The Molen Compiler for Reconfigurable Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op woensdag 20 juni 2007 om 10:00 uur

door

Elena MOSCU PANAINTE

inginer
Universitatea Politehnica Bucureşti
geboren te Adjud, Roemenie

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. S. Vassiliadis

Toegevoegd promotor:
Dr. K. Bertels

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. S. Vassiliadis, promotor	Technische Universiteit Delft
Prof. dr. K. Bertels	Technische Universiteit Delft
Prof. dr. K. Goossens	Technische Universiteit Delft
Prof. dr. R. Hartenstein	Technische Universität Kaiserslautern
Prof. dr. R. Leupers	RWTH Aachen University
Prof. dr. W. Najjar	University of California Riverside
Prof. dr. J. Cardoso	Instituto Superior Técnico Lisboa
Prof. dr. P.M. Sarro	Technische Universiteit Delft, reservelid

ISBN: 978-90-812020-1-5

Keywords: Compiler backend, Compiler optimization, Reconfigurable architecture

Cover: Reconfigurable Computing as a new chess game

Copyright © 2007 E. Moscu Panainte
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without permission of the author.
Printed in the Netherlands

In memoriam Prof. Stamatis Vassiliadis, Prof. Irina Athanasiu, Prof. Ionel Grigoras, Prof. Matei Stan

The Molen Compiler for Reconfigurable Architectures

Elena MOSCU PANAINTE

Abstract

In this dissertation, we present the Molen compiler framework that targets reconfigurable architectures under the Molen Programming Paradigm. More specifically, we introduce a set of compiler optimizations that address one of the main shortcomings of the reconfigurable architectures, namely the reconfiguration overhead. The proposed optimizations are based on data flow analyses at intraprocedural and interprocedural level and take into account the competition for reconfigurable hardware resources and the spatio-temporal mapping. The hardware configuration instructions are scheduled in advance of hardware execution instructions, in order to exploit the available parallelism between the hardware configuration phase and the sequential execution on the core processor. The intraprocedural optimization uses the min s-t cut graph algorithm to reduce the number of executed hardware configurations by identifying the redundant hardware configurations. We also introduce two allocation algorithms for the reconfigurable hardware resources that aim to minimize the total reconfigured area and to maximize the overall performance gain. Based on profiling results and software/hardware estimations, the compiler optimizations and allocation algorithms generate optimized code for the spatio-temporal constraints of the target reconfigurable architecture and input application. Additionally, they guide the selection of hardware/software execution of the operations candidate for reconfigurable hardware execution. In order to evaluate the Molen compiler, we first present an experiment with a multimedia benchmark application compiled by the Molen compiler and executed on the Molen polymorphic media processor with an overall speedup of 2.5 compared to the pure software execution. Subsequently, we estimate that the intraprocedural compiler optimization contributes to up to 94 % performance improvement compared to the pure software execution, while the intraprocedural compiler optimization and the allocation algorithms significantly reduce the number of executed reconfigurations for the considered benchmarks. Finally, we determine that the important performance impact of our compiler optimizations and allocation algorithms increases for the future faster FPGAs.

Acknowledgments

The research presented in this thesis is the result of my work in Computer Engineer group from TU Delft. The first thought of gratitude I have is for Prof. Stamatis Vassiliadis, who was and remains the patron of this group. He created an international working environment with students from all over the world. He will always be a reference model as a researcher and human being, who taught us to tackle our limitations and to enjoy any encountered problem as a provocation to solve it. I appreciate very much his advices and challenging discussions as well as his love for Samos and for good food. I am especially grateful for the countless contributions of my supervisor, Prof. Koen Bertels. During my Ph.D study, he helped me to achieve an academical thinking and improve my technical writing. I specifically thank him for his infinite patience and sense of humour, even when I entered the panic mode. I would like to express my sincere gratitude to Prof. Sorin Cotofana, who helped me find this special group and motivated me to start this research.

Many thanks go to my colleagues, Georgi Kuzmanov and Casper Lageweg for their support and encouragement they have always provided to me. A special contribution to this thesis was provided by my friend and colleague Iosif Antochi, who help me find my way at the beginning of my work. I also want to thank my colleagues Behnaz Pourebrahimi and Yana Yankova for their wonderful company and open discussions. My Romanian friend Andrei Rares, helped me to accommodate in the Netherlands and not to miss my country too much. He proved in many occasions that 'A friend in need is a friend indeed'.

Prof. Irina Athanasiu from Politehnica University of Bucharest is the professor that introduced me in compiler research and created the premises for me to study in the Netherlands. She dedicated her whole life for the future and education of her students. I also want to mention Prof. Ionel Grigoras, Stan Rogoz and Matei Stan, who transmitted to me their love for Mathematics and Physics and directed my steps towards Computer Science and Enginering. I consider myself privileged to have had such wonderful professors, which guided me

with parental love.

I am grateful to my parents and my sister, Nadina for being my best friends and unconditionally trusting and loving me. Their devotement and sacrifices cannot be compensated by anything. I hope they will share my joy when this thesis is completed. Last, but not least, I want to thank my husband and my son for being more than understanding when I was working till late hours and frequently, bringing my work problems at home. Meeting and having so many special persons around me makes me think that I am very lucky and I should become much better.

E. Moscu Panainte | Delft, The Netherlands, 2006

Contents

Abstract i

Acknowledgments iii

1 Introduction 1

1.1 Problem Overview and Dissertation Scope 2

1.2 Motivation, Open Questions and Terminology 3

1.3 Thesis Framework 6

2 Reconfigurable Architectures 9

2.1 FPGA Overview 10

2.2 Classification of Reconfigurable Architectures 11

2.3 Examples of Reconfigurable Architectures 12

2.4 The Molen Programming Paradigm 17

2.5 DelftWorkBench 23

2.6 Conclusion 25

3 The Molen Compiler 27

3.1 The Molen Compiler Framework 27

3.2 The Molen Polymorphic Processor 32

3.3 Molen PowerPC Compiler Backend 34

3.3.1 PowerPC Compiler Backend 34

3.3.2 PowerPC Backend Extensions for the Molen Prototype: 39

3.4 M-JPEG Case Study 41
3.5 Conclusions 45

4 Dynamic SET Instruction Scheduling **47**
4.1 Background and Related Work 48
4.1.1 Control Flow Graphs 48
4.1.2 Data Flow Analyses 51
4.1.3 Related Work 54
4.2 Motivation 55
4.3 Problem Statement 56
4.4 Instruction Scheduling Algorithm 59
4.4.1 Step 1: The Anticipation Subgraph 59
4.4.2 Step 2: Minimum s-t Cut 63
4.4.3 Step 3: Selection of Software/Hardware Execution . . 64
4.5 M-JPEG Case Study 64
4.6 Conclusions 67

5 Interprocedural SET Scheduling **69**
5.1 Motivational Example 70
5.2 Interprocedural SET Optimization 70
5.2.1 Step 1: Construction of the Call Graph 72
5.2.2 Step 2: Propagation of Hardware Configuration Instruction 73
5.2.3 Step 3: Placement of Hardware Configuration Instructions 75
5.3 A MultiMedia Based Evaluation 78
5.3.1 Scenario 1: MPEG 2 Profiling Results for Pure Software Execution 79
5.3.2 Scenario 2: A Simple Hardware Reconfiguration Scheduling 79
5.3.3 Scenario 3: Single Hardware Reconfiguration 82
5.3.4 Scenario 4: Interprocedural Optimization Results . . . 83

5.4 Conclusions 85

6 Compiler-driven FPGA-area Allocation **87**

6.1 Related Work 88

6.2 Problem Overview and Definition 89

6.2.1 Motivational Example 89

6.2.2 Problem statement 91

6.3 FPGA-area Allocation Algorithms 92

6.3.1 FIX/RW Algorithm 93

6.3.2 FIX/RW/SW Algorithm 95

6.4 Results 97

6.5 Conclusions 102

7 Conclusions **105**

7.1 Summary 105

7.2 Contributions 108

7.3 Future Research Directions 109

A Multimedia Design Space Exploration **111**

A.1 The MPEG2 and JPEG Case Study 112

Bibliography **123**

List of Publications **133**

Samenvatting **135**

Curriculum Vitae **137**

Chapter 1

Introduction

Reconfigurable Computing is a computing paradigm based on reconfigurable devices, which are hardware platforms whose functionality and interconnections can be metamorphosed under software control. As a general approach, the computing machines under this paradigm include a General Purpose Processor (GPP) - which provides good performance for a large range of applications - extended with reconfigurable devices - usually a Field-Programmable Gate Array (FPGA) which achieves high performance for application-specific computations. Such hybrid system - denoted as Field-programmable Custom Computing Machine (FCCM) - combines the advantages of the two components: the flexibility of the GPP and performance of the FPGA and provides additional advantages. The hardware flexibility of the reconfigurable devices allow rapid modifications of existing platforms for the continuously changing standards and functional requirements; thus, the time-to-market delay and the prototyping costs are significantly decreased. Due to these features, Reconfigurable Computing is considered a viable solution for the increasing complexity of the current applications and hard requirements imposed for the computation machines.

Although a large number of approaches for Reconfigurable Computing have been proposed in the last decade, the success of this computing paradigm is conditioned and currently limited by the design tools that should transparently exploit the underling reconfigurable machine from the high-level programming application. More specifically, the current state-of-art tools assume the developers have deep understanding of both hardware and software designs and it is their responsibility to fully exploit the benefits of this approach.

In this thesis, we focus on the Molen Compiler backend which addresses a key

component of the design tools that should be adapted for the target FCCM. The presented compiler aims not only to generate code for the target machine, but mainly to apply advanced optimizations that transparently take into account the specific features of the target FCCM.

In this chapter, we present the general problem overview and clearly define the dissertation scope in Section 1.1. Next, we focus on the major open questions that should be answered in the rest of the thesis and define the used terminology. In Section 1.3, we present the organization of this thesis and a brief overview for each chapter.

1.1 Problem Overview and Dissertation Scope

In the last decade, the research in reconfigurable computing leverages the development of new reconfigurable devices, architectures, CAD tools, and methodologies as well as compilation software, hardware-software partitioning and programming paradigms, in an effort to support the ever-increasing demands of a wide range of target applications. These main research topics are covered by two projects which are related to this thesis, namely MOLEN (for the first category related to hardware organization) and Delft WorkBench (for the second category related to the software support).

In this thesis, we address the compilation software area, which aims to generate high-quality binary code for the target reconfigurable architecture. More specifically, the requirements and initial constraints of the proposed research can be summarized as follows:

- Develop compiler extensions in the context of the Molen Programming Paradigm (explained in the next section) for reconfigurable architectures in general, and for the MOLEN Polymorphic processor in particular.
- Investigate which are the main advantages and drawbacks of the target reconfigurable architecture that can be exposed to and positively exploited by the compilation framework.
- Propose compiler optimizations and scheduling algorithms that address the previously mentioned specific features of the target reconfigurable architectures
- Quantify the impact of the proposed algorithms on the overall performance for applications in multimedia domain.

In addition to the initial requirements, we restrict the scope of this dissertation as follows:

- We are concerned with software compilers, which generate assembly/binary code for the target reconfigurable architectures; we do not address hardware compilers which generate the synthesisable code that should be performed on the reconfigurable device.
- The target applications for the compilation software are limited to the multimedia benchmarks, as it is proven (see next section) that the target reconfigurable architecture is appropriate for this application domain.
- The target FCCM is the MOLEN Polymorphic processor (see next section)
- The compiler should follow the Molen Programming Paradigm which is intended (currently) for single program execution. In consequence, we do not address problems specifically for Real-Time Operating systems (RTOS) such as multi-threading, multi process management. Additionally, the parallel execution of tasks on the FPGA represents a separate research direction in Delft WorkBench project and is not the focus of this thesis.
- We do not compare the RC paradigm to other approaches for multimedia applications boosting performance (such as MMX, 3DNow!, SSE) which use dedicated non FPGA related hardware. The focus of this thesis is the compiler support for the Molen Molen Polymorphic processor under the Molen Programming Paradigm.

1.2 Motivation, Open Questions and Terminology

As previously explained, the main idea of the reconfigurable computing paradigm is the use of dynamically configured hardware for implementing new functionalities on a per-application basis. At the Instruction Set Architecture level, the common approach for supporting new functionalities is to add a new instruction for each new functionality executed on the reconfigurable hardware. However, taking into account the common limitation in the number of the unused opcodes and in the instruction encoding, this approach imposes severe restrictions on the type and number of newly added functionalities, while it also requires in-depth hardware modifications of the core processors (GPPs),

at least in the decoding stage - a detailed discussion is included in the next chapter.

A general approach which eliminates the above mentioned shortcomings is offered by the Molen Programming Paradigm and Molen $\rho\mu$-coded processor which require only a small number (see next chapter) of new instructions for a virtual infinite number of new functionalities. In the Molen machine organization, the functionalities are emulated on the reconfigurable hardware using an extended microcode - referred to as reconfigurable microcode. Thus, a generic instruction can cover a large number of functionalities, as long as it addresses their associated reconfigurable microcode.

In order to use the promising features of the RC paradigm from the application level - which, due to the increasing complexity, are developed using usually high-level programming languages (such as C, C++), advanced software tools are required for guiding/supporting the design process, including hardware-software partitioning, compilation and resource management. Among the required software tools, the compiler is a key element, as it can provide information and transformations which are useful for all involved tools. Using modern compiler techniques, the compiler can extract detailed and specific information about the static/dynamic behavior of the target application. Additionally, the compiler is the critical component where the hardware features of the target architecture should be reflected in the generated code. Thus, the compiler addresses both software and hardware features of the target application and architecture and it can/should have a major influence in the whole design process. This observation is particularly suitable for the RC paradigm, where the hardware features of the reconfigurable devices differ significantly from those of GPP and offer interesting, new opportunities for the application and improvement of standard compiler transformations.

Based on these considerations, we formulate four major open questions which are addressed in this thesis, as follows:

1. What are the minimal compiler modifications to transparently generate code for RC under the Molen Programming Paradigm ?

   We investigate the minimal compiler extensions and their practical integration in an existing compiler infrastructure to accommodate to the minimal requirements of the Molen Programming Paradigm for RC. Additionally, we implement a compiler backend specially adjusted for the Molen Polymorphic processor. Once the basic compiler support is provided, specific transformations and optimizations for RC are required, as posed by the following questions.

2. What are the main advantages and drawbacks of RC that are important for the compiler?

   To answer to this question, we analyze the dynamic behavior of a set of multimedia benchmarks in the context of RC and study the advantages and disadvantages offered by the usage of the reconfigurable hardware. As shown later on, we also estimate the impact of the identified features over the overall performance and determine the domains for the target reconfigurable architecture to outperform the GPP alone.

3. What compiler optimizations and instruction scheduling algorithms are appropriate for RC?

   Based on the features identified in the answer of the previous question, we research for a set of advanced compiler optimizations that capitalize the advantages and eliminate/reduce the disadvantages of the target reconfigurable architecture. We also estimate the impact of the proposed transformation on the overall performance of the Molen Polymorphic processor.

4. Can the compiler efficiently guide/manage the allocation of the FPGA resources?

   The resource management in general and of FPGA resources in particular can be handled by both compilers and RTOS. We investigate the compiler's opportunities for guiding the FPGA resource allocation, based on the characteristics of the target applications. Our approach addresses the development of efficient allocation algorithms and the study of their impact on the overall performance. As stated before, we do not address operating systems in this respect.

Terminology: In computer engineering discipline, the term of *computer architecture* (or simply architecture) refers to the conceptual design and fundamental operational structure of a computer system. Basically, it consists of the machine attributes - such as instruction set, operand width and register file - that are exposed to the machine-language programmer of the specific computer.

Reconfigurable hardware is a hardware device that can be modified after fabrication time through user-defined programming both at functional level and at the interconnection level. Accordingly, a reconfigurable architecture is a computer architecture that incorporates reconfigurable hardware. For this thesis, we examine reconfigurable architectures that allow both partial and dynamic reconfigurations.

By *partial reconfiguration*, we refer to the the ability to reconfigure only the part of the device that implements a specific functionality, while leaving unchanged the rest of the device. *Dynamic/run-time reconfiguration* addresses the capability to reconfigure at execution time a part of the reconfigurable device, while the rest of the device is fully operational. Thus, in this thesis, we address reconfigurable architectures that are not used only for fast prototyping, although currently this is one of their main usage.

The complex operations extracted from one application that are implemented and executed on the reconfigurable hardware are addressed in the rest of the thesis as reconfigurable/hardware operations/kernels.

1.3 Thesis Framework

This section presents the organization of the remainder of this dissertation which consists of the following chapters:

- In Chapter 2, we discuss the common approaches for reconfigurable architectures together with the compilation flows and programming paradigms. We proceed by indicating a number of shortcoming of the existing approaches regarding the permitted ISA extensions for the new functionalities performed on the reconfigurable hardware. Next, we present in details the target Molen machine organization and its implementation on the Virtex II FPGA platform denoted as the Molen Polymorphic processor. For programming such hybrid architecture, we present the Molen Programming Paradigm that, although it is particularly suitable for the Molen machine organization, it is a general programming paradigm that can be used for a large range of reconfigurable architectures. Finally, we emphasize the differences that allow for the Molen machine organization and programming paradigm to eliminate/reduce the above mentioned shortcomings of other existing approaches.

