the Molen context, where we can have a maximum number of accelerators that can be executed in parallel by a specific architecture implementation. In this respect, generating hardware accelerators that can fit these a prior defined FPGA slots is very important.

In Chapter 7, we present another hardware specific optimization. This optimization, called SaPA, is based on a relaxation of the traditional software if-conversion technique. The results obtained indicate that this optimization could be universally applied in each hardware compiler, because it does not decrease the accelerator performance (not even in unbalanced if-then-else cases), while, at the same time, the hardware area is negligibly increased.

In Chapter 8, we present important hardware optimizations that allowed us to optimize DWARV2.0 by a factor of 2x to 3x. Furthermore, we present current work oriented towards the automation of selecting and integrating optimizations in a compiler on a case by case basis. The reason behind this work is the fact that including existing standard optimizations randomly in a compiler is not a recipe for success. The order in which these are applied and how they are configured play a very important role, as well.

Finally, Chapter 9 will show comparison results for DWARV3.0 against a newer version of LegUp (i.e. LegUp 3.0) and other three compilers, i.e. Vivado HLS, PandA 0.9.1 and another *CommercialCompiler* (*CC*[2]). Conclusions are presented in Chapter 10, where we summarize the main contributions of this thesis, and we propose a list of open questions and future research directions.

[2]CC is not a real name. This is hidden to avoid license issues w.r.t publication rights

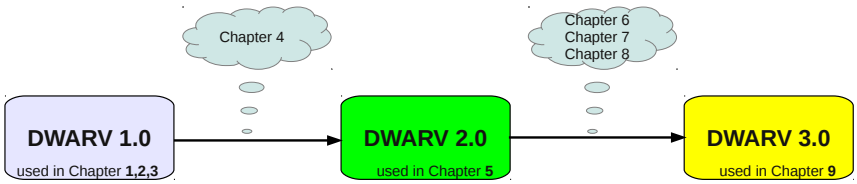


Figure 1.3: DWARV Version Evolution Based on Thesis Chapters.

A final remark is needed before describing each chapter to clarify the DWARV version scheme. Figure 1.3 depicts DWARV's version evolution based on dissertation chapters. We see that three versions of DWARV were involved. We started with a legacy version of DWARV (i.e. 1.0), then, based on arguments described in Chapter 4 we reimplemented the compiler in the CoSy [26] compiler framework to obtain a second major version (i.e. DWARV2.0), to arrive in Chapter 9 at DWARV3.0 by implementing the optimizations described in Chapters 6 to 8.

2
Related Work

HARDWARE compilers that take as input a High-Level Language (HLL), e.g., C, and generate Hardware Description Language (HDL), e.g., VHDL, are maybe not a new but increasingly important research topic. These compilers have been used increasingly in the last decade to generate hardware for various application domains in order to accelerate the computationally intensive part(s), when adopting the Hardware/Software (HW/SW) co-design paradigm. One example is to speedup a MJPEG application by generating VHDL code for the DCT function (called also a kernel), synthesizing it, and merging the generated bit file with the Executable and Linkable Format (ELF) file generated by the software compiler for the rest of the application and running it on a mixed platform, i.e., processor (e.g., ARM, PowerPC) combined with a co-processor (e.g., FPGA) [73].

To do fast design space exploration of the different configuration options available and select the best mapping (i.e., HW/SW partitioning depending on the available area and required throughput), we need to be able to evaluate the hardware implementations for the different functions chosen for hardware execution. Performing this task by hand requires not only hardware design knowledge to implement these application functions in hardware, but also requires the developer to go through the typical iterative implement-test-debug-implement cycle, which is very time consuming. This, in turn, will drastically limit the effectiveness of the design space exploration analysis. As a result, the ever-increasing time-to-market pressure will not be reduced. A solution to this problem are hardware generators, referred to also as high-level synthesis tools, which are essentially HLL-to-HDL compilers. This allows the designer to immediately obtain a hardware implementation and skip the time-consuming iterative development cycle altogether.

2.1 High-Level Synthesis Tools

In this section, we present related research projects that addressed the process of automating HDL generation from HLLs. We will describe here important features such as supported input/output languages, underlying compiler framework upon which the tool has been built (where this information is available), and as a direct consequence, the optimizations available, the target application domains, support for floating- and/or fixed-point arithmetic, and if the tool supports automatic verification by means of automatic test bench generation. Therefore, in this chapter we will emphasize on the HLS state-of-the-art and describe how the DWARV compiler compares to this other work in the field. The goal is to show that our compiler, when compared with the others, accepts a large sub-set of unmodified C-language constructs, and that it generates code for any application domain code, which is one of the design goals behind DWARV. In subsequent chapters, we will show that DWARV is modular, and it can be easily extended by including two custom designed optimizations (Chapters 6 and 7), as well as that it has great potential for further improvement by adding standard CoSy framework optimizations (Chapter 8). Finally, Chapter 9 will show that DWARV3.0's performance, the final version at the time of writing this dissertation, is comparable with commercial compilers, and, that between the compared academic compilers, for the presented applications and requirements, it performs the best.

The tool presentation will be done according to a classification depending only on the design input language as shown in Figure 2.1. We distinguish between two major categories, namely tools that accept **Domain-Specific Language (DSL)s** and tools that are based on **General-Purpose Language (GPL)s**. DSLs are composed of *new languages* invented specially for a particular tool-flow and *C-based dialects*, which are languages based on C extended with pragmas/annotations to convey specific hardware information to the tool. GPLs are also split in two categories, namely *procedural languages* and *object-oriented languages*. Under each category, the corresponding tools are listed in green, red or blue fonts standing for in use, abandoned, respectively, no information is known about the status of the tool. Furthermore, the bullet type, defined in the figure's legend, denotes the target application domain for which the tool can be used. Finally, the underline in the figure means the tool supports also SystemC, that is a combination of both procedural and object-oriented language, extended with constructs to model hardware-specific information.

We emphasize that the focus in this chapter is on existing High-Level Synthesis

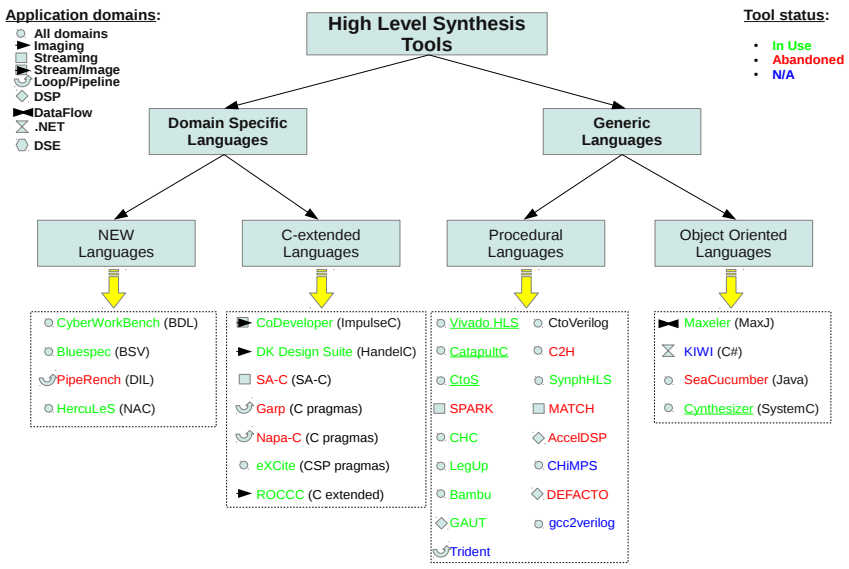


Figure 2.1: Classification of High-Level Synthesis Tools based on Input Language.

(HLS) tools. Complementary approaches intended for defining and programming the different heterogeneous (reconfigurable) hardware system components are considered generic and situated at a higher level of abstraction than the tools described in this work. Therefore, these multi-platform languages are not considered in the reminder of the chapter. Two such well-known programming languages not described here are OpenCL [36] and SystemC [50].

We present next the different compilers available, whereas Tables 2.1 and 2.2 in the next section summarizes all the important findings. However, before we start to describe each related compiler, it is important to note that all these tools differ in one important aspect from DWARV2.0. The tool proposed in this work is built upon a commercial compiler framework, the CoSy compiler framework, inheriting the advantage that it is extremely robust and flexible to extend. Furthermore, the large amount of framework optimizations offer a great potential for experimenting with already existing software optimizations. The CoSy compiler framework will be described in detail in Chapter 3.

2.1.1 Domain-Specific Languages

DSLs are languages created for a specific purpose. As a result, these can give very good results for the domain in which they are applied. However, their restricted applicability by construction limit the application domain. In our work, we target to support all application domains, and as such, our approach is orthogonal to DSLs.

2.1.1.1 New Languages

These are languages that are not based on any previous language or that resemble the syntax of an existing language, but, it adds many and complex extensions that require a considerable amount of time to be learned. The syntax and semantic (extensions) are defined from scratch to take advantage of tool-flow organization and hardware characteristics optimally.

CyberWorkBench

CyberWorkBench (CWB) [62, 94] is a set of synthesis, verification and simulation tools intended for the hardware system-level designer of very large Application-Specific Integrated Circuit (ASIC)s and System on Chip (SoC)s. The tool-set is offered by NEC, a Japanese multinational provider of information technology, since the beginning of the 21st century. However, actual HLS capabilities have been commercially available since 2011. The tool input is Behavioral Description Language (BDL), which is a super-set of the C language, extended with constructs to express hardware knowledge in the high-level description. For example, user-defined variables bit-width, synchronization, explicit clock boundaries specification, and concurrency constructs are some of these C language extensions. Furthermore, the programmer can express in BDL the mapping of variables to wires, arrays to memories or register files, the binding of modules or the amount of loop unrolling.

The synthesis flow of the CyberWorkBench offers the possibility to select between three types of scheduling approaches: fixed scheduling, automatic scheduling with resource sharing between alternative branches, and pipeline scheduling. The fixed scheduling is driven by the user-specified clock boundaries and strictly follows the control flow of the input description. The automatic allows concurrent execution between independent basic-blocks. The pipeline scheduler can be invoked for data-dominated descriptions and relies on user-specified pipeline initiation interval. The tool can generate both VHDL

and Verilog based hardware designs. Furthermore, two types of verification are supported, i.e., formal by running C-RTL equivalence checkers and informal by performing cycle-accurate simulations. The tool supports both floating and fixed-point arithmetic.

Bluespec

Bluespec Compiler (BSC) [13], developed by BlueSpec Inc. and available since 2007, is a tool that uses Bluespec SystemVerilog (BSV) as design language. BSV is essentially a high-level functional HDL based on Verilog and inspired by Haskell, where modules are implemented as a set of rules using Verilog syntax. The rules are called *Guarded Atomic Actions* and express behavior in the form of concurrent cooperating FSMs [63]. The use of these concepts make this language, and implicitly the BSC tool, appropriate only for developers that have hardware design knowledge. Furthermore, verification can be done only by manually writing test benches in BSV as well and connecting them to the generated designs. Although the company claims that it offers solutions very close to the performances and areas obtained by manual designs, the use of the tool requires both manual rewrites in the BSV language as well as hardware knowledge.

PipeRench

PipeRench [32] [85] project was also one of the first that proposed reconfigurable architectures. The research was done at Carnegie Mellon University around 2000. The PipeRench compiler was a restrictive one intended solely for pipeline reconfiguration and generation in stream-based media applications. The source language is a *dataflow intermediate language*, *DIL*, that is basically a single-assignment language with C operators. The output of the tool is a bit stream representing the generated pipeline. In the process of obtain this pipeline, the PipeRench compiler employs automatic bit width inference, unrolls all loops and decomposes operators that exceed the target cycle time.

HercuLeS

HercuLeS [51, 52] is a new commercial product offered by Ajax Compilers since 2013. HercuLeS targets whole-program hardware compilation featuring ease of extension through pluggable analyzes and optimizations. NAC (N-address code) is the IR used which is a new typed-assembly language created

by a frontend available through GCC Gimple. The tool generates RTL VHDL and self-checking VHDL test benches and it supports scalar, streaming and array ports. VHDL-2008 fixed point and IEEE-754 and custom floating point arithmetic can be generated as well. HercuLeS offers both frontend optimizations such as loop unrolling, array flattening through gcc and target specific optimizations such as operation chaining.

2.1.1.2 C-dialect Languages

These are languages that are based on a previous language extended with a few mechanisms (e.g., pragmas, keywords) to model hardware specific concepts such as concurrency. These extensions are fairly easy to learn and do not require a lot of time. Nevertheless, the fact that extensions are still required, the applicability of these languages is impaired, as well.

CoDeveloper - Impulse-C

CoDeveloper is the HLS design environment provided by Impulse Accelerated Technologies. This commercial product first released in 2003 includes an Impulse-C compiler, based on the SUIF compiler framework [89], and related library functions intended for FPGA-based applications. Impulse-C is the design language, and the commercialization of Streams-C [30] academic language developed in 2000 at Los Alamos National Laboratory. Impulse-C is based on a C-language subset and adds CSP style extensions required for parallel programming of mixed processor and FPGA platforms. The generated HDL output can be in the form of both VHDL or Verilog files. Because the basic principle of the CSP programming model consists of processes that have to be independently synchronized and streams through which communication between processes must be performed, the application domain is limited only to image processing and streaming applications. Hence, applications that cannot be described in this model are not supported. In addition, the parallelization of the algorithm has to be performed manually. The communication between the processes and the streams implementation also has to be specified explicitly through pragmas. Therefore, accelerating existing C applications in the context of software/hardware co-execution is not a trivial task because both manual rewrites as well as learning a new programming language are necessary before the application can be compiled.

The tool supports several optimizations such as loop-invariant code motions, common sub-expression elimination, constant propagation or constant fold-

ing. Furthermore, floating point operation can be supported through external libraries. However, fixed point arithmetic is not permitted. Finally, CoDeveloper's CoValidator tool offers automatic verification capabilities by means of generating test vectors and HDL test bench only for stream (co_stream) interfaces as well as scripts to invoke ModelSim for simulating the test bench.

DK Design Suite - Handel-C

DK Design Suite [33] from Mentor Graphics is an integrated environment that since the acquisition of Agility in 2009 includes HLS capabilities by being able to generate VHDL/Verilog from HLL descriptions. The design language is Handel-C [34], first developed at Oxford University in 1996, and which is based on a rich subset of the C language, but extended with language constructs required to aid the hardware synthesis process. Using these extensions, the user needs to specify explicit timing requirements, and to describe the parallelization and synchronization segments in the code explicitly. In addition, the data mapping to different memories has to be manually performed. Because of these language additions, the user needs advanced hardware knowledge. Therefore, the tool is oriented more towards the hardware/FPGA designer rather than the software developer.

The Handel-C input language does not support floating point types. However, the programmer can define data types with variable widths for fixed-point arithmetic. Because Handel-C is based on the Communicating Sequential Process (CSP) programming model, any original C-code has to be rewritten not only to add the Handel-C language directives, but has also to be structurally modified to cope with concepts such as combinational loops, i.e., breaking them by adding extra delay statements in the code on undefined if-else paths. Furthermore, because of the underlying CSP model, the application domain is oriented towards streaming applications. Finally, the user manual downloaded did not describe neither if automated verification through test bench generation is possible nor what hardware compiler optimizations are available. Therefore, using this tool is not trivial and is not intended for the general use considered in this work.

Single-Assignment C

Single-Assignment C (SA-C) [61] is a C language variant in which variables can be set only once, when the variable is declared. The language and its accompanied hardware compiler were developed in 2003 primarily at Colorado State

University. This work provided the inspiration for the later ROCCC compiler. Given that image processing algorithms were the target application domain, this work falls into the category of compilers that have the application domain drawback, making it thus not comparable with DWARV2.0. Furthermore, the language introduces new syntactical constructs, which require application rewriting. Another big limitation is the fact that it did not accept pointers. The authors of SA-C describe it as a language that is the closest to Streams-C, but with the difference that their work focuses on loops and arrays and not on streams and processes. The SA-C compiler included many optimizations to reduce circuit size and propagation delay by performing constant folding, operator-strength reduction, dead-code elimination, invariant-code motion and common subexpression elimination. The output of the compiler was VHDL. However, it did not offer any verification capabilities nor floating or fixed point arithmetic support.

Garp

The *Garp* [17] architecture and C compiler were developed in 2000 at Berkeley University. The main goal of the project was to accelerate loops of general-purpose software applications. It accepts C as input and generates a bitstream for the actual loop module. The compilation process implemented in the SUIF compiler framework tackled two challenges, namely, excess code in loop bodies and how to extract Instruction Level Parallelism (ILP) from sequential code. The solution taken was very similar to those chosen in Very Long Instruction Word (VLIW) processors, and it was based on the hyperblock concept. Advanced techniques such as predication, speculative loads, pipelining and memory queues were employed to obtain efficient designs.

Napa-C

Napa-C [31] project was one of the first to consider high-level compilation for systems which contain both a microprocessor and reconfigurable logic. The Sarnoff Corporation conducted this project around 1998. The Napa-C language was a C variant that provided pragma directives so that the programmer (or an automatic partitioner) can specify where data is to reside and where computation is to occur with statement-level granularity. The NAPA C compiler, implemented in SUIF and targeting National Semiconductor's NAPA1000 chip, performed semantic analysis of the pragma-annotated program and co-synthesized a conventional program executable combined with a

configuration bit stream for the adaptive logic. Loop pipelining was a powerful optimization that Napa-C compiler employed. However, being one chip target specific language, several language restrictions were present, such as pointers usage and control constructs not being allowed. Furthermore, no floating or fixed point operations were possible.

eXCite

eXCite [28] from Y Explorations is one of the first HLS tools available since 2001. The tool distinguishes itself by starting from a C-input that has to be manually partitioned with the help of pragmas and select what parts are to become hardware (both VHDL and Verilog RTL code supported). To perform the communication between the software and hardware communication channels have to be inserted manually as well. This is one of the most important tasks the user has to perform. These channels can be streaming, blocking or indexed (e.g., arrays). Although different types of communications between the software and hardware parts (e.g., streaming, shared memory) are possible, because the channel insertion is done manually, this step is time consuming and requires the original application code to be modified.

eXCite support automated verifications by means of testbench generation that is automatically created from the HLL application after the synthesis step. This testbench can then be used with any RTL simulation tool to verify the same inputs and outputs that were tested on the C behavior. The tool offers also a number of powerful optimizations that can be fine-tuned, e.g., pipelining, bit reduction, constant folding, loop flattening, algebraic eliminations or common subexpression elimination.

ROCCC

The ***R**iverside **O**ptimizing **C**onfigurable **C**omputing **C**ompiler* was one of the first academic high-level synthesis tools, developed at University of California, Riverside, in 2005. The first version of the compiler [38] [39] was built using SUIF2 [89] and Machine-SUIF [86] compiler frameworks from Stanford respectively Harvard Universities. The project focused mainly on the parallelization of the high computational intensity parts within low control density applications. This restricts the application domain to streaming applications mostly, and it means that the input C language accepted must be restricted only to a small subset of the C-language. For example, only perfectly nested loops with fixed stride, operating on integer arrays are allowed. Other examples of

not allowed C-constructs include generic pointers, non-for loops, shifting by a variable amount, multidimensional arrays or stream accesses other than those based on a constant offset from loop induction variables. This last restriction is needed to facilitate the generation of *smart buffers*, which can be defined as customizable and reconfigurable caches in the hardware for the fetched memory values. This is a powerful concept that allows the optimization of the memory sub-system by enabling to fetch live variables (i.e variables that will be used again) only once.

In 2010, the tool underwent a major revision to transition to ROCCC2.0 [93]. At the same time, the tool was branched from being an academic tool to a commercial one offered by Jacquard Computing Inc. [48]. One of the major modifications was the replacement of the underlying compiler framework from SUIF and Machine-SUIF, which were no longer supported, to LLVM [66]. Although the new tool accepts the same fixed C subset and generates VHDL code as well, major improvements have been done regarding modularity and reusability. In the new version, VHDL can be generated for modules or for systems. Modules represent concrete hardware implementations of purely computational functions and can be constructed using instantiations of previously defined modules in order to create larger components. System code performs repeated computations on streams of data implemented as loops that iterate over arrays. System code may or may not instantiate modules, represent the topmost perspective, and generate hardware that interfaces to memory systems through array accesses.

In ROCCC2.0, a number of system specific optimizations (e.g., loop fusion, loop interchange, loop unrolling or temporal common sub-expression elimination), optimizations for both systems and modules (e.g., multiply by const elimination, inline modules, division by const elimination) and low level optimizations (e.g., arithmetic balancing, maximize precision) are available. Furthermore, floating point operations are supported by the use of library components that need to be described in the ROCCC's internal database. However, neither fixed point arithmetic is supported, nor automatic verification by means of automatic testbench generation is possible.

2.1.2 General-Purpose Languages

GPLs are, as the name suggests, existing languages that are used for any application domain.

2.1.2.1 Procedural Languages

Procedural, or imperative, languages are languages where the computations are expressed in functions using abstractions of machine code instructions and executed sequentially. The C-language is used by the majority of the presented tools in this section.

Catapult-C

Catapult-C [78] is a commercial high-level synthesis tool released by Mentor Graphics in 2004, but currently maintained by Calypto Design Systems which acquired it in 2011. Initially oriented towards the ASIC hardware developer, over time, the product has become a powerful, complex but also complete design environment targeting both the ASIC and FPGA hardware design flows. It offers high flexibility in choosing the target technology, external libraries, setting the design clock frequency, mapping function parameters to either register, RAM, ROM or streaming interfaces to name just a few options. Furthermore, it accepts unrestricted ANSI C/C++ and SystemC inputs and generates both VHDL and Verilog register transfer level (RTL) netlists as well as SystemC code.

However, from the perspective of a software programmer, learning to use the tool can be a difficult process given that little automated support is provided by the tool in selecting which optimizations are best to apply. For example, the memory organization and communication protocols have to be defined, resources have to be selected and constrained, code transformations such as loop unrolling, loop merging or loop pipelining have to be manually enabled. This amount of extensive designer input required by the tool with respect to both the applied optimizations and the actual mapping process makes it less viable for a software designer that does not have in-depth hardware knowledge. Nevertheless, good tool documentation and books [14] [29] exist on how to use Catapult-C.

An important feature of the tool is verification. This can generate test benches for ModelSim, for cycle accurate as well as RTL and gate level implementations of the design. On top of that, a verification test bench can be generated that combines the original C++ design and test bench with the generated design which applies the same input data to both designs and compares the output. This speeds up verification substantially. Furthermore, fixed point arithmetic is supported via SystemC fixed-point data types. However, Catapult-C will not convert float or double into floating point hardware but will convert these

operations to fixed point arithmetic. The developer will need to write a floating point class to implement the portion of the floating point arithmetic that is required for the design.

C-to-Silicon

C-to-Silicon (CtoS) [15] [16] is Cadence's high-level synthesis tool dating back to 2008. The tool does not restrict the application domain allowing for both control- and data-flow types of codes to be compiled, with the exceptions of a few minor syntax constructs such as post-incremental arithmetic or multi-dimensional pointers. CtoS can generate different interface types, that is, it can pass function array parameters both as a flat structure or implement an addressing scheme by either sending read and write requests for specific indexes or streaming the array into the design. Loop related optimizations (e.g., loop pipelining, loop unrolling) are the only compiler optimizations possible, and they have to be selected and configured manually by the user. Floating-point operations, variables, constants, and literal values are not supported and they must be replaced with integer approximations or emulated via synthesized integer arithmetic. However, arbitrary bit-width and fixed-point data types are supported through SystemC data types.

CtoS accepts either timed or untimed applications written in C, C++ or SystemC, and it outputs IEEE-1364 Verilog. However, if the input is not SystemC, the tool will generate a SystemC wrapper for the corresponding input files using the *import function design flow*. This separate flow for non-SystemC applications causes the verification flow to be tool-dependent because the generated SystemC wrappers contain extensions that do not comply with the OSCI SystemC standard. As a result, the verification wrapper can be executed only by Cadence IES - Incisiv Entreprise Simulator (SimVision tool). Furthermore, the tool can perform only cycle-accurate verification by means of cycle-accurate test benches, which implies that automated verification based on test bench generation is not possible because if designs change timing (e.g., we want to explore a different unroll factor; therefore, we need to schedule operations differently), the test bench would need to be (manually) changed as well. More precisely, each time we modify the design we would need to obtain accurate latency and throughput numbers to rewrite the test bench accordingly. However, obtaining these numbers for data-dependent applications is not possible, a typical scenario in HLS.

Although imposing SystemC as the main design language and requiring cycle-accurate test benches renders possible a large number of system configurations

(e.g., implementation of different communication protocols between different functions/modules), this highly increases the complexity of the tool as well, making it accessible only for the hardware/ASIC designer. Furthermore, the documentation is not always clear and terms such as combinational loops are not well defined, resulting in a steep learning curve. According to the documentation, "A loop is said to be combinational if at least one program execution from the top to the bottom of the loop does not include waiting for a clock edge." and "Combinational loops must be eliminated before synthesis because they cannot be implemented in hardware. CtoS considers a loop to be combinational if it cannot prove at compile time that any execution from the top to the bottom of the loop takes more than one clock cycle." Furthermore, "some functions can not be synthesizable because different paths through the function require a different number of clock cycles to execute. This can be resolved by inlining the function or by balancing the paths through the function to have the same number of states". However, doing this manually is not always easy nor feasible given limited hardware knowledge.

SPARK

SPARK [41, 84] is a modular and extensible high-level synthesis research system developed at University of California at Irvine in 2003. The SPARK compiler accepts ANSI-C without any restrictions on the input code, and it generates RTL VHDL code. The main goal of the project was particularly targeted to multimedia and image processing applications along with control-intensive microprocessor functional blocks. Hence, they provide support for broader application domains. The optimizations set of the SPARK compiler include frontend transformation such as loop unrolling, loop invariant code motion, copy propagation, inlining, dead code elimination, but also specific scheduling optimizations such as percolation/trailblazing speculative code motions, chaining across conditions or dynamic cse. However, similar to related academic projects of the time, no automated verification of the generated hardware design is possible. In addition, explicit specification of the available hardware resources like adders, multipliers, etc. is required. Finally, no floating or fixed point data types are allowed.

C to Hardware

Altium's *C to Hardware Compiler (CHC)* [8] is a feature available since 2008 inside the Altium Designer integrated design environment. It contains two

operation modes. The default mode synthesizes all functions to hardware as connected modules, whereas the second works in the context of an Altium's *Application Specific Processor (ASP)* defined processing core. This is where accelerators are offloaded from the processor side onto hardware. This is accomplished either by Graphical User Interface (GUI) interaction in the Altium Designer IDE, or by code transformations. The input application has to be specified in the C-language, and the tool generates VHDL or Verilog hardware descriptions. During the compilation process, fixed point arithmetic optimizations are performed to save hardware area. Floating point operations are also supported.

Altium offers a free evaluation license. Verification has to be done manually by loading and executing the generated design on an actual Altium Desktop NanoBoard NB2DSK01 because no simulation capabilities are available to verify the correctness of the generated hardware. Furthermore, before any hardware design can be generated, a completely mapped and routed target platform has to be defined. This includes defining and configuring a TSK3000 which is the only processor supported to run the software application. Because this processor is based on a Harvard architecture, memories for both instructions and data have to be defined along with their interconnections. Finally, the required number of ASPs (i.e., drag and drop a WB_ASP core in the platform design sheet) executing the generated hardware and their memory connections have to be specified as well. As a consequence, the time to design a system in which generated hardware modules can be tested is very long, besides being technology dependent. This is a very important difference between our compiler and CHC as we neither require a complete system configuration to be able to generate hardware designs, nor we restrict the target platform to one particular board/processor combination.

Vivado HSL

Vivado HLS [49], former AutoPilot, was developed initially by AutoESL until it was acquired by Xilinx in 2011. The new improved product, which is based on LLVM as well, was released early 2013, and it includes a complete design environment with abundant features to fine-tune the generation process from HLL to HDL. All applications written in C, C++ and SystemC are accepted as input, and hardware modules are generated in VHDL, Verilog and SystemC. During the compilation process, different optimizations can be selected depending on the final goal. For example, operation chaining, loop pipelining and loop unrolling are some of the optimizations that influence the generated

design's performance. Furthermore, different parameter mappings to memory can be specified. Streaming or shared memory type interfaces are both supported to allow for both streaming and control domain application to be compiled. Because the provided GUI is simple and the drop-down menu options are well described, the time required to compile a function to hardware is minimal. Verification of the generated designs is fast as well due to the powerful test bench generation capabilities. Finally, both floating point and fixed point variables and arithmetic operations on these are supported.

LegUp

LegUp [18, 68] is a research compiler developed in 2011 at Toronto University using LLVM [66]. It accepts standard C-language as input and generates Verilog code for the selected input functions. Its main strength is that it can generate hardware for complete applications or only for specific application functions, i.e., accelerators. In this latter case, a TigerMIPS soft processor [65] is then used to execute the remainder of the application in software. The connection between these two main components is made through an Avalon system bus. This is similar to the Molen machine organization. The tool can automatically generate test benches which allows easy validation of the generated accelerators. Furthermore, the latest release of the tool (version 2.0 in 2013) accepts also floating point arithmetic. This makes the tool one of the research competitors for DWARV, and this tool will be used in later chapters to benchmark against.

Bambu

Bambu is a tool for the high-level synthesis currently under development since 2012 at Politecnico di Milano in the context of the PandA framework [22]. "It integrates compiler optimizations by interfacing with the GCC compiler and implements a novel memory architecture to synthesize complex C constructs (e.g.,, function calls, pointers, multi-dimensional arrays, structs) without requiring three-states for its implementation. It also integrates floating-point units and thus deals with different data types, generating the proper architectures. Moreover, it is also possible to target both ASIC and FPGA technologies by automatically generating customizable scripts for commercial logic and physical synthesis tools. It is also possible to generate different implementation solutions by trading off latency and resource occupation, to support the hardware/software partitioning on heterogeneous platforms. Finally, thanks to

its modular organization, it can be easily extended with new algorithms, architectures or methodologies, targeting different application domains or user's requirements. Constraints, options and synthesis scripts are easily configurable via XML files and it is also possible to generate test benches for automatically comparing the results with the software counterpart". The tool is available for download under a standard free license, and, being similar in design goals as well as how the hardware accelerators are generated, this compiler will be used in Chapter 9 to compare against the results obtained with DWARV3.0.

GAUT

GAUT [20] [21] from Universite de Bretagne-Sud is a HLS tool that generates VHDL from bit-accurate C/C++ specifications. GAUT was designed specially for DSP applications, and since it first appeared in 2010 it can be freely downloaded. A distinct feature for the tool is that besides the processing unit, i.e., the accelerator, GAUT can generate both communication and memory units. Furthermore, to validate the generated architecture, a testbench is automatically generated to apply stimulus to the design and to analyse the results. Fixed point arithmetic is supported through Mentor Graphics Algorithmic C class library. Although GAUT resembles DWARV, there are also major differences. First of all, because GAUT targets mostly the DSP application domain, its generated designs are pipelined and, therefore, offer a streaming interface to the outside system. This implies that all loops are fully unrolled, which is not always a good idea especially when there is a limit on the hardware area available. On the contrary, DWARV assumes a shared memory type of interface. Furthermore, experimenting with the tool revealed a few drawbacks. The function to be compiled has to be renamed to main with type void, and all original function parameters have to be made global. This requires application rewriting. No warning is given to understand why the compiler is not generating any code. Finally, array parameters are not supported and no warnings are given why the code cannot be compiled.

Trident

Trident [55] [82], developed by Los Alamos National Laboratory, is a research compiler available as open source since 2007. It builds on the Sea Cucumber compiler, and it generates VHDL-based accelerators for scientific applications operating on floating point data starting from a C language input. Its strength consists in allowing users to select floating point operators from a variety of

standard libraries, such as FPLibrary and Quixilica, or to import their own. The compiler uses LLVM, and as such, it can include any optimization available in this framework. Furthermore, module scheduling is one of the four scheduling algorithms supported that enables the loop pipelining optimization. However, the tool has some issues such as non comprehensible errors, testbenches not being generated which rendered automated verification impossible and functions not being able to have arguments or return values.

C-to-Verilog

C-to-Verilog [11] [72] is a hardware compiler developed around 2008 which accepts C code and generates Verilog RTL. The compiler is available online from the University of Haifa. The lack of available documentation prevents us to describe it here in detail. However, the brief testing performed revealed major problems. The most important was that it does not support a lot of C language features. Furthermore, from the runs performed with C function/kernels available in the Computer Engineering department showed that c-to-verilog can compile only 38% of those functions. Finally, the simplest test (cmultconj) from the testbench that will be used in Chapter 9 failed to compile without any meaningful error message. As a result, we can conclude that this tool is useful for specific purposes and cannot be compared to DWARV. Therefore, we will not attempt to include it in the final hardware compilers evaluation.

C-to-Hardware

C2H [6] is a HLS tool offered by Altera Corporation since 2006. The tool is target dependent and generates VHDL or Verilog hardware designs from C descriptions only for accelerators that communicate via an Avalon bus with a NIOS II configurable soft processor. Furthermore, using this tool requires advanced hardware design knowledge. To main disadvantage is that it is not possible to create, simulate and evaluate accelerators using the C2H tool fast. The user needs to create a NIOS II valid system before it can create accelerators, which is realized by creating a SOPC Builder system file. Creating and connecting the different components in a system is not a trivial task. Neither floating nor fixed point arithmetic is supported. The available documentation explains that in order to verify designs the IOWR_32DIRECT directive has to be used to generate a testbench. Furthermore, even if this is performed it is very hard to identify the performance of the generated accelerator as this is embedded and only applicable for the NIOS tool-flow. As a consequence of

the fact that this is not a general HLS product, therefore, one needs to build first a fully connected and configured Processor-Memory-Peripherals-Bus Systems before one can generate hardware from a HLL, Altera announced that C2H will be discontinued in future products [7].

Synphony HLS

Synphony HLS [77] product is a HLS tool for hardware DSP design offered by Synopsys. It was acquired in 2010 from Synfora under the name of the PICO tool [5]. The input design languages can be either C or C++ whereas the output is in the form of VHDL or Verilog. The tool can support both streaming and memory interfaces and allows for performance related optimizations to be fine-tuned (e.g., loop unrolling, loop pipelining). Floating point operations are not permitted, but the programmer can use fixed point arithmetic which is supported. Finally, verification is automated by generating test vectors and scripts for RTL simulators. Comparison results published by BDTi [10] showed that performance and area metrics are comparable with those obtained with the former Vivado HLS, i.e., AutoESL.

MATCH

MATCH [9, 88] is a software system developed by the University of Northwestern Illinois in 2000. The goal is to translate and map matlab code descriptions to heterogeneous computing platforms for signal and image processing applications. The MATCH system included besides a MATLAB to VHDL compiler, also two MATLAB to C compilers for the embedded and Digital Signal Processor (DSP) processors of the system. The translation from MATLAB to C was done by converting each function into a VHDL process, each scalar variable in MATLAB into a variable in VHDL, each array variable in MATLAB was assumed to be stored in a RAM adjacent to the FPGA, hence a corresponding read or write function of a memory process was called from the FPGA computation process. Furthermore, control statements such as if-then-else constructs in MATLAB were converted into corresponding if-then-else constructs in VHDL. Assignment statements in MATLAB were converted into variable assignment statements in VHDL. Loop control statements were converted into a finite state machine. That is, for each loop statement, a finite state machine with four states was created. The first state performed the initialization of loop control variables and any variables used inside the loop. The second state was used to check if the loop exit condition is satisfied. If the con-

dition is valid, it transfers control to state four, which is the end of the loop. If the condition is not valid, it transfers control to state three, which performs the execution of statements in the loop body. If there is an array access statement (either read or write), one needs to generate extra states to perform the memory read/write from external memory and wait the correct number of cycles. The MATCH technology was later transferred to a startup company, AccelChip, bought in 2006 by Xilinx.

AccelDSP

AccelDSP [96] was a tool acquired from AccelChip DSP by Xilinx in 2006. However, the tool has been discontinued since 2010, with the release of ISE edition 12.1. The tool was one of the few on the market that started from a MATLAB input description to generate VHDL or Verilog for DSP algorithms. Key features of the product were automation of floating- to fixed-point conversion, generation of synthesizable VHDL or Verilog and testbench generation for verification. Finally, it also offered some optimization possibilities such as loop and matrix multiplication unrolling, pipelining and memory mapping.

CHiMPS

The *CHiMPS* compiler [71] (Compiling High-level Languages into Massively Pipelined Systems), developed by the University of Washington in collaboration with Xilinx Research Labs in 2008, targets applications from the High-Performance Computing (HPC) domain. The platforms targeted are CPU-FPGA based platforms communicating via a shared memory system. The design language is C and the output generated is VHDL. This make the design goals of CHiMPS very similar to DWARV. The distinctive feature of CHiMPS is the *many-cache* which is a hardware model that adapts the hundreds of small, independent FPGA memories to the specific memory needs of an application. This allows for simultaneous memory operations per clock cycle. Furthermore, the programmer can fine tune the generation process via pragmas. For example, cache parameters, separate memory spaces, loop unrolling factor and manual bit-width can be specified manually. However, no floating or fixed point capabilities are mentioned, nor any support for automated accelerator verification.

DEFACTO

DEFACTO [53] is one of the early design environments that proposed hardware/software co-design solutions as an answer to the ever increasing demand in computational power. The research regarding this environment took place at the University of South California in 1999. DEFACTO is composed of a series of tools such as profiler, partitioner and software and hardware compilers. The main benefit of the tool suite was that it allows the designer to perform fast Design Space Exploration (DSE) and choose the best solution given a set of design constraints. SUIF compiler framework was the underlying brick stone use to build the individual tools. The DEFACTO paper does not describe floating, fixed or verification capabilities.

Gcc2Verilog

GCC2Verilog [45] is a tool that, just as the name suggests, translates C code into Verilog. It was developed at the University of South Korea in 2011, and it uses the GCC frontend to translate the C code into the Intermediate Representation (IR) format used by GCC, and as such, it was designed for the purpose of having a HLS tool that uses unrestricted C code. However, this was not accomplished, and problems such as dynamic pointer allocations are left for the software-side of the program. To generate the hardware, a customized back-end then translates the IR into Verilog. The compiler does not make use of any optimization techniques to extract parallelism other than scheduling and the techniques already built into GCC. A Verilog design generated by the compiler consists of a data-path and a Finite State Machine (FSM). The back-end translates each of the instructions from the GCC IR into a part of the data-path and a part of the FSM, adding a new state whenever it is necessary. Finally, no information was available about other features such as floating and fixed point data types and automated verification support.

2.1.2.2 Object-Oriented Languages

Object-oriented languages offer a higher degree of abstraction than procedural languages. This improves the maintenance of the code and allows for a more structured design approach. These languages have been used increasingly in the last two decennia, and, as a result, HLS tools based on C++ or Java were developed as well.