- In Chapter 3, the Molen compiler backend we have implemented for the Molen Polymorphic processor is discussed in details. Additionally, a complete experiment of a real multimedia application compiled for and executed on the Molen Polymorphic processor is presented as a proof of concept and we will show that the expected performance improvements can be achieved but also that additional compiler optimizations are required to fully exploit the target reconfigurable architecture.

- In Chapter 4, we introduce a compiler optimization that is based on the anticipation of the hardware configuration instructions at the intraprocedural level. The optimization uses data-flow analyses to determine the anticipation space for each hardware configuration instructions and a min s-t cut algorithm is applied in order to compute the optimal placement of the hardware configuration instructions. The impact on performance of this optimization is estimated for real multimedia applications and current FPGAs.

- In Chapter 5, we investigate the impact of the reconfiguration overhead on the overall performance and propose an interprocedural compiler optimization to reduce its negative influence. To this purpose, the instructions for hardware reconfiguration are anticipated as soon as possible before the associated hardware execution instructions and redundant reconfigurations are eliminated. The optimization also takes into account the limited reconfigurable hardware resources.

- In Chapter 6, we propose two efficient FPGA area allocation algorithms which are based on profiling information regarding the reconfiguration frequency. The allocation problems are translated in ILP problems with two different objective functions: minimal reconfiguration overhead and maximal performance improvement, in the context of hw/sw partitioning problem.

- In Chapter 7, we present the main conclusions of this thesis emphasizing on the main contributions of the presented research. Finally, we propose future work directions for the compiler research as well as for the Delft-WorkBench project.

Chapter 2

Reconfigurable Architectures

Due to the increased demand of computation power and flexibility, Reconfigurable Computing has been a major research domain in the last decade. However, existing approaches have several important shortcomings and there is a lack of dedicated tools to assist the design process in all its stages. The Molen machine organization and Programming Paradigm address and solve the considered problems while the tools involved in DelftWorkbench project support the designer targeting reconfigurable architectures under the Molen Programing paradigm.

In this chapter, we briefly present the background information and related work regarding reconfigurable architectures. After a short discusson of the physical implementation of the reconfigurable hardware, we propose a set of classification criteria for reconfigurable architectures. In the following section, we present a set of relevant reconfigurable architectures and a discussion on their main problems. In Section 2.4, we describe the Molen machine organization and Programming paradigm and emphasize the architectural features that address the previous problems. In the next section, we introduce the DelftWorkBench project that aims to provide a semi-automatic tool platform for hw/sw co-design and partitioning of applications executed on reconfigurable architectures under the Molen Programming Paradigm. Finally, the chapter is concluded with Section 2.6.

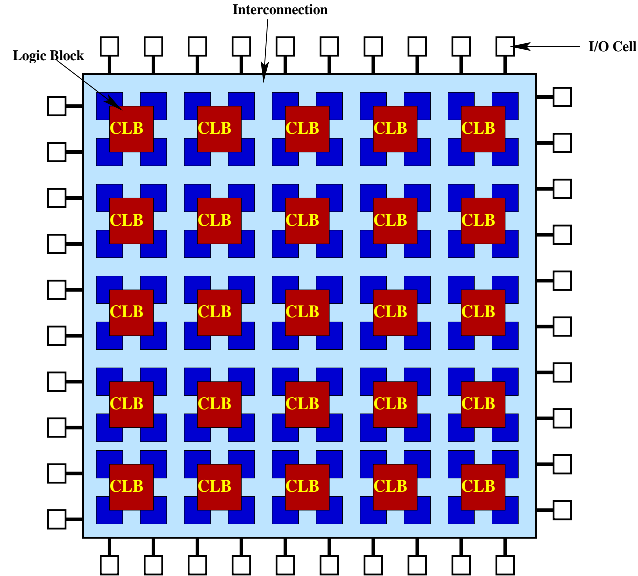

Figure 2.1: Internal structure of an FPGA

2.1 FPGA Overview

The concept of reconfigurable hardware was proposed for several decades, but only the recent advances in technology made it a successfull alternative to dedicated hardware. As reflected by the name, its main strenght resides in the promising combination of the flexibility provided the reconfigurative feature of the hardware and the performance of hardware execution. Several approaches exists for such devices, varying from the first small PLDs (Programmable Logic Devices) usefull to implement small boolean logic equations to the most recent FPGAs needed for register-heavy and pipelined applications. In the rest of this thesis, when reconfigurable hardware is addressed, we consider that FPGA devices are referred to.

An FPGA consists of an array of uncommitted processing elements which can be interconnected in a general way and this interconnection is user-programmable. The typical structure of an FPGA is depicted in Figure 2.1. The main components are the two-dimensional array of logic blocks, the interconnections and the I/O cells.

The logic blocks within an FPGA can be as small and simple as the macrocells in a PLD (a so-called fine-grained architecture) or larger and more complex

(coarse-grained). In most FPGAs, these programmable logic components also include memory elements, which can vary from simple flip-flops to more complete blocks of memories. A typical FPGA logic block consists of a 4-input lookup table (LUT), and a flip-flop.

The interconnection resources contain the segments of wire of different lengths together with the programmable switches that serve to connect the logic blocks to the wires, or the wires themselves. A logic circuit is implemented in the FPGA by decomposing it in individual logic blocks and then connecting the logic blocks via the switches as required in the initial design.

2.2 Classification of Reconfigurable Architectures

In many projects, FPGAs are used just for rapid prototyping. This is not the focus of our research as real multimedia applications cannot fit entirely on current FPGAs and a set of operations (e.g. I/O operations) are not proper for FPGA execution. Instead, we address the combination of an FPGA and a GPP. The main issue is to accelerate the computation intensive tasks using the FPGA while preserving the I/O operations and control dominated tasks on the GPP.

A large number of approaches have been also proposed for such hybrid reconfigurable architectures (see [1] for a complete classification). We can classify them through the following criteria:

- *Configuration granularity:* The granularity of the reconfigurable hardware is defined as the size of the smallest functional unit (CLB) that is addressed by the mapping tools.
  - *Fine-grained* architectures work at the bit manipulation level. Such architectures offer the maximum level of flexibility at the cost of increased area, power and delay requirements due to greater quantity of routing per computation. For such architectures, the reconfiguration overhead has a major influence on performance.
  - *Coarse-grained architectures* [2] [3] [4] [5] perform reconfiguration at processing element level and they are suitable for standard data path applications.
- *Host coupling*: Another important architectural issue is the type of connection between the GPP and FPGAs [6] [7] [8] [9]. One approach is to *tightly* integrate the FPGA as a functional unit of the GPP. In this case, the operations executed on the reconfigurable hardware have a limited

number of input/output operands and they resemble simple GPP instructions. The other approach is to *loosely* connect the FPGA as a coprocessor of the GPP. For such architectures, complex computations can be performed on the FPGA which usually is allowed to access the main memory. More performance improvements are expected for this second category, but the reconfiguration overhead must be taken into account.

- *Explicit reconfiguration*: As previously mentioned, the reconfiguration overhead is an important issue for the reconfigurable architectures, where even for modern FPGAs, a complete configuration takes several milliseconds. In order the reduce the reconfiguration latency, several architectures (see [10] [11] [12] [9] [13] [14]) provide a special instructions for hardware configuration (SET instruction). However, some architectures (see [15] [16] [7] [17] [18]) do not provide such an instructions, either because the reconfiguration overhead is negligible or this issue is not taken into account.

2.3 Examples of Reconfigurable Architectures

In the following, we shortly present a set of representative related reconfigurable architectures emphasizing on the criteria we presented in the previous sections. First, the target architecture is described, followed by the programming model and toolchain, and finally we focus on the compiler related issues regarding code generation and special optimizations for the reconfiguration overhead.

Napa[19][20]
Sarnoff Corporation

One of the early compilers for configurable hardware is Napa C. The target architecture NAPA1000 combines an embedded 32-bit RISC processor with a configurable logic with a 64 x 96 Adaptive Logic array, which is partially and dynamically reconfigurable. Additionally, there are two 32 bits x 2K on-chip memory bank and eight 8 bits x 256 scratchpad memories.

For programming such hybrid architecture, the programmer is provided with a set of pragma directives where he/she can specify the location (external memory, local memory or scratchpad) and the size of a variable, as well as the execution engine of a subroutine or statement (RISC processor or configurable logic). Additional pragmas for concurrency and I/O operations are not yet implemented in the NAPA compiler. It is suggested that the hardware/software

partitioning could be made by an automatic system; however, the programmer has to deeply understand the target architecture and applications in order to perform an efficient mapping.

The NAPA C compiler is based on the SUIF compiler infrastructure. After the identification of the segments of code selected for execution on the configurable logic, the remaining code is unparsed to C and processed by the RISC processor's compiler. Thus, the quality of the code can be seriously decreased, while the opportunity for applying specific optimizations for the configurable logic is mainly lost. Regarding the configuration latency, it is not clear from the available documentation whether there is a special instruction for such purpose. Instead, most of the compiler optimizations address the synthesis of hardware pipelines from pipelineable loops.

Garp[21][22]
University of California, Berkeley

The Garp architecture integrates a single-issue MIPS processor with a reconfigurable array connected as a coprocessor. The reconfigurable hardware has access to the same memories and caches as the MIPS processor. It is mentioned that the GARP chip does not exist as real silicon; circuit simulations are used to estimate the clock speed, power consumption and silicon area.

One main advantage of the GARP compiler is the fact that it does not require the programmer to insert any hints or directives in the source code (standard ANSI C). The compiler automatically identifies the kernels that should be accelerated using profiling and execution time estimates. One main constraint for the considered applications is the size of the reconfigurable array.

The GARP compiler is also based on the SUIF compiler infrastructure. Similar to the NAPA approach, the code for the core processor is also unparsed back to C, with the previously mentioned drawbacks. One important advantage of the GARP compiler is that it can extract whole loops to be executed on the reconfigurable hardware. However, for the considered applications, the loops are large and they do not fit entirely on the available reconfigurable array. For such cases, only the frequently executed paths of the loops are grouped in hyperblocks and executed on the reconfigurable hardware. Regarding the configuration overhead, there is a special instruction for loading a new configuration and there is hardware support to avoid loading a configuration when it is already available. However, the authors assume that reconfigurable hardware is "rapidly" reconfigurable in few cycles, in order to be efficient for short-running loops. Additionally, configurations can be loaded only when the reconfigurable hardware is idle.

Chimaera[23][24]
Northwestern University

The Chimaera micro-architecture is a complementary approach to NAPA and GARP for coupling the reconfigurable hardware to the core processor. In this approach, the reconfigurable hardware is integrated as a new functional unit (Reconfigurable Functional Unit RFU) in the host processor. Such tightly coupling allows faster communication with the host processor as it is interfaced only with a set of registers, but the RFU is limited in accessing the memory and performing control flow operations. An important consequence of such architectural design is that the operations executed on the RFU usually replace only a set of several (up to 10) instructions on the host processor, while in NAPA and GARP approaches whole loops could be executed on the reconfigurable hardware.

The Chimaera compiler does not require the programmer for indications about the operations for reconfigurable hardware. Instead, the compiler automatically combines sets of instructions of the host processor that have maximum 9 inputs and only one output into a new instruction for the reconfigurable hardware.

The Chimaera compiler is based on the GCC compiler version 2.6.3. New compiler optimizations have been added in order to automatically identify the best patterns for the reconfigurable hardware executions. These optimizations - such as Control Localization, SWAR - aim to eliminate the branch instructions and to increase the basic block boundaries in order to better exploit ILP and medium-grain data parallelism. Regarding the reconfiguration overhead, the reported results are based on simulations using different timing models. Moreover, the execution stalls for the duration of configuration loading.

PipeRench[15][25]
Carnegie Mellon University

The Piperench architecture uses the reconfigurable hardware in a different manner from the above mentioned approaches. The main idea behind it is the pipelined reconfigurations, when large pipelined computations are executed on a small piece of reconfigurable hardware by loading the configuration of each stage of the pipeline in one or few cycles. This approach allows the execution of computations that do not fit entirely on the reconfigurable hardware. However, this virtualization of the configurations imposes some additional constraints of the computations that can be executed on such type of hardware. For example, the operations executed in one stage of the pipeline can be dependent only of the operations in the current or previous stage.

The PipeRench compiler is focused on the generation of the hardware configurations for the considered isolated kernels. It automatically synthesizes, places, and routes the design for each kernel, while hiding from the programmer all notions of hardware resources, timing and physical layout. Nevertheless, the programmer is allowed - if wanted - to give additional hints about bit width of variables.

The source language of the considered kernels is DIL (Data Intermediate Language), which is a single assignment language with C operators. The DIL compiler applies a set of compiler optimizations - such as inlining, loop unrolling, in order to determine the minimum data width and to meet the target cycle time. These optimizations are parameterized with architecture-specific information. Regarding the reconfiguration overhead, the architecture imposes the reconfiguration at each cycles. In order to achieve this goal, a wide onchip configuration buffer is connected to the physical fabric.

ADRES[26][27]
IMEC

The ADRES architecture is a coarse-grained reconfigurable architecture composed by a regular array of functional units and register files. Each functional unit contains more configurations and support predicate operations.

The ADRES architecture and compiler are focused on exploiting loop level parallelism. Their goal is to fully implement on the reconfigurable array the considered kernels, using a model of the ADRES architecture in an XML-based language which must be provided to the compiler.

The DRESC compiler is based on the IMPACT compiler framework for VLIW architectures. The main extension is the Modulo scheduling algorithm which performs a mapping of the program graph and architecture graph aiming to achieve the optimal performance. The scheduling algorithm resembles the placement and routing algorithms for FPGAs, but tightly coupled in one framework. Due to the coarse-grained feature, the reconfiguration overhead is not a problem for such architectures.

DLX+FPGA[28] [29]
Politecnico di Torino

The target architecture contains a 32 -bit RISC processor DLX extended with an FPGA as a functional unit. The FPGA is dynamically reconfigurable and each of its cells can store up to 4 configurations that can be instantly interchanged. New instructions are added for the operations executed on the FPGA, with at most 4 inputs and 2 outputs, but maximum 4 registers per instructions.

This very restrictive limitation is due to the encoding limits of the DLX ISA.

The kernels considered for execution on the reconfigurable hardware are manually selected - the selection is guided by standard profilers - and delimited with pragma annotations. A set of tools based on the gcc toolchain is provided for automatic design space exploration, including the compiler, assembler, simulator and debugger. The new instructions for FPGA execution replace a relative small number of DLX instructions, thus the gcc compiler can schedule them without major modifications. Additionally, there is no instruction for loading the configurations on the FPGA and the reconfiguration overhead is not significant.

Based on the presented examples, we can conclude that there are four major shortcomings of current approaches, namely:

1. **Opcode space explosion**: a common approach (e.g. [19], [18], [28]) is to introduce a new instruction for each part of application mapped into the FPGA. The consequence is the limitation of the number of operations implemented into the FPGA, due to the limitation of the opcode space. More specifically stated, for a specific application domain intended to be implemented in the FPGA, the designer and compiler are restricted by the unused opcode space.

2. **Limitation of the number of parameters**: In a number of approaches, the operations mapped on an FPGA can only have a small number of input and output parameters ([28], [23]). For example, in the architecture presented in [29], due to the encoding limits, the fragments mapped into the FPGA have at most 4 inputs and 2 outputs; also, in Chimaera [23], the maximum number of input registers is 9 and it has one output register.

3. No support for **parallel execution** on the FPGA of sequential operations: an important and powerful feature of FPGA's can be the parallel execution of sequential operations when they have no data dependency. Many architectures [30] do not take into account this issue and their mechanism for FPGA integration cannot be extended to support parallelism.

4. No **modularity**: each approach has a specific definition and implementation bounded for a specific reconfigurable technology and design. Consequently, the applications cannot be (easily) ported to a new reconfigurable platform. Further there are no mechanisms allowing reconfig-

urable implementation to be developed separately and ported transparently. That is a reconfigurable implementation developed by a designer A can not be included without substantial effort by the compiler developed for an FPGA implementation provided by a designer B.

A general approach that eliminates these shortcomings is required. In the rest of this chapter, we introduce the Molen machine organization with the Molen Programming Paradigm, and the DelftWorkBench toolchain with special emphasis on the Molen Compiler. We will mainly discuss how this approach addresses the above mentioned problems and eventually solve them.

2.4 The Molen Programming Paradigm

In this thesis, we target reconfigurable architectures following the Molen machine organization, depicted in Figure 2.2. The two main components in the Molen machine organization are the Core Processor, which is a general-purpose processor, and the Reconfigurable Processor (RP), usually implemented on an FPGA. Another key component is the Arbiter which performs a partial decoding of the instructions received from the instruction fetch unit and issue them to the appropriate processor (GPP or RP). Data are fetched (stored) by the Data Fetch unit from(to) the main memory. The Memory MUX unit is responsible for distributing data betwen the reconfigurable and the core processor. The Exchange Registers (XREGs) are used for data communication between the Core Processor and Reconfigurable Processor. However, the Reconfigurable Processor can access the main memory through the Memory MUX.

The Reconfigurable Processor is further subdivided into the $\rho\mu$-code unit and the custom configured unit (CCU). The $\rho\mu$-code unit provides fixed and pageable storage for the reconfiguration bitstreams and controls the CCU. The CCU consists of reconfigurable hardware and it is intended to support and accelerate additional and future functionalities that are not implemented/suitable for the core processor. The Molen machine organization has been implemented in the Molen Polymorphic processor on Virtex II Pro FPGA platform and described in [31].

The envisioned support of operations by the reconfigurable processor can be initially divided into two distinct phases: *set* and *execute*. In the set phase, the CCU is configured to perform the supported operations. Subsequently, in the execute phase, the actual execution of the operations is performed. This

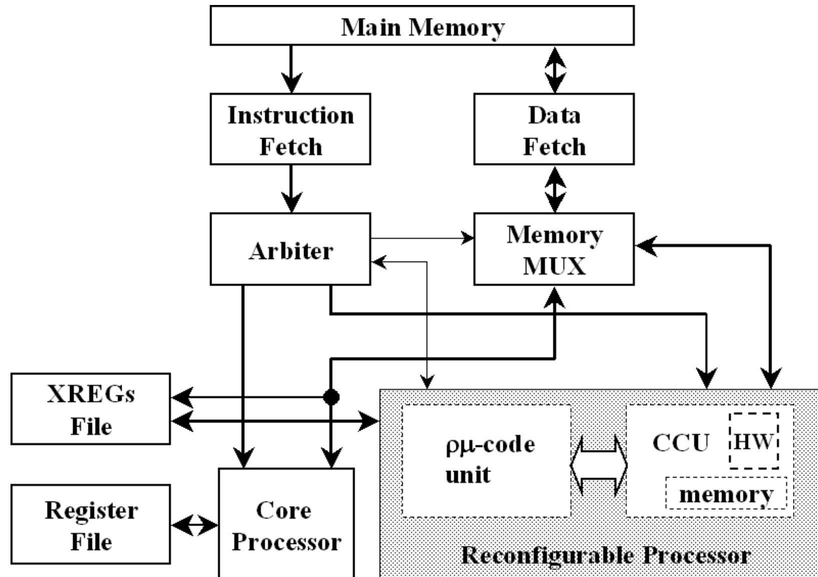

Figure 2.2: The Molen machine organization

decoupling allows the set phase to be scheduled well ahead of the execute phase, thereby hiding the reconfiguration latency. As no actual execution is performed in the set phase, it can even be scheduled upward across the code boundary in the code preceding the RP targeted code.

One main advantage of the Molen machine organization is based on the reintroduction of the microcode for the emulation of the complex operations that are performed on the reconfigurable hardware. The microcode (denoded as $\rho\mu$-code) is a sequence of simpler and smaller basic operations that control both reconfiguration and execution of the CCU. The consequence is that a generic instruction (*set* instruction) can be used for any hardware configuration, as the specific configuration is entirely controlled by the associated $\rho\mu$-code. Additionally, only one generic instruction (*execute* instruction) is provided for starting the execution on the reconfigured hardware of any implemented hardware operation, as its effect is completely depended of its associated microcode. By the introduction of the $\rho\mu$-code, the Molen machine organization eliminates the first shortcoming presented in the previous section and provide solid support for solving the remaining drawbacks.

The Molen Programming Paradigm [32] [33] is a sequential consistency paradigm for programming CCMs possibly including a general purpose com-

putational engine(s). The paradigm allows for parallel and concurrent hardware execution and it is intended (currently) for single program execution. The Molen Programming Paradigm requires only a one-time architectural extension of few instructions to provide a large user reconfigurable operation space. The complete list of the eight required instructions, denoted as polymorphic instruction set architecture (π ISA), is as follows:

Six instructions are required for controlling the reconfigurable hardware:

- Two set instructions: these instructions initiate the configurations of the CCU. When assuming partial reconfigurable hardware, we provide two instructions for such purpose, namely:
  - the partial set (p-set $\langle address \rangle$) instruction performs those configurations that cover common and frequently used functions of an application or set of applications. In this manner, a considerable number of reconfigurable blocks in the CCU can be preconfigured.
  - the complete set (c-set $\langle address \rangle$) instruction performs the configurations of the remaining blocks of the CCU (not covered by the p-set). This completes the CCU functionality by enabling it to perform the less frequently used functions. Due to the reduced amount of blocks to configure, reconfiguration latencies can be reduced.

  We must note that in case no partially reconfigurable hardware is present, the c-set instruction alone can be utilized to perform all configurations.

- execute $\langle address \rangle$: this instruction controls the execution of the operations implemented on the CCU. These implementations are configured onto the CCU by the set instructions.

- set prefetch $\langle address \rangle$: this instruction prefetches the needed microcode for CCU reconfigurations into a local on-chip storage facility (the $\rho\mu$-code unit) in order to possibly diminish microcode loading times.

- execute prefetch $\langle address \rangle$: the same reasoning as for the set prefetch instruction holds, but now relating to microcode responsible for CCU executions.

- break: this instruction is utilized to facilitate the parallel execution of both the reconfigurable processor and the core processor. More precisely, it is utilized as a synchronization mechanism to complete the par-

allel execution. Thus, the shortcoming regarding the support for parallel execution is eliminated.

Two *move* **instructions** for passing values between the register file and exchange registers (XREGs) since the reconfigurable processor is not allowed direct access to the general-purpose register file:

- movtx $XREG_a \leftarrow R_b$: (move to XREG) used to move the content of general-purpose register Rb to XREGa.
- movfx $R_a \leftarrow XREG_b$: (move from XREG) used to move the content of exchange register XREGb to general-purpose register Ra.

The $\langle address \rangle$ field in the instructions introduced above denotes the location of the reconfigurable microcode responsible for the configuration and execution processes, previously described. It must be noted that a single address space is provided with at least 2^{n-op} addressable functions, where n represents the instruction length and op the opcode length. If 2^{n-op} is found to be insufficient, indirect pointing or GPP-like status word mechanisms can extend the addressing of the reconfigurable function space at will. One important observation is that the operands are not directly encoded in the instruction format; instead, the microcode for each operation is responsible to access the associated XREGs. In consequence, the number of input and output values is limited only by the number of available XREGs, which can be mapped in the local memory of the reconfigurable hardware and thus, it is not a real limitation and resolve the second shortcoming regarding reconfigurable architectures.

It should be noted that it is not imperative to include all instructions when implementing the Molen organization. The programmer/implementor can opt for different ISA extensions depending on the required performance to be achieved and the available technology. There are basically three distinctive πISA possibilities with respect to the Molen instructions introduced earlier - the minimal, the preferred and the complete πISA extension. In more detail, they are:

- **the minimal πISA**: This is essentially the smallest set of Molen instructions needed to provide a working scenario. The four basic instructions needed are *set* (more precisely: c-set), *execute*, movtx and movfx. By implementing the first two instructions (*set*/*execute*) any suitable CCU implementation can be loaded and executed in the RP. Furthermore, reconfiguration latencies can be hidden by scheduling the set instruction considerably earlier than the execute instruction. The movtx and movfx

instructions are needed to provide the input/output interface between the RP targeted code and the remainder application code. **Observation:** The minimal πISA extension is assumed in the rest of the thesis

- **the preferred πISA**: The minimal set provides the basic support, but it may suffer from time-consuming reconfiguration latencies, which could not be hidden, and that can become prohibitive for some real-time applications. In order to address this issue, two *set* (p-set and c-set) instructions are utilized to distinguish among frequently and less frequently used CCU functions. In this manner, the c-set instruction only configures a smaller portion of the CCU and thereby requiring less reconfiguration time. As the reconfiguration latencies are substantially hidden by the previously discussed mechanisms, the loading time of microcode will play an increasingly important role. In these cases, the two prefetch instructions (set prefetch and execute prefetch) provide a way to diminish the microcode loading times by scheduling them well ahead of the moment that the microcode is needed. Parallel execution is initiated by a πISA *set*/*execute* instruction and ended by a general-purpose instruction.

- **the complete πISA**: This scenario involves all πISA instructions including the break instruction. In some applications, it might be beneficial performance-wise to execute instructions on the core processor and the reconfigurable processor in parallel. In order to facilitate this parallel execution, the preferred ISA is further extended with the break instruction. The break instruction provides a mechanism to synchronize the parallel execution of instructions by halting the execution of instructions following the break instruction. The sequence of instructions performed in parallel is initiated by an *execute* instruction. The end of the parallel execution is marked by the break instruction. The *set* instructions are executed in parallel according to the same rules.

The Exchange Registers. The XREGs are used for passing operation parameters to the reconfigurable hardware and returning the computed values after the operation execution. Parameters are moved from the register file to the XREGs (movtx) and the results stored back from the XREGs in the register file (movfx).

During the execution phase, the defined microcode can access the parameters of its associated operation from specific XRs and return the result(s). A sequential *execute* instruction does not pose any specific challenge because the

whole set of exchange registers is available. However, when executing multiple *execute* instructions in parallel, additional conventions are introduced in order to avoid the overlapping of the used XREGs. A more detailed discussion is presented in the next chapter.

The Molen paradigm facilitates *modular system design*. For instance, hardware implementations described in an HDL (VHDL, Verilog or System-C) language are mappable to any FPGA technology, e.g., Xilinx or Altera, in a straightforward manner. The only requirement is to satisfy the Molen *set*/*execute* interface. In addition, a wide set of functionally similar CCU designs (from different providers), e.g. sum of absolute differences (SAD), can be collected in a database allowing easy design space explorations. Thus, the fourth shortcoming regarding reconfigurable architecture is eliminated.

Interrupts and miscellaneous considerations. The Molen approach is based on the GPP co-processor paradigm. Consequently, all known co-processor interrupt techniques are applicable. In order to support the core processor interrupts properly, the following parts are essential for any Molen implementation:

1. Hardware to detect interrupts and terminate the execution before the state of the machine is changed, is assumed to be implemented in both core processor and reconfigurable processor.

2. Interrupts are handled by the core processor. Consequently, hardware to communicate interrupts to the core processor is implemented in CCU.

3. Initialization (via the core processor) of the appropriate routines for interrupt handling.

It is assumed that the implementor of a reconfigurable hardware follows a co-processor type of configuration. With respect to the GPP paradigm, the FPGA co-processor facility can be viewed as an extension of the core processor architecture. This is identical with the way co-processors, such as floating point, vector facilities, etc., have been viewed in the conventional architectures.

Regarding the shortcomings presented in the previous section, the Molen Programming Paradigm and the architectural extensions solve the aforementioned problems as follows:

- There is only a one time architectural extension of few new instructions to include an arbitrary number of configuration.

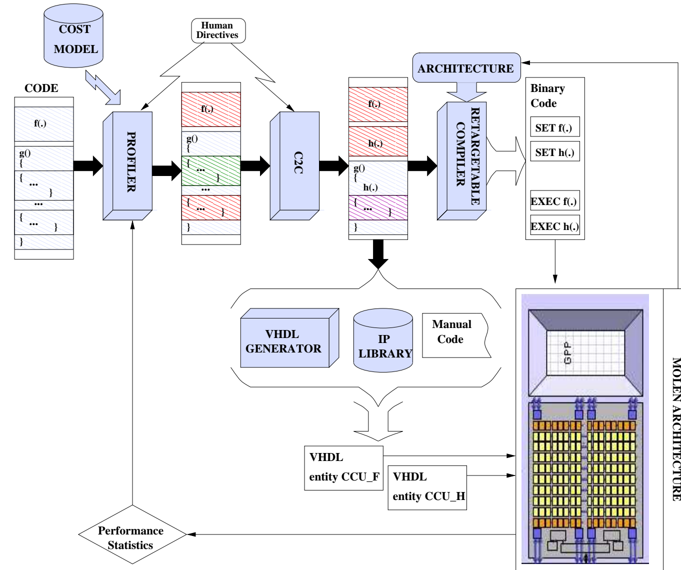

Figure 2.3: The Delft Workbench Design Flow

- The programming paradigm allows for an arbitrary (only hardware real estate design restricted) number of I/O parameter values to be passed to/from the reconfigurable hardware. It is only restricted by the implemented hardware as any given technology can (and will) allow only a limited hardware.
- Parallelism is allowed as long as the sequential memory consistency model can be guaranteed.
- Assuming that the interfaces are observed, modularity is guaranteed because the paradigm allows freedom of operation implementation.

2.5 DelftWorkBench

DelftWorkBench project aims to provide a semi-automatic tool platform for integrated hardware-software co-design targeting heterogeneous computing systems containing reconfigurable components which provide the required support for the Molen Programming Paradigm. Delft Workbench addresses the entire design cycle rather than isolated parts. The design flow is presented in Figure 2.3.

Profiler: As shown in Figure 2.3, the first step is the identifications of the application parts that can provide the required benefit when implemented and executed on the reconfigurable hardware. The target objective can vary significantly, from increased performance to reduced power consumption or a smaller footprint. The profiler can collect and analyze execution traces of the program and use this information in combination with human directives to propose a number of candidate code segments.

In order to quantify the potential benefit of a certain task, the profiler relies on the estimation cost model [34] of the target reconfigurable hardware that will provide preliminary estimation about configuration delays, area usage, power consumption, etc. Such a cost model will allow to filter away those candidates that will not likely result in the anticipated improvement in the view of the target objective. The input for the profiler is ANSI C code and the output is annotated C code with pragma directives to indicate the tasks considered for execution on the reconfigurable hardware.

In the **C2C** step, the kernels proposed by the profiler are further analyzed and transformed in order to better fit on the reconfigurable hardware. One main transformation is graph restructuring [35] [36] that aims to determine which clusters of basic operations are optimal for hardware execution, taking into account their execution frequency and potential benefits. Advanced loop optimization can further be applied to fully exploit the loop level parallelism and to remove the data dependency by using the reconfigurable hardware. After the C2C step, the set of tasks for hardware execution is completely defined.

Retargetable Compiler: Once the kernels have been identified, the compiler generates the appropriate link code for the execution on the reconfigurable Processor, while the rest of the application is compiled for the GPP. The link code mainly contains the following:

- code for passing parameters to the Reconfigurable Processor in XREGs
- instructions for hardware configuration
- instructions for starting the execution on the Reconfigurable Processor
- code for returning the computed results from XREGs

One main goal of the compiler is to generate high quality code tailored to the specific features of the target architecture. In this case, specific optimizations have to be included in the compiler in order to address the distinct characteristics of the reconfigurable hardware such us the reconfiguration overhead,

parallel execution, sw/hw final partitioning, reconfigurable hardware allocation. The compiler is the main focus of this thesis and it is further denoted as the Molen compiler.

VHDL Generation: For the tasks executed on the reconfigurable hardware, the VHDL design can be obtained using three approaches. The first one is the manual VHDL generation and it is appropriate for critical or uncommon tasks. However, this approach is a time-consuming and error prone task. The second approach is to use IP cores which are already available for general tasks such us DCT, IDCT. The third approach [37] is the automatic code generation from the associated C code. As previously discussed, this approach is considered in many research projects (see [38] [21] [39]), but the quality of the generated code is far bellow the expectation and there is a large set of limitations on the C code which can be automatically translated to VHDL. In DelftWorkBench project, the automatic VHDL generation will address these limitations and the research will focus on optimizations and scheduling techniques for loops and memory accesses.

2.6 Conclusion

In this chapter, we presented the background for this thesis and an overview of reconfigurable architectures. We identify the main problems of current approaches and present how the targeted Molen machine organization and programming paradigm eliminates them. The main advantages of the Molen approaches can be summarized as follows:

Compact and transparent ISA extension For a given ISA, only a one time architectural extension of up to 8 instructions is required in order to support a virtually unlimited number of reconfigurable operations. This achievement is mainly based on the introduction of the $\rho\mu$-code which is the emulation code that allows to define generic instructions without concern about their exact implementation on the reconfigurable hardware. Additionally, the proposed ISA extension is application independent and provides ISA compability and portability.

Technology independent and modular design HDL designs can be developed independently of the target reconfigurable hardware and they can be easily integrated in the Molen organization (using vendor's tools for synthesis) as long as the described interfaces are preserved.

Parallel processing The user can select from different levels of parallelism supported by the Molen Programming Paradigm. When parallelism is not the main concern, than minimal πISA extension can be used, while the complete πISA allows for parallel execution on the reconfigurable hardware and GPP.

In the next chapter, we present the basic Molen compiler backend that targets reconfigurable architectures under the Molen Programming Paradigm. In the following chapters, specific compiler optimizations and scheduling algorithms are proposed to take advantage of the distinctive features of the reconfigurable architectures.

Chapter 3

The Molen Compiler

When most alternative reconfigurable architectures rely on simulations and estimations for validation purposes, we disposed of a physical implementation of the Molen machine organization. The Molen Polymorphic processor ([40]) was implemented on a Virtex II Pro FPGA platform which consists of one PowerPC General Purpose Processor immersed into the reconfigurable hardware.

In this chapter, we present the Molen compiler framework we have developed for the Molen Polymorphic processor, with emphasize on the extended PowerPC backend. We first present in Section 3.1 the Molen compiler framework and general extensions required for the Molen Programming Paradigm. We shortly describe the specific features of the Molen Polymorphic processor in Section 3.2 and next we discuss in details the PowerPC compiler backend we have implemented in the Molen compiler. In section 3.4, we present as a proof of concept an experiment with the M-JPEG multimedia application running on the Molen Polymorphic processor with a 2.5 speedup over the PowerPC processor alone. Finally, the chapter is concluded with Section 3.5.