MaxCompiler

MaxCompiler [81] is a data-flow specific HLS tool used by Maxeler since 2010 to program their data-flow hardware. The compiler accepts MaxJ, a Java based language, as input language and generates synthesizable RTL code used to program the hardware data-flow engines provided by the Maxeler's hardware platform. Consequently, besides being oriented towards the streaming data-flow domain, the fact that existing applications would need to be rewritten in MaxJ clearly differentiate the target audience of Maxeler from the one targeted in this work.

Kiwi

The *Kiwi* [35] parallel programming library and its associated synthesis system generates FPGA (Verilog-based) co-processors from C# parallel programs. This system has been developed by the University of Cambridge and Microsoft Research Cambridge in 2008, and it was oriented towards software engineers that are willing to express the application code as parallel programs. The novelty of this system consists in the fact that it allows the programmer to use more parallel constructs such as events, monitors and threads, which are closer to the hardware concepts than the conventional C language constructs. The lack of further information available made it impossible to provide details about the optimizations support, testbench generation capabilities or whether float and/or fixed point arithmetic is allowed.

Sea Cucumber

Sea Cucumber [83] is a Java based compiler that generates directly EDIF netlist files. It was developed in the early 00's (2002) at the Birgham Young University with the goal to generate circuits that exploit the coarse- and fine-grained parallelism available in the input class files. Sea Cucumber adopts the standard Java thread model and augments it with a communication model based on Communication Sequential Processes (CSP). The tool employed many conventional compiler optimizations such as dead-code elimination, constant folding and if-conversion. Furthermore, by using the BitWidth specific Sea Cucumber package, support for both floating and fixed data types and arithmetic was available. However, designs generated by this tool could not be verified automatically because of the lack of test bench generation capabilities.

Cynthesizer

Cynthesizer [79] from Forte Design Systems was the first HLS tool to provide an implementation path from SystemC to Verilog RTL code. The tool was first released in 2004, and since then it became a proven product offering features such as verification and co-simulation, formal equivalence checking between RTL and gates, power analysis, a number of optimizations such as operation chaining, and support for floating point datatypes available in IEEE754 single and double precision as well as other combinations of exponent and mantissa width defined by the user.

2.2 Summary of Tool Features

Tables 2.1 and 2.2 summarize the important aspects of every hardware compiler described in the previous section. For each compiler shown in the first column, we list in the second and third columns the company or university that developed the tool and whether this is commercially or freely available (academic license). The fourth and fifths columns show what input languages are accepted, respectively in what HDL language the output is produced. The next two columns show the year in which the tool was first released and at what application domain it was targeted. For some commercial tools, the application domain was not restricted. This is denoted in the *Domain* column as *All* domains are supported. Based on this observation, we note that if, in the literature available, no information was found about the target application domain of the toolset, particularly true for academic compilers, also an *All* domains was assumed. Finally, the last three columns provide information about verification capabilities of the synthesized hardware designs by means of automatic test bench generation, and whether floating and fixed point arithmetic is supported in the input language of the tool.

One of the biggest difference with most of the early compiler frameworks is that DWARV does not restrict the application domain. That is, it can generate HDL code for both control- and data-based application domain. For example, tools such as ROCCC, Impulse-C, SPARK, CoDeveloper or Gaut restrict the application domain to either streaming applications, image processing, control-based or DSP algorithms. The consequence is that the generated interface for these designs can then support only one particular communication protocol, which is efficient only for one type of application.

Another drawback of the fact that tools limit the application domain, is that

Table 2.1: Overview of Existing High-Level Synthesis Tools.

Compiler	Owner	License	Input	Output	Year	Domain	TestBench	FP	FixP
ROCCC 1.0	U. Cal. River	Academic	C subset	VHDL	2005	Streaming	No	Yes	No
ROCCC 2.0	Jacquard Comp.	Commercial	C subset	VHDL	2010	Streaming	No	Yes	No
Catapult-C	Calypto Design Systems	Commercial	C/C++ SystemC	VHDL/Verilog SystemC	2004	All	Yes	No	Yes
CtoS	Cadence	Commercial	SystemC TLM/C++	Verilog SystemC	2008	All	Only cycle accurate	No	Yes
DK Design Suite	Mentor Graphics	Commercial	Handel-C	VHDL Verilog	2009	Streaming	No	No	Yes
CoDeve-loper	Impulse Accelerated	Commercial	Impulse-C	VHDL Verilog	2003	Image Streaming	Yes	Yes	No
SA-C	U. Colorado	Academic	SA-C	VHDL	2003	Image	No	No	No
SPARK	U. Cal. Irvine	Academic	C	VHDL	2003	Control	No	No	No
CHC	Altium	Commercial	C subset	VHDL/Verilog	2008	All	No	Yes	Yes
VivadoHLS	Xilinx	Commercial	C/C++ SystemC	VHDL/Verilog SystemC	2013	All	Yes	Yes	Yes
LegUp	U. Toronto	Academic	C	Verilog	2011	All	Yes	Yes	No
PandA	U. Polimi	Academic	C	Verilog	2012	All	Yes	Yes	No
HercuLeS	Ajax Compiler	Commercial	C/NAC	VHDL	2012	All	Yes	Yes	Yes
GAUT	U. Bretagne	Academic	C/C++	VHDL	2010	DSP	Yes	No	Yes
Trident	Los Alamos NL	Academic	C subset	VHDL	2007	Scientific	No	Yes	No

Table 2.2: Overview of Existing High-Level Synthesis Tools. (Cont)

Compiler	Owner	License	Input	Output	Year	Domain	TestBench	FP	FixP
CtoVerilog	U. Haifa	Academic	C	Verilog	2008	All	No	No	No
C2H	Altera	Commercial	C	VHDL/Verilog	2006	All	No	No	No
Synphony HLS	Synopsys	Commercial	C/C++	VHDL/Verilog SystemC	2010	All	Yes	No	Yes
MATCH	U. Northwest	Academic	MATLAB	VHDL	2000	Image	No	No	No
Cyber-WorkBench	NEC	Commercial	BDL	VHDL Verilog	2011	All	Cycle/ Formal	Yes	Yes
Bluespec	BlueSpec Inc.	Commercial	BSV	SystemVerilog	2007	All	No	No	No
AccelDSP	Xilinx	Commercial	MATLAB	VHDL/Verilog	2006	DSP	Yes	Yes	Yes
Kiwi	U. Cambridge	Academic	C#	Verilog	2008	.NET	No	No	No
CHiMPS	U. Washington	Academic	C	VHDL	2008	All	No	No	No
MaxCompiler	Maxeler	Commercial	MaxJ	RTL	2010	DataFlow	No	Yes	No
SeaCucumber	U. Brigham Y.	Academic	Java	EDIF	2002	All	No	Yes	Yes
DEFACTO	U. South Cailf.	Academic	C	RTL	1999	DSE	No	No	No
PipeRench	U.Carnegie M.	Academic	DIL	bitstream	2000	Stream	No	No	No
Garp	U. Berkeley	Academic	C subset	bitstream	2000	Loop	No	No	No
Napa-C	Sarnoff Corp.	Academic	C subset	VHDL/Verilog	1998	Loop	No	No	No
gcc2verilog	U. Korea	Academic	C	Verilog	2011	All	No	No	No
Cynthesizer	FORTE	Commercial	SystemC	Verilog	2004	All	Yes	Yes	Yes
eXCite	Y Explorations	Commercial	C	VHDL/Verilog	2001	All	Yes	No	Yes

the input language has to be restricted, modified or extended. For example, SA-C, DK Design Suite, Impulse-C, CyberWorkbench, BlueSpec, PipeRench or Garp all limit in some way the input language by not accepting for instance pointers or control statements. Furthermore, some tools require the addition of specific code/language constructs to convey specific platform information, such as PipeRench or Napa-C. This increases the time required to learn how to use the tool and to transform the existing HLL code to an accepted syntax before any useful output can be generated. DWARV does not require any additional language constructs and can generate code from (almost) any function as it is.

Another important difference is that some tools are available as integrated compilers in larger design environments (e.g., CHC, C2H, Synphony HLS or AccelDSP), most of which are targeted at the hardware designer and not the software developer. That tool is then very hard to use without actually having hardware design knowledge. Considering the scenario assumed in this work where a software designer moves software to hardware to accelerate specific computations (i.e., only a particular function needs to be accelerated), designing the whole (hardware) system is inconvenient and limited to a number of hardware designers. These frameworks are thus not intended for the vast majority of software programmers as DWARV's target audience is, but it is intended to obtain (fast) hardware IPs by a hardware designer performing system-wide DSE before he can choose a solution for its ASIC.

Finally, some of the (early) tools were based on immature research compiler frameworks which led to the their abandonment. We list ROCCC1.0, Napa-C, Garp or AccelDSP. Furthermore, this underlying compiler framework can play a very big role in the high-level language coverage and the compiler optimization support of a hardware compiler. The latest version of DWARV differs from all previous tools by using CoSy compiler framework. This highly modular and extensible framework offers both lowering (i.e., high-level to low-level construct mapping) transformations so that all HLL constructs can be accepted as well as support for automatic selection of which compiler optimizations to be applied removing the user need of performing this selection manually at the command line (or in a GUI).

2.3 Conclusion

In this chapter, we described a number of academic and commercial high-level synthesis compilers. Although we could observe an increase in the amount of

both research tools and commercial products available, especially in the last three to four years, our evaluation of these tools showed that knowledge on how to generate efficient hardware accelerators is still not mature enough to replace the manual design completely. As a result, the need of research in how to design hardware compilers is justified. Furthermore, the last version of the compiler at the time of writing this thesis, i.e., DWARV3.0, is different in at least one aspect from the related compilers as shown in the previous section.

Note.

The content of this chapter was submitted as part of the following publication:

R. Nane, V.M. Sima, K.L.M. Bertels, **A Survey of High-Level Synthesis Tools and Comparison with DWARV 3.0**, Submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, April 2014.

3
Background Work

In this chapter, we present the environment in which the work presented in this dissertation was done. First, we describe the Delft Workbench tool-chain, which consists of several tools to enable automated Hardware/Software (HW/SW) co-design. Subsequently, we present in detail the Molen machine organization, we show the synthesis and verification flows used to validate generated accelerators, and we discuss important similarities and differences between software and hardware compilers. Finally, we present the first version of the DWARV compiler that provided the inspiration for the current version, we describe the CoSy compiler framework used to construct DWARV3.0, and we give a simple C-to-FPGA example showing the tool-flow.

3.1 Introduction

The Delft Workbench (DWB) [12] is a semi-automatic tool chain for integrated HW/SW co-design. The goal is to facilitate heterogeneous computing and to enable fast Design Space Exploration (DSE) on Molen type heterogeneous systems by automating, where possible, the profiling, estimation, mapping and implementation processes that are part of the development process. One of the tools developed in the scope of this workbench was the DWB Automated Reconfigurable VHDL (DWARV) [97] generator. As explained in the introduction, DWARV takes parts (i.e., functions[1]) of high-level programs and transforms them into hardware accelerators to be run on a Molen machine. Although the DWB targets implementations of Molen on Xilinx boards because of its availability and expertise, any other type of heterogeneous computing platform could have been used instead. For example, the Convey [19] HC machines can be considered instantiations of the Molen machine organization

[1] functions and kernels are used interchangeably throughout the text

and could have been used as well in the experiments performed throughout this dissertation.

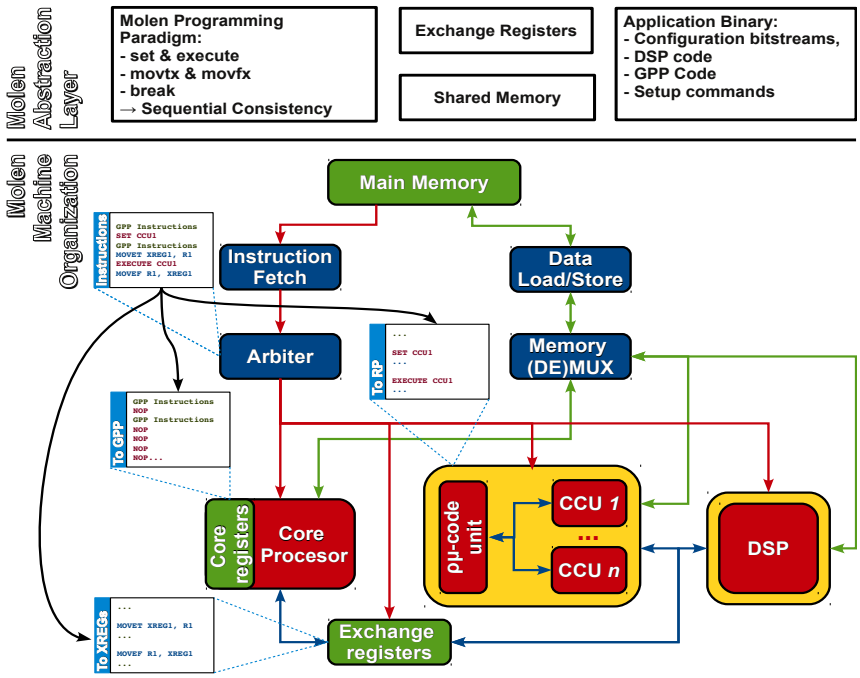


Figure 3.1: An Overview of the Molen Platform with an Indication of the Flow of Instructions Through the Platform [60].

3.2 Molen Machine Organization

The Molen Machine Organization [24, 73] is an architecture developed at TU Delft intended to define how different heterogeneous computing elements can be used together to execute applications faster and in a power efficient way. Figure 3.1 gives a high-level view of the organization in which we can see that it is composed of three main elements. These are a shared memory and a General Purpose Processor(GPP) tightly coupled with Custom Computing Unit (CCU)s. The CCU can be anything such as Graphics Processing Unit (GPU), Digital Signal Processor (DSP) or Field-Programmable Gate

Array (FPGA) based kernels. The Molen organization has been successfully implemented and demonstrated using DSP and FPGA kernels.

This setup allows taking advantage of inherent parallelism in applications from various domains, by allowing to move computationally intensive parts of code to hardware (CCU) while keeping the rest of the application in software (GPP). The Molen supporting tool-chain (i.e., DWB) performs in this respect the following tasks: searches for pragma annotations in the source application code, usually set manually by the user or automatically by a previously running profiling tool, invokes the Molen primitives that deal with loading the function parameter values in the eXchange Register (XREG)s, place from where the CCU can access its input values, and finally, calls the actual function that will run in the FPGA. Depending on the data dependency analysis this can be blocking or non-blocking. The CCU can be the hardware generated by invoking the DWARV compiler described in Chapter 4. This setup enables DSE by allowing the user to perform and evaluate various configurations of HW/SW partitioning.

To exploit the Molen machine organization efficiently, the Molen programming paradigm [91] was devised. The concepts defined by this paradigm are shown in the upper part of Figure 3.1. They address the inherent problem caused by the flexibility of reconfigurable computing where different instructions can be accommodated, and, as a result, there is no fixed instruction set. The solution to this issue consisted in a one-time extension of the (Polymorphic) Instruction Set Architecture (πISA) such that any number of processor extensions with a variable number of arguments can be supported. This extension was implemented by the following instructions:

- *SET* instruction configures (i.e., loads) the computation on the processing element (e.g., FPGA).
- *MOVET/MOVEF* instructions are used to transfer data to and from the processing element's local memory.
- *EXECUTE* instruction gives the start signal for the processing element to begin execution.
- *BREAK* instruction polls the processing element to check whether execution has finished.

The Molen machine organization implements a parallel execution model, and its supporting Molen programming paradigm is compatible with parallel programming frameworks such as OpenMP [69]. As a consequence, it is very

important to note that *sequential consistency* (i.e., execution on the heterogeneous platform produces the same results as when the original program would have been executed sequentially on a General-Purpose Processor (GPP)) is enforced by the above instructions. This is accomplished by adopting the shared memory computational model.

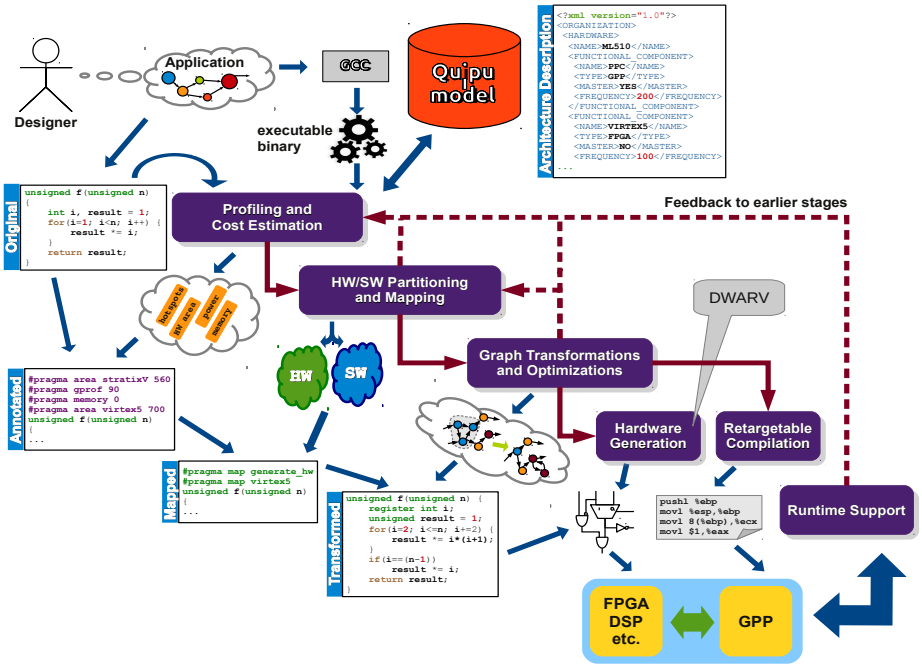


Figure 3.2: Overview of the Delft Workbench Tool-Chain [60].

3.3 Delft Workbench Tool-Chain

Figure 3.2 shows the complete design flow within the DWB. The first process in the flow is the *Profiling and Cost Estimation*. The objective is to identify the computational intensive parts of an application that are good candidates for hardware acceleration. Furthermore, design data such as the maximum performance improvement, resource occupation and power requirements is gathered to drive the subsequent partitioning, mapping and optimization stages. *HW/SW Partitioning and Mapping* uses the previous acquired design characteristics to partition the original application into different parts and map them onto the different computational elements available on the heterogeneous computing plat-

form. *Graph Transformations and Optimizations* processes are used to transform the application code to a form suitable for hardware implementation. For example, parts of the graph can be duplicated or factored out depending on the available hardware resources. *Retargetable Compilation* is an important step in the context of the complete tool-chain. It is here that a retargetable compiler generates code for the embedded GPP and combines generated bitstreams for FPGAs, DSPs, etc., into a final Executable and Linkable Format (ELF) file that can be executed on the heterogeneous platform. The generation of bitstreams is done with technology dependent synthesis tools, but Hardware Description Language (HDL) code required as input by these synthesis tools is obtained in the *Hardware Generation* stage. DWARV is the compiler that generates such HDL (i.e., VHDL) from High-Level Language (HLL) (i.e., C) code and is, therefore, executed in this stage. Finally, *Run-time Support* is intended to offer support for (dynamic) system reconfiguration and management in a multi-tasking environment.

3.4 Back-end Work Flows

There are two possible work flows: the synthesis flow and the simulation flow. The first one can be used for fast prototyping and mapping exploration, while the later is used to verify the generated hardware kernel automatically before running it on the actual hardware. Both flows rely on same high-level concepts.

3.4.1 Synthesis Flow

The synthesis flow is presented in Figure 3.3. The input of the flow is C code while the output is an executable binary for a specific platform (for example the ML510 platform). The blue boxes represent the tools/components of the DWB tool-chain while the grey components represent external components like the gcc compiler or the Xilinx tool-chain. The green boxes show intermediary files related to the application.

The purpose of the frontend is to partition and map the application in preparation to the invocation of the back-end compilers. This was illustrated and described in the previous section (3.3). Then, each part of the application goes to each back-end compiler (e.g., for the ML510 platform). The *platform* compiler will directly generate object code in the ELF format. For the part that has to go to the FPGA, Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) is invoked first to generate the VHDL. The generated

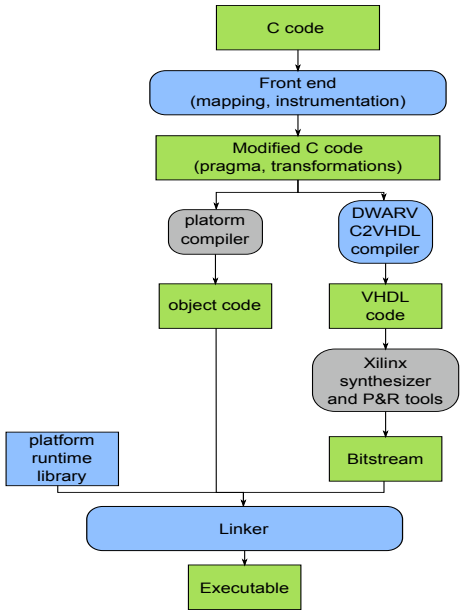


Figure 3.3: Molen Backend Synthesis Flow.

VHDL is passed to the Xilinx tools which perform synthesis, place and route. The final bitstream is included in the executable as an ELF symbol by the linker.

3.4.2 Simulation Flow

The simulation flow can be used to rapidly assess the performance of implementing a function in hardware. We call the hardware implementation of a function a CCU. Another possible usage of the flow is to ensure that the output of the CCU is functionally correct. Because the tool-chain supports floating-point operations, and the rounding and order of operations affect the final result, the simulation is an important step to ensure a successful implementation.

The overall flow is depicted graphically in Figure 3.4. A C file, annotated with pragmas, is provided by the developer or a previous running tool-chain (e.g., DWB frontend) as input. Using special scripts, the memory inputs and outputs of the function that has to be synthesized to VHDL, are dumped to files. Then, the VHDL is generated together with a simulation test bench that provides all the necessary signals for the CCU to run. The VHDL is simulated using the

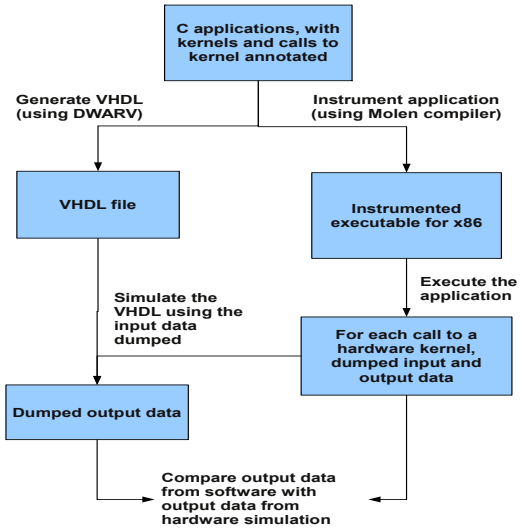


Figure 3.4: Simulation Flow for Verifying DWARV Generated VHDL Kernels.

memory inputs, and the outputs are also dumped in a file. The final step is to compare the outputs obtained from software with the outputs obtained from hardware.

An important note is that all memory locations that are sent through pointer parameters to the CCU-s must be allocated through malloc function calls (no local or global variables). If the function parameters are arrays (and thus, the compiler can determine their size at compile time), any memory location can be sent to the function.

3.5 Software vs. Hardware Compilers

Different books are available on how to write a compiler such as [37]. For the purposes of this section, it is sufficient to highlight that a compiler is composed of several main passes that transforms the input high-level language code to a format accepted by the target machine. We define a software compiler as a compiler that transforms a HLL to assembly type instructions, whereas a hardware compiler is one that transforms a HLL code to an equivalent HDL representation. The first two steps in any compiler are to scan and parse the input

file. After these steps, an Intermediate Representation (IR) form is created that enables transformations and optimizations to be performed on the original input code. Important transformations are lowering transformations, Static Single Assignment (SSA), Data Dependency Graph (DDG) construction, instruction selection, instruction scheduling and register allocation. Important optimizations include Common Subexpression Elimination (CSE), constant folding and propagation, dead code elimination, if-conversion, loop unrolling, loop pipelining, polyhedral analysis and peephole optimizations such as strength reduction. Finally, the last step is the generation of code. This process outputs in a file the schedule obtained by including all the previous information gathered and by printing architectural dependent strings representing the instruction rules matched in the instruction selection step.

Building a hardware compiler is not very different than building a software compiler. In principle, all above steps can be applied. However, because on (raw) (re)configurable hardware the number of resources is not fixed, and, as such, we can (to some extent) accommodate as many resources as required, we have to change how some of the traditional software compiler passes are applied in the context of hardware generation. Although this kind of analysis can apply to various compiler transformations and optimizations, for the scope of this section, we discuss here only a few: CSE, if-conversion, register allocation, operation chaining and memory spaces.

CSE is an optimization that computes common expressions once, places the result in a register or memory, and refers this location for subsequent uses. The advantage is that we can save time and resources by not computing the expression again because loading the already computed value is faster on a Central Processing Unit (CPU). However, for hardware generation this has to be carefully considered. That is, if the allocated register and the expression where this replaced expression should be used is not in the immediate proximity of the calculation, the routing to that place might actually decrease the design frequency. Therefore, applying this optimization is not always useful in High-Level Synthesis (HLS). In this case, replacing only a particular subset and recomputing the expression for others to enforce a better locality of the operations, would provide better results. If-conversion provides the opposite scenario, which will be described in detail in Chapter 7.

Register allocation is a highly important transformation that should be treated differently. Because in hardware we have an unlimited number of registers, the allocation of variables to these can be considered simpler. That is, whenever we need a register, we allocate a new one. However, if the generated design

does not fit into a given area constraint, the register allocation becomes more difficult to solve because it is not easy to decide how many registers should be removed and which ones in order to successfully implement the generated design on the given area. Operation chaining is an optimization that is specially designed for hardware compilers. Because in hardware the clock period is user defined, we use this optimization to schedule dependent instructions in the same clock cycle if their cumulative execution time is less than the given clock cycle time. This operation optimizes significantly the wall clock time (i.e., execution time) of the entire function implemented on hardware. Finally, memory spaces can be allocated for different array parameters so that multiple memory operations can be performed in the same cycle.

3.6 DWARV 1.0

DWARV's [97] first implementation was based on the SUIF compiler framework from Stanford university [89]. The framework provided methods to create an IR from a HLL program and passes/mechanisms to access that IR to process the data in a way most suitable for the compiler developer. However, being a research project, the SUIF framework had many drawbacks such as:

- No longer supported and the documentation was scarce.
- No optimization passes available.
- No dependency analysis and aliasing disambiguation available.
- No extendible IR.

Furthermore, given the fact that no lowering transformations (i.e., from high- to low-level constructs, e.g., from switch statement to simple compare-jump statements) were available in SUIF, not all syntax was supported by DWARV 1.0. Tables 3.1 and 3.2 list all restrictions available in the initial version of DWARV. Given the many drawbacks and the missing features, especially given the fact that development of SUIF was abandoned and it become unsupported, a change to another compiler framework was necessary. Nevertheless, despite all the drawbacks, this first compiler inspired the FSM plus Datapath computational model along with the hardware accelerator VHDL file structure also used in the second version (see Chapter 4).

Table 3.1: DWARV 1.0 Allowed Data Types.

Data Type	Supported	Not Supported
Boolean		_Bool
Integer	Up to 32 bits	64 bit
Real Floating-Point		all
Complex and Imaginary		all
Pointer	data pointer	function pointer
Aggregate	uni-dimensional array	multi-dimensional arrays struct
Union		all

Table 3.2: DWARV 1.0 Allowed Statements.

Statement Type	Supported	Not Supported
Expression	unary,add,mul shift,bitwise,assign	div,mod logic and, logic or, cast
Labeled		case,label
Jump		return, break, goto, continue
Selection	if	switch
Iteration	for	while, do-while
Calls		functions

For the new version of DWARV, four candidate compiler frameworks were considered. *LLVM* was the first one, but because, at that time, the framework was not mature enough (e.g., frontend integration was not easy), it was not selected. *gcc* was another candidate, however, this framework was going through major changes in the IR from version 4.5; therefore, it was not considered stable. *ROSE* is a compiler framework that deals only with high-levels of abstraction, and; therefore, it is used only for HLL to HLL compilers. Furthermore, it does not include backend code generator features that are required for a compiler generating low-level code. Finally, the last candidate was the CoSy compiler framework. Due to its maturity, robust, and extendible features, coupled with the close collaboration with the ACE company in the REFLECT [70] project, this framework was selected to reimplement the DWARV 1.0 compiler.

3.7 CoSy Compiler Framework

The CoSy [26] compiler development system is a commercial compiler construction system licensed by ACE Associated Compiler Experts. The development system is composed of several tools building on innovative concepts that allow the user to build and extend a compiler easily. The central concept in CoSy is the *Engine* that can be regarded as an abstraction of a particular transformation or optimization algorithm. The framework includes a total number of 208 engines that target different parts of the compilation flow, from the frontend (i.e., code parsing and IR creation) processing, high-level (e.g., loop optimization, algebraic optimizer, code analysis, lowering transformations) and low-level (e.g., register allocation, instruction selection and scheduling) transformations and optimizations to backend code generation template rules used in the final emission step. The general functionality of (custom user-defined) engines can be programmed in C or in C++ while for IR specific initializations and assignments, CoSy-C, an extension of the C-language, is used.

Figure 3.5 depicts graphically the connections between CoSy concepts that combined form a CoSy compiler. Each engine accesses the *IR* through specific *Views* generated in the *Supervisor* during compiler build time. This is possible because each engine contains a list of parameters that are typed handles on IR fragments. The IR is described in a CoSy specific description language, i.e., *full-Structure Definition Language* (*fSDL*), and contains the data structures on which engines operate. The strength of this language is its distributed nature in the sense that multiple definitions of the same data structure augment each other. This allows to specify IR extensions easily and locally in each

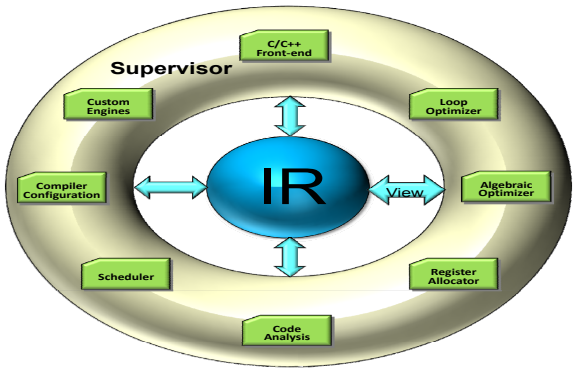


Figure 3.5: CoSy Framework Elements: Supervisor, Engines, Views and IR.

engine by (re)defining and extending the target data structure in the engine's ⟨engine_name⟩*.sdl* file. From this file, specific views are inferred based on the access rights defined for each data structure's operator fields (e.g., new, read, write, walk, etc). That is, a view can be thought of as a special set of accessor and mutator method(s) for a data structure. These concepts improve the productivity of the compiler designer because they enforce at compile time the clear separation between engines, thus making a CoSy based compiler very modular by nature.

The control and data flow through the compiler is coded in **.edl* files that are written in another CoSy-specific language, the *Engine Description Language* (*EDL*). This language provides mechanisms to form an engine class hierarchy by defining composite engine classes as groups of engines. Furthermore, it allows the user to specify different interaction schemes between engines. For example, a particular set of engines can be run sequentially, in a pipeline, in parallel or in a loop. A supervisor is generated from the EDL and fSDL specifications. Therefore, it implements the control and data flow through the compiler and protects the IR against conflicting accesses. To generate code, the supervisor uses the instantiated (i.e., implemented) versions of CoSy template rules (e.g., mirPlus(rs1:register;rs2:register) -⟩ rd:register) defined in **.cgd* files. A *compiler* is composed of the generated supervisor instantiated in a small main program that reads command line options, processes them if necessary, and passes them on to the supervisor. Finally, below the supervisor we find the independent codes of all simple engines that form the bulk of the compiler.

Therefore, the main strength of the framework is that it consist of loosely inte-

grated engines that perform various transformations and optimizations on the IR, engines that can be skipped easily depending on user set options. This enables us to design custom engines that deal with different aspects of the hardware optimization and/or generation phase(s) without conflicting with existing passes or existing IR data types. More information about the CoSy conventions can be found in the Engine Writer's Guide framework documentation [27].

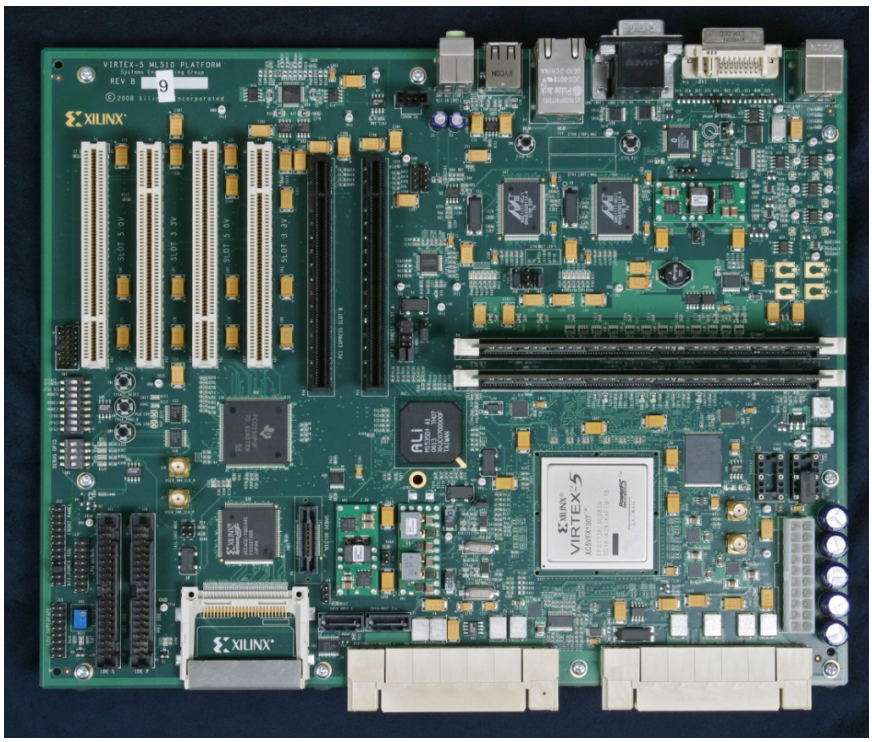


Figure 3.6: Xilinx Virtex-5 ML510 Hardware Platform.

3.8 C-to-FPGA Example

In this section, we describe the complete compilation flow of a simple C application and show what the backend tools generate, as well as how this is integrated into the final target platform executable. One of the hardware platforms used in this thesis is the Xilinx Virtex-5 ML510 board shown in Figure 3.6. This platform is based on the Xilinx Virtex-5 XC5VFX130T FPGA, which has 20480 slices, 1580 Kb distributed memory blocks and two PowerPC 440

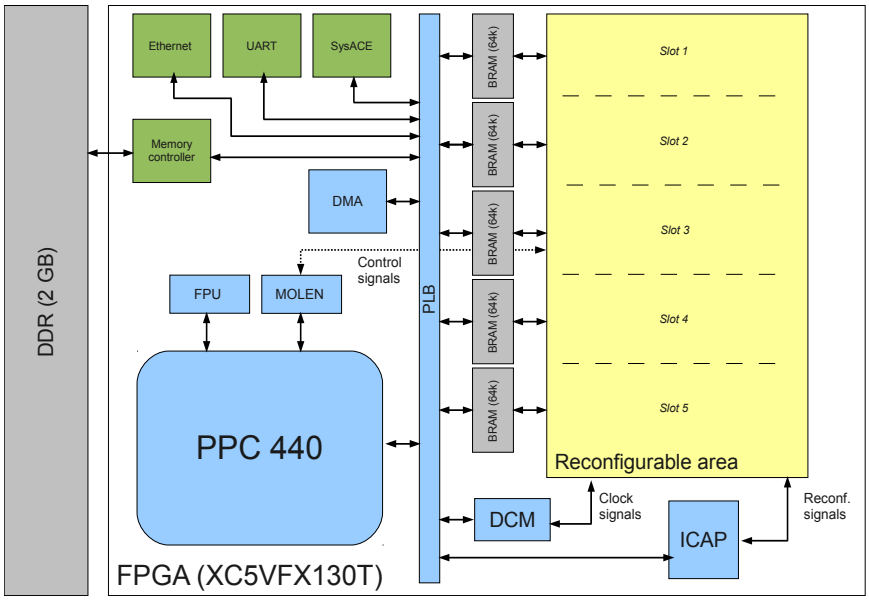


Figure 3.7: Molen Implementation on the ML510 Hardware Platform.

(PPC440) processor hard cores embedded that run at a maximum of 400 MHz.

To use this platform, the Molen machine organization was implemented using a custom Fabric Coprocessor Module (FCM), shown in the simplified platform schematic in Figure 3.7 as the Molen box. The FCM is designed to enable custom instruction extensions of the PPC440 processor's Instruction Set Architecture (ISA). Newly added instructions are recognized in the decode phase by the PPC440 processor's Auxiliary Processor Unit (APU) (not shown in the figure) as custom instructions, and, subsequently, dispatched to the FCM unit for the execution stage. In Figure 3.7, we see five reconfigurable slots (slot one to five) reserved for custom accelerators. Each of these areas can be independently reconfigured and executed at run-time. This is not a limitation of the Molen architecture, but merely a design choice considering the hardware area available on this FPGA board and the reconfigurability requirement.

To show how applications can be accelerated on the ML510 platform using reconfigurable logic, we use the simple C-application given in Listing 3.1i). This is composed of some arbitrary code, that will keep running on the PPC440 processor, followed by a call to the *absval_arr* function, which is the target function for acceleration. To mark that this function should be implemented in hardware, we annotate its call with a *pragma* directive. This annotation

can be performed manually or automatically by executing the frontend tools shown in Figure 3.2, which perform profiling, partitioning and mapping for the complete application. Finally, the *absval_arr* call is followed by *some other code* that will execute on the software processor. This code is translated to standard PowerPC assembly instructions, and, because we focus on the custom instructions, the standard instructions are omitted from the listing. Therefore, in Listing 3.1ii) we show only the custom instructions generated by the Molen *platform compiler* (see Figure 3.3), which is essentially a modified version of the GNU Compiler Collection (gcc) compiler, to perform the necessary steps to invoke a reconfigurable accelerator.