3.1 The Molen Compiler Framework

The Molen compiler [41] currently relies on the Stanford **SUIF2** (Stanford University Intermediate Format)[42] for the front-end and the Harvard **Machine SUIF**[43] backend framework, as presented in Figure 3.1. The **SUIF2** compiler infrastructure was designed for research and development of new compilation techniques that maximize code reuse by providing general ab-

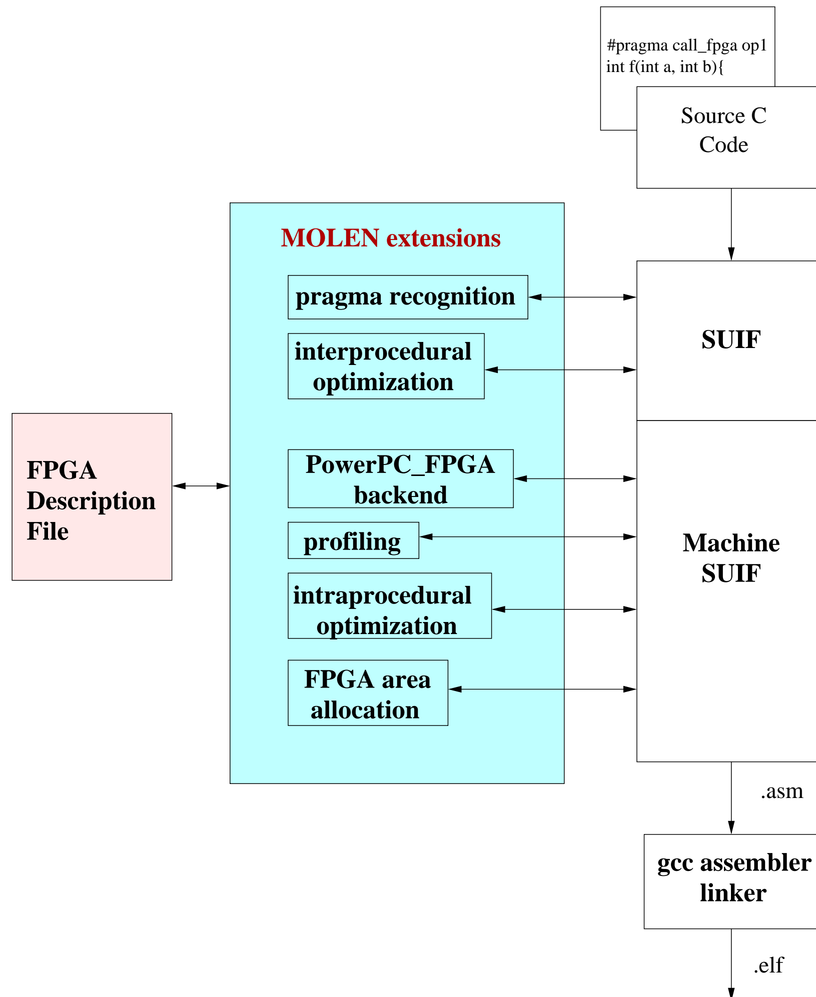

Figure 3.1: The Molen Compiler Structure

stractions and frameworks. Thus, new optimizations and transformations can easily be added and the flexible IR can be extended to express new features of the application or target architecture. It provides advanced analyses for loop parallelism such as affine program transformations and interprocedural program analysis and a C converter.

Machine SUIF is a flexible and extensible infrastructure for constructing compiler backends. Although it is based on the SUIF system, its optimizations and analyses are not SUIF specific as they can be reused in other compiler en-

vironments as long as the Optimization Programming Interface (OPI) is supported. It provides support for building control-flow graphs, control flow analyses and bit vector dataflow analyses, as well as a set of optimizations such as common subexpression elimination, dead code elimination, peephole optimizations. Additionally, a set of backends are already available (e.g. Compaq Alpha, Intel x86, suifvm - SUIF virtual machine), together with a graph-coloring register allocation and support for code finalization, assembly and C printing. Finally, it also supports code instrumentation that allows development of profile-driven optimizations that require accurate and specific profile informations.

The Molen compiler's input is C99 [44], with user's pragmas that indicate the kernel functions implemented on the reconfigurable hardware. Regarding the C preprocessing step, the user has to indicate the appropriate system headers, taking into account that the compilation is usually a crosscompilation (e.g. running on a Linux machine while compiling to Xilinx FPGA platform). A basic compilation flow of the Molen compiler typically contains the following steps:

- **Frontend Processing:**
  - c2s - C to SUIF converter
  - call_fpga - pragma recognition
  - do_lower - SUIF to Low SUIF converter
- **Backend Processing:**
  - do_s2m - SUIF to Machine SUIF converter
  - do_gen - code generation for a target architecture given as a parameter
  - do_il2cfg - converter from instruction list to control flow graph
  - do_raga - register allocation
  - do_cfg2il - converter from control flow graph to instruction list
  - do_fin - code finalization
  - do_il2cfg
  - do_raga - register allocation again for the virtual registers from code finalization
  - do_cfg2il

  – do_m2a - ascii/asm printer

- **Assembler/Linker processing:**

  The GNU assembler and linker have been modified for the target architecture.

Additional optimizations and analyses can be easily included in the compilation flow as independent steps. Such optimizations can be the standard optimizations provided by SUIF/MachineSUIF infrastructure or the Molen optimizations we have developed for reconfigurable architectures.

The Molen Compiler Extensions

In order to generate code according to the Molen Programming Paradigm, the following target-independent Molen extensions have been implemented

- *Code identification*: for the identification of the code mapped on the reconfigurable hardware, we added a special pass (denoted as *call_fpga*) in the SUIF front-end. This identification is based on code annotation with special pragma directives (similar to [19]). More specifically, the definitions of the functions executed on the reconfigurable hardware are preceded by a pragma annotation *call_fpga* and the name of the associated hardware operation, as included in the FPGA description file. In this pass, all the calls of the recognized functions are marked for further modifications.

- *MIR extension*: the MIR *suifvm* has been extended with SET/ EXECUTE and MOVTX/MOVFX instructions at MIR (Medium Intermediate Representation) level.

- *Register file extension*: the Register File Set has been extended with the XRs.

- *MIR Code generation*: code generation for the reconfigurable hardware is performed when translating SUIF to Machine SUIF IR *suifvm*, and affects the function calls marked in the front-end.

An example of the code generated by the extended compiler for the Molen Programming Paradigm is presented in Figure 3.2. In the first part, the C program is given. The function implemented in reconfigurable hardware is annotated with a pragma directive named *call_fpga*. It has incorporated the operation name, *op1* as specified in the FPGA description file. In the central part of

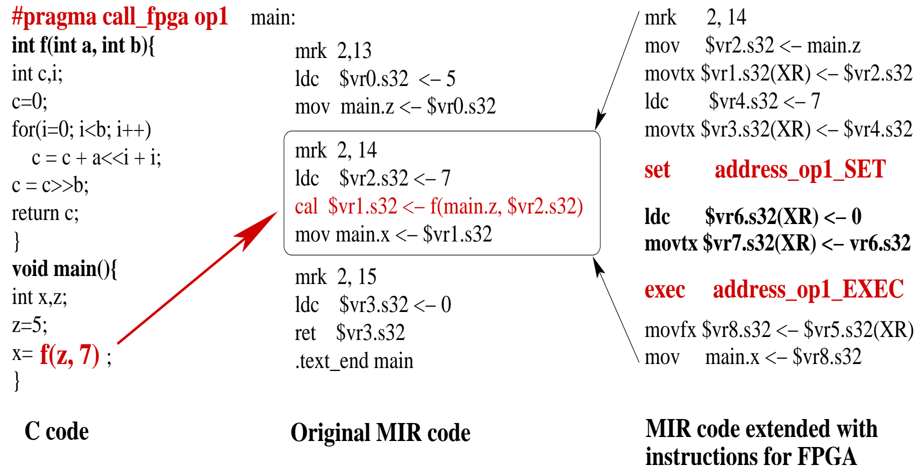

Figure 3.2: Code Generation at MIR level

the picture, the code generated by the original compiler for the C program is depicted. The pragma annotation is ignored and a standard function call is included. The last part of the picture presents the code generated by the compiler extended for the Molen Programming Paradigm; the function call is replaced with the appropriate instructions for sending parameters to the reconfigurable hardware in XRs, hardware reconfiguration, execution of the operation and the transfer of the result back to the GPP. The presented code is at MIR level (before *do_gen* pass) and the register allocation pass has not been applied.

The inter/intraprocedural optimizations and the FPGA area allocation algorithms from Figure 3.1 are general Molen optimizations aiming to reduce the reconfiguration overhead. These optimizations use detailed profile information regarding the kernels executed on the reconfigurable hardware. Software and hardware features of the considered kernels are included in the FPGA Description File; the hardware features are provided by the hardware designers, while the software features address the execution on the GPP alone, and are measured in the profiling phase. The PowerPC_FPGA backend is a specific backend we have implemented for the Molen Polymorphic processor (see Section 3.2) and it is presented in details in Section 3.3.

3.2 The Molen Polymorphic Processor

In this section, we discuss the implementation of a microarchitecture supporting the minimal Molen πISA on the Virtex II Pro with the embedded PowerPC 405 serving as the core processor. The Virtex II Pro family contains platform FPGAs for designs that are based on IP cores and customized modules. The family incorporates up to four IBM PowerPC RISC 405 processor blocks, with the following main features:

- embedded 300+ MHz Harvard Architecture Block
- low power consumption: 0.9 mW/MHz
- five-stage data path pipeline
- hardware multiply/divide unit
- thirty-two 32-bit General Purpose Registers
- 16 KB two-way set-associative instruction cache
- 16 KB two-way set-associative data cache
- memory management unit (MMU)
  - Variable page sizes (1 KB to 16 MB)
- dedicated on-chip memory (OCM) interface
- supports IBM CoreConnect" bus architecture
- debug and trace support
- timer facilities

Virtex-II Pro devices incorporate large amounts of 18Kb Block SelectRAM+ memory. The available memory resources for Virtex II Pro XC2VP20 is around 300 Kb while for XC2VP50 is around to 700 Kb. OCM controllers provide dedicated interfaces between Block SelectRAM+ memory and processor block instruction and data paths for high-speed access routing resources.

These features make the The Virtex II Pro platform suitable for the implementation of the Molen machine organization. The first implementation was performed on the Virtex II Pro XC2VP20 FPGA platform and followed by the implementation on the larger Virtex II Pro XC2VP30 FPGA platform. In both implementations, there is only one core processor, namely IBM PowerPC 405.

A key element is the implementation of the arbiter which is described in detail in [45] [46] [47]. The arbiter controls the proper coprocessing of the core processor and the reconfigurable processor (see Fig. 2.2) by directing instructions to either of these processors. It also arbitrates the data memory access of the reconfigurable and core processors and it distributes control signals and the starting microcode address to the $\rho\mu$–code unit. The arbiter operation is based on the partial decoding of the incoming instructions and either directs instructions to the core processor or generates an instruction sequence to control the state of the core processor. The latter instruction sequence is referred to as arbiter emulation instructions and it is used upon decoding of either a *set* or an *execute* instruction, as explained below.

Software considerations For performance reasons, PowerPC special operating modes instructions were not used - exiting special operating modes is usually performed by an interrupt. Instead, the arbiter emulates a wait state by using the branch to link register (*blr*) instruction and the exit from the wait state by using branch to link register and link (*blrl*) instruction. The difference between these instructions is that *blrl* modifies the link register (LR), while *blr* does not. The next instruction address is the effective address of the branch target, stored in the link register. When *blrl* is executed, the new value loaded into the link register is the address of the instruction following the branch instruction. Thus, the arbiter emulation instructions, are reduced to only one instruction for wait and one for wake-up emulation. The PowerPC architecture allows out-of-order execution of memory and I/O transfers, which has to be taken into account in the implementation. To guarantee that data dependency conflicts do not occur during reconfigurable operation, the PowerPC synchronization instruction (*sync*) can be utilized before a *set* or *execute* instruction. In other out-of-order execution architectures, data dependency conflicts should be resolved by specific dedicated features of the target architectures. In in-order architecture implementations, this problem does not exist.

Instruction encoding In this implementation, the *movtx* and *movfx* instructions are mapped to the existing PowerPC instructions *mtdcr* and *mfdcr*. This solution is imposed by the fact that the Virtex II Pro PowerPC core has a dedicated interface to the so-called Device Control Registers (DCR) [48] [49] [50] and two instructions that support DCR transfers (namely, *mtdcr* and *mfdcr*). It should be noted that this is a PowerPC specific implementation and not applicable in the general case. This leaves only the *set* and *execute* instructions to be encoded. We follow the PowerPC I-form and choose unused opcodes for both instructions. The manner to distinguish a *set* instruction, an *execute* instruction (using the same opcode), and resident/pageable (R/P) microcode addresses is

via instruction modifiers.

3.3 Molen PowerPC Compiler Backend

The first step for compiling for the Molen Polymorphic processor is to have a backend compiler that generates the appropriate binaries to be executed on the PowerPC processor integrated on the Virtex II Pro board. Current MachineSUIF backends excluded the backend for PowerPC architecture. In this section, we present the Molen PowerPC backend we developed for this purpose and we focus on the PowerPC instruction generation, PowerPC register allocation, PowerPC EABI stack frame allocation and software floating-point emulation. We also describe the specific PowerPC backend extensions for the Molen Polymorphic processor.

3.3.1 PowerPC Compiler Backend

In order for one application to utilize external and/or underlying software or hardware, a binary interface - called Application Binary Interface (ABI) has to be defined. For example, many applications have to include a set of libraries (e.g. math) that are compiled using a number of platform dependent conventions. One such set of conventions proposed for PowerPC 405 is the Embedded Application Binary Interface (EABI) with the goal of reducing memory usage and optimizing execution speed, as these are prime requirements of embedded system software. The EABI describes conventions for register usage, parameter passing, stack organization, small data areas, object file, and executable file formats. A description of the key issues for the PowerPC compiler backend we have implemented is presented in the rest of this section.

Register Usage

In user mode, The PowerPC 405 processor provides the following registers:

- General Purpose Registers (GPRs): 32 registers, each 32 bits wide, numbered r0 through r31;
- Condition Register (CR): a 32-bit register that reflects the result of certain instructions and provides a mechanism for testing and conditional branching; for example a branch based on the condition r3 $<$ 64 can be implemented as follows:

```
; CR has 8 fields of 4 bits each
```

```
cmplwi 3, r3, 64   ; CR3 field contain
       ; the result of the comparison
blt 3, LABEL_1  ; branch based on CR3
......
```

- Fixed-Point Exception Register (XER): a 32-bit register that reflects the result of arithmetic operations that have resulted in an overflow or carry;
- Link Register (LR): a 32-bit register that is used by branch instructions, generally for the purpose of subroutine linkage;
- Count Register (CTR): a 32-bit register that can be used by branch instructions to hold a loop count or the branch-target address;
- User-SPR General-Purpose Register (USPRG0): a 32-bit register that can be used by application software for any purpose;
- SPR General-Purpose Registers (SPRG4- SPRG7): 32-bit registers that can be used by system software for any purpose and available with read-only access
- Time-Base Registers: a 64-bit incrementing counter implemented as two 32-bit registers (TBU and TBL) with read-only access

Register	Type	Usage
R0	Volatile	Language specific
R1	Dedicated	Stack Pointer (SP)
R2	Dedicated	Read-only small data area anchor
R3 - R4	Volatile	Parameter Passing/ return values
R5 - R10	Volatile	Parameter Passing
R11 - R12	Volatile	
R13	Dedicated	Read-write small data area anchor
R14 - R31	Nonvolatile	
Fields CR2 - CR4	Nonvolatile	Condition Register
Other CR fields	Volatile	Condition Register
Other registers	Volatile	

Table 3.1: PowerPC EABI Register Usage

The PowerPC EABI register usage conventions are depicted in Table 3.1. Nonvolatile registers must have their original values preserved, therefore, functions

```
FuncX:

        mflr %r0              ; Get Link register
        stwu %r1,-88(%r1)     ; Save Back chain and move SP
        stw %r0,+92(%r1)      ; Save Link register
        stmw %r28,+72(%r1)    ; Save 4 non-volatiles r28-r31

        ....................................................

        lwz %r0,+92(%r1)      ; Get saved Link register
        mtlr %r0              ; Restore Link register
        lmw %r28,+72(%r1)     ; Restore non-volatiles
        addi %r1,%r1,88       ; Remove frame from stack
        blr                   ; Return to calling function
```

Figure 3.3: Function's Prologue and Epilogue

modifying nonvolatile registers must restore the original values before returning to the calling function.

The Stack Frame

In addition to the registers, each function may have a stack frame on the runtime stack. The PowerPC architecture does not have a push/pop instruction for implementing a stack. The EABI conventions of stack frame creation and usage for parameter passing, nonvolatile register preservation, local variables, and code debugging are presented in Fig. 3.4. The following requirements apply to the stack frame:

- The address of the previous frame is stored in Back Chain Word, thereby forming a linked-list of stack frames and it is always located at the lowest address of the stack frame.
- The return address to the calling function is stored in the LR Save Word.
- In order to maintain 8-byte alignment of the stack frame, a Padding Area may be introduced to guarantee such alignment.
- In the Function Parameters Area, additional function arguments are stored when they do not fit into the designated registers R3-R10.
- When the number of local variables is higher than can be contained in the available volatile registers, they are stored in Local Variables Area.

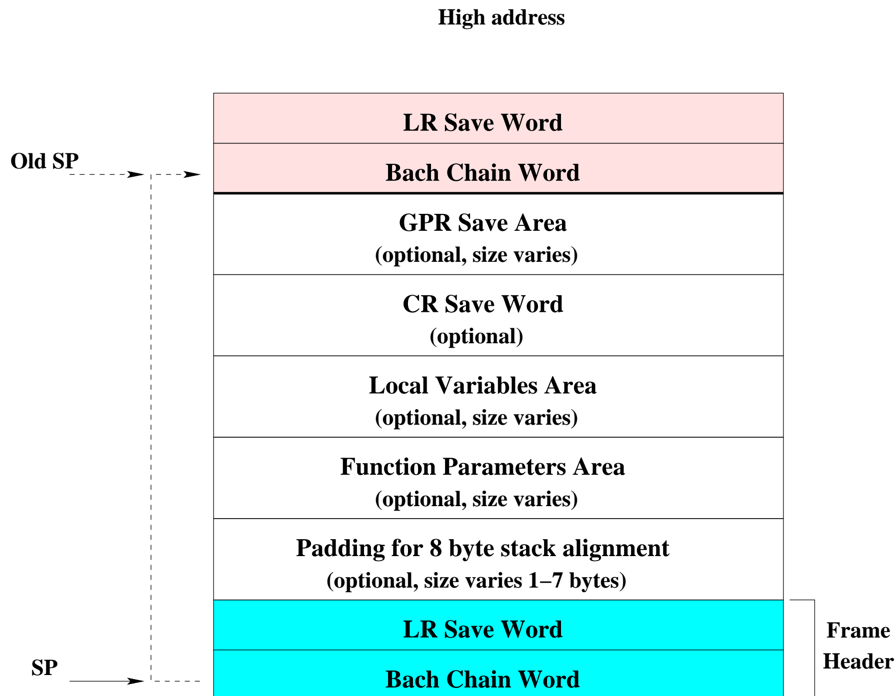

Figure 3.4: PowerPC EABI Stack Frame Organization

- When the nonvolatile CR fields are modified, its content needs to be saved in the CR Save Word.
- GPR Save Area may be introduced to save nonvolatile GPR. When saving any GPR, all the GPRs from the lowest through R31, inclusive, must be saved.

The stack frame is created by a function's prologue code and destroyed in its epilogue code. In Fig. 3.3 is presented an example of function's prologue and epilogue.

Floating-Point Emulation

The PowerPC 405 is an integer processor and does not support the execution of floating-point instructions in hardware. System software can provide floating-point emulation support by supplying a call interface to subroutines within a floating-point run-time library. The individual subroutines emulate the operation of floating-point instructions as presented in [51]. This method requires the recompilation of floating-point software in order to add the call interface

and link in the library routines.

Op Type	unsigned	signed
char	rlwinm r1,0,0xff → r4 rlwinm r2,0,0xff → r5 mullw r4, r5 →r3	
short	rlwinm r1,0,0xffff → r4 rlwinm r2,0,0xffff → r5 mullw r4, r5 →r3	extsh r1 → r4 extsh r2 → r5 mullw r4, r5 →r3
int	mulhwu r1, r2 → r3’ mullw r1, r2 → r3”	mulhw r1, r2 → r3’ mullw r1, r2 → r3”
long long	mullw r2”, r1” → r3”” mulhwu r2”, r1” → r3”’ mullw r2”, r1’ → r10 mulhwu r2”, r1’ → r3” mullw r2’, r1” → r12 mulhwu r2’, r1” → r11 mullw r2’, r1’ → r13 mulhwu r2’, r1’ → r3’ addc r3”’, r10 → r3”’ adde r3”, r10 → r3” addze r3’ → r3’ addc r3”’, r12 → r3”’ adde r3”, r13 → r3” addze r3’ → r3’	mullw r2”, r1” → r3”” mulhw r2”, r1” → r3”’ mullw r2”, r1’ → r10 mulhw r2”, r1’ → r3” mullw r2’, r1” → r12 mulhw r2’, r1” → r11 mullw r2’, r1’ → r13 mulhw r2’, r1’ → r3’ addc r3”’, r10 → r3”’ adde r3”, r10 → r3” addze r3’ → r3’ addc r3”’, r12 → r3”’ adde r3”, r13 → r3” addze r3’ → r3’

Table 3.2: LIR translation of MIR instruction *mul r1, r2 → r3*

The compiler has to manage floating point arithmetic, comparisons, loads, and stores by generating software floating point emulation (sfpe) code, rather than using PowerPC floating point instructions - currently it is not fully supported in the Molen compiler. In sfpe code:

- Floating point single precision scalars shall be treated as long int scalars.
- Floating point double precision scalars shall be treated as long long scalars.
- Whenever a function has a variable argument list, it shall not set condition register bit 6 to 1 (as usual for PowerPC architecture), since no arguments are passed in the floating-point registers (as there are no FPR included in PowerPC 405).

Code Selection

Code selection is typically the first backend phase and maps machine independent IR statements and operations into machine specific processor instructions. This phase is performed in Machine SUIF where each IR statement is translated into equivalent assembly instructions. For example, for the MIR instruction *mul r1, r2* $\rightarrow$ *r3*, the generated LIR set of instructions depends on the operands type as presented in Table 3.2. In the case when both operands are of type unsigned short (2 bytes), then a mask for each operand (to take the lower 16 bits) is required and the result of the single multiplication can be placed in a 32-bit register. For integer operands, two multiplications are required when a 8 byte result is expected.

For RISC targets with homogeneous register files, the translation of each MIR instruction separately provides satisfactory results, since there are hardly complex instructions and late improvement of the selected code is still possible by means of peephole optimization.

3.3.2 PowerPC Backend Extensions for the Molen Prototype:

The pure PowerPC backend we have previously presented has been extended to generate the appropriate code for the Molen Polymorphic processor as follows:

- SET/EXECUTE instructions are included in the MIR (Medium-level Intermediate Representation) and LIR (Low-level Intermediate Representation) of the Molen compiler. In order not to modify the PowerPC assembler and linker, the compiler generates the instructions in binary form. For example, for the instruction *exec 0x80000C* the generated code is *.long 1A000031* where the encoding format (presented in [52]) is recognized by the arbiter.

- MOVTX/MOVFX: The equivalent PowerPC instructions are *mtdct/mfdcr*. Moreover, the XRs (exchange registers) are not physical registers but they are mapped at fixed memory addresses.

- The XRs are allocated in contiguous locations as specified in the Molen Programming Paradigm in order to allow a simple manipulation of the parameters.

In Figure 3.5, we present the code generated by the Molen compiler for the DCT* function call executed on the reconfigurable hardware. In order to correctly generate the instructions for hardware configuration and execution, the compiler needs information about the DCT* hardware implementation. This

```
la 3, 12016(1)          #load the address of the first param
la 12, 12016(1)         #load the address of the second param
mtdcr 0x00000056,3      #send the address of the first parameter
mtdcr 0x00000057,12     #send the address of the second parameter
sync                    #
nop                     #synchronization
nop                     #
nop                     #
bl main._label0         #instr. required by the arbiter impl.
main._label0:
.long 0x1A000031        #exec 0x8000C
nop                     #synchronization
```

Figure 3.5: Code generated by the Molen compiler for the reconfigurable DCT* execution

```
1: NO_XRS        = 512        # number of available XRs
2: START_XR      = 0x56       # the address of the first XR
3: OP_NAME       = dct        # info about the DCT* operation
4: SET_ADDR      = 0x39A100   # the address of SET
5: EXEC_ADDR     = 0x80000C   # the address of EXEC
...............
k: END_OP                     # end of the info about the DCT*
................................  # info about other operations
```

Figure 3.6: Example of an FPGA Description File

information is described in an FPGA Description File, which contains for the DCT* operation the fields presented in Figure 3.6. Line 2 defines the start memory address from where the XRs are mapped. In line 3, the compiler is informed that there is a hardware implementation for the DCT* operation with the microcode addresses for SET/EXECUTE instructions defined in lines 4-5. The *sync* instruction from Figure 3.5 is a PowerPC instruction that ensures that all instructions preceding *sync* in program order complete before *sync* completes. The sequences of *sync* and *nop* instruction are used to flush the processor pipeline. The SET instruction is not included in the above example because it is not supported (yet) by the Molen Polymorphic processor.

3.4 M-JPEG Case Study

In this case study we report the performance improvements of the Molen implementation on the Virtex II Pro for the multimedia video frame M-JPEG encoder.

The Design Flow

The design flow used in our experiments is depicted in Figure 3.7. In the target application written in C, the software developer introduces pragma annotations for the functions implemented on the reconfigurable hardware. In these annotations, the designer indicates the name of the associated hardware implementation as it is specified in the FPGA Description File. Thus, the designer has the opportunity to select from a set of such implementations, which is particularly usefull for design space exploration [53]. These selected functions are translated to Matlab and processed by the COMPAAN[54]/LAURA[55] toolchain to automatically generate the VHDL code. The commercial tools can then be used to synthesize and map the VHDL code on the target FPGA. The application is compiled with the Molen compiler and the executable is loaded and executed on the target Molen FCCM.

M-JPEG, Software and Hardware Implementations

The application domain of these experiments is the video data compressing. We consider a real-life application namely Motion JPEG (M-JPEG) encoder which compresses sequences of video frames applying JPEG compression for each frame. The M-JPEG implementation is based on the public domain implementation described in "PVRG-JPEG CODEC 1.1", Portable Video Research Group, Stanford University. The input video-frames used in these experiments are presented in Table 3.3. The resolution of the input images is relatively small, due to the memory limitations of the target Molen Polymorphic processor.

The most demanding function in M-JPEG application is 2D DCT with preshift and bound transforms (named DCT*). In consequence, DCT* is the first function candidate for hardware implementation. The only modification of the M-JPEG application that indicates the reconfigurable DCT* execution is the introduction of the pragma annotation as presented in Figure 3.7. The hardware implementation for execution of the DCT* function on the reconfigurable hardware is described in [56]. The VHDL code is automatically extracted from the DCT* application code using COMPAAN[54]/LAURA[55] tools. The Xilinx IP core for DCT and the ISE Foundation[57] are used to synthesize and map the VHDL code on the FPGA. After the whole application is compiled

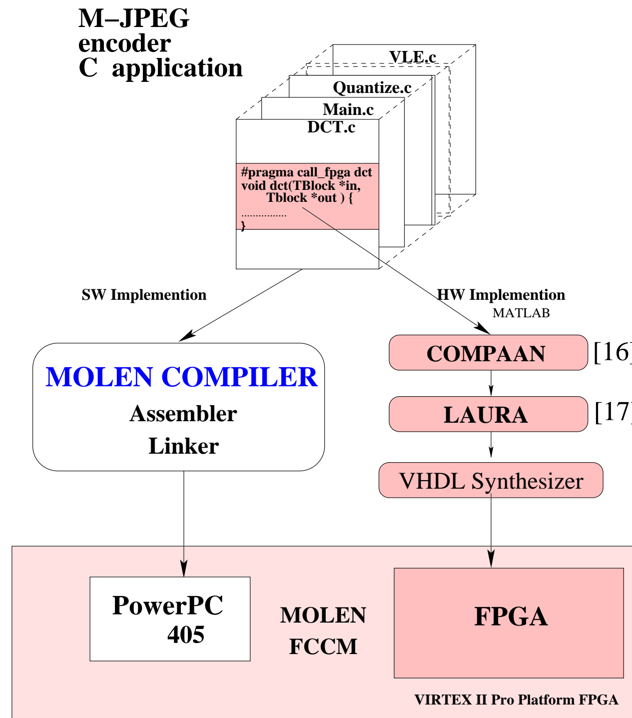

Figure 3.7: The design flow

Name	# frames	Resolution [pixels]	Format	Color/BW
tennis	8	48x48	YUV	color
barbara	1	48x48	YUV	color
artemis	1	48x48	YUV	color

Table 3.3: M-JPEG video sequences

with the Molen compiler described in the previous section, the final step is performed by the GNU assembler and linker with the C libraries included in the Embedded Development Kit (EDK) [58] from Xilinx to generate the application binary files. As previously mentioned, the target FCCM is the Molen Polymorphic processor with the IBM PowerPC 405 processor immersed into the FPGA fabric.

Performance Measurements: The current Molen implementation is a proto-

type version, which imposes the following constraints:

- the memory size for *text* and *data* sections are limited to maximum 64K. In order for the M-JPEG executable to fulfill these limitations, we rewrote the original application preserving only the essential features for compressing sequences of video frames. Moreover, these limitations also restrict the size of the input video frames to 48x48 pixels (Table 3.3, column 3).
- dynamic reconfiguration is not supported (yet) on the Molen prototype. In consequence, we could not measure the impact on performance of repetitive hardware configurations.

In addition, the performance measurements have been performed given the following additional conditions:

- the input/output operations are extremely expensive for the current Molen prototype, due to the standard serial connection implemented by UART at 38400 bps between the Molen Polymorphic processor and the external environment; this limitation can be removed by the implementation of faster I/O system. We therefore did not include the I/O operation impact in our measurements as they are not relevant for RC paradigm
- the DCT* hardware implementation requires a different format for the input data than the software implementation. Consequently, an additional data format conversion is performed in software before and after the DCT* execution on reconfigurable hardware.
- taking into account that the target PowerPC processor included in the Virtex-II Pro platform does not provide hardware floating-point support and that the required floating-point software emulation is extremely expensive, the integer DCT is used for both software and hardware implementation to allow a fair comparison.

The execution cycles for M-JPEG encoder and comparisons are presented in Table 3.4, Table 3.5 and Table 3.6. As we considered a sequence of 8 video frames for *tennis* input sequence, we present only the minimal and maximal values for each measurement in order to avoid redundant information.

Pure Software Execution: In Table 3.4, we present the results of our measurements performed on the the Virtex II Pro platform, when the M-JPEG

	tennis [0-7]		barbara	artemis
	MIN	MAX		
M-JPEG	33,671,821	33,779,813	34,014,157	34,107,893
DCT*	1,242,017	1,242,017	1,242,017	1,242,017
DCT* cumulated	22,356,306	22,356,306	22,356,306	22,356,306
Maximal improvement	66.18%	66.39%	65.73%	65.55%

Table 3.4: M-JPEG execution cycles on the PowerPC processor

application is entirely executed on the PowerPC processor. In row 1, the number of cycles used for executing the whole M-JPEG application is given. In row 2, the cycles consumed by one execution of the DCT* function are given and the next row contains the total number of cycles spent in DCT*. From these numbers, we can conclude that 66% of the total execution time is spent in the DCT* function, given the input set. This 66% represents the maximum (theoretical) improvement that can be obtained by hardware acceleration of the DCT* function. The corresponding theoretical speedup - using *Amdahl's law [59]* - is presented in Table 3.6.

Execution on the Molen Polymorphic processor: In Table 3.5, we present the number of cycles for the M-JPEG execution on the Molen Polymorphic processor. From row 1 we can conclude that an overall speedup of 2.5 (Table 3.6, row 1) is achieved. The DCT* execution on the reconfigurable hardware takes 4125 cycles (row 2) which is around 300 times less than the software based execution on the PowerPC processor (Table 3.4, row 2). However, due to the data format conversion required by the DCT* hardware implementation, the overall number of cycles for one DCT* execution becomes 102,589 (Table 3.5, row 3), resulting in a 10 fold speedup for DCT* and a 2.5 speedup globally.

The performance efficiency (defined as the ratio between the achieved speedup and the maximum speedup) is about 84% as presented in Table 3.6, last column. It is noted that this efficiency is achieved even though:

- the hardware implementation has been automatically but non-optimally obtained (using COMPAAN[54]/LAURA[55] tools)
- additional software data conversion diminished the DCT* speedup in hardware.

	tennis [0-7]		barbara	artemis
	MIN	MAX		
M-JPEG	13,443,269	13,512,981	13,764,509	13,839,757
DCT* HW	4,125	4,125	4,125	4,125
DCT* HW + Format conv.	102,589	102,589	102,589	102,589

Table 3.5: M-JPEG execution cycles on the Molen Polymorphic processor

	tennis [0-7]		barbara	artemis
	MIN	MAX		
Practical speedup	2.50	2.51	2.47	2.46
Theoretical speedup	2.96	2.98	2.92	2.90
Efficiency	84.17%	84.65%	84.70%	84.91%

Table 3.6: Comparison of M-JPEG execution cycles for SW/HW execution

From these measurements, we can conclude that even non-optimized implementation can be used to achieve considerable performance improvements. In addition, taking into account that only one function (DCT*) is executed on the reconfigurable hardware, we consider that an overall M-JPEG speedup of 2.5x from the theoretical speedup of 2.96 x confirm the viability of the presented approach.

In these experiments, the DCT hardware implementation is preloaded on the reconfigurable hardware and it is not changed during the application execution. However, for an extensive usage of the reconfigurable hardware, multiple kernels with overlapping reconfigurable areas should be considered.