Listing 3.1: C-to-FPGA Example Application and Instrumented Assembly Code

```
      i) C code                ii) Assembly code
void main() {
int a[16],b[16],c[16],res;
...  // some other code        ...
                           //set CCU_ID, ADDR_bitstream
                                set       1, 0
//pragma inserted by previous partitioning and
//mapping tool; movet CCU_ID,XREG_ID,addr,size,type
#pragma call_hw VIRTEX5 1      movet     1,1,0,a,0
res = absval_arr(a,b,c);       movet     1,1,1,b,0
                               movet     1,1,2,c,0
                               execute   1
                               break_end 1
                               movef     1,0
...  // some other code        ...
}
```

First, the *set 1,0* instruction is used to configure *slot 1* on the reconfigurable logic with the bitstream at address 0 in the main memory. Number 1 is given as the second parameter to the *call_hw* attribute to represent that this function should use the first slot in the Virtex5 reconfigurable area. The actual (re)configuration of the hardware is performed by the *Internal Configuration Access Port* controller (denoted as *ICAP* in Figure 3.7), which loads the generated configuration bitstream from memory. The actual generation of the bitstream is discussed below. Second, the *movet* instructions are used to transfer the data of the three arrays (a,b,c) from the shared main memory (*DDR 2GB*) via the *Direct Memory Access (DMA)* module to the local slot's *BRAM(64k)*

memory on the FPGA. This is required when the accelerator uses pointers to access data. After the data is transferred, the function is *execute*d in the hardware while the *break_end(1)* instruction polls whether the accelerator in slot 1 has finished execution. Finally, the *movef* instruction retrieves the accelerator's return value located at address 0 in the BRAM of slot 1. The communication between the PPC440 and the reconfigurable logic is performed through the *Processor Local Bus (PLB)*. The *Digital Clock Manager (DCM)* module has the role to drive the slots frequencies, which can be different from each other.

Listing 3.2: C-to-FPGA Example Function Code

```
1:      int absval_arr(int* a, int *b, int *c) {
2:           int i, a_tmp, b_tmp;
3:           for (i=0; i<16; i++) {
4:                a_tmp = a[i] * a[i];
5:                b_tmp = b[i] * b[i];
6:                if (a_tmp < b_tmp)
7:                     c[i] = b_tmp - a_tmp;
8:                else
9:                     c[i] = b_tmp - a_tmp;
10:          }
11:          return c[0];
12:     }
```

To obtain the bitstream required to configure the FPGA reconfigurable slots, we need to synthesize, place, and route a hardware description file corresponding to the function moved to FPGA. These final implementation steps are well-known problems and are addressed by commercial FPGA vendor tools. Therefore, we use these tools in the last phase of the compilation process (see Xilinx box on the right-hand side tool-chain flow in Figure 3.3).

To illustrate the translation process of a computation expressed in a HLL, to a hardware model expressed in a HDL, consider the example shown in Listing 3.2. The first step in the hardware generation process is to transform the input source into an IR that reflects the control and data dependencies in the algorithm. For this purposes, a hierarchical *Control Data Flow Graph (CDFG)* is used. Figure 3.8a depicts the control flow graph of the whole program, in which the nodes are basic blocks containing one or more program statements that are always on the same path through the code.The edges represent control dependencies between these blocks. Figure 3.8b zooms in basic block

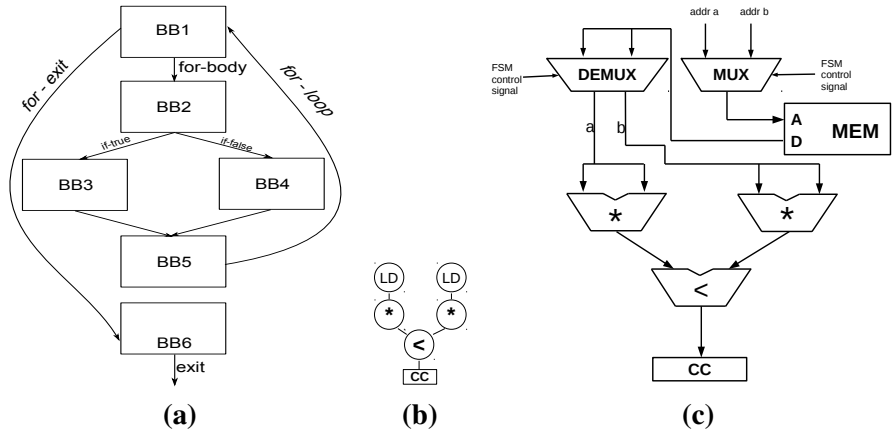


Figure 3.8: C-to-FPGA steps: **(a)** CFG; **(b)** `BB2` DFG; **(c)** `BB2` generated Hardware.

two corresponding to the statements in the for loop before the branch decision is taken about which if path to execute next. That is, two multiplications and one comparison that stores the outcome in the CC conditional register. Because there is no dependency between these instructions, they can run in parallel as illustrated by the data dependency graph. Here, the nodes represent simple hardware instructions to implement the statements and the edges represent the data dependencies between these instructions. The hierarchy levels in the CDFG reflect thus the enclosure level of the statements. The outer nodes of this graph are basic blocks corresponding to grouped statements or called functions. The outer edges of the graph represent the control dependencies, while the basic block inner edges represent data dependencies. The basic blocks nodes contain data flow graphs (DFGs) that describe the data dependencies within the corresponding basic block (Figure 3.8b). From the CDFG, the VHSIC Hardware Description Language (VHDL) compiler has to infer a hardware model for the computation. The arithmetic and logic operations within the basic blocks are mapped to combinational hardware logic (Figure 3.8c). For loop-back edges, i.e. when a control structure such as the `for-loop` in `absval_arr` is found, sequential hardware logic is used.

Both combinational (i.e., simple arithmetic operations that depend only on current inputs) and sequential (i.e., operations that output values based on both current inputs and past outputs) logic is expressed in a HDL. There are several HDLs, but only two of them are widely supported by the FPGA vendors both for simulation and synthesis (i.e., VHDL and Verilog). Because the hardware

logic is always active (i.e., every hardware element is, in every clock cycle, outputting a value based on its current inputs), a controller is required to select which part of the circuit is actually performing useful computations. These controllers, called Finite State Machine (FSM)s, are generated by performing a scheduling of the CDFG.

Listing 3.3: C-to-FPGA Generated VHDL Excerpt for *BB2* DFG

```
when "S" =>
    -- multiply by 4 because i has int type
4:  sig_load_index := var_i << 2;
4:  sig_load_a := var_addr_a + sig_load_index;
5:  reg_load_b <= var_addr_b + sig_load_index;
3:  var_i <= var_i + 1;
4:  DATA_ADDR <= sig_load_a;
    -- FSM instructs now to move to state "S+1"
when "S+1" =>
5:  DATA_ADDR <= reg_load_b;
    -- FSM move to "S+2"(empty)->"S+3"(empty)->"S+4"
when "S+4" =>
    -- assuming a 4 cycles delay to access memory
4:  sig_data_a := READ_DATA;
4:  reg_a_tmp <= sig_data_a * sig_data_a;
    -- FSM move to "S+5"
when "S+5" =>
    -- data reads are pipelined, so read next value
5:  sig_data_b := READ_DATA;
5:  sig_b_tmp := sig_data_b * sig_data_b;
6:  CC <= CMPLT(reg_a_tmp,sig_b_tmp);
    -- FSM move to "S+6"
when "S+6" =>
6:  -- FSM move to IF(CC) ? "S+7" ; "S+8"
```

Listing 3.3 shows an excerpt of how BB2 was scheduled in terms of a simplified VHDL syntax. The listing shows five states from the schedule generated by the hardware compiler when it created the FSM. These states are expressed in terms of the variable S to underline the existing dependencies between the states. The actual FSM control, which is part of a different hardware process, is not shown. The listing shows only instructions that belong to the data-pah process. Nevertheless, comments are placed to highlight what is the next state the

FSM will take and when. Furthermore, the number in the beginning of the line refers to the actual C-code line in Listing 3.2 and is intended to immediately visualize how C-code constructs are mapped to VHDL specifications. Finally, variables preceded by *var_* and *reg_* are registered, whereas those preceded by *sig_* denote that only wires are inferred (i.e., no register is required to store their value) because they are used immediately in the same cycle. As a result, their value does not need to be remembered. *DATA_ADDR* and *READ_DATA* are interface ports of the accelerator through which (dynamic, pointer based) data is loaded and used inside the hardware logic from the local BRAM memory.

One final remark is needed regarding the usability of these kind of C-to-FPGA frameworks. Namely, these frameworks are useful in the embedded software domain, where processor speeds are far less than the full-blown performant CPU frequencies found in modern-day general-purpose computers. For the embedded system domain, other design constraints such as hardware area and power consumption play a very important role, and, as a result, embedded processor speeds are trade-offed with such other constraints. This implies that, when we talk about speedups compared to execution only in software, we refer to software execution on an embedded hard-core software processor.

4

DWARV2.0: A CoSy-based C-to-VHDL Hardware Compiler

In the last decade, a considerable amount of effort was spent on implementing tools that automatically extract the parallelism from input applications and to generate Hardware/Software (HW/SW) co-design solutions. However, the tools developed thus far either focus on a particular application domain or they impose severe restrictions on the input language. In this chapter, we present the DWARV2.0 compiler that accepts general C-code as input and generates synthesizable VHDL for unrestricted application domains. Dissimilar to previous hardware compilers, this implementation is based on the CoSy compiler framework. This allowed us to build a highly modular compiler in which standard or custom optimizations can be easily integrated. Validation experiments showed speedups of up to 4.41x when comparing against another state-of-the-art hardware compiler.

4.1 Introduction

Even though hardware compilers, which take as input a High-Level Language (HLL) and generate a Hardware Description Language (HDL), are no longer seen as exotic technology, they cannot yet be seen as a mature technology to the same extent as software compilers. Hardware compilers are especially used to develop application-specific hardware where for various application domains the computational intensive part(s) are accelerated. They are a vital component of the HW/SW co-design effort needed when FPGA based kernels are involved. In this chapter, we specifically look at FPGA based platforms where parts of the application will stay on the General-Purpose Processor (GPP) and other parts will be transformed into Custom Computing Unit (CCU). To per-

form fast Design Space Exploration (DSE), it is necessary to quickly evaluate the different mappings of the application, with their corresponding HDL implementation, on the hardware platform. For this purpose, hardware compilers allow the designer to obtain immediately a hardware implementation and skip the manual and iterative development cycle altogether.

However, current hardware compilers suffer from a lack of generality in the sense that they support only a subset of a HLL, for example, no pointers or Floating-Point (FP) operations are accepted. Even more, only a few allow the application programmer to use other function calls inside the kernel (i.e., unit) function. This leads to manual intervention to transform the input code to syntax accepted by the compiler, which is both time consuming and error prone. These problems are caused by the fact that hardware generators are typically bound to one particular application domain or are implemented in compiler frameworks that provide cumbersome ways of generating and processing the Intermediate Representation (IR) of the input code. Our contribution is threefold:

- Provide a redesign of the DWARV hardware compiler [97] using the CoSy compiler framework [26] to increase the coverage of the accepted C-language constructs.
- Provide a general template for describing external Intellectual Property (IP) blocks, which can be searched and used from an IP library, to allow custom function calls.
- Validate and demonstrate the performance of the DWARV2.0 compiler against another state-of-the-art research compiler. We show kernel wise performance improvements up to 4.41x compared to LegUp 2.0 compiler [18].

The rest of the chapter is structured as follows. In Section 4.2 we present an overview of existing HDL generators. Section 4.3 gives details about the compiler tool-flow and the template descriptor used for external IP blocks supporting custom function calls. Section 4.4 validates DWARV2.0 by presenting the comparison results while Section 4.5 draws the conclusion.

4.2 Related Work

Plenty of research projects addressed the issues of automated HDL generation. The ROCCC project [93] aims at the parallelization and acceleration of loops.

Catapult-C [14] and CtoS [16] are commercial high-level synthesis tools that take as input ANSI C/C++ and SystemC inputs and generate register transfer level (RTL) code. The optimizations set of the SPARK [41] compiler is beneficial only for control-dominated code, where they try to increase the instruction level parallelism. In addition, the explicit specification of the available hardware resources such as adders, multipliers, etc. is required. In contrast to these compilers, DWARV2.0 does not restrict the application domain and it is able to generate hardware for both streaming and control intensive applications. Furthermore, it does not restrict the accepted input language. DWARV2.0 allows a large set of C constructs including pointers and memory accesses. Finally, no additional user input is necessary.

Altium's C to Hardware (CHC) [8], LegUp [18] and DWARV2.0's predecessor [97] are the compilers that resemble DWARV2.0 the closest. They are intended to compile annotated functions that belong to the application's computational intensive parts in a HW/SW co-design environment (although the latter can compile the complete application to hardware as well). They are therefore intended to generate accelerators for particular functions and not autonomous systems. This is typical for Reconfigurable Computing (RC) Systems and the same assumption is true for DWARV2.0 as well. However, there are also two major differences, the IP reuse and the more robust underlying framework. The first feature allows custom function calls from the HLL code to be mapped to external IP blocks provided they are available in external IP libraries. The second feature enables seamless integration of standard or custom optimization passes.

4.3 DWARV 2.0

In this section, we describe the DWARV2.0 compiler by highlighting the improved aspects comparing with the previous version. We present the engine flow, the new features and describe the IP library support.

4.3.1 DWARV2.0 Engines: The Tool-Flow

DWARV2.0 targets reconfigurable architectures following the Molen [73] machine organization and is built with CoSy [26]. Compilers built with CoSy are composed of a set of *engines* which work on the IR of the input program. In the following text, the engines in *italics* are standard CoSy engines available for use in the framework, and as such, they were used by simply plugging

them in the DWARV2.0 compiler. The engines in **bold** are custom written engines designed specifically for the hardware generation process. The initial IR is generated by the C frontend, which is a CoSy standard framework engine. To generate VHDL from C code, DWARV2.0 performs standard and custom transformations on the combined Control Data Flow Graph (CDFG) created in the IR by the *CFront* engine. Figure 4.1 depicts this process graphically, highlighting on the left side the three main processing activities required for C-to-VHDL translation. On the right side of the same figure, we show in clockwise order an excerpt of the most important engines used in each activity box shown on the left side.

The *CFront* (ANSI/ISO C front end) creates the IR. The *cse* and *ssa* engines perform common subexpression elimination and static single assignment transformations. The *match* engine creates rule objects by matching identified tree patterns in the IR while the **psrequiv** engine annotates which register/variable actually needs to be defined in VHDL. **fplib** searches and instantiates hardware templates found in the library. **hwconfig** reads in parametrizable platform parameters, e.g., memory latency. **setlatency** places dependencies on def/use chains for registers used by IP cores. It also sets the latencies on memory dependencies. *sched* schedules the CDFG and *dump* prints IR debug information. Finally, the **emit** engine emits IEEE 754 synthesizable VHDL. The engines given in bold in Figure 4.1 are custom and thus written specifically for VHDL generation. The remaining ones are standard framework engines. A total of 52 (43 standard - 9 custom) engines were used in DWARV2.0.

4.3.2 New Features and Restrictions

Tables 4.1 and 4.2 summarize DWARV2.0's new features. Leveraging the availability of generic lowering engines, which transform specific constructs to basic IR operations, most of the previous syntax restrictions were removed. The best example is the support for structured aggregate data types. Another major development was the FP and the template library. This not only facilitates the addition of FP operations, but provides also a generic mechanism to support function calls.

To add support for the basic FP arithmetic, we use the Xilinx tool *coregen* first to generate FP cores (e.g., for multiplication). Then, we describe these generated IP cores in a library that DWARV2.0 is able to search for an appropriate core for each of the floating point operations. Important fields that DWARV2.0 must know in order to find the proper core, instantiate, and schedule it in the design are IP name, list of input and output ports, operation type and operand

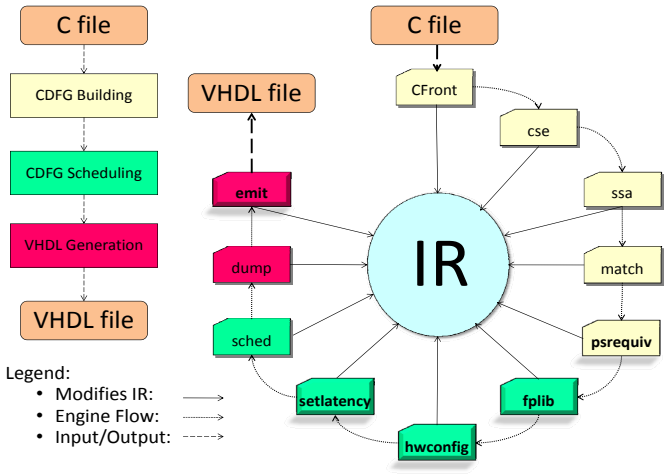


Figure 4.1: DWARV 2.0 Engines. Clock-wise Sequential Execution of Engines Starting from CFront.

Table 4.1: DWARV 2.0 vs. DWARV 1.0 Allowed Data Types.

Data Type	**DWARV 1.0 Supported**	**DWARV 2.0 New Features**	**DWARV 2.0 Not Supported**
Boolean		_Bool	
Integer	32 bits	64 bit	custom sizes
Real Floating Point		all	
Fixed Point		DSP-C	
Complex and Imaginary			all
Pointer	non-local pointer	limited local pointer	func ptr ptr to ptr
Aggregate	1-dim array	n-dim array global struct	local struct
Union		all	

Table 4.2: DWARV 2.0 vs. DWARV 1.0 Allowed Statements.

Statement Type	DWARV 1.0 Supported	DWARV 2.0 New Features	DWARV 2.0 Not Supported
Expression	unary,add,mul shift,bitwise,assign	div,mod, cast logic and, logic or	global variables
Labeled		case,label	
Jump		return, break goto, continue	
Selection	if	switch	
Iteration	for	while, do-while	
Calls		functions	recursion

sizes as well as latency and frequency of the core. The same syntax can be used also to describe and support generic function calls. The only exception is that for the operation field name, instead of using an *operation type* identifier, we simply use the function name.

Although the new version eliminates most restrictions from the first version of DWARV, there are still some restrictions left. The first two restrictions are related to the fact that there is no stack on an FPGA. This implies that functions can not be recursive and that static data is not supported. Implementing a stack would be possible, but would defeat the purpose of hardware execution because it will limit the available parallelism. The third restriction is that mathematical functions present in the standard C library are not available. This restriction can be lifted in the future using the described function call support.

4.4 Experimental Results

To assess the performance of DWARV2.0, we compared cycle, frequency and area information obtained by generating and simulating the CCU hardware for eight kernels against the hardware IP produced by the LegUp 2.0 compiler from Toronto University [18]. In this section, we briefly describe the LegUp 2.0 compiler, the platform and the comparison experiments.

LegUp 2.0 Compiler

LegUp 2.0 [18] is a research compiler developed at Toronto University which was developed using LLVM [66]. It accepts standard C-language as input, and generates Verilog code for the selected input functions. Its main strength is that it can generate hardware for complete applications or only for specific application functions, i.e., accelerators. In this latter case, a TigerMIPS soft processor [65] is then used to execute the remainder of the application in software. The connection between these two main components is made through an Avalon system bus. This is similar to the Molen machine organization, therefore comparing the execution times of accelerators generated by this tool is relevant to assess the performance and development state of DWARV2.0. LegUp 2.0 was reported [18] to perform close to an industrial HLS compiler, i.e., eXCite [28], which, assuming transitivity of results, was another reason to use LegUp 2.0 as our benchmark.

Experimental Platform

To compare the DWARV2.0 and LegUp 2.0 compilers, we followed a two-step approach. First, we simulated the generated kernels to obtain the cycle information. The simulation infrastructure for DWARV2.0 is in such a way designed to return only the execution time for the individual kernel invocation. However, for the LegUp 2.0 simulation, care has to be taken to obtain only the execution time for the kernel itself and not for the complete testbench as it is currently reported when the hybrid execution is chosen. To obtain the correct number, the ModelSim wave-form had to be opened, and the difference between the start and finish signals had to be computed.

Subsequently, we ran a full post-place and route synthesis to obtain the maximum frequency and area numbers for the Xilinx Virtex5 ML510 development board. To obtain a meaningful comparison, we needed to integrate the LegUp 2.0 generated kernels in the Molen workflow to target the same board. To this purpose, we wrote wrappers around the LegUp 2.0 interface. Note that these wrappers do not influence the performance comparison. We use these wrappers only for integration purposes to be able to target a different platform than the one for which LegUp kernels were generated. Doing so, we are interested only in the area numbers obtained for Xilinx instead of Altera board. The performance numbers are computed thus in the original setting without any wrappers included. Given that the tools target similar heterogeneous platforms with the accelerated kernels running on the Xilinx board as co-processors of the Pow-

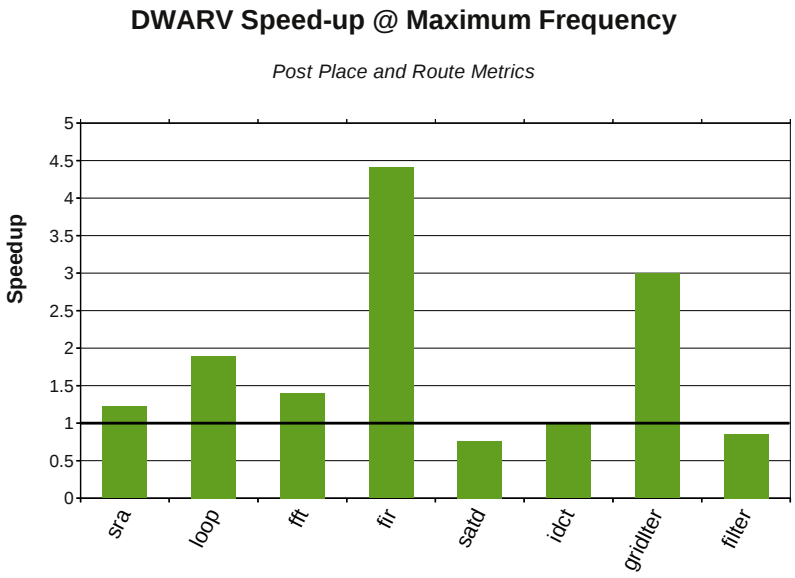


Figure 4.2: DWARV 2.0 Speedups vs. LegUp 2.0 times.

erPC processor vs. Altera/TigerMIPS platform, the mismatch in interfaces was minimal and easy to correct. Both interfaces contained ports to start the accelerator, query its status and read/write data from the shared memory. Therefore, bridging the gap between these interfaces was only a matter of connecting the proper ports to each other, e.g., *DATA_ADDR* to *memory_controller_address*.

DWARV vs. LegUp Comparison

We perform two kinds of comparisons: one that focuses on speedup and area consumption and one on the restrictions imposed on the C-code. To measure the speedup and area, we have selected eight kernels for testing. The first four, i.e., *loop, sra, fft, fir*, were extracted from the examples directory in the LegUp 2.0 distribution, whereas the other four were taken from DWARV2.0's testbench. All eight functions compiled without any C-language syntax modifications in both tools. Furthermore, the approach described above was followed. The results are summarized in Table 4.3, whereas Figure 4.2 shows DWARV2.0 speedup information for all test cases relative to the times obtained for LegUp 2.0. The computed speedups were obtained by considering the number of cycles at the maximum frequency reported by the Xilinx post place and route synthesis, except for the *idct* kernel. For this kernel, the initial

Table 4.3: Evaluation Numbers - DWARV2.0 vs. LegUp 2.0.

Kernel	**Slices**	**Cycles**	**Max. Freq** (xst[2])	**ETAMF**[1] (xst)	**Speedup** (xst)	**Max. Freq** (real[3])	**ETAMF** (real)	**Speedup** (real)
sra-legup	370	70	261	0.27	0.82	202	0.35	0.82
sra-dwarv	338	64	290	0.22	1.22	225	0.28	**1.22**
loop-legup	122	292	352	0.83	1.24	251	1.16	**1.30**
loop-dwarv	122	380	368	1.03	0.80	252	1.51	0.77
fft-legup	1980	7377	125	59.02	0.76	98	75.28	0.71
fft-dwarv	3198	8053	180	44.74	1.32	150	53.69	**1.40**
fir-legup	320	223	124	1.80	0.33	80	2.79	0.23
fir-dwarv	1063	127	213	0.60	3.02	201	0.63	**4.41**
satd-legup	1189	132	175	0.75	1.29	150	0.88	**1.31**
satd-dwarv	1201	265	272	0.97	0.77	230	1.15	0.76
idct-legup	N/A	24004	88	320.05	*1.00*	N/A	N/A	N/A
idct-dwarv	9519	41338	151	273.76	*1.00*	75	551.17	N/A
gridIterate_fixed-legup	455	471348	102	4621.06	0.26	100	4713.48	0.33
gridIterate_fixed-dwarv	1343	355810	294	1210.24	3.82	226	1574.38	**2.99**
filter_subband_fixed-legup	342	21464	158	135.85	1.46	103	208.39	**1.17**
filter_subband_fixed-dwarv	386	55137	278	198.33	0.68	226	243.97	0.85

[1]**E**xecution **T**ime **A**t **M**aximum **F**requency

[2]Estimated Maximum Frequency after Behavioural Synthesis

[3]Real Maximum Frequency after Post Place and Route Synthesis

maximum frequency estimation was used. LegUp 2.0 *idct* kernel could not be synthesized targeting Xilinx because it contained an instantiation of an Altera proprietary/specific IP block used for integer division. We compared the execution times at the kernel level only which gives an indication of the quality of the generated HDL.

Analyzing the last column in Table 4.3, we observe that performance-wise, DWARV2.0 gave a speedup for four kernels, *icdt* provided no improvement or degradation (6th column), whereas the other three functions incurred a decrease in performance. These speedup numbers were computed by first calculating the **E**xecution **T**ime achieved **A**t **M**aximum **F**requency (ETAMF) reported for the two hardware versions, i.e., $ETAMF = Cycles/Max.Freq.$. Next, $Speedup_{dwarv} = ETAMF_{legup}/ETAMF_{dwarv}$.

With respect to the hardware area, DWARV2.0 produces less than optimal hardware designs because no optimization passes that target area reduction were used. Our primary focus was functional correctness and to obtain a basis for comparison for future research. As an example of such future research, consider the loop kernel case study. Only by integrating the standard CoSy framework engines *loopanalysis* and *loopunroll*, which annotate respectively unroll simple loops, we decreased the number of cycles for this kernel from 380 to 113. Given the newly obtained frequency of 256 MHz, we were able to obtain a speedup of 1.90 for this example as well (initial numbers are given in Table 4.3 where we can see that the first implementation in DWARV2.0 gave a 0.77 slowdown). Figure 4.2 shows the final results obtained after this simple optimization was applied. Even though the *loopunroll* engine can provide considerable performance benefits, determining an unroll factor is not a trivial problem. If the unroll factor is too big, the generated VHDL will not synthesize due to lack of available area. Chapter 6 will address this problem.

To gain more insight on how the two tools compare and decide if a tool is performing better than the other one, we illustrate in Figure 4.3 the performance ratios in terms of area ratios. This graph shows that for three kernels (i.e., *fir*, *gridderate* and *fft*), DWARV2.0 generated designs perform better, but at the cost of more hardware logic. This means that these points are not actually comparable because LegUp 2.0 generates more balanced area-performance designs, while for DWARV2.0 the focus is only on performance. *filter*, *loop* and *satd* are kernels for which DWARV2.0 is slower than LegUp 2.0, given area is about the same. Finally, *sra* is faster in DWARV2.0 under the same conditions.

The second comparison focused on the extent that the compilers are capable of compiling a large subset of the C-language without requiring substantial

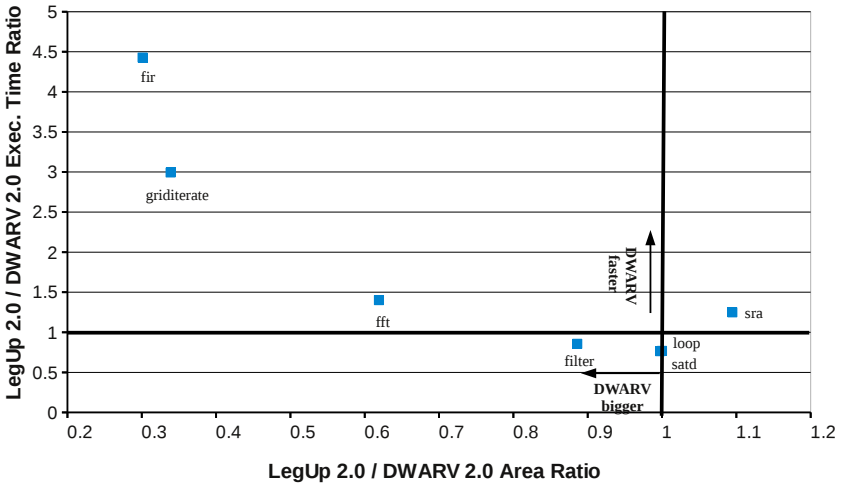


Figure 4.3: DWARV 2.0 vs. LegUp 2.0 Execution Time per Area Comparison.

rewrites. To do this, we used our internal benchmark, which is a database of 324 kernels from a wide variety of application domains. For example, the cryptography domain contains 80 kernels and the mathematical domain contains 70 kernels. Other domains available in the benchmark are physics, multimedia, DSP, data processing and compression. Simply invoking the two compilers with this database, we observed that DWARV2.0 is able to generate synthesizable VHDL for 82.1% of the kernels, whereas LegUp 2.0 for 65.7%. However, LegUp 2.0 does not support FP operations and, as such, the ability to generate correctly VHDL for our kernel library is degraded. When we ignored the kernels containing FP operations, the performance increased to 87.7%.

4.5 Conclusion

In this chapter, we presented the DWARV2.0 compiler, and we did a performance comparison with another academic hardware compiler. We conclude that the current version provides a good basis for future research of hardware related optimizations. One of the most important advantage of DWARV2.0, compared to the previous version, is that it is highly extensible. Extending the compiler can be achieved by including standard CoSy or custom (new) engines, or can involve extensions in the IR. CoSy's mechanism of extending the IR guarantees that the correctness of the code already written is not affected.

Note.

The content of this chapter is based on the following paper:

R. Nane, V.M. Sima, B. Olivier, R.J. Meeuws, Y.D. Yankova, K.L.M. Bertels, **DWARV 2.0: A CoSy-based C-to-VHDL Hardware Compiler**, 22nd International Conference on Field Programmable Logic and Applications (FPL 2012), Oslo, Norway, September 2012.

5

IP-XACT Extensions for Reconfigurable Computing

MANY of today's embedded multiprocessor systems are implemented as heterogeneous systems, consisting of hardware and software components. To automate the composition and integration of multiprocessor systems, the IP-XACT standard was defined to describe hardware Intellectual Property (IP) blocks and (sub)systems. However, the IP-XACT standard does not provide sufficient means to express Reconfigurable Computing (RC) specific information, such as Hardware-dependent Software (HdS) metadata, which prevents automated integration. In this chapter, we propose several IP-XACT extensions such that the HdS can be generated and integrated automatically. We validate these specific extensions and demonstrate the interoperability of the approach based on an H.264 decoder application case study. For this case study we achieved an overall 30.4% application-wise speedup and we reduced the development time of HdS from days to a few seconds.

5.1 Introduction

Heterogeneous systems combine different hardware cores (e.g., different processor types) to enable the execution of software on different pieces of hardware to increase performance. A trivial example is a general-purpose processor tightly coupled with a co-processor to perform floating-point operations fast. Such a system can thus be used to off-load computational-intensive application codes onto specialized hardware units. The objective is to gain performance. However, such a system is beneficial only in predefined scenarios. The system's specialized hardware units are not useful for other software algorithms.

To extend the range in which heterogeneous systems can be applied, reconfigurable hardware devices were introduced. These have the advantage that the hardware design can be customized and configured on a per case basis.

A widely adopted practice within Reconfigurable Computing (RC) design is to accelerate part(s) of applications, using custom hardware architectures, that are specifically tailored for a particular application. These specialized architectures can be IP blocks written at the Register Transfer Level (RTL) by a designer, or IP blocks generated by a High-Level Synthesis (HLS) tool from a functional specification written in a High-Level Language (HLL) [97]. To cope with the diversity of IP blocks coming from different sources, IP-XACT [1] was introduced. Using IP-XACT, hardware components can be described in a standardized way. This enables automated configuration and integration of IP blocks, aiding hardware reuse and facilitating tool interoperability [56].

In a Hardware/Software (HW/SW) system, connecting the different HW and SW components, using for instance buses or point-to-point connections, is not sufficient to fully implement a system. Typically, a SW component connected to a HW component needs a driver program, also known as *Hardware-dependent Software* (*HdS*) [98], to control the HW component. IP blocks that can be controlled from a SW component are typically shipped with particular HdS to ensure proper control from SW. However, in RC systems, the IP blocks are automatically generated by HW tool-chains for application kernels selected for hardware acceleration. Therefore, the HdS driving these new hardware blocks has to be generated automatically as well. The compilation process in such RC systems, i.e., from HLL application source code to a combined HW/SW executable, is done by different tools, such as partitioning, mapping and HW/SW generation tools. This implies that there is no central place from where the HdS can be generated. That is, the compiler used to generate the IP has no knowledge about what HW primitives are used for example to communicate data in the system, which prevents it from generating a proper driver. This information is available, however, in the partitioning and mapping tool. Therefore, we adopt a layered solution in which different parts of the HdS are generated at different points in the tool-flow. Furthermore, to allow the tools involved in this HdS generation process to communicate seamlessly with each other, we need to describe the software requirements of each step in IP-XACT as well. One example of such a software requirement is the number of function input parameters. However, unlike the RTL constituents of an IP block, which can be already described using the current IP-XACT standard, there is no standardized way to describe the driver information for an IP.

In this chapter, we elaborate on the expressiveness of IP-XACT for describing HdS metadata, addressing the second challenge of this dissertation. Furthermore, we address the automation of HdS generation in the RC field, where IPs and their HdS are generated on the fly, and, therefore, are not fully predefined. The contribution of this chapter can be summarized as follows:

- We combine two proven technologies used in MPSoC design, namely IP-XACT and HdS, to automatically integrate different architectural templates used in RC systems.
- We investigate and propose IP-XACT extensions to allow automatic generation of HdS in RC tool-chains.

The rest of the chapter is organized as follows. Section 5.2 presents IP-XACT, other HdS solutions, and already proposed IP-XACT extensions. Section 5.3 describes a HdS generation case study and investigates the IP-XACT support for automation. Section 5.4 elaborates on the identified shortcomings and proposes IP-XACT extensions to support software related driver descriptions. Section 5.5 respectively Section 5.6 validates the automated integration and concludes the chapter.

5.2 Related Work

The IP-XACT standard (IEEE 1685-2009) [1] describes an XML schema for metadata modeling IP blocks and (sub)systems. The metadata is used in the development, implementation, and verification of electronic systems. In this chapter, we focus on *Component* schema for associating HdS to HW IP blocks and we focus on *Generator-Chain* schema to express compiler specific requirements. The current schema provides limited support for software descriptions. Namely, one can only attach software file-sets to a component and describe the function parameter's high level types. However, it does not offer means to assign semantics to the attached file-set and how it should be used during integration of a complete system. Furthermore, it lacks means to model tool chains in which complex software generators are to be integrated. In Section 5.4, we propose solutions to these problems.

The OpenFPGA CoreLib [95] working group focused on examining the IP-XACT Schema and proposed extensions for facilitating core reuse into HLLs. Wirthlin et al. [4] used XML to describe common IP block elements and defined their own schema using IP-XACT syntax. They proposed a lightweight

version intended for Reconfigurable Computing (RC) systems, such as interface specifications and capturing HLLs data types information.

Other IP-XACT related research is focusing on extending the schema to incorporate semantic information about IP elements. Kruijtzer et al. [54], proposed adding *context labels* to provide additional meaning to IP-XACT components. They use this to assess the correctness of interconnections in the system. Strik et al. [76] studied aspects regarding IP (re)configurability to reuse these after a partial change of some parameters. They underline that IP-XACT is missing *expression evaluation* fields to support flagging illegal (sub)system composition. However, all proposed extensions discussed so far in this section consider only the HW IP block. As mentioned in Section 5.1, for systems involving both HW and SW, one also needs to describe the HdS belonging to the HW IP to enable automated integration of a system. Therefore, we propose software related extensions for IP-XACT.

5.3 Integrating Orthogonal Computation Models

To investigate the IP-XACT capabilities to model HW/SW co-design tool chains supporting HdS generation and tool interoperability, we used an H.264 decoder application implemented on an *Field-Programmable Gate Array* (*FPGA*) as a case study. The goal is to integrate different tools and models such that we can automatically generate application specific MPSoC implementations of sequentially specified applications. To realize this, we use the Daedalus [64] system-level synthesis toolset to implement an MPSoC from sequential C code. In particular, from the Daedalus tool set we use the PN compiler [92] to partition a sequential application into *Polyhedral Process Networks* (*PPN*) and ESPAM [42] to map the partitioned application onto an FPGA. Outside the Daedalus toolset, we use DWARV2.0 (see Chapter 4) to automatically generate hardware IP blocks for performance critical parts of the sequential C code. We first describe the problems observed when integrating the two tools in Section 5.3.1. Subsequently, we present our extended framework in Section 5.3.2.

5.3.1 IP Core Integration

In [90], the PICO compiler from Synfora Inc. [3] was incorporated in the Daedalus tool-flow. This approach was used to achieve higher performance for PPN implementations by replacing computational-intensive nodes with func-

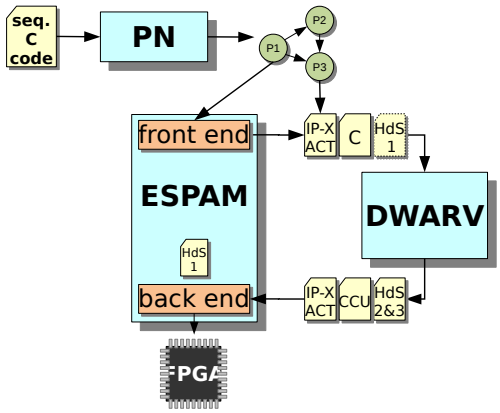


Figure 5.1: H.264 Generation Tool-Chain Flow.

tionally equivalent hardware IP cores generated by PICO from the available C code. The replacement was done smoothly, as both tools were operating under the same memory model, i.e., a distributed memory model. However, several restrictions were imposed on the C code that can be processed by the PICO compiler. For instance, each loop body could contain only one other loop. Therefore, using PICO as the hardware compiler was not feasible for the H.264 application where multiple nested loops are present. DWARV2.0 is a compiler which has less restrictions than PICO, making it suitable for generating hardware blocks for our case study.

However, integration of a *Custom Compute Unit* (*CCU*), generated by DWARV2.0, in a PPN created by Daedalus is not straightforward. The CCU has an interface suitable for interacting with Molen [73] based platforms, which employ a shared memory model. The PPN node, on the other hand, into which the CCU has to be integrated, has multiple input and output FIFO channels typical for the stream-based distributed memory model. The challenge therefore is to find a way to specify the requirements for Daedalus such that DWARV2.0 can automatically generate the correct interface.

5.3.2 Framework Solution

We show our solution in Figure 5.1. We use the PN compiler to create a PPN from the sequential C code of the H.264 top level function. Subsequently, we use ESPAM to implement the H.264 PPN as a system of point-to-point connected MicroBlaze processors on an FPGA, as shown in the left part of Figure 5.1. This means the functional part of each process is implemented as

a software program running on a MicroBlaze. Based on profile information we have decided to accelerate the Inverse Discrete Cosine Transform (*IDCT*) process using a specialized hardware component. We use the DWARV2.0 C-to-VHDL compiler to generate a CCU from the C code of the IDCT function, which requires the C function to be communicated from ESPAM to DWARV2.0.

To solve the interface mismatch problem between DWARV2.0-generated CCUs and Daedalus' PPNs, DWARV2.0 generates a wrapper for the CCU. This wrapper provides memory to the CCU which stores the input/output channel data before/after the CCU is started/stopped.

The HdS controlling the CCU is structured into three different layers. The right side of Figure 5.2 shows the HdS hierarchy. We distinguish platform primitives (layer 1), IP- and OS-specific driver code (layer 2) and an application layer (layer 3). The primitives in layer 1 strongly depend on the processor the HdS is running on, and the way the CCU is connected to the processor running the HdS. For instance, for one processor these primitives use memory-mapped I/O, whereas for another processor dedicated instructions are available. This information is known only by ESPAM. Therefore, the HdS layer 1 primitives are generated by ESPAM. HdS layer 2 provides functions that control the CCU by sending and receiving commands and data to and from the CCU using the primitives provided by layer 1. The separation of HdS layers 1 and 2 makes the HdS layer 2 code independent of the actual platform. HdS layer 3 provides user level functions, which are invoked by a user application to perform the task for which the CCU was designed. The functions in layer 3 only use functions provided by HdS layer 2. The HdS layer 3 function(s) provide a transparent interface to the CCU, essentially making the CCU available as a regular software function.

5.4 IP-XACT Extensions

In this section, we elaborate on the expressiveness of the current IP-XACT standard to describe the scenario presented in the previous section. Based on this analysis, we describe three possible extensions, namely hardware compiler input, HdS and tool-flow integration related extensions. We implemented the proposed extensions using the *vendorExtensions* extension construct that is already part of IP-XACT. This allows vendor specific information to be added to IP-XACT descriptions.

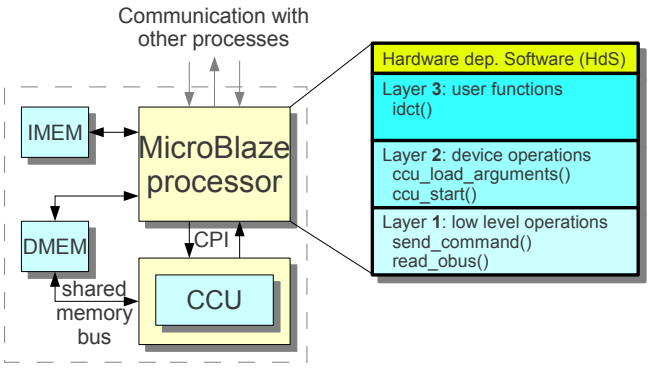


Figure 5.2: Connection between CCU and processor (left) and HdS layers (right). IMEM is the instruction memory of the processor, while DMEM is the data memory that is shared between both the processor and the CCU.

5.4.1 Hardware Compiler Input

The DWARV2.0 compiler accepts a C function as input and generates a VHDL component that implements the original C function. In our case study, we send the C function together with an IP-XACT metadata description to DWARV2.0. DWARV2.0 assumes that a function argument can be both input and output at the same time, if the argument is a pointer into shared memory. However, arguments to functions inside a PPN process are always unidirectional. For each function argument, unidirectional First In, First Out (FIFO) channels are created according to the process network topology. Therefore, we need to inform DWARV2.0 about the direction of each function argument, such that the appropriate FIFO input or output connection can be generated. We therefore add a **direction** field to the IP-XACT description that is passed along with the C file defining the function implementation. The values this field can take are: *in*, *out* and *inout*.

5.4.2 Hardware-Dependent Software

Using HdS in heterogeneous MPSoCs abstracts hardware and OS details away from the application level. In our case study, we have partitioned the HdS into three different layers, as described in Section 5.3.2. HdS layer 1 is generated by the Daedalus environment and then passed to DWARV2.0. This enables DWARV2.0 to generate HdS layers 2 and 3 that make use of the primitives

```
<spirit:function>
  <spirit:entryPoint>send_data</spirit:entryPoint>
  <spirit:fileRef>f-hds1_h</spirit:fileRef>
  <spirit:returnType>void</spirit:returnType>
  <spirit:argument spirit:dataType="int">
    <spirit:name>data</spirit:name>
    <spirit:value>0</spirit:value>
  </spirit:argument>
  <spirit:vendorExtensions>
    <spirit:hdstype>write</spirit:hdstype>
  </spirit:vendorExtensions>
</spirit:function>
```

Figure 5.3: HdS IP-XACT extensions for layer 1.

provided by HdS layer 1.

To create a semantic link between two different HdS layers, we need to specify the purpose of the functions found in HdS1. For HdS layer 1, we classify a function as *read*, *write* or *command*. An example of such a description in IP-XACT is shown in Figure 5.3. The *read* identifier classifies the function as one that reads data from the *CCU-Processor Interface* (*CPI*), which has been implemented using two small FIFO buffers. The *write* identifier classifies the function as one that writes application data to the CPI and the *command* identifier classifies a function as one that writes control data to the CPI. Because hardware primitives are typically limited in number, we define a new IP-XACT type **HdS type** to establish a semantic link between layers 1 and 2.

Similarly, we can create a link between layers 2 and 3. However, layer 2 is concerned with abstracting OS specific implementation details for the custom IP block, and since there is no OS present in our case study, we leave the definition of this type as future work. Nevertheless, we imagine that this type could include identifiers for the *POSIX* standard such as opening and closing file handles.

5.4.3 Tool Chains

To fully automate the tool flow shown in Figure 5.1, IP-XACT provides means to model generator chains. For example, the current IP-XACT standard provides a *generatorExe* field which contains the executable to be invoked for a *generator*. However, we observe that IP-XACT currently lacks a way to describe the tool-specific configuration files. For example, DWARV2.0 uses an external Floating-Point (FP) library description file listing the available FP cores, such that floating-point arithmetic in the C code can be implemented us-

ing available FP cores. To allow seamless cooperation of different tools from different vendors, we observe the need to include tool-specific descriptions and files in IP-XACT generatorChain schema.

5.5 Experimental Results

In this section, we report on two kinds of results. First, we show the applicability and usefulness in a real world application and second, we report the overall productivity gain. We base this on our experience with the H.264 case study, for which the first implementation was done manually.

5.5.1 Validation of Approach

In our experiments, we target a Xilinx Virtex-5 XC5VLX110T-2 FPGA and use Xilinx EDK 9.2 for low-level synthesis. We use *QCIF* (176x144 pixels) video resolution and a 100 MHz clock frequency. To validate the approach, we implement the H.264 decoder application twice. The first time we map all processes of the PPN onto MicroBlaze processors, which means all PPN processes are implemented as software. This serves as our reference implementation. The second time we replace the software IDCT node with a hardware version obtained using the methodology described in the previous sections. We obtain a speedup of approximately 30.4%.

5.5.2 Productivity Gain

Besides proving the usefulness of the approach to obtain a faster implementation of a PPN, we discuss the productivity gain observed when adopting an automated IP-XACT based approach. If the automated support was not available, manually patching the tools would have been time consuming and error-prone. Depending on the application knowledge of the system designer and application complexity, activities like writing the HdS or the CCU wrapper can take from a few hours up to even weeks. Moreover, validation may take a similar amount of time. For example, a memory map has to be specified as C code in the HdS and as VHDL in the RTL. For correct operation of the system, these two representations need to be fully consistent, which may be an important source of errors when manual action is involved. We eliminate such errors by taking the information from a central source (e.g., an IP-XACT description) and then automatically generate the different representations. This

substantially reduces the time needed for validation. To fully understand the specific challenges and properly design the modifications required by the tools to enable automated integration, our first implementation of the system was manual. Based on this experience, we estimate that building a fully working system for the H.264 decoder application by hand would take one week. Using the approach described in this work, one could obtain a working system in less than an hour, which is a considerable gain in productivity.

5.6 Conclusion

In this chapter, we have presented a new approach for automated generation of RTL implementations from sequential programs written in the C language. This is achieved by combining the Daedalus framework with the DWARV2.0 C-to-VHDL compiler with the aid of the IP-XACT standard. With these concepts, even different architectural templates can be reconciled. We investigated the capabilities of the IP-XACT standard to model automated integration of MPSoCs consisting of both hardware and software components. We found that the Hardware Dependent Software needed to control the hardware component cannot be described in the current IP-XACT standard. We identified three possible concepts that could be added as extensions to the IP-XACT standard to realize automated integration of HW/SW systems. Using an H.264 video decoder application we validated our approach.

Note.

The content of this chapter is based on the following paper:

R. Nane, S. van Haastregt, T.P. Stefanov, B. Kienhuis, V.M. Sima, K.L.M. Bertels, **IP-XACT Extensions for Reconfigurable Computing**, 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2011), Santa Monica, USA, September 2011.

6

Area Constraint Propagation in High-Level Synthesis

HARDWARE compilers, which generate hardware descriptions from high-level languages, are rapidly gaining in popularity. These generated descriptions are used to obtain fast implementations of software/hardware solutions in heterogeneous computing platforms. However, to obtain optimal solutions under certain platform constraints, we need intelligent hardware compilers that choose proper values for the different design parameters automatically. In this chapter, we present a two-step algorithm to optimize the performance for different area constraints. The design parameters under investigation are the maximum unroll factor and the optimal allocation of resource[1] types. Experimental results show that generated solutions are mapped into the available area at an occupancy rate between 74% and 99%. Furthermore, these solutions provide the best execution time when compared to the other solutions that satisfy the same area constraint. Finally, a reduction in design time of 42x on average can be achieved when these parameters are chosen by the compiler compared to manually selecting them.

6.1 Introduction

Heterogeneous multi-core architectures are a direct consequence of the end of Moore's law. In many cases, this heterogeneity is being implemented by means of FPGA based custom computing units. The Xilinx Zynq at the embedded side and the Convey HC-1 on the supercomputing side are just a couple of the telling examples. The FPGA blades allow to provide application-specific hardware support, which can even be modified at run-time, and thus, provide a tai-

[1] resources and functional units are used interchangeably in the text

lored support for different application domains. The gain that can be obtained by combining a traditional processor with a Reconfigurable Architecture (RA) can be tremendous (e.g., between 20x and 100x [40]). However, before the potential of this technology can be fully exploited, a number of challenges have to be addressed. One of the challenges is the automatic generation of the hardware units through e.g., C-to-VHDL generation, while a second important challenge is to have an efficient way to explore the design space. This chapter primarily focuses on the second challenge.

The strength of RAs is that they offer much more design freedom than a General-Purpose Processor (GPP). In this work, we rely on such architectures to maximize the application performance by automatically exploiting the available parallelism subject to area constraints. In particular, we look at application loops in more detail as these constructs provide a greater source of performance improvement, also in hardware synthesis. Considering the scenario where Hardware Description Language (HDL) code is automatically generated, two important parameters have to be explored, namely, the degree of parallelism (i.e., the loop unrolling factor) and the number of functional modules used to implement the source High-Level Language (HLL) code. Determining these parameters without any human intervention, is a key factor in building efficient HLL-to-HDL compilers and implicitly any Design Space Exploration (DSE) tools.

This chapter presents an optimization algorithm to compute the above parameters automatically. This optimization is added as an extension to the DWARV2.0 hardware compiler (see Chapter 4) which generates synthesizable VHDL on the basis of C-code. The contributions of this chapter are:

- The automatic determination of the maximum unroll factor to achieve the highest degree of parallelism subject to the available area and the function characteristics.
- The automatic computation of the number of functional units instantiated by the compiler, to optimize the performance given the previously identified unroll factor and respecting the given design constraints.
- The validation of the algorithm through an implementation on an operational platform.

The rest of the chapter is organized as follows. Section 6.2 presents the background and related research. In Section 6.3 the details of the algorithm are

presented while in Section 6.4 the experimental results are discussed. Finally, Section 6.5 draws the conclusions and highlights future research activities.

6.2 Background and Related Work

The MOLEN Machine Organization [73] is an architecture developed at TU DELFT that facilitates Hardware/Software (HW/SW) co-design. It includes three main components, a GPP, Custom Computing Unit (CCU) used as an accelerator and a shared memory between them. The CCUs are implemented on a reconfigurable (FPGA based) platform. In order to create an accelerator for this platform, we use DWARV2.0, a C-to-VHDL compiler, that generates a CCU for an application kernel. More information about DWARV2.0 will be given below. The generated CCU complies with a simple interface that contains ports to enable the exchange of data and control information. This allows changing the CCU while the system is running, without modifying the hardware design, thus allowing multiple applications and CCUs to execute at the same time. Given enough resources, multiple CCUs can be executed in parallel, taking advantage of inherent application parallelism. To manage the reconfigurable area we divide it (logically) into slots, in which one CCU can be mapped. Each slot can be used by a different application. However, these slots can be combined to allow different sized kernels to be mapped corresponding to different design goals. For example, possible layouts include 5 CCUs that each use an equal area or 2 CCUs, having an area ratio of 3/2. Having only one slot using all the available area is another possible scenario.

DWARV2.0 (see Chapter 4) is a C-to-VHDL hardware compiler built with CoSy Compilers Framework [26]. Compilers built with CoSy are composed of a set of *engines* which work on the Intermediate Representation (IR) generated based on the input program. The initial IR is generated by the frontend. To generate VHDL from C code, DWARV2.0 performs standard and custom transformations to the combined Control Data Flow Graph (CDFG). The ROCCC project [48, 93] aims at the parallelization and acceleration of loops. Catapult-C [14] and CtoS [15] are commercial high-level synthesis tools that take as input ANSI C/C++ and SystemC inputs and generate register transfer level (RTL) code. However, these compilers are complex and require extensive designer input in both the applied optimizations and the actual mapping process, making it less viable for a software designer that does not have in-depth hardware knowledge. Both Altium's C to Hardware (CHC) [8] and LegUp [18] compilers are intended to compile annotated functions that belong to the ap-

plication's computational intensive parts in a HW/SW co-design environment.

However, none of these compilers, including DWARV2.0, currently posses any capabilities to generate hardware that satisfies a particular area constraint, while maximizing performance. More precisely, requiring that a function's generated HDL takes no more than a given area is not possible. Neither the unroll factor, nor the number of functional units used are determined taking into account the execution time or the area. Performing this optimization automatically would enable high-level tool-chains to analyze different application mappings in a shorter amount of time. For example, the algorithm presented in [74], where the best mapping of CCUs is selected given a fixed number of resources and fixed kernel implementations, would be improved if the implementations would be generated according to some determined area constraint.

6.3 Area Constrained Hardware Generation

In this section, we present the model that allows the compiler to decide the unroll factor and the number of resources. In the first part, we describe and define the problem while, in the second part, we elaborate on the details of the algorithm. Finally, we conclude the section by showing how this model has been integrated in the DWARV2.0 compiler.

6.3.1 Motivational Example and Problem Definition

To describe the problem, we make use of a synthetic case study. We consider a simple function that transforms each element of an input array based on simple arithmetic operations with predefined weights (as in e.g., FIR filter). The function has no loop-carried dependencies as each array element is processed independently of the others. Figure 6.1(b) shows this graphically, where L_b marks the beginning of the loop (i.e., compute the new value for each of the array elements) and the number four in the superscript represents the total number of loop iterations (i.e., we use an input array size of four elements). The body of the loop is delimited by the rectangle box. The L_e marks the end of the four loop iterations. Furthermore, we see that, in this body, three operations are performed taking one cycle each for a total function execution time of 12 cycle time (C_T). Unrolling once and given there are no loop-carried dependencies, we can speedup the application by a factor of two if we would double the resources. However, the overall speedup will depend on the available hardware resources and is thus constrained by this. For instance, if we use

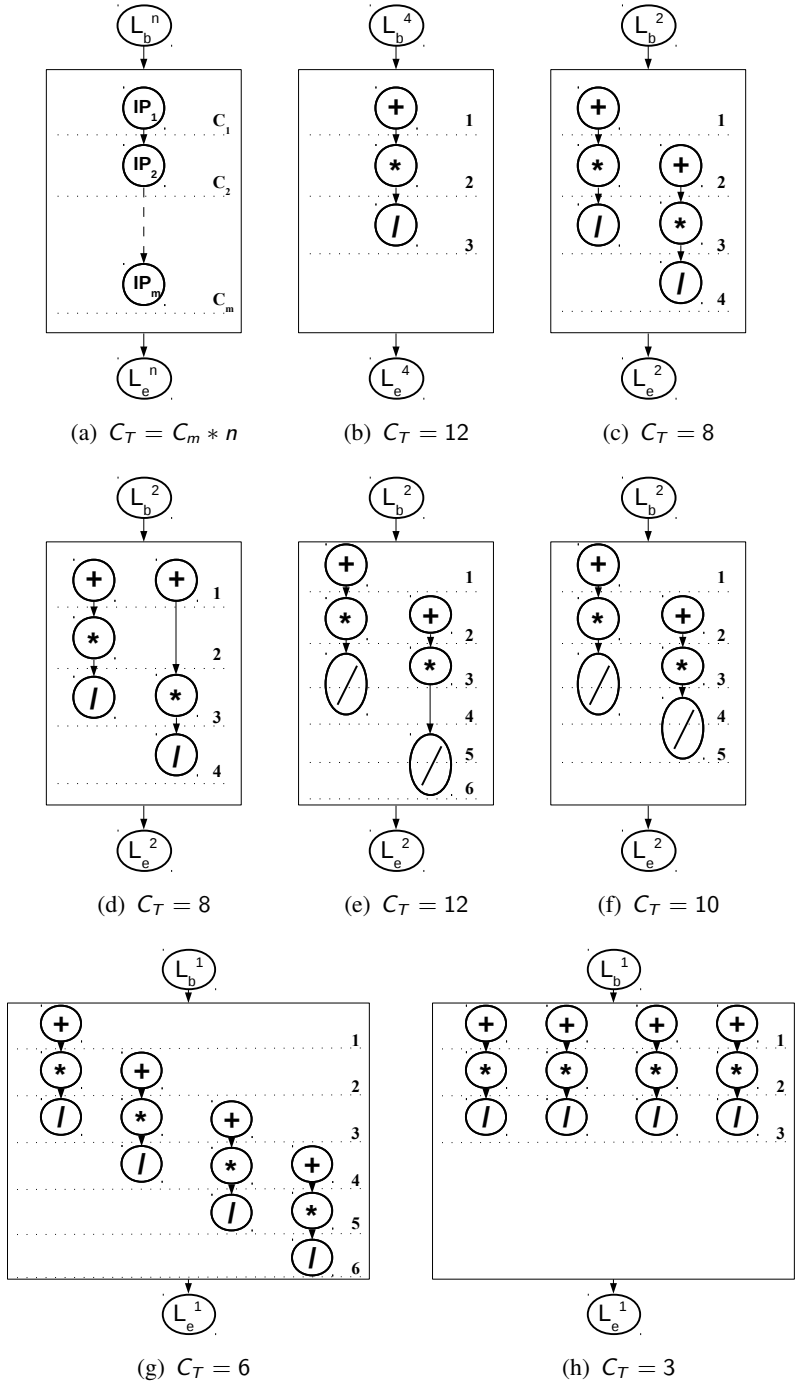


Figure 6.1: Motivational Examples: a) Formal Representation; b) No Unroll and 1+, 1*, 1/ units; c) 2 Unroll and 1+, 1*, 1/ units; d) 2 Unroll and 2+, 1*, 1/ units; e) 2 Unroll and 1+, 1*, 1/ units; f) 2 Unroll and 1+, 1*, 2/ units; g) 4 Unroll and 1+, 1*, 1/ units; h) 4 Unroll and 4+, 4*, 4/ units;

only one resource of each type as in the initial case, the speedup would be less than the maximum possible ($C_T = 8$ in Figure 6.1(c)).

Doubling only one resource type, as, for example, the addition unit (Figure 6.1(d)), and keeping the initial assumption that each of the three computations takes the same number of clock cycles (i.e., one clock cycle), does not decrease the execution time while increasing the area. This is a suboptimal allocation of resources for the achieved execution time. It is important to note that this scenario can falsely allow one to draw the conclusion that, for such loop body types, the number of resources that should be used for each type is the number corresponding to the resource with the minimum count in the loop. However, this is true only for the case where all resources compute the output in the same number of cycles. However, if this is not the case, which is a fair assumption for real world applications, having different counts of resources is possible without obtaining a suboptimal resource allocation. This is illustrated in Figures 6.1(e) and 6.1(f). In the first illustration, we have the scenario when one resource of each type is used, while, in the second, only the number of division units is doubled. This leads to a decrease in C_T from 12 to 10 because the number of cycles for one loop iteration is decreased by one due to the availability of a second division unit that can be used before the first division finished execution.

Finally, fully unrolling the loop and using only one resource of each type achieves a yet better execution time. This is illustrated in Figure 6.1(g) where C_T has been reduced to six cycles. Nevertheless, the best execution time (C_T = 3) is achieved when fully unrolling the loop and using the maximum possible units for each operation as shown in Figure 6.1(h). However, this can increase the area past the available area for the function. This is specially important in our scenario, where the reconfigurable area is divided among differently sized slots. For example, in the current implementation of the MOLEN machine organization, we have available a maximum of five slots. Given run-time and partial reconfigurability, the slots can be merged or split depending on the application's requirements. This would lead to a different area requirement for one kernel. Therefore, it is necessary to have hardware compilers that automatically generate hardware designs that map onto the available area given as an input constraint. This would avoid iterating over the design choices to find the optimal unroll factor and the maximum number of resources of each type, thus reducing the design space exploration time.

Summarizing, there is a trade-off between the number of unrolls one can do and the number of resources used for each unroll. The goal of our model is

to compute the best combination given performance as the main design objective. Given the general form of applications with loop body types as shown in Figure 6.1(a), we define the problem as follows: be the loop delimited by L_b^n and L_e^n iterating n times, performing operations that use m different hardware modules, IP_1 to IP_m, for which we know the sizes and latencies c_1 to c_m respectively, determine the *unroll factor* and the maximum number of modules for each of the m IP types such that the input area constraint is satisfied while the performance is maximized.

6.3.2 Optimization Algorithm

The algorithm consists of two parts, one for each parameter that needs to be determined. In the first step, we determine the unroll factor based on the available FPGA area as well as the area increase due to *wiring* when more parallelism is available after an unroll is performed. *Wiring* is the total amount of hardware logic (i.e., wires) that is used for routing register and memory resources to corresponding arithmetic unit(s) logic and vice-versa. To obtain the unroll factor (uf) we solve inequality 6.1 for the maximum value of uf:

$$A_i + uf * wi <= A_t \tag{6.1}$$

,where A_i, A_t and wi represent the initial area of the kernel, the total area available for the kernel on the target FPGA and the wiring increase per unroll, respectively. Furthermore, we have $uf \in N$ and have obtained both A_i and wi after a complete behavioral synthesis and estimation of the kernel's generated HDL. The initial kernel refers to the code 'just as is', i.e., the code without any unroll optimization applied and using the minimum number of resources for each of the required IPs necessary to implement its functionality. The compiler is executed once without activating any specific optimization regarding the number of IPs and unroll. The compiler is executed a second time with the loop unrolled once. Then, the wi is the difference between the estimated area of these two generated kernels. The estimation is based on [59].

In the second step, we determine the necessary component counts to fit the hardware design onto the FPGA area available. This step assumes the IP sizes are available and can be loaded from an external library. However, if these are not available, the netlist of the IP should be synthesized for the specific device, and the number of slices required for it should be saved in the external database. Furthermore, the available parallelism for each IP has been fixed by the previous step which unrolled the loop body. That means that no more

than some value n of IPs of type m (conditions 6.5 and 6.6 below) can execute in parallel (inequality 6.4). The second constraint according to the problem definition involves the area of the IPs itself, which leads to the constraint that the sum of all IP sizes multiplied by the number of their instantiations should not exceed the total area given as a parameter (inequality 6.3). Finally, the objective is to minimize the time it takes to compute each level in the graph by instantiating as many resources of that type possible (6.2). The solution of the algorithm is obtained by selecting the maximum between these minimums chosen for each level in the CDFG. Note that minimizing the number of cycles only for a level in the CDFG is ineffective if the other levels with different operations are not minimized, as well. The complete set of (in)equations is shown below:

$$min : MAX\{countIP_i/x_i + countIP_i\%x_i?1 : 0\} \tag{6.2}$$

$$\sum_{i=1}^{m} x_i * area\{IP_i\} <= A_t \tag{6.3}$$

$$x_i <= t_{IP_i} * uf \tag{6.4}$$

$$countIP_i = MAX\{t_{IP_i} * uf\} \tag{6.5}$$

$$x_i \in N, \forall i \in \{1, ..., m\} \tag{6.6}$$

,where *area* is the area of the component accounting for both the number of slices and on board Digital Signal Processor (DSP)s cores it requires. x_is are the variables which represent how many IPs of type i can be used inside the total area available (A_t). Furthermore, t_{IP_i} represents how many instances of type IP_i are used in the initial version of the code when no unroll has been performed.

6.3.3 Integration in DWARV2.0

The algorithm is shown in Figure 6.2 and includes two function calls corresponding to the above two steps described. In *determineUnrollFactor()*, we first load the necessary parameters to solve inequality 6.1. These are obtained by (pre)compiling the kernel and using any estimation tool (such as Quipu [60] from the Delft Workbench tool-chain) on the generated HDL to predict the initial area. Therefore, the result of the estimation is used to extract and store the parameters required for the algorithm. These are fed into the compiler in the second run for the same kernel. Figure 6.3 shows how the compiler flow

```
int determineUnrollFactor(int At) {
      wi = importWI(); //import estimated wiring increase (see Section III.C)
      return solveUF(wi,At)// solve inequality (1);
}

struct areagenparams optimizeForArea(int At) {
      I)  uf = determineUnrollFactor(At);
      ipsizes = loadIPsizes(); // import component sizes
      II) return solveIPcounts(uf,ipsizes,At); // return solution found by (2)
}
```

Figure 6.2: *optimizeForArea* Main Function of the Algorithm.

has been extended to compute these values and how the algorithm has been integrated in DWARV2.0. In the stripped upper box, we see the internals of the DWARV2.0 compiler as it was described in section 6.2. This is composed of a series of loosely integrated engines, executing different optimizations and transformations on the globally available IR. The small engine box denoted *Plug-ins* represents the implementation of our algorithm, which performs no action if there are no input parameters, i.e., for the first two runs to obtain the parameters for the initial area and the area of the kernel with the loop unrolled once. The difference between these estimated areas gives us the estimated wiring increase for unrolling.

After these preliminary compilations (maximum two), the algorithm can be applied. Using the computed wiring increase and the initial area obtained, the unroll factor can be computed by solving inequality 6.1. The solution is feed into the unroll engine which obtains a new intermediary version of the code with an increased parallelism. Note that the unroll engine is encompassed by the ”Plug-ins” engine which was built around it, i.e., it is composed actually of two smaller engines corresponding to the two steps. One it is run before and the other after the unroll engine. Finally, based on the determined parallelism, the inequalities 6.2 to 6.6 are initialized and solved in function *solveIPcounts()*.

6.4 Experimental Results

In this section, we describe the experimental environment, the test cases and provide a discussion of the obtained solutions for the different design options available.

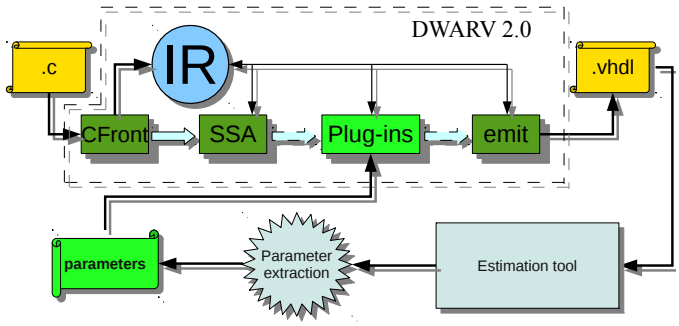


Figure 6.3: Algorithm Integration with DWARV2.0 Compiler.

6.4.1 Experimental Environment

The environment used for the experiments is composed of three main parts:

i) The C-to-VHDL DWARV2.0 compiler, extended with an optimization engine applying the two-step algorithm described in the previous section,

ii) the Xilinx ISE 12.2 synthesis tools, and

iii) the Xilinx Virtex5 ML510 development board.

The board contains a Virtex 5 xc5vfx130t FPGA consisting of 20480 slices and 2 PowerPC processors. From these, 9600 slices are allocated to the reconfigurable part of the MOLEN design as presented in section 6.2, and constitute the maximum area that DWARV2.0 generated designs target. More precisely, we use in the experiments 1920, 2880, 4800 and 9600 slices corresponding to 20%, 30%, 50% and respectively the full area of the reconfigurable part to test the capability of the algorithm to generate designs that during synthesis will fit within these predefined area constraints.

6.4.2 Test Cases

To validate the correctness and usefulness of the algorithm, we used three case studies based on real applications. These are a simple vector summation of 128 elements, a 10x2 with a 2x10 matrix multiplication and a 5-tap-FIR filter computing 27 values. Figures 6.4, 6.5 and 6.6 graphically show the characteristics of these kernels. The vector summation contains 64 additions in parallel, the

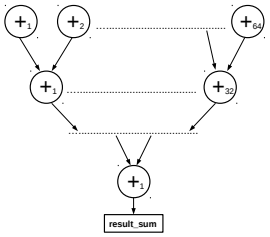


Figure 6.4: VectorSum test case.

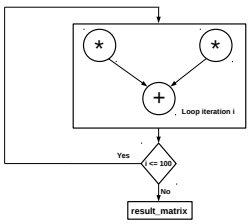


Figure 6.5: MatrixMult test case.

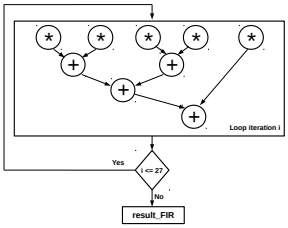


Figure 6.6: FIR test case.

matrix multiplication iterates 100 times to compute each element of the 10x10 resulting matrix by doing two parallel multiplications followed by an addition, whereas the FIR test case consists of 5 parallel multiplications and 2 parallel additions for each computed element. All the arithmetic operations are done on floating-points numbers, therefore, the IPs (i.e., resources) of interest in the experiments are floating-point adders and multipliers. However, the general approach described in the previous section can apply to any IP block, not just floating-point adders and multipliers.

The first step of the algorithm is to determine the unroll factor. To do this, we first compute the set of values available for the unroll parameter based on the loop bound. This set is composed of the loop bound's divisors and is necessary because we restrict the search space only to unroll factors that divide the loop bound. Furthermore, we consider only compile-time known loop bounds and leave as future research the case when these bounds are variable. For example, in the matrix multiplication test case that iterates 100 times, the set of unroll factors is composed of the divisors of 100, i.e., $\{1,2,4,5,10,20,25,50,100\}$, with one representing no unroll and 100 the fully unrolled version of the code. Next, the inequality for determining the unroll factor is derived. This is achieved by taking the increase in the area due to wiring when unrolling once, multiplying it with the unroll variable and adding the area size of the minimum kernel implementation (i.e., no unroll). The increase in the area due to unrolling is computed by subtracting the size of the estimated area for the 'no unroll' implementation from the area obtained for the '2 unroll' implementation. Both of these numbers are obtained using the methodology described in Section 6.3.3.

The reason why this model is viable is that the wiring increase per unroll factor is the biggest between the 'unrolling' once' and 'no unroll' kernel implementations. If the body of the loop is unrolled further, the wiring increase

decreases because more resources become available for reuse. More specifically, by packing the routing logic into logic already partially occupied by previous unrolls, the increase in the area for subsequent unrolls is becoming smaller. Therefore, solving inequality 6.1 using the maximum wiring implies that the unroll factor obtained will satisfy the area constraint for unrolls bigger than two. Our experiments showed that this is true. Nevertheless, in future research, we will investigate this matter in more depth and propose a formal model for this increase.

To verify that the obtained *uf* is valid, we use the divisor set obtained previously and check that the value is included in the set. However, note that the unroll factor can fall in between the values present in the set. In this case, we select the next smaller value in the set because the design would be guaranteed to fit into the constrained area. Furthermore, the slack area that becomes available by choosing a smaller unroll factor than the computed one allows the second step of the algorithm to duplicate more IP cores and thus achieve a higher degree of parallelism for the computation.

The second step is to decide how many IPs can be instantiated based on the available code parallelism after unrolling the number of times identified in the previous step (Table 6.1 third column). This is done by solving the system composed of inequalities 6.2 to 6.6 described previously in section 6.3.2. The parameters are set according to the operations with their predefined area loaded from the target-dependent external library. Furthermore, we set the main constraint for the objective function, i.e., maximum area available in terms of slices. Table 6.1 fifth column shows the various solutions obtained for the corresponding area constraints given in the second column.

6.4.3 Discussion

To gain insight in the results and understand why the design process benefits from such an optimization, we look at the generated hardware under various area constraints, different unroll factors and number of IP cores. That is, we analyze the Pareto points obtained for different configurations. Due to resemblance reasons, we discuss only the matrix multiplication example and give for the other two only the final results. This function consists of a loop iterating 100 times to compute each of the elements of the resulting 10x10 matrix. Each iteration contains one addition and two parallel multiplications. Fully unrolling this code would lead to a maximum of 100 additions and 200 multiplications to execute in parallel. Clearly this can easily lead to a HDL implementation that would not fit into the available area. Therefore, we perform an analysis

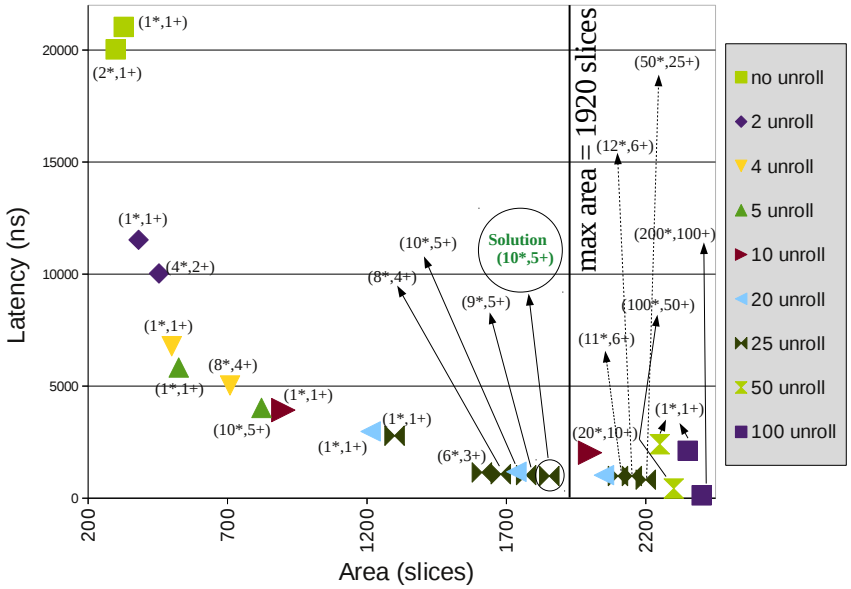


Figure 6.7: Matrix multiplication: 20% area design constraint.

with the DWARV2.0 compiler extended with the new optimization engine and investigate its capability to generate CCUs that fit into different sized slots available in the current MOLEN design.

We begin by constraining the available area for the CCU to the smallest size slot that has 1920 slices accounting for 20% of the available reconfigurable FPGA area. Figure 6.7 shows different points corresponding to different configurations of the matrix function. To explain the points in the graph, we define the tuple $\langle x, y*, z+\rangle$ notation that represents the solution point with the loop body unrolled x times, instantiating y multipliers and z adders. In the graphs, the first element of the tuple is represented by a different shape and color. For example, the most left side point in the figure is $\langle 1, 2*, 1+\rangle$ denoting the implementation of the initial code (i.e., no unroll) and using two multiplication and one addition units. This implementation executes in 20030 ns and occupies 299 slices. Note that the implementation using the minimum number of resources, i.e., $\langle 1, 1*, 1+\rangle$ is slower (21030 ns) and occupies 29 slices more. The execution time is bigger because using only one multiplication core we do not take full advantage of the available parallelism, whereas the increase in the area is due to the more slices required to route multiple inputs to one unit compared to the increase obtained by duplication. This confirms also the wiring

increase model used, which assumes that the wiring increase is the highest when there is a small area available for reuse as is for this minimum kernel implementation.

The vertical line on the right of Figure 6.7, denoted by *max area*, represents the threshold after which the generated configurations will not fit into the requested area constraint. The fastest solution obtained by fully unrolling the loop and using the maximum number of cores for both operations, i.e., $\langle 100, 200*, 100+\rangle$, does not meet the design constraint of 1920 slices being situated on the right of the threshold line. The rest of the points show representative design configurations. That is, for each possible unroll factor we show the minimum (i.e., using one core of each unit) and the maximum implementations. However, for each unroll we can have more configurations than the minimum and maximum, for example, the $\langle 20, 10*, 5+\rangle$ and $\langle 25, 8*, 4+\rangle$ solutions, with the second being faster and smaller.

Analyzing all these solutions manually is very time consuming; therefore, the benefit of having the compiler perform this automatically has tremendous effects in terms of saved design time. Considering the 20% area constrained matrix example and assuming a binary search through the design space, we would need to evaluate at least seven designs manually to obtain the $\langle 25, 10*, 5+\rangle$ optimal solution. Assuming 30 minutes to obtain the bitstream for each implementation, we would need at least 210 minutes to arrive at the optimal solution. The automation of this search and running the compiler along with the estimator takes on average five minutes. This leads to an average speedup in design time of 42x.

Figure 6.8 shows the return on investment (ROI) for the 20% area design constraint. ROI is computed as the factor-wise increase in performance over the factor-wise increase in area. We show the ROI at discrete points on the x-axis obtained by the division of the corresponding (new) solution area number in terms of slices to the initial solution. The origin of the graph represents the initial solution, which is occupying 328 area slices and executes in 21030 ns. Such a graph is important when we are confronted with many optimization choices and choosing the best solution given a limited budget is not trivial. In this case, we can see immediately what is the effect of unrolling vs. increasing the number of resources. For example, if resources are at a premium, a cost efficient solution is usually one chosen from the ones with a smaller unroll factor and less functional units. Appendix B shows the ROI for the other three cases of the matrix multiplication example described in this section.