3.5 Conclusions

In this chapter, we presented the Molen compiler framework with emphasis on the Molen PowerPC compiler backend we have developed for the Molen Polymorphic processor. The compiler allows the automatic translation of the application source C code using the extensions following the Molen Program-

ming Paradigm. We also presented a complete experiment that evaluated the effectively realized speedup of reconfigurable hardware execution of the DCT* function of the M-JPEG application. The generated code was executed on the Molen Polymorphic processor and showed a 2.5 speedup. This speedup consumed 84% of the total achievable speedup which amounts to 2.9. Taking into account that hardly any optimization was performed and only one function ran on the reconfigurable fabric, a significant performance improvement was nevertheless observed. We emphasize that we do not compare the RC paradigm to other approaches for multimedia applications boosting performance (such as MMX, 3DNow!, SSE) which use dedicated hardware accelerators. The focus of this chapter was rather on the compiler support for the Molen FCCM under the RC paradigm.

In the following chapters, we will investigate the specific features of the target reconfigurable architectures and propose advanced compiler optimizations and analyses to efficiently exploit the advantages of the underlying reconfigurable hardware.

Chapter 4

Dynamic SET Instruction Scheduling

The latest commercial FPGA platforms now offer support for partial and dynamic hardware configurations. Nevertheless, one of their main drawback remains the huge reconfiguration latency. In order to hide this latency, compiler support is fundamental to automatically schedule and optimize the compiled application code for efficient reconfigurable hardware usage.

When dealing with reconfigurable hardware, the compiler should be aware of the competition for the reconfigurable hardware resources (FPGA area) between multiple hardware operations during the application execution time. A new type of conflict - called in this thesis ”FPGA area placement conflict” - emerges between the hardware configurations that cannot coexist together on the target FPGA due to a predefined spatial mapping.

In this chapter, we propose a general instruction scheduling algorithm that automatically minimizes the number of required hardware configurations taking into account both the ”FPGA area placement conflicts” and the characteristics of the compiled software application. More specifically, the algorithm anticipates the hardware configurations in less frequently executed application points avoiding the ”FPGA area placement conflicts”.

The chapter is organized as follows. In the following section, we present background information and related work. In section 4.2, we describe the goals and the motivation of the proposed algorithm. A formal description of our scheduling problem is included in Section 4.3. Section 4.4 introduces the instruction scheduling algorithm. The M-JPEG case study is discussed in Section 4.5 and finally, we present conclusions and future work.

```
int fib( int n){
  int f0 = 0, f1 = 1, fn, i;

  if(n <= 1){
    fn = n;
  }
  else {
    for(i = 2; i <= n; i++){
      fn = f0 + f1;
      f0 = f1;
      f1 = fn;
    }
  }
  return fn;
}
```

Figure 4.1: C code for the computation of Fibonacci numbers

4.1 Background and Related Work

In this section, we present the basic compiler optimization background including data flow analysis and control flow graph representation that is used by the proposed instruction scheduling algorithm described in detail in Section 4.4. Additionally, we present related approaches for the proposed scheduling algorithm in the context of reconfigurable hardware usage.

4.1.1 Control Flow Graphs

A common representation of the input application used for compiler optimizations and analyses is the *control flow graph* (CFG), which is a graph that portrays all paths that might be traversed during the application execution. Each node in the graph is a *basic block*, i.e. a maximal sequence of consecutive instructions, which may be entered only at the beginning and exited only after the execution of the last instruction of the sequence. The successor/predecessor relation between the nodes of the graph reflects the control flow of the application.

As an example, we present in Figure 4.1 the C code for computing Fibonacci numbers with the associated intermediate code as generated in the Molen com-

piler depicted in Table 4.1. The control flow graph for the C program is shown in Figure 4.2. As shown in Table 4.1, the instructions are grouped in basic blocks which are the nodes of the final CFG from Figure 4.2. For example, the *for* loop from Figure 4.1 consists of B3, B4 and B5 basic blocks with B4 and B5 included in the cycle of the CFG from Figure 4.2.

The algorithm for partitioning a list of instructions in basic blocks starts with the identification of the *leader* instructions, which are the first instructions of the basic blocks. As presented in [60] and [61], a leader instruction can be:

- the entry point of a procedure
- an instruction which is a target of a branch instruction
- an instruction which immediately follows a branch or return instruction

The *leader* instructions are deliminators of the basic blocks; each basic block start with a leader instruction and includes all consecutive instructions till the next *leader* instruction (see Figure 4.1).

After the identification of the basic blocks, the control flow graph is constructed by first adding two special basic blocks: *entry* and *exit*. These two nodes are added for the simplicity of the algorithms for further optimizations and transformations. The *entry* node is connected to the initial basic block (with no predecessor) and the *exit* node is connected to the final node (with no successors). The edges of the graph are added to represent the control flow of the application. An edge (Bi, Bj) is included in the CFG if:

- there is a branch from Bi to Bj, or
- Bi and Bj are consecutive basic blocks and the final instruction of Bi is not an unconditional branch

Throughout this thesis, we denode a control flow graph as directed graph $G =< N, E >$, with N the set of nodes, $entry \in N$, $exit \in N$, and $E \subseteq N \times N$. An edge from a node n to a node m is denoted as (n,m). Further, we define the set of successors and predecessors as follows:

$Succ(n) = \{m \in N \mid \exists(n, m) \in E\}$

$Pred(n) = \{m \in N \mid \exists(m, n) \in E\}$

B0	**entry**
B1	**mrk 2, 1** ldc $vr0.s32 ← 0 mov fib.f0 ← $vr0.s32 ldc $vr1.s32 ← 1 mov fib.f1 ← $vr1.s32 ldc $vr2.s32 ← 1 bgt fib.n,$vr2.s32,fib._FibonacciTmp7
B2	**mrk 2, 5** mov fib.fn ← fib.n jmp fib._FibonacciTmp8
B3	**fib._FibonacciTmp7:** mrk 2, 8 ldc $vr3.s32 ← 2 mov fib.i ← $vr3.s32
B4	**fib._FibonacciTmp5:** bgt fib.i,fib.n,fib._FibonacciTmp4
B5	**mrk 2, 9** add $vr4.s32 ← fib.f0,fib.f1 mov fib.fn ← $vr4.s32 mrk 2, 10 mov fib.f0 ← fib.f1 mrk 2, 11 mov fib.f1 ← fib.fn mrk 2, 8 mov fib._FibonacciTmp6 ← fib.i ldc $vr6.s32 ← 1 add $vr5.s32 ← fib._FibonacciTmp6,$vr6.s32 mov fib.i ← $vr5.s32 jmp fib._FibonacciTmp5
B6	**fib._FibonacciTmp4:** **fib._FibonacciTmp8:** mrk 2, 14 ret fib.fn
B7	**exit**

Table 4.1: MIR intermediate code for the C code from Figure 4.1

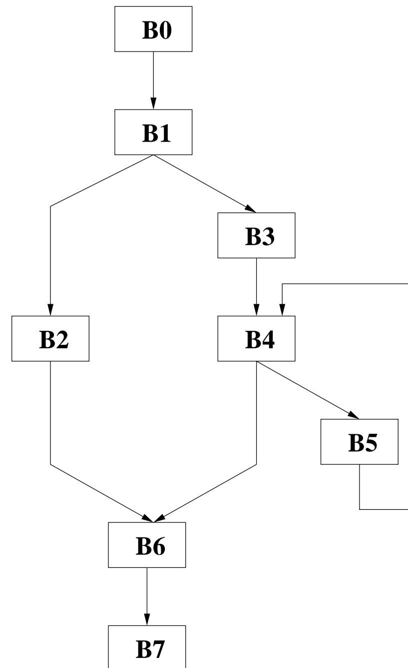

Figure 4.2: Control flow graph for the C code from Figure 4.1

4.1.2 Data Flow Analyses

A large number of compiler optimizations require the examination of the entire program. For example, when an assignment of a variable (without side effects) is not used further in the application, it can be safely removed. In order to determine if the variable is used later, the whole application must be analyzed.

A typical example for data flow analysis is the well-known reaching definition problem. A definition *d* of a variable *v* reaches a point *p* in the application if:

- there is a path from *d* to *p*
- variable *v* is not redefined on the path from *d* to *p* (also expressed as *d* is not killed).

The first step of computing the reaching definitions is to identify the definitions of the target application and to associate them to a unique identifier. In Figure 4.3, we present a simplified version of the control flow graph from Figure 4.2 and Table 4.1, with the nine definitions represented as $d_1 d_2 ... d_9$.

	GEN		KILL	
B0	-	000000000	-	000000000
B1	$d_1d_2d_3$	111000000	d_7d_8	000000110
B2	d_4	000100000	d_6	000001000
B3	d_5	000010000	d_9	000000001
B4	-	000000000	-	000000000
B5	$d_6d_7d_8d_9$	000001111	$d_2d_3d_4d_5$	011110000
B6	-	000000000	-	000000000
B7	-	000000000	-	000000000

Table 4.2: GEN and KILL sets for the CFG from Figure 4.3

In the next step, the local information associated with each basic block is computed. More specifically, for the considered problem, we determine two sets of elements, namely GEN and KILL. The GEN set (generated definitions) for basic block B contains all the definitions included B which reach the end of the basic block. The KILL set (killed definitions) of a basic block B includes all the definitions outside B that redefine identifiers already defined in B.

For an efficient computation of these sets and of additional data flow information, the sets of definitions (or application objects, for the general case) are represented as bit vectors. In many dataflow analyses, bit representation (0/1) is a powerfull mechanism that efficiently represents the required informations. In the considered example, we assume that the definition d_i is associated with bit i from a bit vector $b_1b_2...b_n$. The GEN and KILL sets for the CFG from Figure 4.3 are presented in Table 4.2.

The next step is the propagation of the local information in the CFG using the following data flow equations for each basic block i:

$$RDout(i) = GEN(i) \cup (RDin(i) - KILL(i)) \tag{4.1}$$

$$RDin(i) = \bigcup_{j \in Pred(i)} RDout(j) \tag{4.2}$$

$$RDin(entry) = \emptyset \tag{4.3}$$

The above equations are used for the computations of two sets of elements RDout and RDin which reflect the reaching definition for the entire procedure. Thus, a definition is reaching the output of a basic block i if (see Equation 4.1):

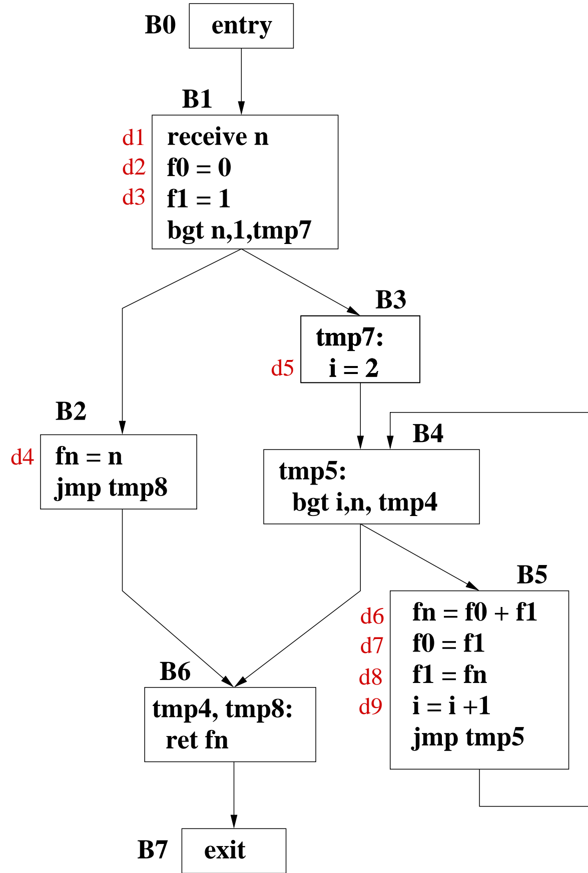

Figure 4.3: Schematic CFG for the C code from Figure 4.1

- it is generated by the basic block i, or
- it is reaching the input of basic block i and it is not killed in the basic block i.

From Equation 4.2, a definition is reaching the input of a basic block i if it is reaching the output of one of the predecessors of i. In the initialization phase (see Equation 4.3), we assume that there is no definition that reaches the entry node.

For solving the above equations, the most frequently used method is the iterative computation, where the input and output sets (RDin and RDout) are iteratively computed till they converge to a fixed set of values, which will not

	Iteration 1		Iteration 2	
	IN	OUT	IN	OUT
B0	000000000	000000000	000000000	000000000
B1	000000000	111000000	000000000	111000000
B2	111000000	111100000	111000000	111100000
B3	111000000	111010000	111000000	111010000
B4	111010000	111010000	111011111	111011111
B5	111010000	100001111	111011111	100001111
B6	111110000	111110000	111111111	111111111
B7	111110000	111110000	111111111	111111111

Table 4.3: RDin and RDout sets for the CFG from Figure 4.3

change in the following iterations. For the reaching definition problem, the results of each iteration are presented in Table 4.3. For the considered example, only two iterations are required to reach the fixed set of values. Based on these results, we can conclude that all definitions can reach the final node B7.

4.1.3 Related Work

As presented in the previous chapter, the code generated by the Molen compiler for a hardware operation (an operation performed on the reconfigurable hardware) includes i) parameter passing, ii) the SET instruction, iii) the EXECUTE instruction and iv) returning the computed results. This sequence of instructions where the SET instruction is immediately followed by the associated EXECUTE instruction is referred to in the rest of this thesis as ”simple scheduling”.

In [62], it has been reported that this simple scheduling produces significant performance decrease due to the huge reconfiguration latency of current FPGA. Many approaches for reducing the reconfiguration latency are hardware approaches, such as coarse grain reconfigurations or multi context FPGAs (see [27]) and hardware configuration prefetching (see [63] [64]). However, the reconfigurable hardware market is still mainly dominated by FPGAs and there are few approaches that address the reconfiguration overhead issue at compiler level. In order to deal with this drawback, an instruction scheduling algorithm has been proposed in [65] for a particular case when there is only one hardware operation in the whole application. The main idea is to move the SET instructions outside loops in order to eliminate redundant hardware configurations.

In order to achieve significant performance improvement for real applications, more than one operation is usually implemented in hardware. As the available area of the reconfigurable platforms is limited, the coexistence of all hardware configurations on the FPGA for all application execution time may be restricted. Moreover, hardware implementations of these operations can be developed by different IP providers that can impose a predefined FPGA area allocated for each operation, resulting ”FPGA-area placement conflicts”. Two hardware operations have an ”FPGA-area placement conflicts” (or just conflict in the rest of the thesis) if i) their combined reconfigurable hardware area is larger than the total available FPGA area or ii) the intersection of their hardware areas is not empty. In Figure 4.4(a) we sketch a possible FPGA area allocation for three operations performed on the FPGA. We observe that op1 and op2 cannot fit together on the FPGA (thus op1 conflicts with op2) while op2 and op3 have a common overlapping area (thus op2 conflicts with op3).

In [66], a run-time heuristic scheduling algorithm is proposed for applications with deterministic behaviour. Such scheduling requires detailed information for all tasks (including all software tasks) of the application and imposes limitations due to the deterministic behaviour of the target application. A compiler approach that considers the restricted case of two consecutive and non-conflicting hardware operations is presented in [67]. In this approach, the hardware execution of the first operation is scheduled in parallel with the hardware configuration of the second operation. Our approach is more general as it performs scheduling for any number of hardware operations at procedural level and not only for two consecutive hardware operations. The performance gain produced by our scheduling algorithm results from reducing the number of performed hardware configurations. The proposed algorithm is similar to the standard compiler optimization for partial expression redundancy elimination presented in [68], with emphasis on the ”FPGA-area placement conflicts” between the hardware operations, as described in Section 4.4.

4.2 Motivation

Figure 4.4(b) shows the control-flow graph of a procedure, when op1, op2 and op3 operations are performed on the reconfigurable hardware and they are placed on the FPGA as presented in Figure 4.4(a). The numbers associated with each edge of the graph represent the execution frequency of the edge. One first observation is the redundant repetitive execution of SET op1 instruction from B5 in the loop B4-B5-B6. Additionally, it should be noticed that

moving this SET op1 instruction on (B1, B2) edge will also make redundant the SET op1 instruction from B13. In the initial simple scheduling, the FPGA is configured for op1 100 times in B5 and 10 times in B13. As a result of our scheduling algorithm, the hardware configuration for op1 will be executed only 20 times. The hardware configuration for op2 from B10 cannot be moved further then B7, as it will change the hardware configuration for op3 that must be performed in B7. There are no redundant configurations for op3, thus the hardware execution of op3 has to be preceded each time by the hardware configuration. When the hardware configuration consumes all the performance gain produced by the hardware execution of op3, the scheduler can switch to its software execution on the GPP (General-Purpose Processor).

In this chapter, we propose a general approach for intraprocedural instruction scheduling of the hardware configuration instructions taking into account the ”FPGA-area placement conflicts”. It is based on the state-of-art compiler optimization for partial expression redundancy elimination presented in [68]. In order to incorporate the ”FPGA-area placement conflicts” between the hardware operations, we introduce a new data-flow analysis as described in Section 4.4. Additionally, it can switch for one operation from hardware execution to its software execution when the hardware operation provides no performance improvement even after the scheduling phase.

4.3 Problem Statement

We represent the control flow graph of a procedure as a directed graph G $< N, E, w >$ where the nodes N represent the basic blocks, the edges E represent the control flow dependencies and the weight function w: E $\rightarrow R^+$ represents the execution frequency of each edge. The operations implemented in hardware are included in HW set. We define DEF_{op} the set of basic blocks n $\in$ N that contain an instruction *SET op* immediately followed by *EXEC op* instruction - the input graph G contains the SET and EXECUTE instructions for an operation in consecutive order according to the simple scheduling. A node n $\in DEF_{op}$ is called a definition node for op. In our example from Figure 4.4, B5 and B13 are definition nodes for op1.

An ”FPGA-area placement conflict” between two operations op1 and op2 is represented as op1 $\not\leftrightarrow$ op2. The information about these conflicts is provided by a symmetric function f : HW x HW $\rightarrow$ $\{$0,1$\}$, where f(op1, op2) = 1 if op1 $\not\leftrightarrow$ op2, and 0 otherwise. We define the set of conflicting nodes for an

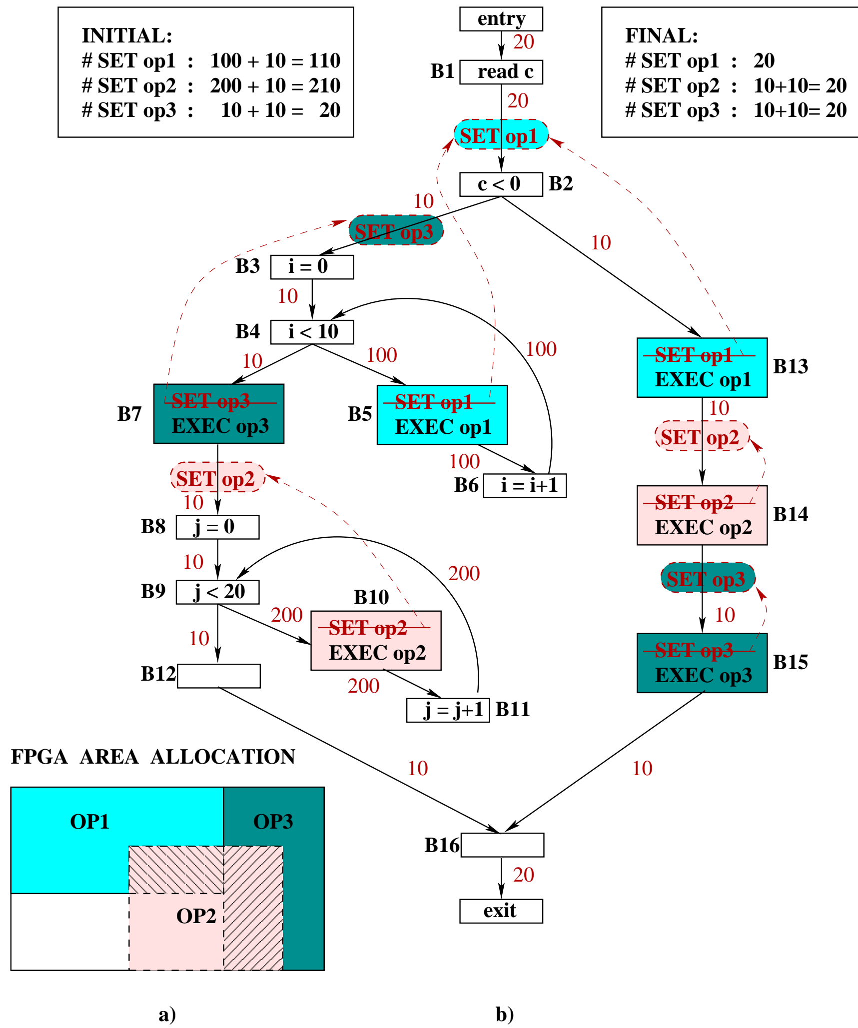

Figure 4.4: Motivational example for instruction scheduling of hardware configurations (b) with FPGA-area placement constraints (a)

operation op as follows:

$$Conflict_{op} = \{n \in N | \exists op_i \in HW, n \in DEF_{op_i} \wedge op \not\leftrightarrow op_i\}. \tag{4.4}$$

A node n $\in Conflict_{op}$ is called a conflict node for op. In Figure 4.4, B10 and B14 are conflict nodes for both op1 and op3.

In order to simplify this discussion, we make the following assumptions. We assume that there is a single **entry** node with no predecessor (pred(entry) = $\emptyset$, where pred(n)=$\{$m $\in$ N | (m,n) $\in$ E$\}$) and a single **exit** node with no successor (succ(n) = $\emptyset$, where succ(n)=$\{$m $\in$ N | (n,m) $\in$ E$\}$). Also, we assume that a node cannot be simultaneous in DEF_{op} and $Conflict_{op}$. In consequence, when more conflicting operations are included in the same basic node, this node must be split into a set of nodes, one for each operation. The final assumption is that only the SET/EXECUTE instructions included in the CFG affect the reconfigurable hardware.

For each operation op, we consider a set of insertion edges $\delta_{op} \subseteq E$. The merit of δ_{op} is measured by the function $W_\delta = \sum_{e \in \delta_{op}} w(e)$. Loosely stated, the objective of our algorithm is to move upwards the SET instructions from DEF_{op} on less frequently executed edges, in order to reduce the total number of performed SET instructions. A formal description of this problem is as follows:

PROBLEM *Given a directed, weighted graph G $< N, E, w >$ and a set of hardware operations HW, each defined in $DEF_{op} \subseteq N$ and with conflicts in $Conflict_{op} \subseteq N$, find a set of insertion edges $\delta \subseteq E$ for each op $\in$ HW which minimizes W_δ under the following constraints:*

- *$\forall$ n $\in DEF_{op}$, for all paths from* **entry** *to n, there is an insertion edge $(u, v) \in \delta$,*
- *$\nexists$ k $\in Conflict_{op}$ such that k is included in any subpath from v to n*

The minimization of W_δ guarantees that a smaller or equal number of SET instructions will be performed in the final CFG graph than in the input graph. The first constraint reflects the requirement that hardware configuration (the SET instruction) must precede the hardware execution (the EXEC instruction) on all paths. The second constraint assures that no conflict operation will change the hardware configuration before the operation execution.

4.4 Instruction Scheduling Algorithm

The problem of removing redundant hardware configurations is similar to the well-known problem of removing redundant expressions. As hardware configurations do not cause any exception, we can use an aggressive speculative scheduling for the hardware configurations in order to anticipate them on less frequently executed paths and thus, to make redundant the hardware configurations from frequently executed paths. We introduce the scheduling algorithm that solves the problem defined in the previous section in three steps. In the first step, the subgraphs where the hardware configurations can be anticipated are constructed. Next, a minimum s-t cut algorithm is applied to find the optimal insertion edges δ_{op} for each hardware operation. Finally, a switch from hardware to software execution is introduced for the cases when the expense of hardware configurations in the newly inserted nodes still outperforms the performance gain of hardware execution.

4.4.1 Step 1: The Anticipation Subgraph

Constructing the anticipation graph is a key step in our algorithm. The main goal is to eliminate from the initial graph the edges that cannot propagate upwards the SET instructions due to hardware conflicts. This step contains two uni-directional data-flow analyses and one pass for constructing the anticipation subgraph by removal of non-essential edges.

Partial Anticipability

A hardware configuration for operation op is partially anticipated in a point m if there is at least one path from m to the exit node that contains a definition node for op and none of the paths from m to the first such definition node contains a conflict node for op.

A confluence conflict node n is a node with two successors s1 and s2 such that op1 is partially anticipated at the entry point of s1, op2 is partially anticipated at the entry point of s2 and op1 $\nleftrightarrow$ op2. Due to hardware conflicts, op1 and op2 cannot be both anticipated in the confluence conflict node n. We consider a restricted partial anticipability analysis where the confluence conflict nodes limit the partial anticipability for both op1 and op2. This is a backward data-flow problem, where the data-flow equations for a basic block i are defined as follows:

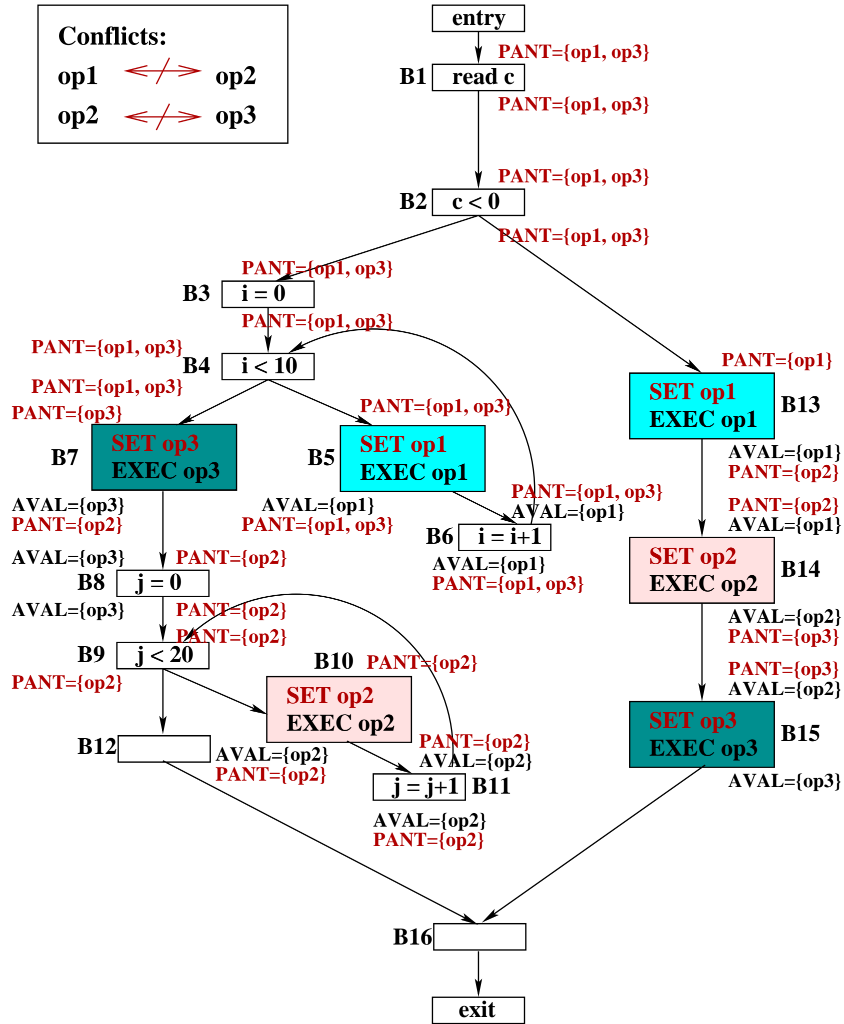

Figure 4.5: Set of PANT and AVAL values for the input graph from Figure 4.4

$$PANTin(i) = Gen(i) \cup (PANTout(i) - Kill(i)) \quad (4.5)$$

$$PANTout(i) = \biguplus_{j \in Succ(i)} PANTin(j) \quad (4.6)$$

$$PANTout(exit) = \emptyset \quad (4.7)$$

In the first equation, Gen(i) is the set of hardware operations generated in the basic block i. A hardware operation op1 is generated in a basic block i if i $\in DEF_{op}$. The set Kill(i) includes all hardware operations that are in conflict with the operations generated in the basic block i.

According to Equation 4.5, a hardware operation op $\in$ PANTin(i) is partially anticipated at the *entry* of a basic block i if it is generated in i or if it is partially anticipated at the exit of i and it is not killed in i.

The second equation differs from standard data-flow equations involved in iterative data-flow analysis where the join operator is $\bigcup$ or $\bigcap$. The operator $\biguplus$ is a conditional reunion that excludes the conflicting hardware operations and defined as follows:

$A \biguplus B = \{x \in A \bigcup B | \ \nexists y \in A \bigcup B, x \nleftrightarrow y\}$

This operator is used to stop the partial anticipability of the operations with hardware conflicts at confluence points. According to 4.6, a hardware operation op $\in$ PANTout(i) is partially anticipated at the exit of a basic block i if it is partially anticipated at the entry of any successor of i and i is not a conflict confluence node for op. In Figure 4.5, we present the values for PANT for the input graph presented in Figure 4.4. Based on these values, we can determine that the SET instructions for op1 and op3 can be safely anticipated till B1. For the basic blocks where these values are missing, there are implicitly assumed as $\emptyset$.

Availability

A hardware configuration for operation op is available at a point p if every path from the entry node to p contains a definition node for op and after this node, there is no conflict node prior to reaching p. We use the standard forward data-flow analysis for availability described by the following set of data-flow equations:

$$AVALout(i) = Gen(i) \cup (AVALin(i) - Kill(i)) \tag{4.8}$$

$$AVALin(i) = \bigcap_{j \in Pred(i)} AVALout(j) \tag{4.9}$$

$$AVALin(entry) = \emptyset \tag{4.10}$$

The Gen and Kill sets are the same as in the previous data-flow problem for partial anticipability. As the availability problem is related to the paths from the entry node, it is expressed as a *forward* data-flow analysis. According to Equation 4.8, an operation is available at the exit of a basic block i if it is generated in the basic block i or it is available at the input and it is not killed in the basic block i. Subsequently, an operation is available at the input of a basic block i if it is available at the exit of all predecessors of basic block i, as expressed in equation 4.9.

This analysis is used to eliminate the hardware configurations when they are already available. The values for AVAL for our example graph are presented in Figure 4.5. For example, we notice that in B14, op1 is available at the input but not at the exit of B14, as it is killed by the hardware configuration of the conflicting operation op2.

Constructing the Anticipation Graph

Based on the previously presented data-flow analysis results, for each operation op $\in$ HW we eliminate from the initial graph the nodes which are not essential as follows. We call an edge (u,v) an essential edge for op if

$$Ess(u, v) = \{(u, v) \in E | op \notin AVALout(u) \wedge op \in PANTin(v)\}. \tag{4.11}$$

The reduced graph G_{rd} contains the nodes

$N_{rd} = \{n \in N | \exists m \in N, Ess(n, m) \vee Ess(m, n)\}$

and the edges

$E_{rd} = \{(u, v) \in E | Ess(u, v)\}$.

By construction, the reduced graph may contain a set of disconnected subgraphs. In order to connect them, we introduce a new pseudo entry node (called s) and a pseudo exit node (called t) and the edges $E_{st} = \{(s, n) | n$

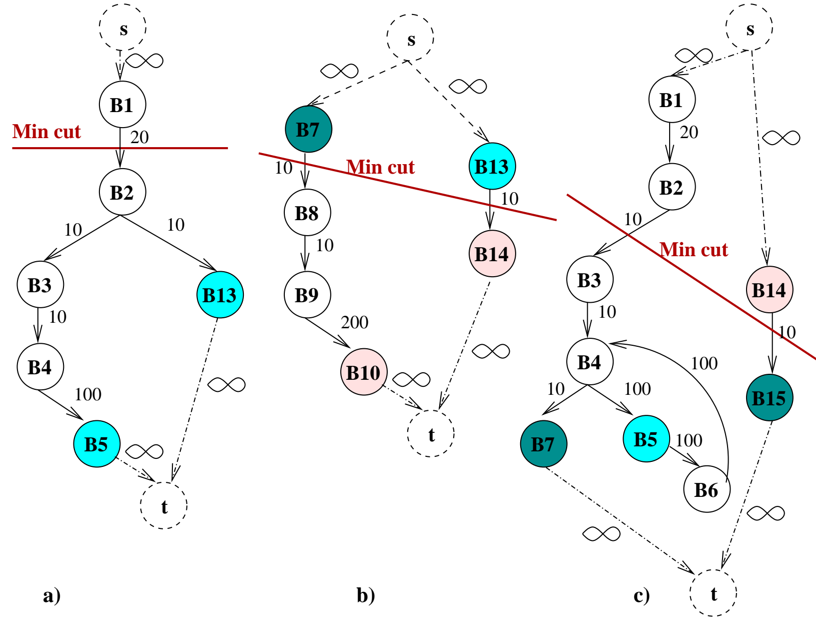

Figure 4.6: The anticipation graph for op1 (a), op2 (b) and op3 (c) from Figure 4.4

has no predecessor in $N_{rd}\} \bigcup \{(n,t)|n$ has no successor in $N_{rd}\}$ with infinite execution frequency, in order to guaranty that the newly introduced edges will not be included in the final set of insertion edges. For our example from Figure 4.4, the anticipation graphs are presented in Figure 4.6.

4.4.2 Step 2: Minimum s-t Cut

In this step, the set of insertion edges from our problem definition is determined by applying a minimum s-t cut algorithm. The purpose of the min cut algorithm is to select the less frequently executed edges from the anticipation graph on all paths to the definition nodes. In consequence, the min cut algorithms assures the minimization requirement and the first constraint from our problem definition, while the construction of the anticipation graph secures the second constraint.

One of the important advantages of using a min cut algorithm is to avoid moving upwards SET instructions on edges inside loops. In our optimization, we used Edmonds-Karp minimum s-t cut algorithm[69] from Mascopt [70]. For the three hardware operations from Figure 4.4, their minimum cuts are pre-

sented in Figure 4.6. We notice that for op3 (depicted in Figure 4.6 (c)), the SET instruction from B7 can propagate further then B2 (on edge (B1, B2)). The minimum cut algorithm chooses the edge (B2, B3) as its execution frequency is smaller (10 versus 20 for (B1,B2)).