Figure 6.9 illustrates the design points for the 30% area constraint. For this

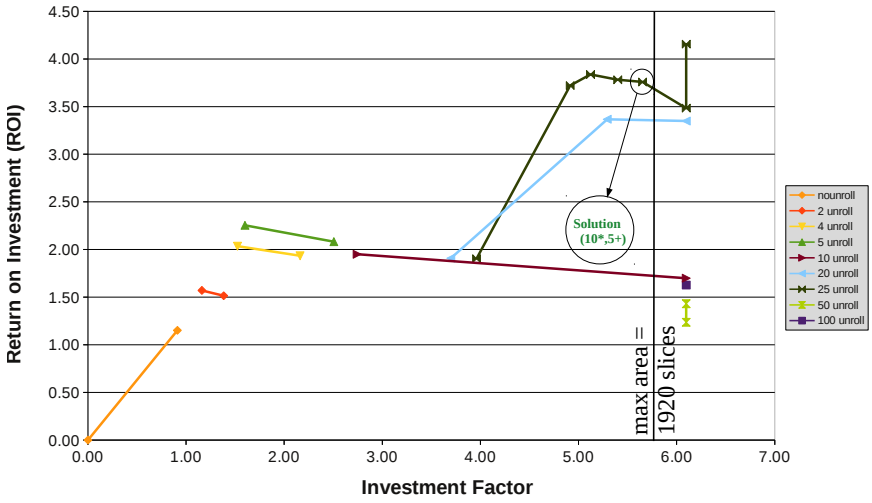


Figure 6.8: Matrix multiplication ROI for 20% area design constraint.

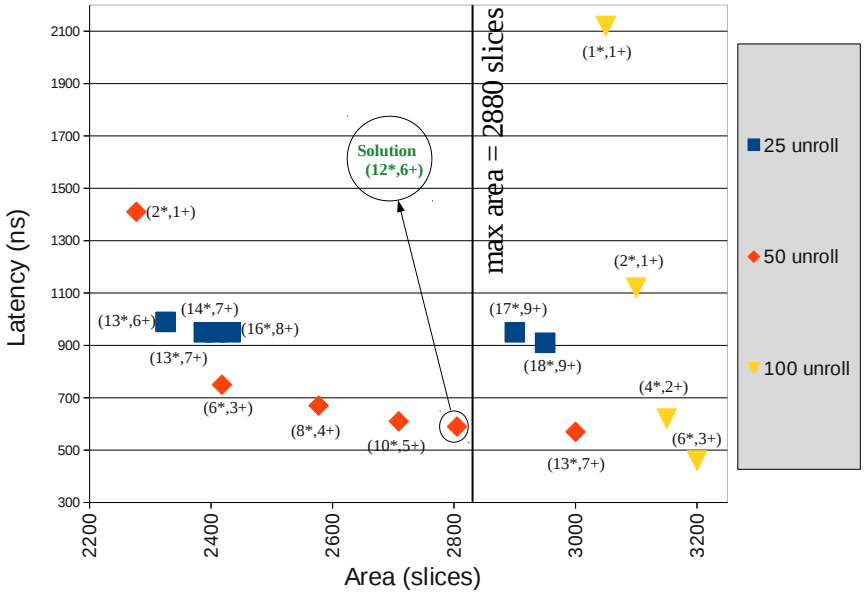


Figure 6.9: Matrix multiplication: 30% area design constraint.

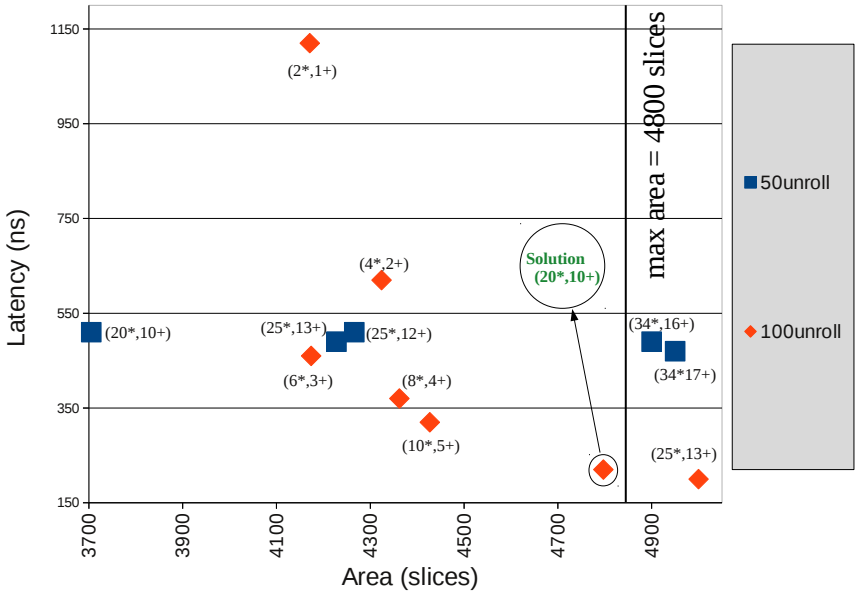


Figure 6.10: Matrix multiplication: 50% area design constraint.

experiment, the figure shows only configurations after the loop body was unrolled by at least a factor of 25 because, from the previous experiment with the smaller area constraint, we know that the design choices up to 25 unroll factor cannot be the optimal solution. Therefore, we highlight several possible implementations for the 25, 50 and full unroll factors. Analyzing the performance for these implementation, we observe that unrolling has a bigger impact on performance than using more parallelism with a smaller unroll factor. For example, the $\langle 50, 6*, 3+\rangle$ kernel implementation is faster and occupies less area than the one represented by the $\langle 25, 18*, 9+\rangle$ point. However, if we unroll too much, we might not obtain any valid solutions as is the case with all the 100 unroll points. Therefore, finding the optimum unroll factor is a key step in the area constraint driven HDL generation. The optimal solution obtained for the 30% area constraint is the $\langle 50, 12*, 6+\rangle$ implementation point.

Finally, Figures 6.10 and 6.11 illustrate design points for the 50% and 100% area constraint, respectively. The solutions obtained are close to the minimum latency achieved when all 200 multiplications and 100 additions are used in the fully unrolled version. However, because of the high number of arithmetic units required by this implementation, and given the maximum number of slices available in the current setup, this best solution cannot be achieved.

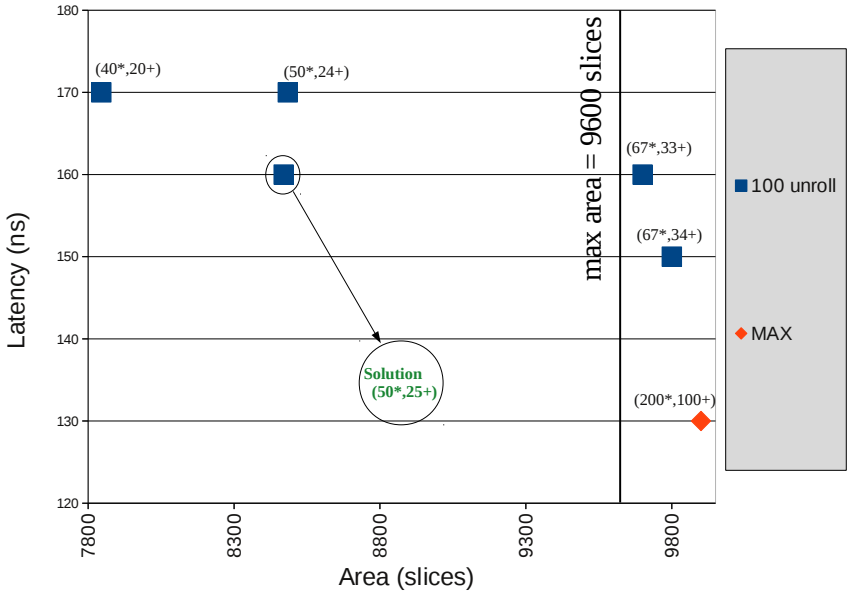


Figure 6.11: Matrix multiplication: 100% area design constraint.

Therefore, a compiler that cannot handle area constraints and always fully unroll loops to achieve the highest code parallelism using the maximum possible resources will always fail to give a valid solution.

The same experiments were subsequently performed for the VectorSum case study, as well. Because this function does not contain any loops, only the second step of the algorithm was applied. This case study shows thus that the algorithm can be applied also for applications that do not contain loops and thus it can be used in a general way. The solutions are summarized in Table 6.1. For each constraint, we verified that the solutions are the optimal ones by checking that allocating a smaller number of resources gives a bigger latency for the kernel execution time. At the same time, synthesizing the next solution confirmed that this fails to meet the area constraint.

Finally, we tested the FIR kernel that has as input an array of 32 values and 5 parameters to compute 28 output values. Each of the output values is computed thus in a loop iterating 28 times. Therefore, the unroll factor set is composed of $\{1,2,4,7,14,28\}$. Table 6.1 summarizes the solutions obtained for all case studies. The second column shows the area restriction for which the results are given. The third column shows the unroll factor obtained in the first step of the algorithm while the next column gives the maximum operations of each type

Table 6.1: Experimental results of the test cases and their corresponding solutions for different area design constraints.

Function	Case	Unroll factor	Max. IPs	Solution	Area (slices)	Occupancy (%)	Latency (ns)	Freq. (MHz)	Power (mW)
VectorSum	20%	N/A	64+	10+	1888	98	925	213	106
	30%		64+	22+	2817	97	885	240	126
	50%		64+	64+	4526	94	865	338	294
	100%		64+	64+	5581	78	865	338	294
Matrix	20%	25	50*, 25+	10*, 5+	1854	96	990	300	95
	30%	50	100*, 50+	12*, 6+	2805	97	590	283	140
	50%	100	200*,100+	20*,10+	4797	99	220	280	324
	100%	100	200*,100+	50*,25+	8470	88	160	280	403
FIR	20%	14	70*,28+	8*, 3+	1897	98	1020	250	106
	30%	14	20*,28+	14*, 7+	2738	95	900	325	155
	50%	28	140*,56+	28*,12+	4738	98	500	339	245
	100%	28	140*,56+	47*,19+	7174	74	480	327	293

that could be executed in parallel for the previous unroll factor. The solution of the second step of the algorithm for how many instances to use for each operation is shown in the fifth column. The next two columns highlight the number of slices the obtained solution takes as well as how much of the available area for the experiment is used. Kernel latency and frequency information are showed in columns eight and nine. This maximum frequency reported is obtained after behavioral synthesis. We do not report the post place and route frequency because we set the target clock cycle to 6.66 ns. This restriction is imposed by the 150 MHz frequency required by the MOLEN static part.

It is important to look at the power consumption and how this is influenced by the size of the generated implementation. The dynamic power consumed by the unoptimized, initial kernel implementations (i.e., using only one IP of each type and not unrolling) is 81 mW, 8 mW, and 38mW for the vector summation, the matrix, and the FIR function, respectively. The power consumption for the solution points for the experimented area constraints is given in the last column of the results table. The power data offers another motivation why a hardware compiler should be able to generate different hardware implementation based on different input constraints. We mostly discussed in this chapter the case where the user would want to map different kernels on the reconfigurable area and optimize the performance. However, using the area as a design constraint, the user could just as well optimize the power consumption. This is especially true for cases when the area increases rapidly, but only with a small increase in performance. For example, consider the vector summation solutions for the 20% and 100% experiments. The speedup of the second compared to the first is small, i.e., 1.1x, however, the increase in power is 2.8x. If the performance of the system is not critical, the designer could choose to restrict the area to reduce the power consumption. Therefore, the presented algorithm could be used to minimize the power consumption as well, not only the performance. Nevertheless, a formal power model is needed in order to include the power constraint in the decision model. Such an extension is part of the future work of this approach.

6.5 Conclusion and Future Research

In this chapter, we presented an optimization algorithm to compute the optimal unroll factor and the optimum allocation of resources during the HLL-to-HDL generation process when this is subject to area constraints. The described algorithm was added to an existing C-to-VHDL hardware compiler, and three

case studies were used to validate the optimization. The experiments showed that the generated solutions are mapped into the available area at an occupancy rate between 74% and 99%. Furthermore, these solutions provide the best execution time when compared to the other solutions that satisfy the same area constraint. Furthermore, a reduction in design time of 42x on average can be achieved when these parameters are chosen automatically by the compiler.

Future research includes analyzing other applications and investigating how different graph characteristics influence the optimization presented. Another model extension involves dealing with variable loop bounds. In addition, more accurate prediction models for the wiring increase as well as for the power consumption are needed.

Note.

The content of this chapter is based on the following paper:

R. Nane, V.M. Sima, K.L.M. Bertels, **Area Constraint Propagation in High Level Synthesis**, 11th IEEE International Conference on Field-Programmable Technology (FPT 2012), Seoul, South Korea, December 2012.

7

A Lightweight Speculative and Predicative Scheme for HW Execution

IF-CONVERSION is a known software technique to speedup applications containing conditional expressions and targeting processors with predication support. However, the success of this scheme is highly dependent on the structure of the if-statements, i.e., if they are balanced or unbalanced, as well as on the path taken at run-time. Therefore, the predication scheme does not always provide a better execution time than the conventional jump scheme. In this chapter, we present an algorithm that leverages the benefits of both jump and predication schemes adapted for hardware execution. The results show that performance degradation is not possible anymore for the unbalanced if-statements as well as a speedup between 4% and 21% for all test cases.

7.1 Introduction

As the increase in frequency of the general-purpose processors is becoming smaller and harder to obtain, new ways of providing performance are investigated. One of the promising possibilities to improve the system performance is to generate dedicated hardware for the computational-intensive parts of the applications. As writing hardware involves a huge effort and needs special expertise, compilers that translate directly from high-level languages to hardware languages have to be available before this method is widely adopted. As C and VHDL are the most popular used languages in their fields, of embedded and hardware system development respectively, we will focus on compilers for C-to-VHDL. The algorithm presented here can be applied in theory to any such compiler. A C-to-VHDL compiler can share a significant part with a compiler

targeting a general-purpose architecture, still, there are areas for which the techniques must be adapted to take advantage of all the possibilities offered.

In this context, this chapter presents an improved predication algorithm, which takes into account the characteristics of a C-to-VHDL compiler and the features available on the target platform. Instruction predication is an already known compiler optimization technique; however, current C-to-VHDL compilers do not fully take advantage of the possibilities offered by this optimization. More specifically, we propose a method to increase performance in the case of unbalanced if-then-else branches. These types of branches are problematic because, when the jump instructions are removed for the predicated execution if the shorter branch is taken, slowdowns occur because (useless) instructions from the longer branch still need to be executed. Based on both synthetic and real world applications we show that our algorithm does not substantially increase the resource usage while the execution time is reduced in all cases for which it is applied.

The chapter is organized as follows. We begin by presenting a description of the predication technique and previous research, emphasizing on the missed optimization possibilities. In Section 7.3 we present our algorithm and describe its implementation. The algorithm is based on a lightweight form of speculation because it does not generate logic to roll back speculated values. It employs a lightweight form of predication because only some branch instructions are predicated, as well as keeping jump instructions. Section 7.4 discusses the results and Section 7.5 concludes the chapter.

7.2 Related Work and Background

Given the code in Figure 7.1 (a), the straightforward way of generating assembly (or low-level code) is presented in Figure 7.1 (b). We note that, for any of the two branches, there is at least one jump that needs to be taken. If the block execution frequency is known, an alternative approach exists in which the two jumps are executed only on the least taken branch.

Branches are a major source of slowdowns when used in pipelined processors as the pipeline needs to be flushed before continuing if the branch is mispredicted. Furthermore, branches are also scheduling barriers, create I-cache refills and limit compiler scalar optimizations. In order to avoid these negative effects, the concept of predication was introduced, which does not alter the flow but executes (or not) an instruction based on the value of a predicate. An example is given in Figure 7.1 (c). In this scheme, no branches are intro-

duced, but, for a single issue processor, (sometimes) useless instructions are executed. In case of a multiple issue processor, such instructions can be "hidden" because the two code paths can be executed in parallel. We emphasize that the advantage of the predication for processors comes from the fact that there are no branches in the code.

```
                        cond = cmp x,0
                        branchf cond, else
                        add r,a,b
if (x)                  branch end
   r = a + b;       else:                  cond = cmp x,0
else                    sub r,c,d           [cond] add r,a,b
   r = c - d;       end:                   [!cond] sub r,c,d

     (a)                 (b)                    (c)
```

Figure 7.1: (a) C-Code; (b) Jump- ; (c) Predicated-Scheme.

The predication scheme assumes that the penalty of the jump is huge, and thus branching has to be avoided. This is no longer true in the case of VHDL code. For the VHDL code, there are no "instructions" but states in a datapath, controlled by a Finite State Machine (FSM). A straightforward implementation in VHDL of the jump scheme is presented in Figure 7.2. We will present in the later sections the implications of the fact that the jumps are not introducing a huge delay. For this case, applying predication decreases the number of states from 4 to 2. We will show in the later sections how our algorithm can reduce the number of states even for unbalanced branches, a case not treated in the previous work.

A seminal paper on predication is [58], where a generic algorithm is presented that works on *hyperblocks* which extends the concept of basic blocks to a set of basic blocks that execute or not based on a set of conditions. It proposes several heuristics to select the sets of the basic blocks, as well as several optimizations on the resulted hyperblocks and discusses if generic optimizations can be adapted to the *hyperblock* concept. Compared to our work, their heuristic does not consider the possibility of splitting a basic block and does not analyze the implications for a reconfigurable architecture, e.g., branching in hardware has no incurred penalty.

The work in [57] proposes a dynamic programming technique to select the fastest implementation for if-then-else statements. As with the previous approach, any change in the control flow is considered to add a significant performance penalty. In [43], the authors extend the predication work in a generic way to support different processor architectures. In this work, some instructions are moved from the predicated basic blocks to the delay slots, but as delay

```
datapath
   state_1:
       cond = cmp x,0
   state_2:
   state_3:
       r=a+b;
   state_4:
       r=a-b;
   state_5:
       .... -- code after if-statement
FSM
   state_1:
       next_state = state_2
   state_2:
       if(cond)
         next_state = state_3
       else
         next_state = state_4
   state_3:
       next_state = state_5
   state_4:
       next_state = state_5
   state_5:
       ....
```

Figure 7.2: Jump Scheme

slots are very limited in nature there is no extensive analysis performed about this decision.

Regarding the C-to-VHDL compilers, we mention Altium's C to Hardware (CHC) [8] and LegUp [18]. They translate functions that belong to the application's computational-intensive parts in a hardware/software co-design environment. Neither of these compilers considers specifically predication coupled with speculation during the generation of VHDL code.

7.3 Speculative and Predicative Algorithm

In this section, we describe the optimization algorithm based on two simple but representative examples which illustrate the benefit of including Speculative and Predicative Algorithm (SaPA) as a transformation in High-Level Synthesis (HLS) tools.

```
void balanced_case(int *a, int *b, int *c, int *d, int *result) {
    if (*a > *b)
            *result = *c + *d;
    else
            *result = *c - *d;
}
```

Figure 7.3: Balanced if branches.

```
void unbalanced_case(int *a, int *b, int *c, int *d, int *result) {
    int tmp;
    if (*a > *b) {
            tmp = *c + *d;
            *result = tmp /5; }
    else
            *result = *c - *d;
}
```

Figure 7.4: Unbalanced if branches

7.3.1 Motivational Examples

To understand the problems with the predication scheme (PRED) compared to the jump scheme (JMP), we use two functions that contain each one if-statement. The first, shown in Figure 7.3, considers the case when the then-else branches are balanced, i.e., they finish executing the instructions on their paths in the same amount of cycles, whereas the second case deals with the unbalanced scenario (Figure 7.4). In these examples, we assume the target platform is the Molen machine organization [73] implemented on a Xilinx Virtex-5 board. This setup assumes that three cycles are used to access memory operands, simple arithmetic (e.g., addition) and memory write operations take one cycle, whereas the division operation accounts for eight cycles.

The FSM states corresponding to the two examples are listed in Figure 7.5(a) and 7.5(b). For each example, the first column represents the traditional jump scheme ((1) and (4)), the middle columns ((2) and (5)) represent the predicated one and columns (3) and (6) shows the SaPA version. This column will be explained in more detail in the next section as it is presenting the solution to the problem described here. Because each state executes in one cycle, the first five states are needed to load the *a* and *b* parameters from memory. In the first two states, the address of the parameters is written on the memory address bus. State three is an empty state and, therefore, is not shown in the figures. Finally, in states four and five the values of the parameters are read from the

```
S1: ld *a
S2: ld *b
S4: read a;
S5: read b;
S6:TB = cmp_gt (a,b)
S7: if (TB) jmp S16;
S8: ld *c;
S9: ld *d;
S11: read c;
S12: read d;
S13: result = c-d;
S14: write result;
S15: jmp S23;
S16:  ld *c;
S17: ld *d;
S19: read c;
S20: read d;
S21: result = c+d;
S22: write result;
S23: return;
```

(1) JMP_B

```
S1: ld *a
S2: ld *b
S4: read a;
S5: read b;
S6: TB = cmp_gt (a,b)
S7: ld *c;
S8: ld *d;
S10: read c;
S11: read d;
S12: if (TB)
        result = c+d;
     else
        result = c-d;
S13: write result;
S14: return;
```

(2) PRED_B

```
S1: ld *a
S2: ld *b
S3: ld *c
S4: read a;
    ld *d;
S5: read b;
S6: read c;
   TB = cmp_gt (a,b)
S7: read d;
S8: if (TB)
      result =  c+d;
    else
      result = c-d;
S9: write result;
S10: return;
```

(3) SaPA_B

(a) Balanced Example

```
S1: ld *a
S2: ld *b
S4: read a;
S5: read b;
S6: TB = cmp_gt (a,b)
S7: if (TB) jmp S16;
S8: ld *c;
S9: ld *d;
S11: read c;
S12: read d;
S13: result = c-d;
S14: write result;
S15: jmp s32;
S16:  ld *c;
S17: ld *d;
S19: read c;
S20: read d;
S21: tmp = c+d;
S22: INIT → tmp/5;
S30 result← tmp/5;
S31: write result;
S32: return;
```

(4) JMP_U

```
S1: ld *a
S2: ld *b
S4: read a;
S5: read b;
S6:TB = cmp_gt (a,b)
S7: ld *c;
S8: ld *d;
S10: read c;
S11: read d;
S12: tmp = c+d;
     If (!TB) result = c-d;
S13: INT→  tmp/5;
S21: if (TB)
       result ← tmp/5;
S22: write result;
S23: return;
```

(5) PRED_U

```
S1: ld *a
S2: ld *b
S3: ld *c
S4: read a;
      ld *d;
S5: read b;
S6: read c;
   TB = cmp_gt (a,b)
S7: read d;
S8: if (TB) tmp =  c+d;
   else { result = c-d;
          jmp S18; }
S9: INT→ tmp/5;
S17: if (TB)
       result ← tmp/5;
S18: write result;
S19: return;
```

(6) SaPA_U

(b) Unbalanced Example

Figure 7.5: Synthetic Case Studies.

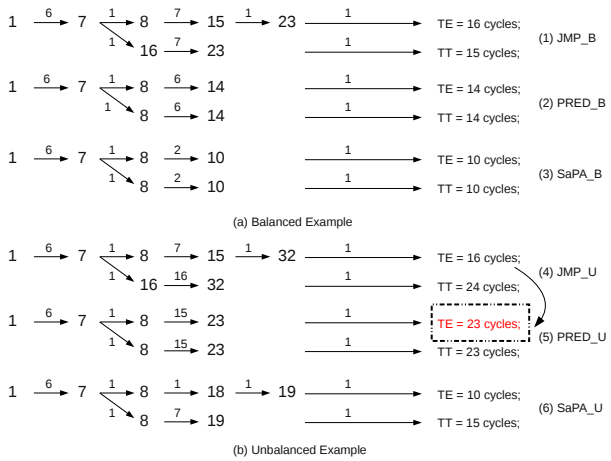


Figure 7.6: Execution Sequence of FSM States.

data bus. These operations are common for all possible implementations (i.e., for all combinations of balanced/unbalanced case study and JMP/PRED/SaPA schemes) shown by the (1) to (6) numbered columns in the two figures. Subsequently, the then-branch (TB) predicate is evaluated for the JMP cases (column (1) and (4)). Based on this value, a jump can be made to the then-branch states (states 16 to 22), or, in case the condition is false, execution falls through to the else-path (states 8 to 15). The number of states required for the unbalanced case, i.e., (4) JMP_U, is larger due to the additional division operation present in the then-branch. That is in state 22 we initialize the division core with the required computation, whereas, in state 30, we read the output.

Applying the predication scheme to the balanced example results in a reduction in the number of states. This is achieved by merging both then- and else-branches and by selecting the result of the good computation based on the predicate value. This optimization is ideal for HLS tools because decreasing the number of states reduces the total area required to implement the function. For the examples used in this section, a reduction of nine states was possible, i.e., when comparing (1) and (4) with (2) and (5) respectively. However, because branches can be unbalanced, merging them can have a negative impact on performance when the shorter one is taken. For example in column (5) PRED_U, when the else-path is taken, states 13 to 21 are superfluous and introduce a slowdown for the overall function execution.

Figure 7.6 shows all possible paths for both examples as well as their execution times in number of cycles, e.g., from state 1 to state 23. TE represents the execution **T**ime for the **E**lse path while TT is the **T**ime when the **T**hen path is

taken. The upper part of the figure corresponds to the balanced if-function and the lower for the unbalanced case. Furthermore, there is a one-to-one correspondence between the columns in Figure 7.5 and the scenarios in Figure 7.6. The numbers on the edges represent the number of cycles needed to reach the next state shown. The last arrow in the paths represents the cycle required to execute the return statement in the last state of the FSM. First, considering the balanced flows, we observe that the predication scheme improves performance compared to the jump scheme (i.e., (2) is better than (1)). This is because jump instructions are removed.

However, care has to be taken to avoid performance degradation when shorter paths are taken. This is shown in Figure 7.6 (5) compared to (4), where the execution time increased from 16 to 23 cycles. Therefore, for hardware execution the predication scheme has to be adjusted to cope with unbalanced branches. This is described next.

7.3.2 Algorithm Description and Implementation

To alleviate the short branch problem from the PRED scheme, we need to introduce a jump statement when the shorter branch is finished. Fortunately, for hardware execution this is possible without any penalty in cycles as opposed to conventional processors. This extension to the predicated scheme is shown in state S10 of Figure 7.5 (6). Including jump instructions in the FSM whenever a shorter path has finished guarantees that no extra cycles are wasted on instructions that are superfluous for the path taken. This is possible because, in hardware execution, there is no penalty when performing jumps. This motivates that this transformation can be applied for hardware generation because a hardware kernel is always seen as running on an n-issue slot processor with a jump penalty equal to 0. Furthermore, the flows in (3) and (6) of Figure 7.6 show that speculation improves performance even more by starting the branch operations before the predicate is evaluated. It is important to note that speculation in the case of hardware execution comes without any penalty as we do not have to roll back if the predicate value did not select the proper branch for execution. In hardware we can use more resources to accommodate speculative operations, i.e., sacrifice area, in favor of improving performance.

The compiler modifications required to implement the algorithm are shown in Figure 7.7 in the lower dashed rectangle. In the upper part of the figure, the global flow of the DWARV 2.0 compiler is shown. Here, standard and custom engines performing various transformations and optimizations are called sequentially to perform the code translation from C to VHDL. In this existing

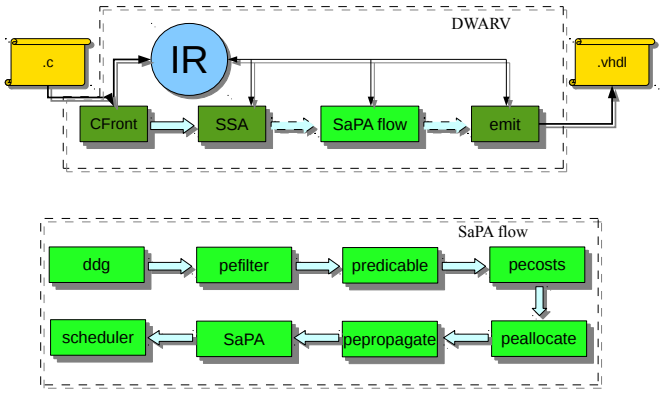
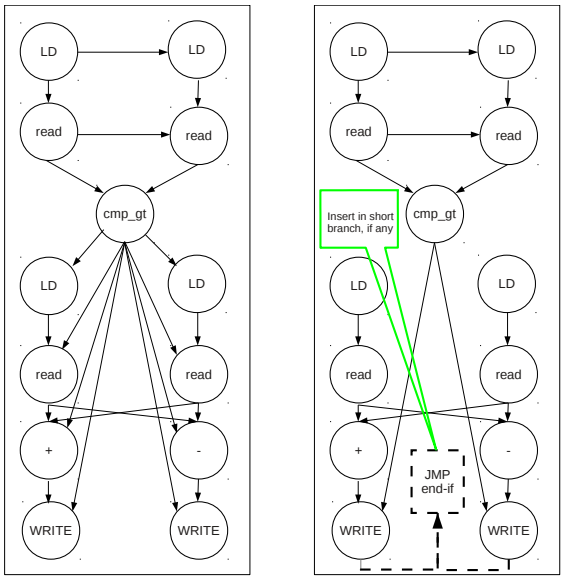


Figure 7.7: Engine Flow to Implement SaPA.

flow, the *SaPA flow* wrapper engine was added. This wrapper engine is composed of seven standard CoSy engines and one custom engine that implements the SaPA algorithm.

The first engine required is the data dependency graph (*ddg*) engine, which places dependencies between the predicate evaluation node and all subsequent nodes found in both branches of the if-statement. Next, the *pefilter* engine is called to construct the if-then-else tree structures. That is, basic blocks containing *goto* information coming from an if-statement are included in the structure, however, basic blocks with goto information coming from a loop are not. The *predicable* engine annotates the Intermediate Representation (IR) with information about which basic blocks can be predicated. The compiler writer can also express in this engine if he does not want the if-construct to be predicated. *pecosts* computes the cost for each branch of the if-statement based on the number and type of statements found in these and decides what scheme should be used to implement this if-statement. For hardware generation, this engine was reduced to simply returning SaPA. *peallocate* allocates registers in which if conditions are stored, whereas *pepropagate* propagates those registers to instructions found in both if-branches.

The SaPA engine implements the lightweight predication by introducing a jump instruction in the case of unbalanced branches. That is, when one of the branches has reached the end. Whenever this point is reached, a jump to the end of the other branch is inserted. The *SaPA* engine performs this step. Furthermore, the control flow edges from the predicate register to the expressions found in both branches are also removed here. That is, for simple expressions with local scope, dependency edges coming from the if-predicate are not



(a) Predicated Graph (b) SaPA Graph

Figure 7.8: Data Dependency Graphs.

needed. These expressions can be evaluated as soon as their input data is available. We name this lightweight speculation because, by removing the control flow dependencies from the if-predicate, we enable speculation; however, we do not introduce any code to perform the roll back in case the wrong branch was taken as this is not necessary in our hardware design. The dependencies to the memory writes, however, remain untouched to ensure correct execution. Figure 7.8(b) exemplifies the removal of unnecessary dependency edges for the balanced case study. It shows as well in the dashed box the (optional) insertion of the jump instruction that in the case illustrated, was not necessary.

Finally, the scheduler is executed which can schedule the local operations before the if-predicate is evaluated, i.e., speculating. Furthermore, when the FSM is constructed, whenever the branches become unbalanced, a conditional jump instruction is scheduled to enforce the SaPA behaviour. If the predicate of the jump instruction is true, the FSM jumps to the end of the if-block, therefore, avoiding extra cycles to be wasted when the shorter branch is taken. This ensures no performance degradation is possible with this scheme. If the predicate is false, the default execution to move to the next state is followed.

7.4 Experimental Results

The environment used for the experiments is composed of three main parts: i) The C-to-VHDL DWARV 2.0 compiler (see Chapter 4), extended with the flow presented in Section 7.3, ii) the Xilinx ISE 12.2 synthesis tools, and iii) the Xilinx Virtex5 ML510 development board. This board contains a Virtex 5 xc5vfx130t FPGA consisting of 20,480 slices and 2 PowerPC processors. To test the performance of the algorithm presented, we used seven functions. Two are the simple synthetic cases introduced in the previous section while the other five were extracted from a library of real world applications. These applications contain both balanced and unbalanced if-branches.

Figure 7.9 show the speedups of the PRED and SaPA schemes compared to the JMP scheme. The *balanced* function shows how much speedup is gained by combining the predication scheme with speculation. Similar to this, the speedup of the *unbalanced* function, tested with inputs that select the longer branch (LBT), shows a performance improvement compared to the JMP scheme due to speculation. However, when the shorter branch is taken (SBT), the PRED scheme suffers from performance degradation. Applying the SaPA scheme in this case allows the FSM to jump when the shorter branch finishes and, therefore, obtaining the 1.14x speedup.

The execution times for both schemes for the *gcd* function are the same because both paths found in this arithmetic function have length one, i.e., they perform only one subtraction each. Therefore, the only benefit is derived from removing the jump-instruction following the operands comparison. It is important to note that applying speculation is useful in all cases where both branches and the if-predicate computation take more than one cycle. Otherwise, the PRED scheme is enough to obtain the maximum speedup. Nevertheless, the speedup that can obtained by simply predicating the if-statement and thus saving one jump instruction per iteration can be considerable, e.g., 20% when the gcd input numbers are 12365400 and 906. *mergesort* provided a test case with a balanced if-structure where each of the paths contains more than one instruction. Therefore, the benefit of applying SaPA was greater than using the PRED scheme. This example confirms that whenever the paths are balanced, the application can not be slowed-down.

Finally, the last three cases show results obtained for unbalanced cases with inputs that trigger the shorter branches in these examples. As a result, for all these functions the PRED scheme generates hardware that performs worse than the simple JMP strategy. Applying the SaPA algorithm and introducing

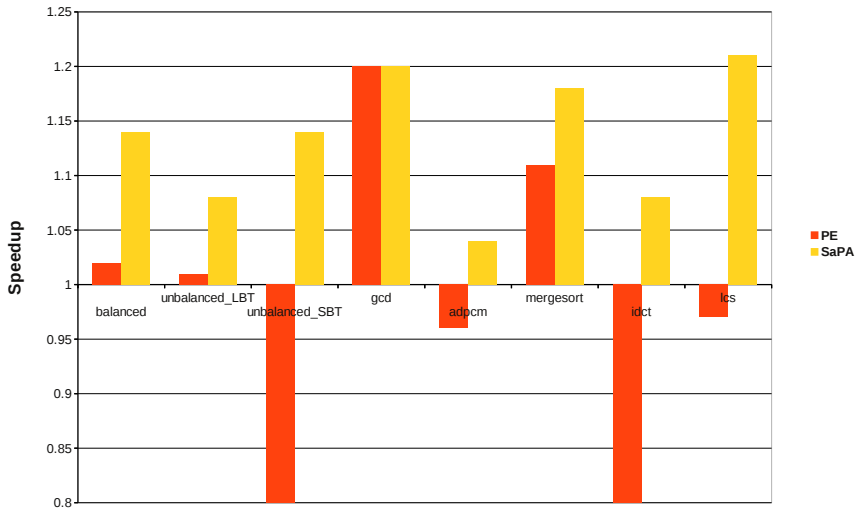


Figure 7.9: Predicated Execution (PE) and SaPA speedups vs. JMP Scheme.

a jump instruction after the short branch will allow the FSM to break from the "predication mode" execution of the if-statement and continue further with executing useful instructions.

To verify that the presented algorithm does not introduce degradation of other design parameters, i.e., area and frequency, we synthesized all test cases using the environment described in the beginning of this section. Table 7.1 summarizes the outcomes for the SaPA, PRED as well as the base JMP scheme. Column three shows the number of FSM states needed to implement the corresponding scheme from column two. Column four shows how much the complete generated design, i.e., FSM with datapath, took in actual FPGA slices for the target device. Finally, the estimated frequency is reported in column five. Studying the numbers, we can observe that the area does not increase nor does the frequency decrease substantially when we combine both paths of if-statements. Our experimental results, therefore, support the claim that SaPA brings additional performance improvement while not substantially increasing the area nor negatively affecting the execution frequency. Nevertheless, future research is necessary to investigate the impact in terms of frequency and area for a large number of test cases.

Table 7.1: Implementation metrics for the different schemes.

Function	Scheme	FSM states	Area (slices)	Freq. (MHz)
balanced	JMP	31	258	644
	PRED	20	461	644
	SaPA	16	411	644
unbalanced	JMP	61	780	644
	PRED	50	910	629
	SaPA	46	855	335
gcd	JMP	16	161	350
	PRED	14	153	357
	SaPA	14	153	357
adpcm	JMP	113	1409	328
	PRED	100	1470	328
	SaPA	95	1424	352
mergesort	JMP	102	726	324
	PRED	73	666	304
	SaPA	69	740	360
idct	JMP	240	2361	211
	PRED	162	2048	211
	SaPA	151	2409	211
lcs	JMP	76	748	329
	PRED	52	740	300
	SaPA	49	786	300

7.5 Conclusion

In this chapter, we argued that the typical JMP and PRED schemes found in conventional processors are not performing ideally when we consider hardware execution. The problem with the first is that we lose important cycles to jump from the state where the if-condition is evaluated to the correct branch state corresponding to the path chosen for execution. The PRED scheme solves this issue; however, it suffers from performance degradation when the if-branches are unbalanced.

To combine the advantages of both schemes, we presented a lightweight speculative and predicative algorithm which does predication in the normal way; however, it introduces a jump instruction at the end of the shorter if-branch to cope with the unbalanced situation. Furthermore, to leverage the hardware parallelism and the abundant resources, SaPA also performs partial speculation, i.e., no roll back necessary. This gains extra cycles in performance when the if-predicates take more than one cycle to evaluate.

We demonstrated based on two synthetic examples the benefit of this optimization and we validated it using five real world functions. The results show that performance degradation will not occur anymore for unbalanced if-statements and that the observed speedups range from 4% to 21%. Therefore, SaPA is a transformation that should be considered for inclusion in any HLS tool. Future research will analyze the impact of SaPA on a large number of kernels and investigate if any systematic relation can be observed between the input parameters (e.g., number of instructions per path or nested ifs) and the measured hardware metrics (i.e., area, frequency).

Note.

The content of this chapter is based on the following paper:

R. Nane, V.M. Sima, K.L.M. Bertels, **A Lightweight Speculative and Predicative Scheme for Hardware Execution**, IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig 2012), Cancun, Mexico, December 2012.

8

DWARV 3.0: Relevant Hardware Compiler Optimizations

In this chapter, we discuss several optimizations that have a great potential of improving the performance of DWARV2.0. In particular, we discuss in Section 8.2 period-aware scheduling and memory space allocation hardware specific optimizations, as well as an optimization to distribute the single conditional code register to selection operations in order to take advantage of the inherent hardware space locality propriety. Furthermore, in Section 8.3 we investigate several existing optimizations, available in CoSy as default engines, to assess the impact of using an optimization intended for generating assembly code in a hardware compiler. Software pipelining is one such optimization.

8.1 Introduction

In Chapter 4, we presented the DWARV2.0 compiler and described how it is implemented using the CoSy compiler framework. This first CoSy version included only basic code transformations, such as common subexpression elimination or static single assignment, but no other more advanced optimizations were analyzed and integrated in DWARV2.0. As a result, the performance of this version (i.e., 3.0) of the compiler greatly increased by using both hardware specific optimizations as well as studying and integrating the standard CoSy optimizations into DWARV.

In this chapter, we consider these two types of optimization sources and describe how the identified optimizations were integrated. First, we look at optimizations designed specially for the hardware generation process. In this respect, we consider two optimizations that have the role to increase the usability and applicability of generated hardware modules in the context of reconfig-

urable computing. The goal is to be able to exploit the flexibility offered by a configurable hardware platform to implement different operating frequencies and different memory system architectures.

These optimizations are period-aware scheduling, also called operation chaining, and memory space allocation optimization, respectively. The first has the clock period of the module as a design parameter and performs an operation scheduling within clock cycles. Concretely, it optimizes the number of cycles the design requires to complete execution. That is, if two dependent and subsequent operations (i.e., they are the vertexes of a dependency edge in the data dependency graph) can both complete execution in a time smaller than the given clock period for the system, then they can be placed in the same cycle. If that is not the case, then for these two operations at least two cycles are needed to finish execution. The second optimization relates to the designed memory system architecture, that is, it allows the compiler to partition the hardware's memory array parameters into different system memory banks to take full advantage of a distributed memory system architecture that is typically available on a reconfigurable platform.

In addition, we also want to investigate how existing optimizations designed for the generation of assembly code can perform when integrated into a compiler that generates Hardware Description Language (HDL). This is very useful to leverage the vast amount of CoSy framework standard optimizations. In section 8.3, we present work performed with M.Slotema in the context of his master project that investigates these optimizations individually to assess what contribution a particular transformation has on the generated hardware.

8.2 Hardware-Specific Optimizations

In this section, we provide implementation details of how we integrated into DWARV3.0 the distributed condition code, operation chaining and memory space allocation hardware specific optimizations.

Distributed Condition Codes

One of the advantages that hardware devices such as FPGAs have over conventional CPUs is their inherent predication support. Because of this feature, we showed in Chapter 7 that a High-Level Synthesis (HLS) tool targeting FPGAs should consider an optimization pass that performs if-conversion to enable the predicated execution. This was demonstrated in the same chapter, where a

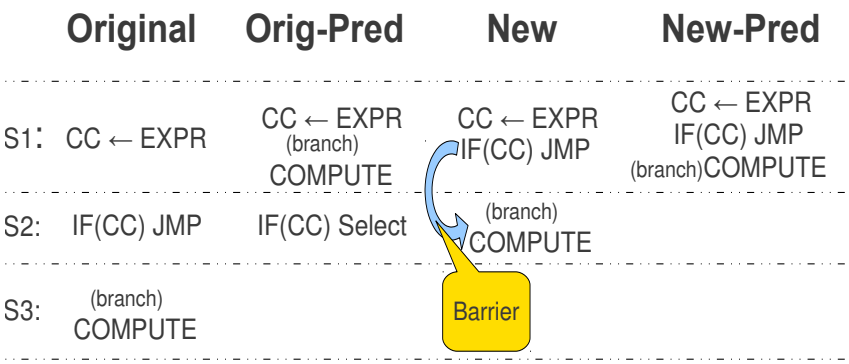


Figure 8.1: Various If Resolution Possibilities.

lightweight scheme was proposed that always performs at least as good as the typical jump scheme found in processor based systems. However, the concluding remark for the presented optimization was that it is beneficial only in cases where the predicate evaluation was performed in multiple cycles. This is an understatement because the observed results were negatively biased by the way DWARV2.0 was implemented.

That is, given a better design of the compiler, the optimization could achieve better results also for the cases where the predicate evaluation takes only one cycle, which is more common in real world applications. To illustrate this, consider the following abstract example showed in Figure 8.1 - *Original*. Here, we have a simple if statement, denoted by *IF (CC) JUMP-SCHEME*, where the jump-scheme represents the typical then-else paths that have to be followed based on the true/false evaluation of the predicate. *CC* is the (**C**onditional **C**ode) register holding the evaluation of the predicate and *COMPUTATION* represents both computational paths that could be taken after a decision was made based on the if-condition. In DWARV2.0, this was taking a minimum of 3-cycles (COMPUTATION contains one or more operations totalling minimum 1 cycle of actual computation) because each of the described steps was kept in a different cycle.

This suboptimal approach was caused by the allocation of one register for the condition values as is usually the case in a processor. This register, CC, was thus always used to hold the value of the predicate; therefore, we needed to have one cycle between its definition and use in the if-evaluation. Furthermore, because an if-statement is considered in every compiler framework as a barrier, we had to evaluate it completely before any subsequent instruction

could be executed; therefore, introducing another cycle. However, because in hardware we can have multiple CC registers, we redesigned the compiler to use locally distributed CC registers. These *if* local registers can be actually transformed into hardware variables that can allow the definition and use to be performed in the same cycle. This is illustrated in Figure 8.1 - *New* where the whole computation takes fewer cycles, i.e., minimum two cycles. Therefore, this modification improves both the execution time and the area required to implement all logic because now we have fewer wires connecting local variables to if-statements opposed to long wires that cross the whole chip going from the central unique CC register location used previously.

This *New* scenario described above is equivalent to the case shown in Figure 8.1 - *Orig-Pred*, where the if-conversion optimization also reduced the minimum required cycles from three to two by allowing the *COMPUTATION* to be performed at the same time with the *CC* evaluation, with the computation result being selected in the second cycle. Therefore, in cases where the CC evaluation took only one cycle, the predicated single CC register implementation gives the same performance results as in the new situation, that is, when the condition codes are distributed locally to selection operations, but then without any if-conversion optimization applied (Figure 8.1-*New*). However, in the new design, the removal of the if-statement with barrier by enabling if-conversion reduces the clock cycles to one. This situation is depicted graphically in Figure 8.1 - *New-Pred*. Therefore, this last case is possible when we distribute the condition code register locally to selection operations (e.g., if statements), taking advantage of the inherent hardware space locality propriety.

Period-Aware Scheduling

We implemented the operation-chaining optimization in CoSy by making the standard framework scheduling engine aware of a global period set by the user. This was implemented in the form of an additional engine that performs Data Dependency Graph (DDG) annotations and allows the scheduler to consider different edge latencies when performing the final scheduling. This can be seen as an extension to the CoSy standard framework scheduling and DDG engines. The flow of engines to support the period-aware scheduling is depicted graphically in Figure 8.2.

The engines in dark green boxes represent the engines written for this transformation, whereas the other two, *sched* and *ddg*, are CoSy standard engines. The first engine in the flow, **setperiod**, sets the global period in the IR for the current function. Furthermore, it initializes all nodes with a MAX period so that

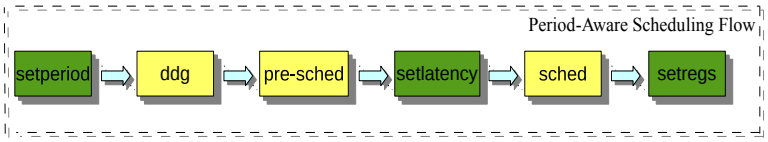


Figure 8.2: Period-Aware Scheduling Flow.

we start from the case where each in/out dependency edge between two nodes represents one cycle in the final schedule. To obtain these edges, we need to call the *DDG* engine that creates these edges between IR nodes. Subsequently, we run a version of the sched engine, *pre-scheduler*, to obtain an initial schedule based on the default edge latencies. This information is needed later by the **setlatency** engine to consider only the edges of nodes that are scheduled one cycle apart for possible chaining. The **setlatency** is thus the core engine that modifies the edge latencies defined by the *ddg* engine for nodes that are scheduled one cycle apart. The algorithm that performs these annotations is shown in Listing 8.1.

Listing 8.1: Engine **setlatency** Excerpt

```
edgeSlack = 'psc->slackPeriod - 'psc->nodePeriod;
      if(edgeSlack >= 'tgt->nodePeriod) {
         'edge->latency = 0;
         if (edgeSlack < 'tgt->slackPeriod &&
                              !'tgt->beginCycle)
            'tgt->slackPeriod = edgeSlack;
      }
      else {
         'edge->latency = 1;
         'tgt->beginCycle = TRUE;
         'tgt->slackPeriod = 'tgt->period;
      }
```

First, for each dependency edge, its (computational) slack time is computed (*edgeSlack*) as the difference between the slack and actual period of the DDG source node. If the difference is bigger or equal to the edge target node's period, i.e., the target node can execute in the same cycle as its predecessor from DDG, then the latency of the edge is made zero. A dependency edge

latency of zero implies that the source and target nodes can be scheduled in the same cycle. Furthermore, if this remaining slack is the smallest between all incoming node predecessor slack periods and this target node was not pushed to the next cycle (i.e., beginCycle is false), its slack period is updated to this new value. Otherwise, the latency of the DDG dependency edge is made one so that the scheduler knows to create a new cycle whenever it needs to schedule the target node. Finally, the target node is annotated to mark the beginning of a new cycle and its slack period is reset to the original value.

With this new information annotated on the DDG edges, we can call the scheduler again to generate the final schedule in which instructions are merged together because their periods can be executed within the global period. The flow is completed with the *setregs* engine that is used to annotate register variables in the IR to mark the fact that this virtual register is also a (VHDL) variable because of the cycle merge. Furthermore, those registers that are now used only as variables are annotated accordingly so that no register will be generated for them any more in the final emit phase. Finally, the last modification required is to annotate all the rules that generate instructions with a computational class that represents the maximum time required to complete the computation. For example, a shift operation is classified as *conversion* and will be initialized with 100 picoseconds. This operation is faster than, for example, an integer addition that is classified as *arithmetic_fast*. The following list enumerates all the classes used in DWARV3.0 as well as their allocated computation time. The initialization of the mentioned times is done in the *setperiod* engine.

- *conversions*: 100 picoseconds
- *memory*: 1000 picoseconds
- *comparison*: 1500 picoseconds
- *arithmetic_fast*: 3000 picoseconds
- *arithmetic_slow*: 4000 picoseconds

We note first that these class execution times are now experimental (they are based on approximations given to various computations extracted from Xilinx documentation manuals) and should be further evaluated to allocate more realistic execution times. However, the absolute time that such a computational class can take depends on multiple factors, such as design density, computational cores and their location relative to the computation or the routing of the design. Therefore, deriving more accurate numbers is not trivial and it will

be considered in future work. We envision some heuristic model that could dynamically allocate different node periods based on different situations. Second, when validating this transformation, we observed that it has a bonus effect by always scheduling register initializations with constants in the same cycle with the corresponding use of that constant; therefore, allowing the synthesizer tool to remove these redundant registers efficiently. This reduces the total area. Finally, as mentioned in the beginning of the section, this transformation allows also for the predication to take effect for cases when the if computation does not take more than one cycle.

Memory Space Optimization

The previous DWARV implementation was designed on the MOLEN machine organization which defined one global shared memory from which all function parameters were accessed. This suboptimal design was motivated by the fact that typical software applications also have only one shared memory which contain the application data. Therefore, to support the same computational model, MOLEN was designed to keep these differences to a minimum when moving specific kernels to hardware. As a result, the generated kernel hardware had to access its parameters from a similar on FPGA single shared memory. However, this is a severe restriction for hardware kernels running on an FPGAs that contain multiple memory modules in the form of distributed BRAMs across the entire area. Therefore, DWARV2.0 was under-performing because only one memory operation could be scheduled each cycle. The logical and simple solution to this problem is to use different BRAMs for different function parameters, thus, allowing the scheduling to perform multiple memory operation in one cycle if different parameters are used in that cycle. This would be possible because, in this new case, we would not have a memory contention for the same BRAM. This feature is known in the embedded systems domain as using different memory spaces.

For this optimization, we assume that a previous running partitioning tool (see Chapter 3) places restrict clauses on parameters. This implies that the semantics of the program are not affected by this transformation. Any parameter that cannot be disambiguated and, therefore, it does not have a restrict clause before it, will be placed by default in the generic memory space.

The CoSy framework supports memory spaces through its embedded C99 [47] frontend extension. This allows memory spaces to be defined and used in the compiler. The code snippet shown in Listing 8.2 illustrates how a simple function prototype has been extended to state explicitly that parameter *p1* is

Listing 8.2: Example of a Procedure Declaration with Multiple Memory Spaces.

```
void exampleMultipleBramSpaces
(
   bram1 int* p1, bram2 int* p2, bram3 int* p3
)
{
        ....
}
```

defined in *bram1* memory space, *p2* in *bram2* respectively *p3* in *bram3*. However, given that the DWARV compiler is part of a complex Hardware/Software (HW/SW) co-design tool-chain that does not accept as input Embedded C99, in which the whole application has to be run and instrumented by gcc based tools, the definition of such bram keywords in the function prototype is not allowed. Without the explicit definition of these keywords representing memory spaces in the function prototype, the CoSy frontend would assume that we have only one memory space, the generic one. This leads to a schedule where memory operations have to be given different cycles even if they are not dependent. Furthermore, the CoSy frontend generates only one memory space metadata structure in the IR to be referred by all types in the function. This is important to keep in mind for the subsequent implementation details.

To implement support for different memory spaces such that the complete tool-chain can still compile the whole application automatically, we must not annotate the function definition with CoSy-specific memory space identifiers, but we need to use one of the following two methods:

1. Use pragmas to define the parameter to memory space mapping.
2. Assume each function parameter is mapped to a different memory space.

For the current implementation, it is sufficient to use the second option. Therefore, we assume that each new encountered function parameter will be allocated in $\langle bram \rangle$i memory space, where i is the number of the parameter in the list of the respective function prototype. For example, parameter p2 from the function prototype in Listing 8.2 will be allocated implicitly in *bram2*.

The next step after the creation of different memory space metadata structures in the Intermediate Representation (IR) is to modify the parameter types to use the new metadata instead of the original space metadata representing the

generic memory space. In CoSy, to change the memory space information for a type, we need to clone that type and make its memory space pointer point to one of the new created memory space metadata. This step is implemented in the *hwconfig* engine and is essentially a walk over the list of parameters, getting and cloning their type, increasing an iteration index after it is concatenated into a *bram* string which is allocated to a newly created space structure's name. Finally, we associate the space with the new type, and the type with the parameter. After completing this step, we have each parameter pointing to unique types with a different memory space each.

However, performing this cloning and dissociating the parameters from the original type, we introduce an invalid state in the CoSy IR. That is, parameters have a new type now, whereas the expressions and statements that use these parameters still point to the old type with the generic memory space. To keep memory space related modification consistent, we have to propagate these new types to all expressions and statements accordingly. This is implemented in a new engine called *setmbramtypes*. First, to propagate the type to expressions we perform a depth-first traversal with a pre- and post-action function for each node traversed. The CoSy framework offers a specific function for such traversals named **prepost_exprs_in_proc**, which takes as arguments the function to perform the pre-action, a function to perform the post-action and a base parameter to a structure to keep common information through all calls. Furthermore, we use a stack to remember the expression that has the pointer type on higher levels. For this one, we need to set the type to the type obtained from the leaf parameter node underneath it. This setting is done in the post-action of the traversal.

Second, to propagate the types also to statements and expressions containing locals derived from pointer parameters, we need to iterate over all statements in the procedure as well and look at local definitions and instantiations from pointer parameters. For these pointer type assignments, we copy the type of the right hand side to the local on the left leaf side of the assignment tree. We subsequently make a global $\langle modified \rangle$ variable true, so that we remember to iterate once more over all expressions, as well. We need to perform this action to propagate these newly allocated real space types to locals up the tree of expressions, as well. We perform these expression and statement traversals as long as we still have modifications. The final touches are to implement the disambiguation function to return true for different memory spaces and create resources, templates, and dynamic rule function templates for the CoSy *sched* scheduler engine. The disambiguation function is called implicitly by the default sched engine, whereas the resource, template, and rule function

templates are used to select the resources which have to be used by each rule instance. Because for each memory parameter we define different (memory) resources, the memory disambiguator will report that memory operations with different memory spaces assigned are independent. Therefore, we can now perform two or more memory operations in parallel.

We mention that the current implementation of CoSy is restrictive from the point of view of creating above elements dynamically. In other words, we cannot accommodate on the fly the necessary number of resources and templates that are to be associated with the corresponding number of memory spaces. These have to be specified when DWARV is built from sources and have a maximum limit that is defined at compile time (currently set to 60). However, considering the current FPGA implementation and the allocated number of BRAMs available on them, we believe that this limitation will not restrict in any way the working of the compiler.

8.3 CoSy Compiler Optimizations

In this section, we present one work in progress which is aiming at taking existing software optimizations and customizing them to exploit the hardware characteristics fully. We will do this by restricting ourself to the optimizations available in CoSy in the form of engines. In Chapter 3, we presented all engines available in CoSy, and in this section we select only a few for further investigation. That is, the techniques covered here are loop unrolling, loop invariant code motion, software-sided caching and algebraic simplifications. Table 8.1 shows how these techniques are divided into CoSy engines and it describes briefly what specific task the engine performs. We believe that leveraging on existing software optimizations and extending or relaxing them for the hardware compilation process, will allow us to close the performance gap between a manually written versus generated HDL code. The goal of this section is thus to investigate software optimizations from a hardware perspective in order to be able to understand how these existing optimizations could be applied for the hardware generation process. The inherent benefit is to possibly leverage on the large amount of previous research done for software compilers. In this sense, work has been started which concluded in a master thesis [75]. The discoveries are briefly summarized in this section.

Table 8.1: Selected Optimisation Engines.

Engine name	Brief description
algebraic	Performs algebraic simplifications
cache	Reducing memory accesses by delaying memory writes
constprop	Performs constant propagation
demote	Reduces the bitwidth of operations
loopfuse	Rewrites two successive loops into a single loop
loophoist	Applies loop-invariant code motion on statements
loopinvariant	Applies loop-invariant code motion on expressions
loopscalar	Reduces memory accesses by delaying memory writes
loopunroll	Performs loop unrolling
scalarreplace	Rewrites structures into local variables
lrrename	Rewrites a variable with separate life ranges into different variables

Engines and Test Cases

To present the issues that arise when integrating (software) optimizations into a hardware compiler‘s engine-flow, we use 19 test cases. Using these kernels, we study both the individual and cumulative effect on the compiler performance when integrating the selected engines shown in Table 8.1. The test cases were carefully selected to contain different C-language characteristics, i.e., loops without conditional paths that can be easily unrolled, loops that contain conditional paths and which are not easy to parallelize after unrolling, loops containing loop-invariant code or accessing repetitively array entries, loops with variable or constant bounds, functions operating on floating-point and fixed-point numbers. This variety of C-language characteristics allows to show objectively if, when, and how a particular optimization is useful.

Optimization Order

Before we investigate which optimizations are useful from a hardware generation perspective, it is important to understand that not only the particular optimization should be investigated if and how it should be applied in the new context, but also in which order the optimizations should be applied. That is, given a sequence of compiler transformations, the question of where to insert a particular optimization is as important as the content of the optimization.

To demonstrate this, we inserted the *loopunroll* engine in different places in DWARV2.0's sequence of transformations.

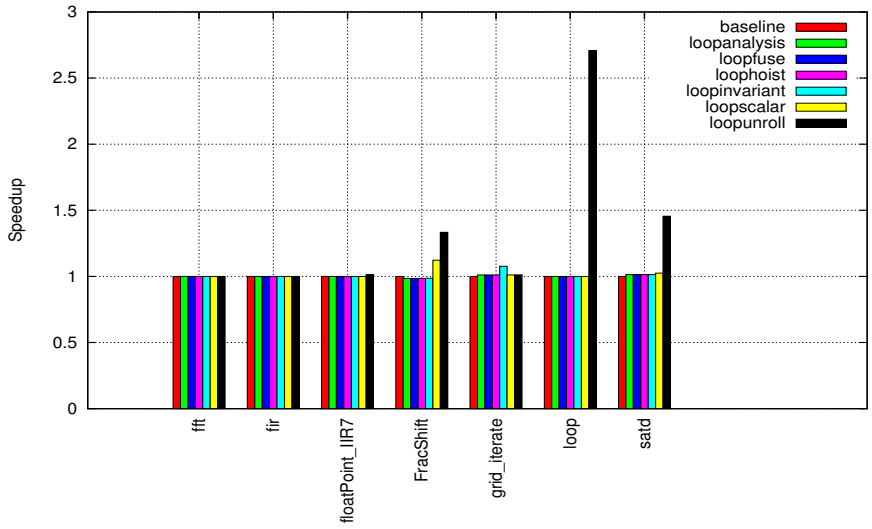


Figure 8.3: Results for Placing Loop-Optimising Engines after SSA Engines.

First, we inserted *loopunroll* before the SSA transformation is performed. Considering the **satd** kernel that contains two for-loops, which can be unrolled four times, a speedup should have been observed. However, this was not the case. Detailed output from the *loopunroll* and *loopanalysis* engines showed that the loop induction variable is not recognized. This is caused by two engines: the *ssabuild* engine and the *ssaconv* that combined perform the SSA transformation in CoSy. Because we do not have only one loop induction variable, all memory access that belong to different variables originating from the induction variables are serialized, and as a result, no parallelism between memory operations can be extracted. The solution is to move *loopunroll* after SSA engines. Figure 8.3 shows that, in this case, a speedup of 1.5 can be achieved, confirming that the order in which compiler optimizations are applied is very important in the context of hardware compilers.

Optimization Configuration

An optimization (using the same configuration/values for its parameters) can have different impacts on different kernels. That is, for one kernel it can be beneficial and, therefore, when the optimization is applied, it will lead to a boost in performance. However, for other kernels the same optimization can experience a slowdown. To illustrate that an optimization is beneficial only in certain circumstances, we took the same *loopunroll* engine and investigated its impact on different kernels for different values of the maximum unroll factor configurable parameter.

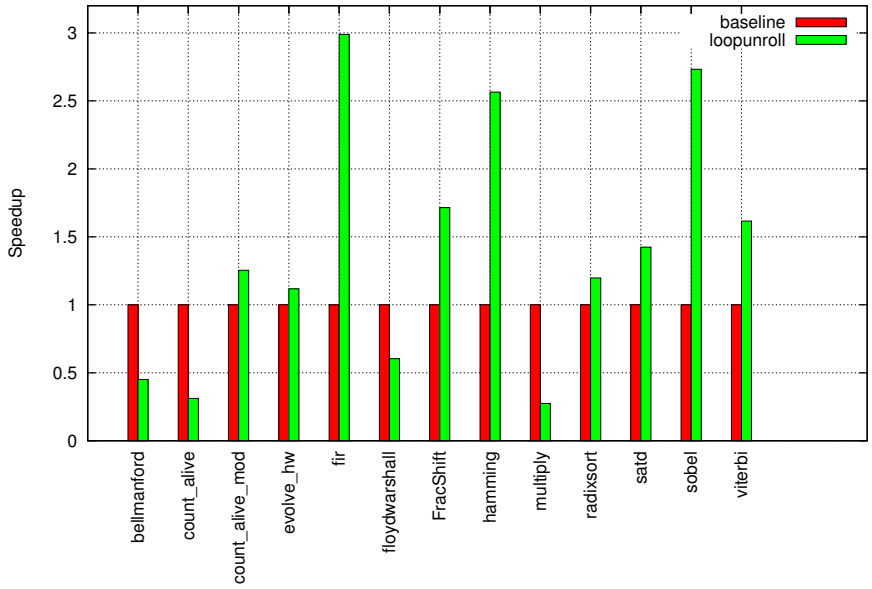


Figure 8.4: Comparison of DWARV without (baseline) and with loop-unrolling (unroll factor set to 128).

Figure 8.4 shows the effects of loop unrolling on different kernels. Five of the kernels, the *fir*, *FracShift*, *hamming*, *sobel* and *viterbi* kernels, show a good speedup of over 1.5, some even approaching three times speedup. Three kernels show a decent speedup upto 1.5. Four kernels show a significant drop in performance. These kernels are the *bellmanford* kernel, the *count_alive* kernel, the *floydwarshall* kernel and the *multiply* kernel. The average speedup of the *loopunroll* over all kernels is 1.26. In order to understand what is the reason,

we analyzed the kernels more closely.

Listing 8.3: The loop of the *count_alive* kernel

```
for( x = i - 1; x <= i + 1; x++ ) {
        for( y = j - 1; y <= j + 1; y++ ) {
                if(( x == i ) && ( y == j ))
                     continue;
                if(( y < size ) && ( x < size )
                  && ( x >= 0 ) && ( y >= 0 ) ) {
                        a += CELL( x, y );
                }
        }
}
```

The *count_alive* kernel is part of a Game of Life implementation. The kernel is responsible for the counting of the number of neighbors which are alive. This is achieved by looping through three rows, starting from the row above the cell, and three columns, starting from the column to the left of the cell. However, the kernel is written in an unusual manner. Instead of looping from -1 to 1, the kernel loops from $i-1$ to $i+1$, where i is one of the coordinates of the center cell. This loop is shown in Listing 8.3. Because of this, the *loopunroll* engine cannot determine these fixed bounds of three by three. A modified version of the kernel, the *count_alive_mod* kernel, loops from -1 to 1 and uses additional variables to determine the coordinate of the cell that is to be checked. The modified loop is shown in Listing 8.4. According to the results from Figure 8.4, the *count_alive_mod* kernel achieves a speedup of 1.25.

Listing 8.4: The modified loop of the *count_alive* kernel

```
for( k = -1; k <= 1; k++ ) {
        x = i + k;
        for( l = -1; l <= 1; l++ ) {
                y = j + l;
                ...
        }
}
```

The problem with the *bellmanford* and *floydwarshall* kernels is the structure of the loop bodies. The bodies of these kernels are governed by if-statements.

Because of this conditional statement, statements from the ith loop iteration cannot be merged into the same basic block as statements from iteration $i - 1$ or $i + 1$ after performing loop-unrolling. Thus, no additional Instruction Level Parallelism (ILP) is exposed. The main loop of the *bellmanford* kernel is shown in Listing 8.5. However, this structure is not the only cause. The *count_alive_mod* kernel has a similar structure but does actually experience a speedup. The major difference between the *count_alive_mod* kernel and the other kernels is the fact that the *count_alive_mod* kernel contains a constant number of iterations, and, when a loop has a non-constant number of iterations, a pre-processing loop is required. This pre-processing loop in combination with no exposed ILP results in the execution time decrease. Something similar happens with the *multiply* kernel. The statements in the main loop of this kernel are dependent on the results of the previous iteration of the loop. Due to this, the amount of ILP is limited. Combined with the pre-processing loop, a drop in the execution time of this kernel occurs.

Listing 8.5: The main loop of the *bellmanford* kernel

```
for (i=0; i < nodecount; ++i) {
  for (j=0; j < edgecount; ++j) {
    if (distance[source[j]] != INFINITY) {
      new_distance = distance[source[j]]+weight[j];
      if (new_distance < distance[dest[j]])
                distance[dest[j]] = new_distance;
    }
  }
}
```

These first results indicate that an optimization is not always useful, showing that compilers used for the generation of hardware behave in the same way as compilers generating assembly code (i.e., software). For the latter category, previous research [2] extensively studied the relationships between optimizations passes and concluded that optimizations are sensitive to the particular order in which they are called as well as the test case for which they are applied. To exemplify this from a hardware point of view, we make use of the unrolling factor configurable parameter of the *loopunroll* engine and show the influence of this option on the execution time. Please note that the result graph does not show all kernels, even though they may have a decent speedup. A representative selection of kernels was made to keep the description brief. The kernels that are not depicted here behave similar to kernels that are depicted.

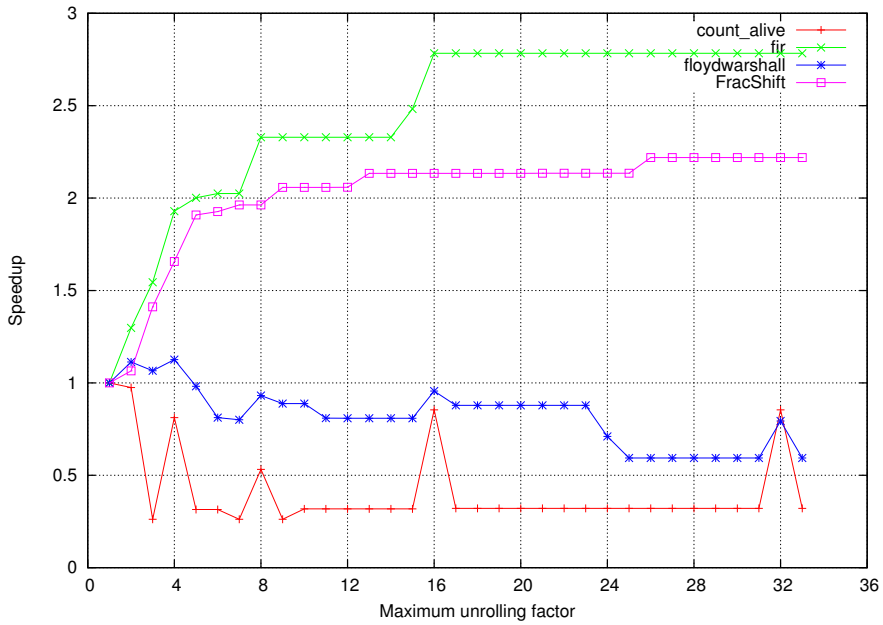


Figure 8.5: Influence of the maxfactor option on the execution time.

The influence of the *maxfactor* option can be seen in Figure 8.5. For the *fir* and *FracShift* kernels, as well as most other kernels not depicted in the figure, the higher the unrolling factor, the better the result. However, this is not true for the *count_alive* and the *floydwarshall* kernels. These kernels do not perform well due to overhead in the pre-processing that is necessary. However, Figure 8.5 shows that when the maximum factor is a power of two, the results are improved. This happens because, inside the pre-processing loop, a modulo operation is normally necessary. However, when the unrolling factor is a power of two, this module is replaced by a logical operator. Nevertheless, for these two kernels, the best execution times (i.e., no slowdown) are obtained when the unroll factor is zero, that is, when no unrolling is applied, proving that an optimization is not always beneficial.

Discussion

Figure 8.6 shows the total speedup for six optimizations both individually and cumulative. These optimization techniques were integrated with DWARV and included loop unrolling, loop invariant code motion, software-sided caching of

memory operations and algebraic simplifications. The speedup in this figure is with respect to DWARV2.0 without any added optimizations. The goal of the exercise was to show that different aspects such as the optimization sequence order, as well as the optimization configuration, play an important role in the final performance of the compiler. The relative minor 1.45 average speedup shown in Figure 8.6 can be explained by the fact that these six optimizations were neither extensibility inspected on how to integrate with the rest of the compiler's existing transformations, nor properly configured on a per case basis. As an example, in the introductory chapter of [75], the *satd* kernel was modified manually resulting in a speedup of 3.3, whereas when applying the six optimizations only a 1.4 speedup could be achieved. This proves that these six optimizations should be both better configured and integrated in the engine flow. Furthermore, other optimization engines could be applied.

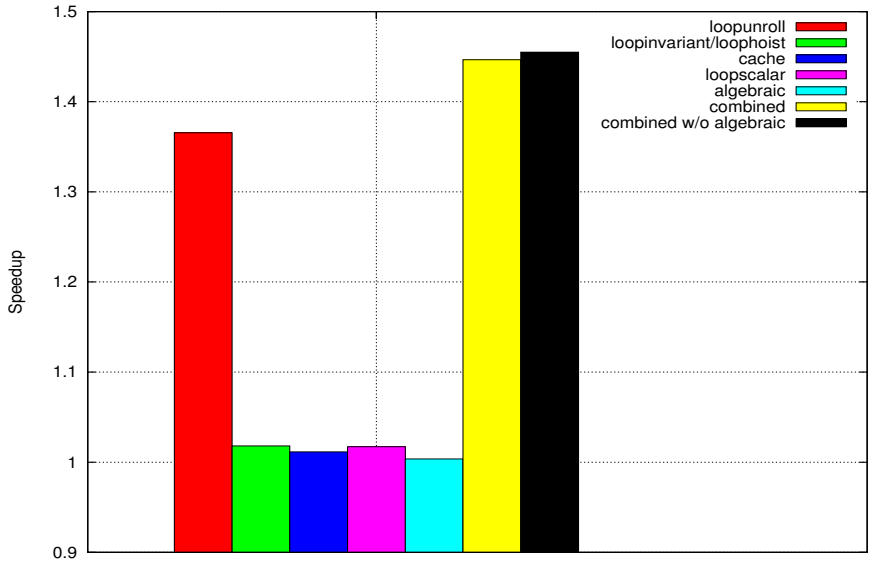


Figure 8.6: Average execution time speedup of the different optimization engines.

To show that other optimizations are needed, such as software pipelining, polyhedral optimization, array partitioning or peephole optimizations like strength reduction, a comparison with LegUp has been performed with respect to its optimized output. LegUp makes use of optimization passes built into the LLVM framework [66]. As described in [75], the impact of these passes on the performance was evaluated and compared to the results obtained with DWARV2.0. Figure 8.7 shows both the impact of the six optimizations inte-

grated in DWARV2.0 with respect to DWARV2.0 without any optimizations, and the impact of the LLVM optimization passes on LegUp, with respect to LegUp without any optimizations. Overall, the optimizations in LegUp produce a speedup in every kernel, except for the *fft* kernel. The difference with DWARV2.0 is due to the fact that LegUp simply uses more optimization techniques than DWARV2.0.

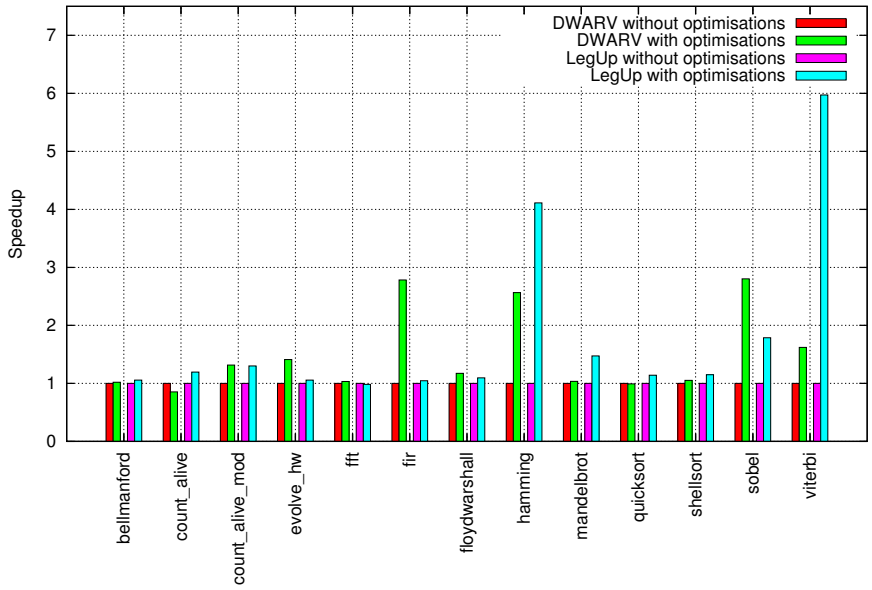


Figure 8.7: Impact of optimisations for DWARV2.0 and LegUp 2.0. The graph shows pairwise normalized results of optimized vs baseline version for each compiler. The goal is to show the optimization potential. Results between compilers are thus not comparable.

The figure also shows that there are promising optimization techniques not yet integrated with DWARV2.0 that can produce good speedups. This can be seen in the *hamming* kernel, the *mandelbrot* kernel and the *viterbi* kernel where the speedup obtained by LegUp is much more than the one observed in DWARV2.0. Nevertheless, the impact of the optimizations on the *count_alive_mod*, *evolve_hw*, *fir*, *floydwarshall* and *sobel* kernels is larger than the impact that the LLVM optimization passes have. This confirms the hypothesis that when more CoSy optimization engines are included, noticeable speedups will be achieved. However, future research is needed.

One of the research directions is to study how an optimization parameter has to be configured on a per case basis. More concretely, an optimization has different parameters that can be tweaked, such as loop unroll maximum factor or the number of nested loop levels that will be unrolled. To obtain a good speedup, simply including a software optimization with its parameters set to default values is not sufficient. One would need to provide sensible values for these parameters on a per case basis to obtain good results. Furthermore, we need not only to change the configurations of the different optimizations, but we need also to include/exclude or change the optimizations sequence based on a per case basis. Anything less than that will not give optimal results as it was shown in this section. Future work will analyze what strategies are necessary to devise algorithms able to decide automatically when and how to include a particular optimization in a hardware compiler.

One final remark is needed to acknowledge the software pipelining optimization as one of the most promising to integrate in DWARV3.0. According to discussions with Hans van Sommeren from ACE bv. and Stephen Neuendorffer from Xilinx Research Labs, this optimization has potential of speeding up kernels by a few orders of magnitude. The CoSy framework has the advantage of supporting the software pipelining optimization in the form of a set of standard engines that can be included in any compiler. To test the benefit of this powerful optimization for hardware designs, we simply plugged the software pipelining related engines in DWARV3.0. The implementation was straightforward, and it was done by following the CoSy documentation that gives precise step-by-step instructions about what engine to include in the engine flow. However, the experiments performed on simple tests revealed that this optimization is rarely enabled. This is mostly caused by the vast amount of configurable options that need to be fine-tuned to make this optimization fully functional. Given the time limit, we leave the proper configuration and performance analysis of the software pipelining optimization as future work.

8.4 Conclusions

In this chapter, we presented a few hardware relevant techniques intended to improve the performance of the unoptimized DWARV2.0 hardware compiler. We argued that and described how supporting predicated execution, chaining operations and using multiple memory spaces reduces this performance gap. Results for the new DWARV3.0 compiler that includes these optimizations will be presented in the next chapter. Finally, section 8.3 showed that there is room

Table 8.2: Overview of New Optimizations in DWARV 3.0.

Optimization Name	Optimization Origin	Included in Evaluation
Area Constrained Generation	Chapter 6	No
If-conversion	Chapter 7	No
Distributed Condition Codes	Chapter 8	Yes
Period-Aware Scheduling	Chapter 8	Yes
Memory Space Allocation	Chapter 8	Yes
algebraic	CoSy Framework	Yes
loop unroll	CoSy Framework	No

for improving DWARV3.0 with existing optimizations available in the CoSy framework. However, we did not include these already because, as showed in the mentioned section, this is not a simple matter of just plugging-and-using an engine (similar research efforts are also performed in the LegUp compiler and published in [44]). First, we need to adapt these engines to better suit the hardware generation process. Second, we also need to understand the order in which to integrate them. As a consequence, future research is necessary to devise algorithms of how and when a particular optimization is useful and, as a result, it should be activated. In this sense, CoSy offers great mechanisms (skip engine option) that can be used to do this activation/deactivation of engines and a powerful mechanism to pass command line options to optimization engines.

Table 8.2 summarizes the optimizations added in this version of DWARV. First column list the optimization name, the second column lists the origin of the optimization, and finally, third column highlights whether this will be included in the compiler used in the final evaluation of the next chapter.

Note.

The content of this chapter was submitted as part of the following publication:

R. Nane, V.M. Sima, K.L.M. Bertels, **A Survey of High-Level Synthesis Tools and Comparison with DWARV 3.0**, Submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, April 2014.

9

Hardware Compilers Evaluation

In this chapter we evaluate state-of-the-art High-Level Synthesis (HLS) tools by investigating their performance with respect to a number of predefined performance criteria. After formally introducing the goal of this chapter, we present in Section 9.2 the four selection criteria we require to include a tool in the comparison. Subsequent section describes the tools selected. Sections 9.4 and 9.5 present software and hardware metrics for both the benchmark functions used in the comparison as well as for the generated hardware kernels, respectively. The comparison results are shown in Section 9.6 while the last section draws the conclusions of the evaluation.

9.1 Introduction

To obtain the most realistic state-of-the-art quantification, we need to evaluate as many compilers as possible. The related work section presented an abundant number of available hardware compilers. Because the number of compilers available today both in the industry and academia is not small, coupled with the fact that not all tools from industry offer an evaluation license, we could not include all compilers in the evaluation. From the list of tools presented in Chapter 2, we investigated approximately half of them. However, after a closer investigation, we decided to exclude some of them because they did not fulfil all the selection criteria (i.e., the tool output is not comparable to DWARV3.0) defined in the next section. We will briefly describe in Section 9.3 what were the specific issues that led to the decision of not including a particular tool into the final comparison. However, before reasoning about this tool selection, we need first to describe the specific selection criteria.

9.2 Tool Selection Criteria

We number the criteria with (1), (2), (3) and (4) respectively, for ease of reference in the next section:

1. The first and most important requirement to include a tool in the comparison was to generate accelerators that are able to connect with a shared memory via their generated interface. The reason is that we want to focus on tools that are closely related to the design principles of DWARV3.0, and, therefore, allow us to compare and evaluate their performance against that of DWARV3.0. Figure 9.1 depicts the minimum requirements for the generated accelerator's interface graphically. Signals such as set_data_address_Xi, get_data_value_Xi, write_data_value_Xi, write_enable_Xi should be part of the interface to allow values for parameter named Xi to be read/written from/to the shared memory. This requirement is directly derived from one of the assumptions in DWARV3.0, namely from the MOLEN context where its generated accelerators are integrated. Although derived from MOLEN, this requirement is generic for any shared/distributed type of system. Other signals necessary for synchronization are reset, clock, start and done of the accelerator.

2. The second criterion concerns the availability and completeness of tool documentation, if it is still maintained, the time required to learn how to use it and the time required to adapt the benchmark kernels to syntax accepted by the compiler. This implies that compilers requiring a modified or extended subset of the C language will fail this criterion because it is time consuming to learn the necessary language extension to rewrite the benchmark kernels to a form accepted by the tool.

3. The third criterion concerns the availability of a free evaluation or academic license. When this was not the case, no further action was taken to include it in the evaluation.

4. The fourth and final criterion requires that the tool in question can generate a test bench. This is needed not only to test the correctness of the generated hardware, but also to run simulations to obtain actual simulation cycle counts based on a particular input data set.

Before we start describing the tools and present the results, it is worth noting that, for the various tools, there are different optimizations and features avail-

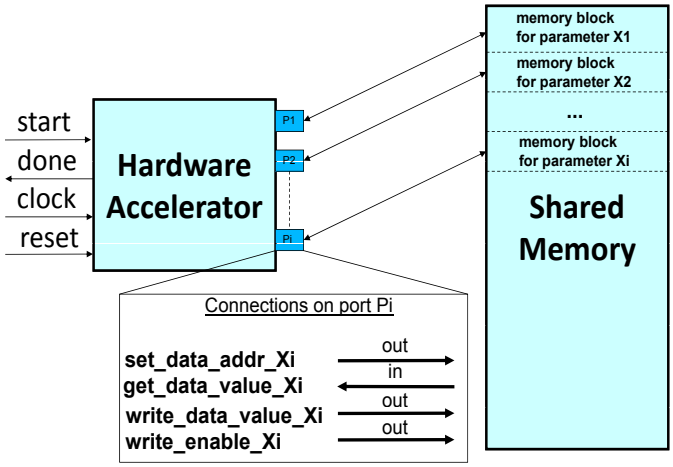


Figure 9.1: Hardware Accelerator Required Memory Connections.

able, especially in the commercial tools. For practical reasons, we describe only the performance out-of-the-box. That is, no effort is made to enable any optimization or feature manually. This is important because we also want to assess the automation support in applying hardware optimizations. Ideally, optimizations should not be enabled manually, but should be included automatically if beneficial. This would make the transition and acceptance between software developers faster and seamless. Furthermore, due to a limited time budget, the evaluation of tool specific optimizations and features is out of the scope of this work. Therefore, the presented results are for a *simple* comparison. Nevertheless, we intend to show as well the optimization support and results as future work. However, this would be possible only with the compiler company's help, which knows best how and what would be the best particular optimizations choice for a specific kernel. Only in this way we could obtain realistic and accurate optimized performance results that would not be influenced by the unintended omission of an optimization or of its wrong appliance by a neutral reviewer.

9.3 Overview Selected Compilers for Evaluation

In Chapter 2, we described a plethora of hardware compilers. In this section, we will revisit those compilers, and reason about the possibility to include

Table 9.1: Overview Selected Compilers.

Compiler	Owner	License	Year	Interface	Input	Available	TestBench	CNM
ROCCC1.0	U. Cal. River.	Academic	2005	Stream	C subset	Evaluation	No	1,2,4
ROCCC2.0	Jacquard Comp.	Commercial	2010	Stream	C subset	Evaluation	No	1,2,4
Catapult-C	Calypto Design	Commercial	2004	All	C/SysC	No	Yes	3
CtoS	Cadence	Commercial	2008	All	SysC	TUD lic.	Only Cycle	2,4
DK Design Suite	Mentor Graphics	Commercial	2009	Stream	HandelC	No	No	1,2,3,4
CoDeveloper	Impulse Accelerated	Commercial	2003	Stream	ImpulseC	Evaluation	Yes	1,2
SA-C	U. Colorado	Academic	2003	Stream	SaC	No	No	1,2,3,4
SPARK	U. Cal. Irvine	Academic	2003	All	C	No	No	3,4
CHC	Altium	Commercial	2008	ASP	C	Evaluation	No	1,4
Vivado HLS	Xilinx	Commercial	2013	All	C/SysC	TUD lic.	Yes	
LegUp	U. Toronto	Academic	2011	Shared	C	Free	Yes	
PandA	U. Polimi	Academic	2012	Shared	C	Free	Yes	
HercuLeS	Ajax Compiler	Commercial	2012	All	C	No	Yes	3
GAUT	U. Bretagne	Academic	2010	Stream	C/C++	Free	Yes	1
Trident	Los Alamos NL	Academic	2007	Shared	C subset	Free	No	2,4
CtoVerilog	U. Haifa	Academic	2008	N/A	C	Free	No	1,4
C2H	Altera	Commercial	2006	ASIC	C	TUD lic.	No	1,4

Table 9.2: Overview Selected Compilers (Cont).

Compiler	Owner	License	Year	Interface	Input	Available	TestBench	CNM
Synphony HLS	Synopsys	Commercial	2010	All	C/C++	No	Yes	3
MATCH	U. Northwest	Academic	2000	DSP	Matlab	No	No	1,2,3,4
CyberWorkBench	NEC	Commercial	2011	All	BDL	No	Cycle/Formal	2,3,4
Bluespec	BlueSpec Inc.	Commercial	2007	All	BSV	No	No	2,3,4
AccelDSP	Xilinx	Commercial	2006	DSP	Matlab	No	Yes	1,2,3
Kiwi	U. Cambridge	Academic	2008	.NET	C#	No	No	1,2,3,4
CHiMPS	U. Washington	Academic	2008	Shared	C	No	No	3,4
MaxCompiler	Maxeler	Commercial	2010	Stream	MaxJ	No	No	1,2,3,4
SeaCucumber	U. Brigham Y.	Academic	2002	All	Java	No	No	2,3,4
DEFACTO	U. South Cailf.	Academic	1999	Stream	C subset	No	No	1,2,3,4
PipeRench	U.Carnegie M.	Academic	2000	Pipes	DIL	No	No	1,2,3,4
Garp	U. Berkeley	Academic	2000	Loop	C subset	No	No	1,2,3,4
Napa-C	Sarnoff Corp.	Academic	1998	Loop	C subset	No	No	1,2,3,4
gcc2verilog	U. Korea	Academic	2011	Shared	C	No	No	3,4
Cynthesizer	FORTE	Commercial	2004	All	SysC	No	Yes	2,3
eXCite	Y Explorations	Commercial	2001	All	C+pragmas	Evaluation	Yes	2

them in the final evaluation. To present the selection in a compact way, we make use of Tables 9.1 and 9.2. These tables show in the first four columns the name of the tool, who distributes it , under what type of license it is available and since what year. The fifth column highlights what type of interface the tool can generate, i.e., streaming, shared, or both, while the sixth column lists how the tool input should be specified. The next two columns indicate if the tool can generate test benches and whether the tool is available for download. If the latter is true, this column will specify if it is freely available (or if TU Delft had a license already for it or if it is available under an evaluation license). Finally, the **C**riteria **N**ot **M**et (CNM) column, shows which of the four above mentioned criteria are not met by the tool, and, therefore, will not be included in the tool comparison. By negation, a tool is included in the comparison only when this column is empty, that is, all criteria were fulfilled.

Tables 9.1 and 9.2 show thus that it is impossible to obtain results for all the tools given in the related work chapter (Chapter 2). As a result, in the subsequent we will focus on tools intended for multiple domains and that are able to generate shared memory interface based accelerators. Although some of the tools can support different types of interfaces, to make results comparable, we only look at this particular interface. This is because we want to quantify the performance of DWARV3.0 at the same time as well, and as this supports only the shared memory interface, we restrict ourselves only to similar tools.

9.4 Benchmark Overview

The test bench used for the evaluation is composed of 11 carefully selected kernels from six different application domains. Although, CHSTONE is becoming the test bench used for benchmarking HLS tools, we didn't use it entirely because of two reasons. First, we wanted to be objective and evaluate compilers that were not particularly designed to give good performance only for kernels in this test suite. Therefore, we selected kernels from the examples directory found in the releases of different tools, in particular we used kernels from Vivado HLS, DWARV and LegUp. Please note that LegUp supports the CHSTONE benchmark, therefore, the kernels selected from its test bench are actually kernels originating from CHSTONE. However, these have been manually changed to transform global parameters to function parameters to enable DWARV compilation. The second reason for not using CHSTONE entirely, is thus that DWARV is not supporting global parameters, and the CHSTONE kernels not selected make heavily use of these variables. As DWARV will sup-

Table 9.3: Comparison Benchmark Characteristics.

Function Name	**Application Domain**	**Benchmark Suite**	**C-language Characteristics**									
			Loop	Add	Mul	Div	Shift	IFs	LDs	STs	FuncCalls	#lines
matrixmult	Mathematics	Vivado HLS	3	1	1	0	0	0	3	2	0	8
adpcm-encode	Telecom.	CHSTONE	2	17	9	0	7	2	15	3	15	65
aes-encrypt	Cryptography	CHSTONE	3	1	1	0	0	7	3	0	6	50
aes-decrypt	Cryptography	CHSTONE	3	1	0	0	1	7	3	0	6	60
gsm	Telecom.	CHSTONE	0	0	0	0	0	0	0	0	4	5
sha	Cryptography	CHSTONE	1	0	0	0	0	0	2	0	3	10
cmultconj	Mathematics	DWARV	0	5fp	3fp	0	0	0	8	2	0	6
satd	Multimedia	DWARV	2	32	0	0	0	0	49	21	0	50
sobel	Image Proc.	DWARV	4	14	5	0	9	5	4	1	0	50
viterbi	Telecom.	DWARV	10	17	0	0	36	8	12	13	0	120
bellmanford	Logistics	DWARV	4	2	0	0	2	3	12	3	0	30

port global variables in the future, we didn't allocate time to rewrite all the CHSTONE kernels to be DWARV compliant.

The functions selected for hardware acceleration are introduced by means of Table 9.3, where we mention in the second and third columns the application domain of the corresponding kernel as well as the belonging benchmark from where the function was extracted. The fourth column shows C-code characteristics, e.g., number of code lines, number of loops and arithmetic int operations.

9.5 Generated Hardware Overview

The HLS tools were configured to generate hardware for the Kintex7 board xc7k325t-2-ffg900. Because we now use a newer Virtex board to relate to the previously obtained results (see Section 4.4), the following adjustment can be made: to compare the area for the Virtex 5 platform the number of registers and LUTs should be roughly divided by 2 because the newer version of boards have more logic elements in a slice [80]. Furthermore, in obtaining the hardware metrics, we used Xilinx ISE 14.3 with no physical constraints for the synthesis.

Before we present the actual results, it is important to note that the tools generate code differently. For example, some of the tools generate modules for functions called inside the kernel, whereas others do inline them. Therefore, for the latter the result will be a big file containing the whole design. Table 9.4 summarizes the characteristics for all generated hardware by the different tools, in terms of lines of generated code, number of FSM states, number of generated modules, registers and files. The metrics are extracted thus by looking at the HDL files generated. No automatic post-processing was performed to extract the numbers. This implies that the presented numbers are approximations of the actual numbers. Nevertheless, the style of declaration of the modules and registers was checked to make sure results are comparable (e.g., we checked that only one line is used for each new port declaration). However, comments or empty lines were not removed. In other words, the data extraction process was manual, and some approximations were introduced in some cases, but overall, we ensured that the results are comparable.

The numbers given in Table 9.4 can be studied to gain more knowledge about the static features of the DWARV3.0 generated designs, i.e., understand if there is more room for reducing the FSM or the number of registers, but also to understand how these features relate to the actual implementation numbers given in later tables in terms of Lookup Table (LUT)s, Flip Flop (FF)s or Maximum

Table 9.4: Generated Accelerator Characteristics Showed as <#FSM : #registers> and <#lines:#components:#files> Tuples.

Kernel	Vivado HLS	CC	DWARV2	DWARV3	LegUp2	LegUp3	PandA0	PandA1
matrixmult	6 : 44 307:1:0	11:105 719:4:3	39:41 500:1:1	24:35 387:1:1	52:141 2621:1:0	39:100 2299:1:0	18:56 1042:12:0	21:36 1066:18:0
adpcm-encode	88:654 3125:30:31	23:666 2791:3:6	670:330 4026:1:0	217:585 6337:2:0	341:1404 19696:14:0	90:561 7826:1:0	275:398 8324:54:48	275:398 8324:54:48
aes-encrypt	144:1377 926:6:14	117:1213 5408:3:8	2879:2344 18080:3:0	1193:1018 17046:3:0	1018:2179 35029:8:0	509:1785 28208:12:0	959:1436 33930:47:8	754:1307 29694:52:5
aes-decrypt	143:1420 996:6:12	117:1139 5735:3:5	2895:2349 18167:3:0	1156:999 17657:3:0	1093:2287 36858:8:0	570:1876 29761:12:0	923:1380 32378:48:9	797:1298 30035:51:5
gsm	72:968 718:4:18	81:708 3813:2:6	966:707 6660:1:0	492:423 5951:1:0	400:2295 25216:15:0	304:1998 22514:19:0	631:1023 22772:63:1	666:1005 21873:71:4
sha	40:343 868:3:4	N/A N/A	364:648 3626:0:0	198:324 3804:0:0	371:1111 15023:9:0	284:1121 14666:9:0	164:324 7150:47:6	148:292 7066:55:1
cmultconj	18:30 544:4:9	5:29 298:2:3	50:34 532:3:0	37:25 422:3:0	16:94 1048:3:0	72:142 2309:3:0	30:47 1026:9:3	140:187 4982:37:10
satd	39:282 1240:1:1	27:155 849:2:3	93:228 1593:0:0	48:120 1084:0:0	50:725 6061:3:0	37:654 5401:3:0	144:319 6437:19;0	214:344 8546:45:8
sobel	19:107 631:2:2	14:98 733:2:3	133:116 1207:2:0	45:42 735:2:0	52:458 4682:3:0	40:457 4266:3:0	32:108 1952:24:0	188:294 7201:57:6
viterbi	20:204 1166:2:2	31:501 2371:2:8	185:199 1698:0:0	117:126 1881:1:0	N/A N/A	120:571 7111:3:0	98:223 4429:29:0	82:170 3994:39:3
bellmanford	13:43 455:0:0	20:111 845:2:3	92:62 818:0:0	50:43 686:0:0	44:168 2283:3:0	43:197 2646:3:0	47:77 1851:16:0	192:358 7164:40:4

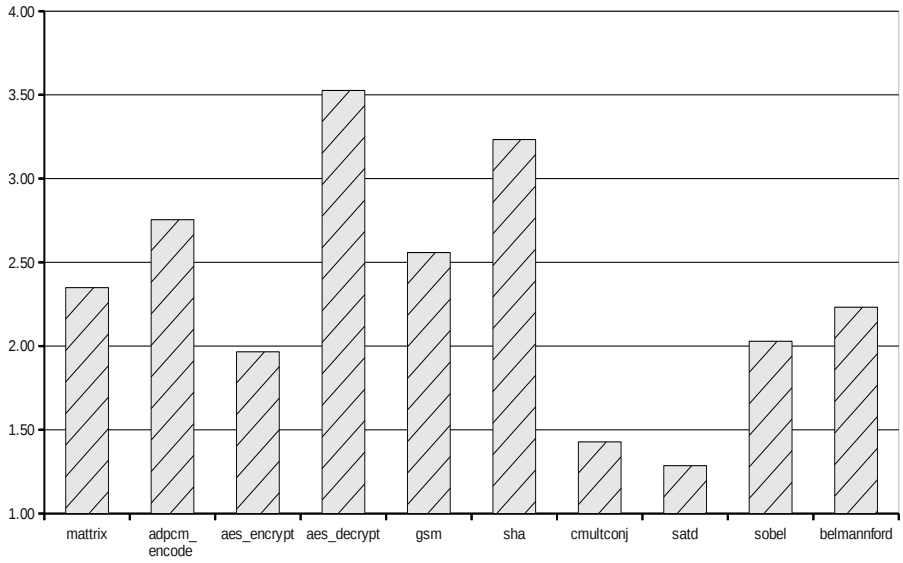


Figure 9.2: Execution Time Speedups of DWARV 3.0 compared to DWARV 2.0 .

Frequency (FMax). For example, we can see that, on average, the CC compiler (see below) gives the smallest numbers of FSM states, however, when we compute the execution time (i.e., #cycles / FMax) Vivado HLS generated hardware designs execute in less time (see Section 9.7). This implies that generating as few states as possible is not a criterion of success.

9.6 Experimental Results

We will now present the obtained performance results for the selected compilers. Besides DWARV2.0 and DWARV3.0, we investigate the compilers selected in section 9.3, that is Vivado HLS, LegUp and PandA. For the last two compilers, the latest two releases are investigated to track their evolution as well in a similar manner to the one performed for DWARV. The complete results, including area numbers for all compilers, are presented in Appendix A while, in this section, we show only the performance related metrics. Besides these compilers, we include a fifth tool, called here *CommercialCompiler (CC)* to hide the actual name of the compiler. We do this in order to avoid any license related issues regarding publishing these results. We note that for Figures 9.3 and 9.4, the smaller the compiler bar, the better its performance.

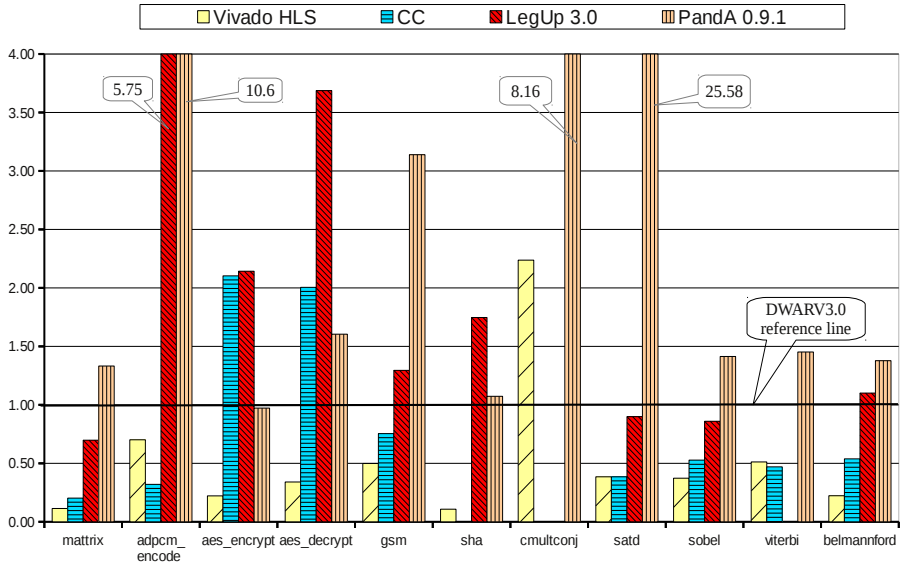


Figure 9.3: Execution Times Normalized to DWARV3.0 Execution Time.

Figure 9.2 illustrates the speedup of the last version of DWARV compared to the previous one, which was used in the evaluation of Chapter 4. The new version improved every test case from the benchmark with speedups between 1.25x and 3.51x. To keep the figures readable, we show in the subsequent figures only the comparisons for the last versions of the compilers. Results that show the evolution for the other two academic compilers is included in the tables presented in Appendix A.

Figure 9.3 shows normalized execution times with respect to the corresponding execution times obtained with DWARV3.0. The horizontal line at *1.00* denotes the baseline execution time, obtained with DWARV3.0 for that particular test case. It is important to note that the presented results are concerned only with one execution of the kernels. That is, they were executed only with the predefined input with which the kernel test bench came. No attempt has been made in trying to derive a different valid input which could trigger a different execution path in the kernel to obtain different performance metrics. Given the fact that the input was defined by a third party that created the application and its test bench, we considered this input as valid and meaningful to obtain the execution time metric. Figures 9.4 and 9.5 show the normalized cycle counts and the corresponding frequencies obtained. Please note that all numbers presented in this section are obtained after behavioral synthesis. No design was

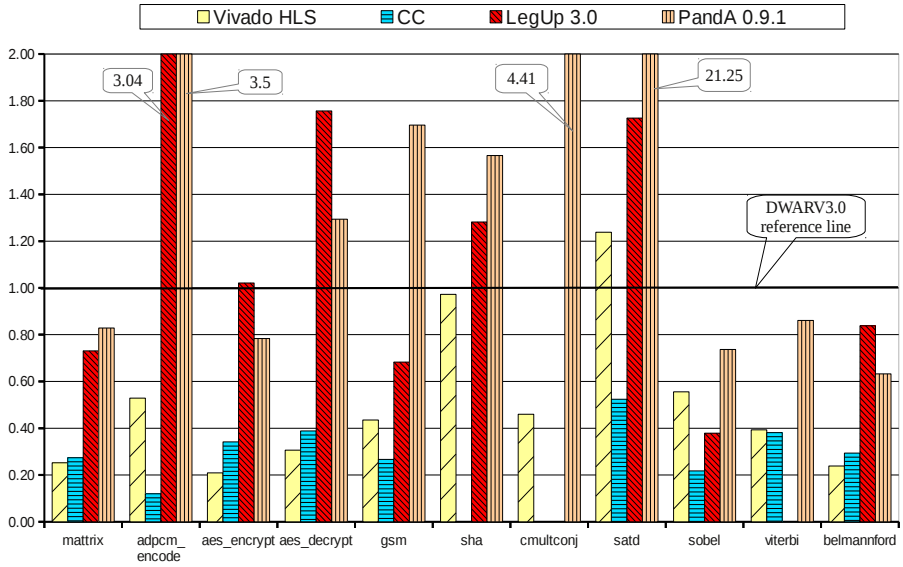


Figure 9.4: Execution Cycles Normalized to DWARV3.0 Cycles.

fully implemented, and as such the final numbers could be different. Referring to Chapter 4 where we performed both behavioral synthesis and implementation, we can estimate that the execution times could be potentially higher due to actual lower obtained frequencies and the area numbers could also be higher.

We observe in figure 9.3 that Vivado HLS generated the fastest executing hardware designs for the majority of kernels. *adpcm_encode* and *viterbi* are the only exceptions for which only CC was faster. The reason for this is the considerable smaller number of cycles generated by CC (Figure 9.4). Analyzing this figure in more detail, we observe that CC was able to generate more compact (i.e., smaller number of cycles) designs than Vivado HLS for the other three kernels. However, because the maximum obtainable frequency plays an equally important role, the execution time for these three kernels was smaller or equal for Vivado HLS (i.e., *gsm*, *satd* and *sobel*). This can be explained by the higher operation density in each cycle (i.e., many operations scheduled in a cycle) for CC generated kernels that led to a very big clock period (i.e., small frequency shown in Figure 9.5).

Comparing DWARV3.0 to the commercial compilers, we notice that, execution time wise, both Vivado HLS and CC generated hardware designs were faster than the Custom Computing Unit (CCU)s generated by DWARV3.0. The only exceptions are the *aes_encrypt||decrypt* designs generated by CC, which can be

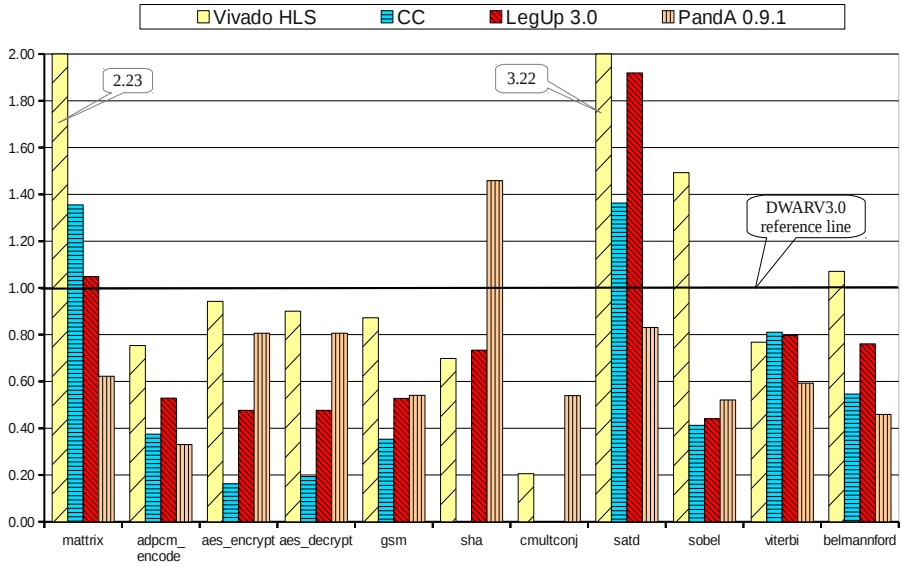


Figure 9.5: Estimated Max. Frequencies Normalized to DWARV3.0 Frequency.

explained by the fact that the frequency was 5x faster for DWARV3.0, whereas the number of cycles obtained with CC was only 2.5x smaller than that of DWARV3.0. This shows again that correctly balancing operations over a minimum number of cycles is key. This can be seen when comparing, for the same kernels, DWARV3.0 with Vivado HLS. The latter obtained even smaller cycle counts than CC, but because Vivado HLS was able to split the operations more efficiently over the small number of cycles generated, the frequency obtained was only marginally smaller than that of DWARV3.0. As a result, Vivado HLS execution times were 4.51x respectively 2.94x faster than those of DWARV3.0.

Finally, comparing DWARV3.0 to the academic compilers, we make two observations. First, DWARV3.0 generated designs were better than the ones generated by both LegUp 3.0 and Panda 0.9.1 with the exception of three kernels, i.e., *matrix*, *satd* and *sobel*. These three kernels were obtained with LegUp 3.0, and the speedup was due to the smaller number of cycles generated while the estimated frequency was comparable with the one of DWARV3.0. For the other kernels, DWARV3.0 is faster. Second, we observe that PandA 0.9.1 uses on average the same number of cycles for the generated designs as DWARV3.0, however, because the longer operations are not split in multiple cycles by using arithmetic multiplier cores, the design frequencies are all smaller than the ones of DWARV3.0.

Table 9.5: Execution Time Slowdowns compared to Vivado HLS.

F. Name	V.HLS	CC	D2.0	D3.0	L2.0	L3.0	P0.9.1
matrixmult	1	1.79	20.77	8.84	11.81	6.16	11.77
adpcm-encode	1	0.46	3.93	1.43	5.83	8.20	15.11
aes-encrypt	1	9.49	8.87	4.51	15.84	9.67	4.39
aes-decrypt	1	5.90	10.37	2.94	21.03	10.85	4.72
gsm	1	1.51	5.13	2.01	38.04	2.60	6.30
sha	1	N/A	N/A	9.28	20.09	16.22	9.96
cmultconj	1	N/A	0.64	0.45	1.78	N/A	3.65
satd	1	1.00	3.34	2.60	2.28	2.34	66.49
sobel	1	1.42	5.45	2.69	2.50	2.31	3.80
viterbi	1	0.92	N/A	1.95	N/A	N/A	2.84
bellmanford	1	2.41	10.02	4.49	6.82	4.94	6.18
AVERAGE	**1**	**3.11**	**7.92**	**3.74**	**12.60**	**7.03**	**12.29**
GEOMEAN	**1**	**1.96**	**5.53**	**2.81**	**8.10**	**5.66**	**7.24**

9.7 Conclusion

In this chapter, we compared a number of hardware compilers that comply with all the criteria defined in Section 9.2 for a hardware compiler evaluation in which DWARV can be included, as well. In particular, we looked at Vivado HLS, ComercialCompiler, LegUp2.0 and 3.0, PandA 0.9.0 and 0.9.1, and two versions of DWARV, i.e., 2.0 and 3.0. Table 9.5 shows the final execution time slowdowns when comparing to Vivado HLS, which on average generated the most efficient hardware. The final row shows the geometric means for the performance of all tools compared to Vivado HLS. We can see here that DWARV3.0 is **2.81x** slower than Vivado HLS, but also that a visible progress has been obtained since DWARV2.0 only by adding the optimizations described in Chapter 8. That is, DWARV3.0 is 1.96x faster than DWARV2.0. Finally, the performed experiments revealed that DWARV3.0 performed the best between all versions of all academically available compilers.

Note.

The content of this chapter was submitted as part of the following publication:

R. Nane, V.M. Sima, K.L.M. Bertels, **A Survey of High-Level Synthesis Tools and Comparison with DWARV 3.0**, Submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, April 2014.

10 Conclusions and Future Work

In this dissertation, we addressed different problems related to the process of automatic hardware generation for reconfigurable architectures. Concretely, we analyzed, designed, and implemented a hardware compiler in a commercial retargetable compiler framework. Subsequently, we studied two optimization problems concerning how area input constraints should be propagated in a High-Level Synthesis (HLS) tool as well as how the software if-conversion optimization can be applied in a hardware generation context. At the same time, we started to investigate the general problem of how existing software optimizations can be included in a hardware compiler. Furthermore, we investigated how and what is required to be generated, so that automatically generated hardware blocks can be integrated in complex (i.e., hardware/software co-designed) systems automatically. Finally, we performed a thorough evaluation of past and present HLS tools, while, at the same time, we benchmarked DWARV3.0 against both state-of-the-art commercial and academic compilers.

10.1 Summary

The different problems addressed in this work are split in chapters as follows:

In Chapter 2, we presented related work in which a vast number of past and present hardware compiler were described. These were categorized based on the design language in domain-specific respectively generic HLS tools. The particular tool description included information such as for what application domain the tool can be used, what extensions are required, if the tool offers verification support, as well as under what type of license it is available (commercial or academic). Finally, we commented for each tool how it is different from DWARV.

In Chapter 3, the underlying concepts and the compiler framework used throughout the work were presented. We described the Molen Machine Organization, the first version of the DWARV compiler which provided the inspiration for the current version, and we provided detailed information regarding the CoSy compiler framework used to implement the new version of DWARV. We also described the simulation and synthesis flows used to validate and implement automatically generated hardware designs. Finally, we discussed important similarities and differences between software and hardware compilers and we described the complete C-to-FPGA tool-flow based on a simple example.

Chapter 4 presented the first DWARV implementation in CoSy. The performance of this version was benchmarked by comparing and evaluating it against the LegUp 2.0 academic compiler. The results obtained showed that DWARV2.0 was outperforming LegUp 2.0.

In Chapter 5, we described the need for Hardware-dependent Software (HdS) when integrating automatically generated hardware blocks in complex System on Chip (SoC) systems. Furthermore, the use of HdS for hardware blocks led to the proposition of HdS related extensions to the IP-XACT standard, standard that facilitates the automatic integration of existing hardware components used by hardware designers for SoC design.

In Chapter 6, an optimization algorithm to generate hardware kernels subject to input area constraints was presented. These area constraints are highly important in the Molen context, where a maximum number of accelerators can be executed in parallel by a particular machine implementation. In this respect, generating hardware accelerators that can fit these previously defined FPGA slots is very important. However, the approach is generic enough to be applicable on different hardware platforms.

Next chapter, Chapter 7, presented another hardware specific optimization. This optimization, called Speculative and Predicative Algorithm (SaPA), is based on a relaxation of the traditional software if-conversion technique. The results obtained indicate that this optimization could be universally applied in each hardware compiler because it does not decrease the accelerator performance in unbalanced if-then-else cases, while, at the same time, the hardware area is negligibly increased.

In Chapter 8, we presented important hardware optimizations that allowed to optimize DWARV2.0 by a factor of 2x to 3x. Furthermore, we initiated work towards the automation of selecting and integrating optimizations in a compiler on a case by case basis. The reason behind this work is the fact that including existing standard optimizations randomly in a compiler is not a recipe for suc-

cess. The order in which these are applied and how they are configured play a very important role, as well.

Finally, Chapter 9, showed comparison results for DWARV3.0 against a newer version of LegUp (i.e., LegUp 3.0) and other three compilers, i.e., Vivado HLS, PandA 0.9.1 and another *CommercialCompiler*.

10.2 Dissertation Contributions

In recent years, we have seen an increase in the use of reconfigurable devices such as FPGAs, as well as an increase in the availability of commercial heterogeneous platforms containing both software processors and reconfigurable hardware. Examples include Convey HPC, Xilinx Zync and IBM Power7 with FPGA blades. However, as we argued in the introduction, programming these devices is still very challenging. As a result, the main goal of this dissertation was to advance the field of HLS. Concretely, we proposed methods to both integrate and optimize hardware compilers used for generating efficient hardware code that can be executed on such heterogeneous computers.

The specific contributions of this thesis are summarized as follows:

- We designed, implemented, and evaluated a new research compiler based on the CoSy commercial compiler framework. This new version of DWARV has a higher coverage of accepted C-language constructs. This is because the underlying compiler framework offers standard lowering (i.e., from high- to low-level constructs mapping) transformations, which essentially allows the developer to implement just the important primitives (e.g., goto) from which all high-level constructs are composed. Furthermore, using CoSy, we obtained a highly robust and modular compiler that can be integrated in different tool-chains by extending it with custom compiler transformations to process third party information (e.g., coming from aspect oriented descriptions) and configure the process of hardware generation accordingly. We validated and demonstrated the performance of the DWARV2.0 compiler against another state-of-the-art research compiler. We showed in this initial comparison kernel wise performance improvements up to 4.41x compared to LegUp 2.0 compiler (Chapter 4).

- We proposed IP-XACT extensions and showed that HdS should accompany hardware kernels to make them generally integrable into third party

tool(-chains). Therefore, we elaborated on the expressiveness of IP-XACT for describing HdS meta-data. Furthermore, we addressed the automation of HdS generation in the Reconfigurable Computing (RC) field, where Intellectual Property (IP)s and their associated HdS are generated on the fly, and, therefore, are not predefined. We combined in this respect two proven technologies used in MPSoC design, namely IP-XACT and HdS, to integrate automatically different architectural templates used in RC systems. We investigated and proposed IP-XACT extensions to allow this automatic generation and integration of HdS in RC tool-chains (Chapter 5).

- We proposed an optimization to control the unroll factor and the number of components when the area available for the kernel is limited. We assumed thus that the hardware area for which a to be generated hardware accelerator is limited. In this respect, two important parameters had to be explored namely, the degree of parallelism (i.e., the loop unrolling factor) and the number of functional modules (e.g., Floating-Point (FP) add operation) used to implement the source High-Level Language (HLL) code. Determining without any human intervention these parameters is a key factor in building efficient HLL-to-Hardware Description Language (HDL) compilers, and implicitly any Design Space Exploration (DSE) tools. To solve this problem, we proposed an optimization algorithm to compute the above parameters automatically. This optimization was added as an extension to the DWARV2.0 hardware compiler (Chapter 6).

- We proposed a predication scheme suitable and generally applicable for hardware compilers called SaPA. This technique takes into account the characteristics of a C-to-VHDL compiler and the features available on the target platform. Instruction predication is an already known compiler optimization technique, however, according to our knowledge and literature searches, current C-to-VHDL compilers do not take fully advantage of the possibilities offered by this optimization. More specifically, we proposed a method to increase the performance in the case of unbalanced if-then-else branches. These types of branches are problematic because, when the jump instructions are removed for the predicated execution if the shorter branch is taken, slowdowns occur because (useless) instructions from the longer branch still need to be executed. Based on both synthetic and real world applications we showed that our algorithm does not substantially increase the resource usage while the execution time is reduced in all cases for which it is applied (Chapter 7).

- We provided an evaluation of state-of-the-art hardware compilers against DWARV3.0. At the same time, a thorough retrospection of existing HLS tools has been performed. The comparison included a number of hardware compilers that comply with some predefined criteria in which DWARV can be included, as well. In particular, we looked at VivadoHLS, another ComercialCompiler, LegUp2.0 and 3.0, PandA 0.9.0 and 0.9.1, and two versions of DWARV, i.e., 2.0 and 3.0. The results obtained showed how all these compilers compare to Vivado HLS, which on average generated the most efficient hardware (Chapters 2, 8 and 9).

10.3 Future Work

We identify five major follow up research directions that we summarize below:

- The first and most important one is the continuation of the investigation related to how existing software optimizations can be applied to the hardware generation process. In this respect, questions such as what is the best place to insert an optimization in the compiler's flow of transformations, or how should a software optimization be customized and/or configured to generate *optimal* hardware, are very important. The importance of these questions, explained in Chapter 8, can be summarized briefly by stating that only including existing standard optimizations randomly in a compiler is not a recipe for success. The order in which these are applied and how they are configured are also important. Furthermore, given that there is still a considerable performance gap between the manually written hardware designs and those automatically generated, the search and invention of new specific hardware optimizations can play a major role also in the success and wide adoption of HLS tools. In this respect, the search of such new optimizations can be inspired by existing software optimizations, as it was the case with SaPA presented in Chapter 7.

- The second major research direction concerns the further investigation of the IP-XACT standard to support automatic generation of complete reconfigurable systems that embed automatically generated hardware blocks. That is, given that, for an individual core, we know how and what to generate to facilitate its integration in a SoC system (see Chapter 5), the question of what IP-XACT extensions are needed to allow the inference of system knowledge about interface and interconnect types

necessary to generate complete SoC systems automatically should be researched subsequently.

- Third, the model presented in Chapter 6 should be extended to allow dealing with variable loop bounds. In addition, more accurate prediction models for the wiring increase as well as for the power consumption are needed.

- Fourth, we intend to extend the comparison performed in Chapter 9 to create a state-of-the-art benchmarking reference result point to which all future compilers can relate. We will do this not only restricting it for the general domain, but also extending it to a broader scope of including particular domains and their corresponding audience tools. For example, one such domain would be the streaming one, where tools such as ROCCC could be included.

- Finally, a very interesting topic, which was not addressed in this dissertation, is to study the impact on performance of data partitioning and/or memory architecture generated by HLS tools. Topics, such as how many memory banks are sufficient for a particular hardware accelerator to satisfy the required performance throughput, or what type of accelerator interconnection can be used to communicate/transfer data efficiently in the system are examples of such system level design questions.

A

Complete DWARV 3.0 Comparison Results

In this Appendix, we present with the help of Tables A.1 to A.4 the complete set of results obtained in Chapter 9 for all compilers. Furthermore, we explain for each tool the difficulties faced in compiling particular examples or we comment on the correctness of the result.

Vivado HLS and CommercialCompiler (CC)

We start the presentation with the commercial compilers. The results obtained from (behaviorally) synthesizing the kernels obtained by compiling the C benchmark functions, with the clock period set to 10 ns, are shown in Table A.1. We note for these tools two benchmarks that we need to explain. First, the *cmultconj* caused problems for both commercial compilers. Because CC does not have floating-point capabilities, it could not compile this kernel at all. VIVADO HLS was able to compile it; however, the result obtained is the only one that is performing worse than the corresponding result obtained with DWARV3.0. A closer examination revealed that this is caused by the placement of the floating-point core in relation to the input register. This critical path with total delay of 16.096 ns is composed of 5.465 ns logic and 10.631 ns routing. Why Xilinx ISE's *xst* compiler was unable to place the input register closer to the floating-point core, is not clear. A possible cause could be an incompatibility between Vivado and ISE synthesis algorithms. That is, we suspect the problem could be caused by the fact that the hardware design obtained with Vivado HLS was synthesized using ISE's *xst* tool, which relies on older synthesizing algorithms. Finally, the *sha* function could not be compiled with CC because the tool gave a segmentation fault in this case.

Table A.1: Complete Performance and Area Metrics for Vivado HLS and CommercialCompiler tools.

Function Name	**Vivado HLS - xc7k325t-2-ffg900**					**CC - xc7k325t-2-ffg900**				
	Cycles	FMax	Regs	LUT	Exec. Time (us)	Cycles	FMax	Regs	LUT	Exec Time (us)
matrixmult	106	685	29	43	0.15	115	416	60	111	0.28
adpcm-encode	269	171	2804	3227	1.57	61	85	4908	6080	0.72
aes-encrypt	1660	180	18963	14500	9.22	2713	31	2928	4466	87.52
aes-decrypt	2434	172	18783	13941	14.15	3088	37	2581	4108	83.46
gsm	4288	205	2019	3853	20.92	2626	83	2762	5037	31.64
sha	206584	160	4055	3904	1291.15	N/A	N/A	N/A	N/A	N/A
cmultconj	17	62	511	4355	0.27	N/A	N/A	N/A	N/A	N/A
satd	104	399	970	921	0.26	44	169	755	1188	0.26
sobel	7461	427	569	570	17.47	2918	118	484	830	24.73
viterbi	13275	215	4655	4090	61.74	12874	227	1205	1708	56.71
belmannford	1598	394	373	603	4.06	1968	201	435	634	9.79

DWARV 2.0 and DWARV 3.0

Table A.2 shows the complete results for the last two versions of DWARV, without and with the modifications of Chapter 8. We notice two cases here, i.e. *sha* and *viterbi* functions that DWARV2.0 was able to compile, however, the simulation results did not match the golden reference obtained by running the function in software. As a result, we considered these hardware designs incorrect, and we didn't attempt to synthesize them.

LegUp 2.0 and LegUp 3.0

Table A.3 gives the results for 2.0 and 3.0 versions of LegUp. We see in the table that, for the older version, the *viterbi* function did not compile. This is denoted by the *ERR* entry in the table. We note that all the kernels compiled with LegUp 2.0 were synthesized with Quartus toolset 10.0 for the Stratix IV EP4SGX70HF35C2 FPGA. For LegUp 3.0, we used two FPGAs, one from Altera and one from Xilinx. The reason is because we wanted to keep the consistency in the comparison by comparing outputs of the same synthesizing compiler for the same target FPGA. However, because LegUp is designed for Altera, we could not synthesize all kernels for Xilinx. The ones requiring special cores, i.e. a *lpm_divide* Altera specific divider, were synthesized with Quartus 11.1sp for the Cyclone 2 EP2C35F672C6 FPGA. We did not choose for the Stratix version because LegUp 3.0 gives errors when we try to select it. Furthermore, with LegUp 3.0, two functions caused problems. *cmultconj* function could not be compiled due to missing floating-point cores and *viterbi* kernel did not give correct results in simulation.

PandA 0.9.0 and PandA 0.9.1

Finally, Table A.4 presents the results for PandA 0.9.0 and PandA 0.9.1 tool versions. We notice that the former version could not generate test benches for all the cases, although the tool was able to compile that particular function. This is the case for the *adpcm-encode*, *aes-encrypt||decrypt* and *sha* functions. As a result, we were not able to calculate performance numbers for them. However, the latest tool version is able to generate both kernels and their corresponding test benches correctly for all benchmark functions.

Table A.2: Complete Performance and Area Metrics for DWARV 2.0 and 3.0 tool versions.

Function Name	**DWARV 2.0 - xc5vfx130t-2-ff1738**					**DWARV 3.0 - xc7k325t-2-ffg900**				
	Cycles	FMax	Regs	LUT	Exec. Time (us)	Cycles	FMax	Regs	LUT	Exec Time (us)
matrixmult	678	211	651	475	3.21	420	307	391	291	1.37
adpcm-encode	1217	197	7621	5933	6.18	509	227	2766	3452	2.24
aes-encrypt	14151	173	31412	14619	81.80	7948	191	17582	10039	41.61
aes-decrypt	26278	179	30222	13274	146.80	7951	191	15871	8199	41.63
gsm	18778	175	14588	12230	107.30	9857	235	3728	6419	41.94
sha	ERR	ERR	ERR	ERR	ERR	212499	229	13722	33549	927.94
cmultconj	71	406	785	491	0.17	37	302	999	880	0.12
satd	243	279	3295	3105	0.87	84	124	713	1762	0.68
sobel	25055	263	1394	1186	95.27	13428	286	711	796	46.95
viterbi	ERR	ERR	ERR	ERR	ERR	33782	280	5782	8146	120.65
belmannford	11098	273	1224	1035	40.65	6701	368	716	623	18.21

Table A.3: Complete Performance and Area Metrics for LegUp 2.0 and 3.0 tool versions.

Function Name	**LegUp 2.0 - EP4SGX70HF35C2**					**LegUp 3.0 - xc7k325t-2-ffg900 ‖ *EP2C35F672C6***				
	Cycles	FMax	Regs	LUT	Exec. Time (us)	Cycles	FMax	Regs	LUT	Exe Time(us)
matrixmult	296	162	716	962	1.83	307	322	648	1179	0.95
adpcm-encode	697	76	4099	4439	9.17	1548	120	2702	3795	12.90
aes-encrypt	11105	76	13276	14356	146.12	*8114*	*91*	*9701*	*13307*	*89.16*
aes-decrypt	19937	67	15463	16633	297.57	*13967*	*91*	*11281*	*15557*	*153.48*
gsm	89921	113	6710	6609	795.76	6733	124	5846	8073	54.30
sha	303361	151	9081	7190	2009.01	*272428*	*168*	*9190*	*11281*	*1621.60*
cmultconj	75	154	395	397	0.49	N/A	N/A	N/A	N/A	N/A
satd	122	205	2265	2889	0.60	145	238	2124	3686	0.61
sobel	5117	117	2448	2756	43.74	5085	126	2217	2852	40.36
viterbi	ERR	ERR	ERR	ERR	ERR	N/A	223	2180	2917	N/A
belmannford	5590	202	692	951	27.67	5614	280	625	1143	20.05

Table A.4: Complete Performance and Area Metrics for PandA 0.9.0 and 0.9.1 tool versions.

Function Name	**PandA 0.9.0 - xc7k325t-2-ffg900**					**PandA 0.9.1 - xc7vx330t-1-ffg1157**				
	Cycles	FMax	Regs	LUT	Exec. Time (us)	Cycles	FMax	Regs	LUT	Exec Time (us)
matrixmult	267	162	389	803	1.65	348	191	391	515	1.82
adpcm-encode	ERR	75	3771	15260	ERR	1783	75	3771	15260	23.77
aes-encrypt	ERR	15	5583	35315	ERR	6229	154	8342	18887	40.45
aes-decrypt	ERR	15	5721	42743	ERR	10287	154	8515	21482	66.80
gsm	5236	114	3364	12868	45.93	16723	127	6484	9904	131.68
sha	ERR	17	1388	11108	ERR	332733	334	4261	6694	996.21
cmultconj	28	44	237	2962	0.64	163	163	1963	4157	1.00
satd	248	136	1223	5163	1.82	1785	103	2199	6519	17.33
sobel	7267	128	744	1765	56.77	9890	149	1843	4345	66.38
viterbi	26883	105	564	4186	256.03	29072	166	1478	2743	175.13
belmannford	5907	142	273	1461	41.60	4239	169	2089	4501	25.08

B

Return on Investment Graphs

In this Appendix, we illustrate in Figures B.1, B.2 and B.3 the corresponding return on investment (ROI) for the other three cases of the matrix example described in Chapter 6.

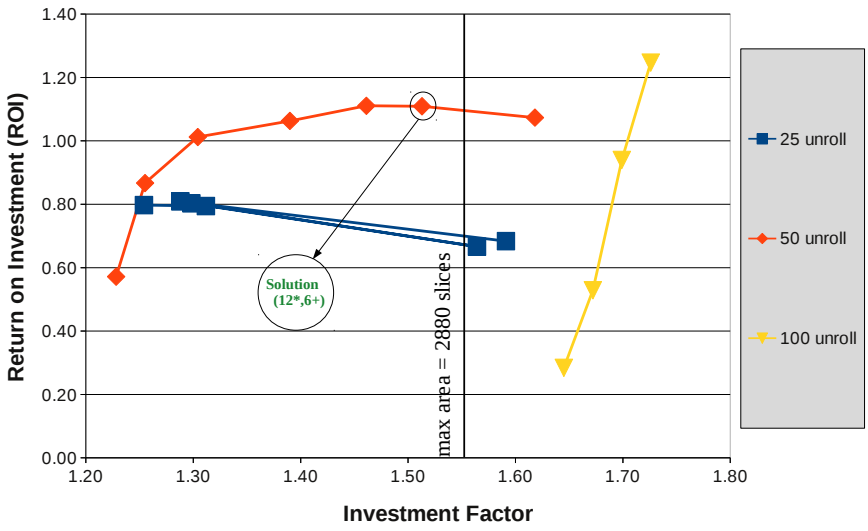


Figure B.1: Matrix multiplication ROI for 30% area design constraint.

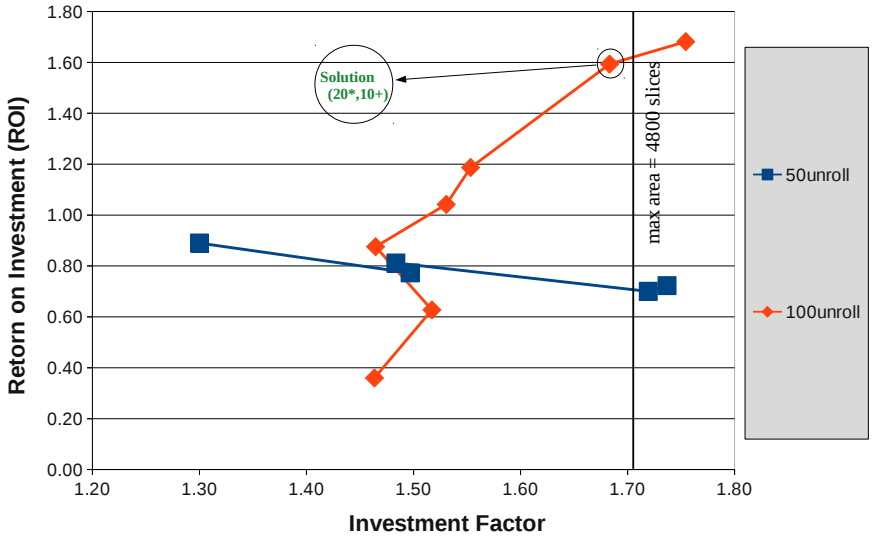


Figure B.2: Matrix multiplication ROI for 50% area design constraint.

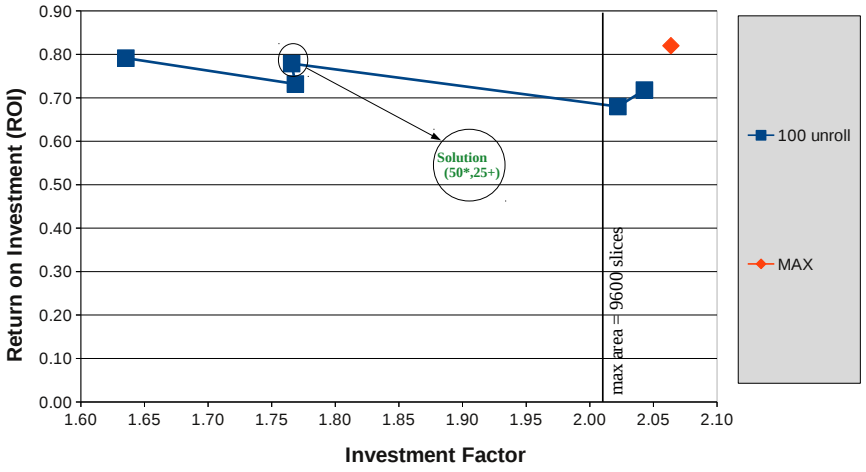


Figure B.3: Matrix multiplication ROI for 100% area design constraint.

Bibliography

[1] Ip-xact ieee 1685-2009 standard. [Online]. Available: http://www.accellera.org/home. 6, 72, 73

[2] Milepost project. [Online]. Available: http://ctuning.org/wiki/index.php?title=Dissemination:Projects:MILEPOST. 129

[3] Synphony c compiler. pico technology. [Online]. Available: http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx. 74

[4] N. Rollins A. Arnesen and M. Wirthlin. A multi-layered xml schema and design tool for reusing and integrating fpga ip. In *Field Programmable Logic and Applications*, FPL '09, pages 472 – 475. 73

[5] Shail Aditya and Vinod Kathail. Algorithmic synthesis using pico. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis: from Algorithm to Digital Circuit*, pages 53–74. Springer Netherlands, 2008. 30

[6] Altera. C2h compiler. [Online]. Available: http://www.alterawiki.com/wiki/C2H?GSA_pos=1&WT.oss_r=1&WT.oss=c2h. 29

[7] Altera. C2h compiler - discontinued. [Online]. Available: www.altera.com/literature/pcn/pdn1208.pdf. 30

[8] Altium. Altium designer: A unified solution. [Online]. Available: http://www.altium.com/en/products/altium-designer. 25, 61, 83, 104

[9] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A matlab compiler for distributed, heterogeneous, reconfigurable computing systems. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 39–48, 2000. 30

[10] BDTi. Bdti high-level synthesis tool certification program results. [Online]. Available: http://www.bdti.com/Resources/BenchmarkResults/HLSTCP. 30

[11] Y. Ben-Asher and N. Rotem. Synthesis for variable pipelined function units. In *System-on-Chip, 2008. SOC 2008. International Symposium on*, pages 1–4, 2008. 29

[12] Koen Bertels, Stamatis Vassiliadis, Elena Moscu Panainte, Yana Yankova, Carlo Galuzzi, Ricardo Chaves, and Georgi Kuzmanov. Developing applications for polymorphic processors: The delft workbench. Technical report, Delft University of Technology, 2006. 39

[13] BlueSpec. High-level synthesis tools. [Online]. Available: http://bluespec.com/high-level-synthesis-tools.html. 17

[14] Thomas Bollaert. Catapult synthesis: A practical introduction to interactive c synthesis. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis: from Algorithm to Digital Circuit*, pages 29–52. Springer Netherlands, 2008. 23, 61, 83

[15] Cadence. C-to-silicon compiler. [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx. 24, 83

[16] Cadence. Cadence c-to-silicon compiler delivers on the promise of high-level synthesis. Technical report, Cadence, 2008. 24, 61

[17] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The garp architecture and c compiler. *Computer*, 33(4):62–69, April 2000. 4, 20

[18] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz 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, pages 33–36, New York, NY, USA, 2011. ACM. 8, 27, 60, 61, 64, 65, 83, 104

[19] Convey Computer. The hc series. [Online]. Available: http://www.conveycomputer.com/products/hcseries/. 39

[20] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. Gaut: A high-level synthesis tool for dsp applications. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 147–169. Springer Netherlands, 2008. 28

[21] Universite de Bretagne-Sud. Gaut - high-level synthesis tool from c to rtl. [Online]. Available: http://hls-labsticc.univ-ubs.fr/. 28

[22] Politecnico di Milano. Bambu: A free framework for the high-level synthesis of complex applications. [Online]. Available: http://panda.dei.polimi.it/?page_id=31. 27

[23] J. Diamond, M. Burtscher, J.D. McCalpin, Byoung-Do Kim, S.W. Keckler, and J.C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 32–43, 2011. 2

[24] K. Bertels E. Panainte and S. Vassiliadis. Compiling for the molen programming paradigm. In *In Field-Programmable Logic and Applications (FPL)*, pages 900–910, 2003. 4, 40

[25] G. Estrin. Reconfigurable computer origins: the ucla fixed-plus-variable (f+v) structure computer. *Annals of the History of Computing, IEEE*, 24(4):3–9, 2002. 3

[26] ACE Associated Compiler Experts. Associated compiler experts ace: Cosy compiler platform. [Online]. Available: www.ace.nl. 12, 49, 60, 61, 83

[27] ACE Associated Compiler Experts. Engine writers guide. [CoSy Release]. 51

[28] Y Explorations. excite: C to rtl behavioral synthesis. [Online]. Available: http://www.yxi.com/products.php. 21, 65

[29] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010. 23

[30] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56, 2000. 18

[31] M.B. Gokhale and J.M. Stone. Napa c: compiling for a hybrid risc/fpga architecture. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 126–135, 1998. 20

[32] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, R. Reed Taylor, and R. Reed. Piperench: A reconfigurable architecture and compiler. *Computer*, 33:70–77, 2000. 17

[33] Mentor Graphics. Dk design suite: Handel-c to fpga for algorithm design. [Online]. Available: http://www.mentor.com/products/fpga/handel-c/dk-design-suite. 19

[34] Mentor Graphics. Handel-c synthesis methodology. [Online]. Available: http://www.mentor.com/products/fpga/handel-c. 19

[35] David Greaves and Satnam Singh. Kiwi: Synthesis of fpga circuits from parallel programs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2008. 33

[36] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. [Online]. Available: https://www.khronos.org/opencl/. 15

[37] D. Grune, K. van Reeuwijk, H. Bal, C. Jacobs, and K.G. Langendoen. *Modern Compiler Design (2nd edition)*. Springer, 2012. 45

[38] Zhi Guo, Betul Buyukkurt, and Walid Najjar. Optimized generation of data-path from c codes for fpgas. In *Int. ACM/IEEE Design, Automation and Test in Europe Conference (DATE 2005*, pages 112–117. IEEE Computer Society, 2005. 21

[39] Zhi Guo, Walid Najjar, and Betul Buyukkurt. Efficient hardware code generation for fpgas. *ACM Trans. Archit. Code Optim.*, 5(1):6:1–6:26, May 2008. 21

[40] Zhi Guo, Walid Najjar, and Frank Vahid. A quantitative analysis of the speedup factors of fpgas over processors. In *Processors, Int. Symp. Field-Programmable gate Arrays (FPGA*, pages 162–170. ACM Press, 2004. 82

[41] 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*, pages 461–466, 2003. 25, 61

[42] T. Stefanov H. Nikolov and E. Deprettere. Systematic and automated multi-processor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27, March 2008. 74

[43] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, H. Meyr, G. Bette, and B. Singh. Retargetable code optimization for predicated execution. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 1492–1497, 2008. 103

[44] Qijing Huang, Ruolong Lian, A. Canis, Jongsok Choi, R. Xi, S. Brown, and J. Anderson. The effect of compiler optimizations on high-level synthesis for fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 89–96, 2013. 134

[45] Giang Nguyen Thi Huong and Seon Wook Kim. Gcc2verilog compiler toolset for complete translation of c programming language into verilog hdl. *ETRI*, 33(5):731–740, October 2011. 32

[46] IBM. The cell project at ibm research. [Online]. Available: https://www.research.ibm.com/cell/. 3

[47] ISO / IEC. Tr 18037: Embedded c. [Online]. Available: www.open-std.org/jtc1/sc22/wg14/www/projects#18037. 121

[48] Jacquard Computing Inc. Roccc 2.0: Intelligent code accelerator solutions. [Online]. Available: http://www.jacquardcomputing.com/. 22, 83

[49] Xilinx Inc. Vivado design suite - vivadohls. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/index.htm. 26

[50] Accellera System Initiative. Systemc standard - ieee std. 1666™-2011. [Online]. Available: http://www.accellera.org/downloads/standards/systemc. 15

[51] Nikolaos Kavvadias. The hercules high-level synthesis tool. [Online]. Available: http://www.nkavvadias.com/hercules/. 17

[52] Nikolaos Kavvadias and Kostas Masselos. Automated synthesis of fsmd-based accelerators for hardware compilation. *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 0:157–160, 2012. 17

[53] Phillip Duncan John Granacki Mary Hall Rajeev Jain Heidi Ziegler Kiran Bondalapati, Pedro Diniz. Defacto: A design environment for adaptive computing technology. In *Reconfigurable Architectures Workshop (RAW)*, 1999. 32

[54] Wido Kruijtzer, Pieter van der Wolf, Erwin de Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge de Paoli, and Emmanuel Vaumorin. Industrial ip integration flows based

on ip-xact standards. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 32–37, New York, NY, USA, 2008. ACM. 74

[55] Los Alamos National Laboratory. Trident high level synthesis tool. [Online]. Available: http://trident.sourceforge.net/. 28

[56] Christopher K. Lennard, Victor Berman, Saverio Fazzari, Mark Indovina, Cary Ussery, Marino Strik, John Wilson, Olivier Florent, François Rémond, and Pierre Bricaud. Industrially proving the spirit consortium specifications for design chain integration. In *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*, DATE '06, pages 142–147, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 72

[57] R. Leupers. Exploiting conditional instructions in code generation for embedded vliw processors. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 105–109, 1999. 103

[58] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 103

[59] R. Meeuws, C. Galuzzi, and K. Bertels. High level quantitative hardware prediction modeling using statistical methods. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 140–149, 2011. 87

[60] R.J. Meeuws. *Quantitative hardware prediction modeling for hardware/-software co-design*. PhD thesis, TU Delft, 2012. xiii, 40, 42, 88

[61] W.A. Najjar, W. Bohm, B.A. Draper, J. Hammes, R. Rinker, J.R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8):63–69, 2003. 19

[62] NEC. Cyberworkbench: System lsi design environment to implement all-in-c concept. [Online]. Available: http://www.nec.com/en/global/prod/cwb/. 16

[63] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, 2004. 17

[64] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 574–579, New York, NY, USA, 2008. ACM. 74

[65] University of Cambridge. The tiger mips processor 2010. [Online]. Available: http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html. 27, 65

[66] University of Illinois at Urbana-Champaign. The llvm compiler infrastructure. [Online]. Available: http://llvm.org, 2008. 22, 27, 65, 131

[67] University of Pennsylvania. Eniac computer. [Online]. Available: http://www.seas.upenn.edu/about-seas/eniac/. 1

[68] University of Toronto. Legup high-level synthesis tool. [Online]. Available: http://legup.eecg.utoronto.ca/. 27

[69] OpenMP. The openmp api specification for parallel programming. [Online]. Available: http://openmp.org/wp/. 41

[70] Reflect Project. Rendering fpgas to multi-core embedded computing. [Online]. Available: http://www.reflect-project.eu/. 6, 49

[71] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers. Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 173–178, 2008. 31

[72] Nadav Rotem. C to verilog: automating circuit design. [Online]. Available: http://www.c-to-verilog.com/. 29

[73] G. N. Gaydadjiev K. Bertels G. Kuzmanov S. Vassiliadis, S. Wong and E. M. Panainte. The molen polymorphic processor. In *IEEE Transactions on Computers*, pages 1363–1375, 2004. 4, 13, 40, 61, 75, 83, 105

[74] V.-M. Sima, E.M. Panainte, and K. Bertels. Resource allocation algorithm and openmp extensions for parallel execution on a heterogeneous reconfigurable platform. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 651–654, 2008. 84

[75] Marcel Slotema. Integration of existing optimisation techniques with the dwarv c-to-vhdl compiler. [Online]. Available: http://repository.tudelft.nl/view/ir/uuidb5d4/, September 2012. 124, 131

[76] Marino Strik, Alain Gonier, and Paul Williams. Subsystem exchange in a concurrent design process environment. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 953–958, New York, NY, USA, 2008. ACM. 74

[77] Synopsys. Synphony c compiler. [Online]. Available: http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx. 30

[78] Calypto Design Systems. Catapult: Product family overview. [Online]. Available: http://calypto.com/en/products/catapult/overview. 23

[79] Forte Design Systems. Cynthesizer 5. [Online]. Available: http://www.forteds.com/products/cynthesizer.asp. 34

[80] CORE Technologies. Fpga logic cells comparison. [Online]. Available: www.1-core.com/library/digital/fpga-logic-cells/. 142

[81] Maxeler Technologies. Maxcompiler. [Online]. Available: https://www.maxeler.com/products/software/maxcompiler/. 33

[82] J.L. Tripp, M.B. Gokhale, and K.D. Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3):28–37, 2007. 28

[83] Justin L. Tripp, Preston A. Jackson, and Brad 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, pages 875–885, London, UK, UK, 2002. Springer-Verlag. 33

[84] San Diego Univ. Of California. Spark: A parallelizing aproach to the high-level synthesis of digital circuits. [Online]. Available: http://mesl.ucsd.edu/spark/. 25

[85] Carnegie Mellon University. Piperench: Carnegie mellon's reconfigurable computer project. [Online]. Available: http://www.ece.cmu.edu/research/piperench/. 17

[86] Harvard University. Machine-suif. [Online].Available: http://www.eecs.harvard.edu/hube/software/, 2004. 21

[87] Iowa State University. Atanasoff berry computer. [Online]. Available: http://jva.cs.iastate.edu/operation.php. 1

[88] Northwestern University. Match compiler. [Online]. Available: http://www.ece.northwestern.edu/cpdc/Match/. 30

[89] Standford University. Suif compiler system. [Online].Available: http://suif.stanford.edu, 2004. 18, 21, 47

[90] Sven van Haastregt and Bart Kienhuis. Automated synthesis of streaming c applications to process networks in hardware. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 890–893, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association. 74

[91] Stamatis Vassiliadis, Georgi Gaydadjiev, Koen Bertels, and Elena Moscu Panainte. The molen programming paradigm. In *in Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 1–10, 2003. 41

[92] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007:19–19, January 2007. 74

[93] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 127–134, Washington, DC, USA, 2010. IEEE Computer Society. 22, 60, 83

[94] 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.*, 19(12):1507–1522, November 2006. 16

[95] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov. Openfpga corelib core library interoperability effort. *Parallel Comput.*, 34:231–244, May 2008. 73

[96] Xilinx. Acceldsp synthesis tool. [Online]. Available: http://www.xilinx.com/tools/acceldsp.htm. 31

[97] Yana Yankova, Georgi Kuzmanov, Koen Bertels, Georgi Gaydadjiev, Yi Lu, and Stamatis Vassiliadis. Dwarv: Delftworkbench automate d reconfigurable vhdl generator. In *VHDL generator, the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, pages 697–701, 2007. 39, 47, 60, 61, 72

[98] Sungjoo Yoo, Mohamed wassim Youssef, Aimen Bouchhima, and Ahmed A. Jerraya. Multi-processor soc design methodology using a concept of two-layer hardware-dependent software. In *In Proceedings of Design Automation and Test in Europe, DATE'04*, 2004. 72

List of Publications

International Journals

1. R. Nane, V.M. Sima, K.L.M. Bertels, **A Survey of High-Level Synthesis Tools and Comparison with DWARV 3.0**, To be submitted to *IEEE TCAD* Transactions on Computer-Aided Design of Integrated Circuits and Systems, April 2014.

2. J.M.P. Cardoso, T Carvalho, J G de F. Coutinho, R Nobre, R. Nane, P. Diniz, Z. Petrov, W. Luk, K.L.M. Bertels, **Controlling a complete hardware synthesis toolchain with LARA aspects**, *MICPRO* Microprocessors and Microsystems, volume 37, issue 8, November 2013.

3. R.J. Meeuws, S.A. Ostadzadeh, C. Galuzzi, V.M. Sima, R. Nane, K.L.M. Bertels, **Quipu: A Statistical Modelling Approach for Predicting Hardware Resources**, *ACM TRETS* Transactions on Reconfigurable Technology and Systems, volume 6, issue 1, May 2013.

International Conferences

1. J G de F. Coutinho, J.M.P. Cardoso, T Carvalho, R Nobre, S Bhattacharya, P. Diniz, L Fitzpatrick, R. Nane, **Deriving resource efficient designs using the REFLECT aspect-oriented approach** (extended abstract), *9th International Symposium on Applied Reconfigurable Computing*, Los Angeles, USA, 25-27 March 2013.

2. R. Nane, V.M. Sima, K.L.M. Bertels, **A Lightweight Speculative and Predicative Scheme for Hardware Execution**, *International Conference on ReConFigurable Computing and FPGAs*, Cancun, Mexico, 5-7 December 2012.

3. R. Nane, V.M. Sima, K.L.M. Bertels, **Area Constraint Propagation in High-Level Synthesis**, *International Conference on Field-Programmable Technology*, Seoul, Korea, 10-12 December 2012.

4. R. Nane, V.M. Sima, B Olivier, R.J. Meeuws, Y.D. Yankova, K.L.M. Bertels, **DWARV 2.0: A CoSy-based C-to-VHDL Hardware Compiler**, *22nd International Conference on Field Programmable Logic and Applications*, Oslo, Norway, 29-31 August 2012.

5. R. Nane, S. van Haastregt, T.P. Stefanov, B. Kienhuis, V.M. Sima, K.L.M. Bertels, **IP-XACT Extensions for Reconfigurable Computing**, *22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, Santa Monica, USA, 11-14 September 2011.

6. J.M.P. Cardoso, R. Nane, P. Diniz, Z. Petrov, K Kratky, K.L.M. Bertels, M Hubner, F Goncalves, G. Coutinho, G Constantinides, B Olivier, W. Luk, J.A. Becker, G.K. Kuzmanov, **A New Approach to Control and Guide the Mapping of Computations to FPGAs**, *International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, 18-21 July 2011.

Book Chapters

1. R. Nobre, J.M.P. Cardoso, B. Olivier, R. Nane, L. Fitzpatrick, J.G.F. de Coutinho, J. van Someren, V.M. Sima, K. Bertels and P.C. Diniz, **Hardware/Software Compilation**, Chapter 5 in *Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach*, Springer, August 2013

2. J.M.P. Cardoso, P. Diniz, Z. Petrov, K. Bertels, M. Hübner, J. van Someren, F. Gonçalves, J.G.F. de Coutinho, G.A. Constantinides, B. Olivier, W. Luk, J. Becker, G. Kuzmanov, F. Thoma, L. Braun, M. Kühnle, R. Nane, V.M. Sima, K. Krátký, J.C. Alves, and J.C. Ferreira, **REFLECT: Rendering FPGAs to Multi-core Embedded Computing**, Chapter 11 in *Reconfigurable Computing - From FPGAs to Hardware/Software Codesign*, Springer, August 2011.

Local Conferences

1. R. Nane, V.M. Sima, K. Bertels, **DWARV 2.0 Support for Scheduling Custom IP Blocks**, *ICT Open - ProRisc 2012*, Rotterdam, The Netherlands, 22-23 October 2012.

2. R. Nane, V.M. Sima, J. van Someren, K.L.M. Bertels, **DWARV: A HDL Compiler with Support for Scheduling Custom IP Blocks**, *48th Design Automation Conference* Work-In-Progress poster session, 5-10 June 2011, San Diego, USA.

3. R. Nane, S. van Haastregt, T.P. Stefanov, B. Kienhuis, K.L.M. Bertels, **An HdS Meta-Model Case Study: Integrating Orthogonal Computation Models**, *Workshop DATE 2011 : Hardware Dependent Software (HdS) Solutions for SoC Design*, Grenoble, France, 18 March 2011.

4. R. Nane, K.L.M. Bertels, **A Composable and Integrable Hardware Compiler for Automated Heterogeneous HW/SW co-design Tool-Chains**, *6th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES 2010)*, Terrassa, Spain, 11-17 July 2010.

5. S. van Haastregt, R. Nane, B. Kienhuis, **HdS Generation**, *SoftSoc Workshop in Embedded Systems Week (ESWeek 2009 Workshop)*, Grenoble, France, 16 October 2009.

6. R. Nane, K.L.M. Bertels, **Scheduling in the Context of Automatic Hardware Generation**, *8th Architectures and Compilers for Embedded Systems Symposium (ACES 2008)*, Edegem, Belgium, 17-18 September 2008.

Samenvatting

Herconfigureerbare architecturen (RA Reconfigurable Architectures) zijn snel populairder geworden tijdens het laatste decennium. Dit heeft twee redenen. De eerste is vanwege de processor klokfrequenties die een omslagpunt hebben bereikt waarbij het energieverbruik een groot probleem wordt. Als gevolg hiervan moest er naar alternatieven worden gezocht om toch de prestaties te kunnen blijven verbeteren. De tweede reden is dat, omdat zij een substantiele toename hebben ondervonden wat betreft oppervlakte (dat wil zeggen het aantal transistoren per mm2), systeemontwerpers Field Programmable Gate Array (FPGA) apparaten konden gaan gebruiken voor een toenemend aantal (complexe) toepassingen. Echter, het in gebruik nemen van herconfigureerbare apparaten bracht een aantal gerelateerde problemen met zich mee, waarvan de complexiteit van het programmeren als een zeer belangrijke beschouwd kan worden. Eén manier om een FPGA te programmeren is door stukken code uit een hoge programmeertaal automatisch om te zetten naar een hardware taal (HDL). Dit wordt High Level Synthesis genoemd. De beschikbaarheid van krachtige HLS tools is cruciaal om met de steeds toenemende complexiteit van opkomende RA systemen om te gaan en zo hun enorme potentie voor hoge prestaties tot uiting te laten komen. Huidige hardware compilers zijn nog niet in staat om ontwerpen te genereren die qua prestaties vergelijkbaar zijn met handmatig gecreëerde ontwerpen. Om deze reden is onderzoek naar het efficiënt genereren van hardware modules van groot belang om dit verschil te verkleinen. In deze dissertatie richten we ons op het ontwerpen, integreren, en optimaliseren van de DWARV 3.0 HLS compiler.

Anders dan eerdere HSL compilers is DWARV 3.0 gebaseerd op het CoSy compiler framework. Hierdoor kunnen we een zeer modulaire en uitbreidbare compiler bouwen waar standaard- of specifieke optimalisaties gemakkelijk in kunnen worden geïntegreerd. De compiler is ontworpen om een grote subset van de C programmeertaal te accepteren als invoer en om VHDL code te genereren voor onbeperkte applicatiedomeinen. Om DWARV 3.0 externe tool-integratie mogelijk te maken stellen we verscheidene IP-XACT (een XML-gebaseerde standaard voor tool-interoperabiliteit) uitbreidingen voor zodat hardware-afhankelijke software automatisch gegenereerd en geïntegreerd kan worden. Verder stellen we twee nieuwe algoritmes voor die respectievelijk de prestaties optimaliseren voor verschillende invoer wat betreft oppervlakte beperking, en die toelaat de voordelen te exploiteren van zowel jump en predication schemes die op een meer efficiënte wijze kunnen worden uitgevoerd in

vergelijking met conventionele processoren.. Ten slotte hebben we een evaluatie gedaan in vergelijking met de meest geavanceerde HLS tools van dit moment. De resultaten laten zien dat wat betreft de uitvoeringstijd van verschillende toepassingen, DWARV 3.0 gemiddeld het best presteert van alle academische compilers.

Curriculum Vitae

Răzvan Nane was born on 17^{th} of August 1980 in Bucharest, Romania. He received his B.Sc. degree in Computer Science and his M.Sc degree in Computer Engineering both from Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science in 2005 and 2007, respectively. In 2008 he started work as a Technical Programmer in the Computer Engineering group, where he was involved in compiler support and extension activities. In 2011 he started his Doctoral Research at the Computer Engineering Laboratory, TU Delft, where he worked under the supervision of prof. dr. Koen Bertels. He was involved in multiple projects related to reconfigurable architectures such as Reflect, SoftSoc, hArtes and IFEST. The work focused on compiler technology for high-level synthesis. His current research interest include: high-level synthesis, compiler technology, reconfigurable architectures, hardware/software co-design and embedded systems.