4.4.3 Step 3: Selection of Software/Hardware Execution

In the cases when, even after our scheduling, the hardware configuration and execution is more expensive than the pure software execution, the scheduling algorithm can switch for this operation from hardware execution to software execution. In this case, all the SET instructions for this operation are eliminated and its EXECUTE instructions are replaced by standard calls to the associated software function. In our example from Figure 4.4, op3 may be in this case if one hardware configuration and one hardware execution is more expensive than one software execution.

In order to determine which operations are switched to software execution, the compiler reads the needed information about software/hardware features of each operation from a special file, named FPGA description file. The relevant information include the reconfiguration overhead, FPGA area, software/hardware latency, execution frequency, etc.

4.5 M-JPEG Case Study

The presented instruction scheduling algorithm has been implemented as a MachSUIF pass [43] within the Molen compiler which generates code for the Molen Polymorphic processor implemented on the Virtex II Pro FPGA platform. The overall time-complexity of the scheduling algorithm is mainly determined by the two data-flow analyses and the min s-t cut algorithm. For the data-flow analyses, the time complexity in the worst-case is $O(|N| * (b + 2))$, where b is the maximum number of back edges on any acyclic path in the graph G. The time complexity of the Edmonds-Karp algorithm is $O(|N| * |E|^2)$, thus the proposed scheduling algorithm has an overall polynomial time complexity. The compilation time spent in the scheduling phase for the considered applications is negligible (around 1 s) compared to the overall compilation time.

The target C application of this case study is the multimedia benchmark Motion JPEG (M-JPEG) encoder and the input sequence contains 30 color frames from ”tennis” in YUV format with a resolution of 256x256 pixels. The operations performed on the FPGA are *DCT* (2-D Discrete Cosine Transform),

	HW Execution			SW Execution	
Op Name	EXEC [cycle]	Area [slice]	SET [cycle]	One call [cycle]	%Total M-JPEG
DCT	416	848	431771	44396	80 %
Quant	73	397	202073	1494	3 %
VLC	272	193	98237	6921	12.5 %

Table 4.4: HW/SW features for the operations that candidate for hardware implementation

Quantization and *VLC* (Variable Length Coding). The Xilinx IP cores for DCT [71], Quantization [72] and VLC [73] are used for the estimations of the hardware implementations. The GPP included in the Molen prototype is the IBM PowerPC 405 processor at 250 MHz.

We present in Table 4.4 the characteristics of DCT, Quantization and VLC hardware and software executions. Based on the characteristics of the XC2VP20 chip, for which a complete configuration of 9280 slices takes about 20 ms, we estimated the configuration time for each operation (Table 4.4, column 4) in terms of PowerPC processor cycles.

The profiling results for the software execution from Table 4.4 are based on simulations using the PowerPC simulator from Simics [74]. Comparing the values from Table 4.4 (column 4 and 5), we notice that the hardware configuration alone is about 10 times more expensive than the complete software execution. Using Amdahl's law[59], we determine that the simple scheduling (as described in Section 4.1) for DCT will slowdown the M-JPEG benchmark up to 10x. For this reason, we compare the performance of our scheduling algorithm to the pure software approach rather than the *inefficient* simple scheduling.

The estimated performance for the M-JPEG application for different possible conflicts between the three hardware operations are presented in Figure 4.7. The standard unit of this comparison is the pure software execution (SW) when the M-JPEG benchmark is completely performed on the GPP alone. The performance of our instruction scheduling algorithm for the real Xilinx hardware implementations is denoted as REAL. As recently some hardware approaches [75] have been proposed for reducing the hardware configuration time, we also analyze the impact of our scheduling algorithm when the hardware configuration is accelerated by a factor of 20x[1] compared to the current values from

[1] The factor has been chosen arbitrarily. Mutatis mutandis, similar observations will then

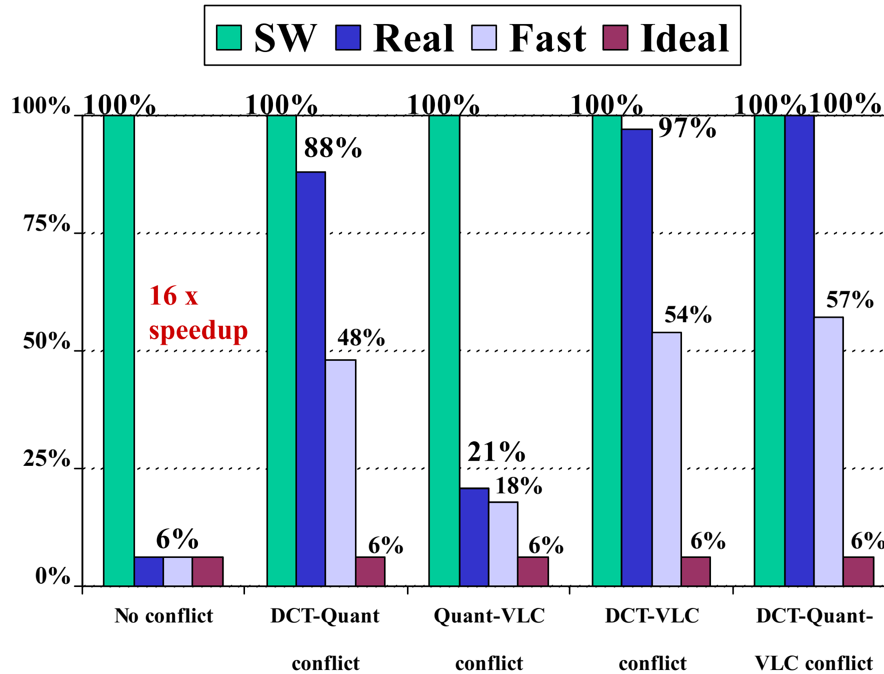

Figure 4.7: Comparison of estimated performance for our scheduling algorithm for M-JPEG benchmark

Table 4.4, column 4. The performance of our instruction scheduling algorithm combined with this faster hardware configuration is presented in Figure 4.7 as FAST. For completeness, we also present the IDEAL case when the hardware configuration is performed in zero cycles.

We notice that for the ”no conflict” case, the performance improvement is about 94 % (equivalent to a 16x speedup) for both REAL and FAST scheduling and very close to the IDEAL performance. In this case, the instruction scheduling algorithm moves the hardware configurations for all three operations at the procedure entry point. In consequence, there is only one hardware configuration for each hardware operation, thus the difference between REAL and FAST is negligible.

For the rest of the ”conflict” cases, the scheduler for REAL will switch from hardware execution to software execution for the conflicting operations. For example, when there is a DCT - Quantization conflict, the scheduler will move

hold.

both DCT and Quantization operation in software, while the third non conflicting operation VLC remains in hardware; its hardware configuration needs to be performed only once, at the procedure entry point.

For the FAST scheduling, even when one operation has a conflict, it may remain in hardware, thanks to the 20x faster hardware configuration. For the case with DCT - Quantization - VLC conflicts, both DCT and VLC are performed in hardware and produce a performance improvement of 43 % as the fast hardware configuration does not consume all performance gain of the hardware execution. The scheduler selects the software execution for Quantization, in order to prevent a performance decrease produced by its hardware configuration and execution (16 % for Quantization). Therefore, the performance improvement for simple scheduling (all operations executed on the reconfigurable hardware) and 20x faster reconfigurations is 27 % while our scheduling algorithm contributes to a performance improvement between 43 % and 94 % compared to SW.

In consequence, we notice that for the non-conflict case, our algorithm capitalizes the maximum performance gain that can be obtained by hardware execution of the considered operations. Finally, the results presented in Figure 4.7 emphasize the important performance impact of our scheduling algorithm even for the future faster FPGAs.

4.6 Conclusions

In this chapter, we have introduced a general scheduling algorithm for hardware configuration instructions. This algorithm takes into account specific features of the reconfigurable hardware such as the ”FPGA area placement conflicts” and the reconfiguration latencies of each hardware operations. Based on the characteristics of the compiled application, the scheduling reduces the number of performed hardware configurations preserving the application semantics. It combines advanced compiler techniques (such as data flow analyses) with powerful graph theory algorithms (min s-t cut).

The results of our case study show that the performance is dramatically improved by using our scheduling algorithm, and this improvement will hold for future faster FPGA platforms. In the next chapter, we propose an extension of the presented scheduling algorithm where the anticipation of the hardware configuration instructions is applied at the interprocedural level.

Chapter 5

Interprocedural SET Scheduling

As presented in the previous chapter, the potential speedup of the kernel hardware executions can be completely wasted by inappropriate repetitive hardware reconfigurations. In this chapter, we thoroughly investigate the impact on the overall performance of hardware reconfiguration overhead and present an interprocedural optimization that extend the anticipation of the hardware configuration instructions at the interprocedural level. More specifically, we study and compare four cases: a) the pure software approach when the whole application is executed only on the GPP; b) the simple scheduling of hardware configurations when each hardware execution is preceded by the corresponding hardware configuration; c) the execution of only one function on the reconfigurable hardware when only a single hardware configuration is needed and d) the proposed optimization that anticipates the hardware reconfiguration instructions.

The chapter is organized as follows: in the next section, we present a motivational example to illustrate the proposed optimization and its results for a real application. The interprocedural optimization algorithm is described in details in Section 5.2. Consequently, we present a profiling experiment and analyze the impact on performance of the hardware reconfiguration for the MPEG 2 benchmark for the considered study cases. Finally, we conclude with Section 5.4.

5.1 Motivational Example

In order to illustrate the goals and the main features of the proposed interprocedural optimization, we present in Figure 5.1 a motivational real example. The presented subgraph is included in the call graph of the MPEG2 encoder multimedia benchmark where an edge $< p_i, p_j >$ represents a call from procedure p_i to procedure p_j. We consider that the procedures SAD, DCT and IDCT are executed on the reconfigurable hardware and that initially the hardware configuration (a SET instruction) is performed before each hardware execution (an EXEC instruction), according to the simple scheduling presented in Section 4.2.

One first observation is that the configuration for the SAD operation can be safely anticipated in the **motion_estimation** procedure. This anticipation will significantly reduce the number of performed hardware configurations as it will not be performed for each macroblock but only for each frame of the input sequence. This observation also holds for the DCT configuration in **transform** and the IDCT configuration in **itransform**. Moreover, the SAD configuration from **motion_estimation** can be moved upwards in the **putseq** procedure, immediately preceding the call site of **motion_estimation** in **putseq**. Additionally, it can be noticed that the propagation of the SAD configuration from **putseq** to the **main** procedure depends on the FPGA area allocation for SAD, DCT and IDCT. When the SAD operation does not have any *FPGA-area placement conflict* (see Section 4.2) with the other two hardware operations DCT and IDCT, its configuration can be safely performed only once, at the entry point in the **main** procedure.

The optimization proposed in this chapter allows to anticipate the hardware configurations at interprocedural level, while prior work was limited to optimizations at procedural level (intraprocedural). Secondly, although the interprocedural optimizations are considered to provide little benefit and significantly increase the compiler complexity, we show that our optimization significantly reduces the number of hardware configurations (a major drawback of the current FPGAs), while the complexity is not significantly increased.

5.2 Interprocedural SET Optimization

The main goal of the proposed interprocedural optimization presented in this section is to appropriately schedule the SET instructions taking into account the hardware conflicts between the available hardware operations. As such

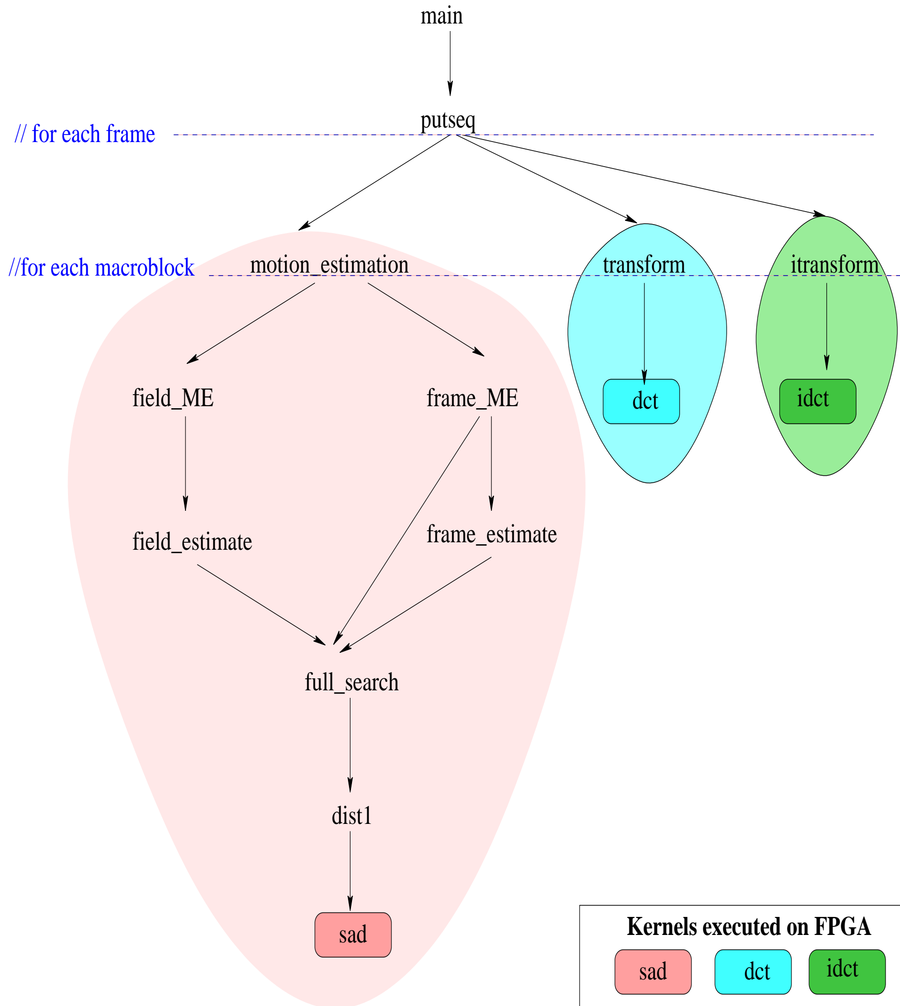

Figure 5.1: Motivational example for MPEG2 encoder

Interprocedural Optimization Algorithm **INPUT:** Call graph $G =< N, S, r >$, hardware conflicts $f : HWxHW- > \{0, 1\}$ **OUTPUT:** Insertion edges L
1. //Verify assumptions for G check if G is DAG 2. //RMOD computation traverse G in reverse topological order compute for each procedure p $RMOD(p) = LRMOD(p) \bigcup_{s \in Succ(p)} RMOD(s)$ //Compute CF for each procedure p $CF(p) = \{op_1 \in RMOD(p) \mid \exists op_2 \in RMOD(p), op_1 \not\leftrightarrow op2\}$ 3. //Compute the insertion edges $L = \emptyset$ for each edge $< p_i, p_j >$ for each $op \in [RMOD(p_j) - CF(p_j)] \cap CF(p_i)$ $L = L \cup < p_i, p_j, op >$ for each $op \in [RMOD(r) - CF(r)]$ $L = L \cup < r, r, op >$

Table 5.1: The interprocedural optimization algorithm for hardware configuration instructions

hardware configuration does not cause an exception, a speculative algorithm is used for scheduling the hardware configuration instructions. As shown in Table 5.1, the interprocedural optimization consists of three steps. In the first step, the application's call graph is constructed based on an interprocedural control-flow analysis. Next, the set of live hardware configurations for each procedure is determined using an interprocedural data-flow analysis. Finally, the hardware configuration instructions are anticipated in the call graph taking into account the available conflicting operations.

5.2.1 Step 1: Construction of the Call Graph

Starting point of the proposed optimization is the construction of the application's call graph. Given an application *P* consisting of a set of procedures $< p_1, p_2, ..., p_n >$, the application's call graph of P is the graph $G =< N, S, r >$ with the node set $N = \{p_1, p_2, ..., p_n\}$, the set $E \subseteq N$ x N, where $< p_i, p_j > \in E$ denotes a call site in p_i from which p_j is called,

and the distinguished entry node $r \in N$ representing the main entry procedure of the application. An example of a real call (sub)graph is presented in Figure 5.1.

The construction of the call graph for a application written in C is straightforward as there are no higher-order procedures in the C programming language. For this purpose, we used the *sbrowser_cg* library included in the *suifbrowser* package available in the SUIF environment. We also used *link_suif* pass already available in SUIF2 in order to link all SUIF files of the initial application in one large SUIF file, which is used for the construction of the call graph. The constructed call graph is the input of the optimization algorithm as presented in Table 5.1. As explained in the next subsection, the constructed graph is required to be a DAG (Directed Acyclic Graph) (see Table 5.1, step 1).

5.2.2 Step 2: Propagation of Hardware Configuration Instruction

The goal of the interprocedural data-flow analysis is to determine what hardware operation can modify the FPGA configuration as a side effect of a procedure call. We define LRMOD(p) (Local Reconfigurable hardware MODification) as the set of hardware operations associated with a procedure p. In order to simplify this discussion, we assume that there is at most one hardware operation that can be associated with a procedure. More specifically, $op_1 \in LRMOD(p)$ if there is a pragma annotation that indicates that procedure p is executed on the reconfigurable hardware and its associated hardware operation is named op_1. $RMOD(p)$, Reconfigurable hardware MODification, represents the set of all hardware operations that may be executed by an invocation of procedure p and it can be computed using the following data-flow equation:

$$RMOD(p) = LRMOD(p) \bigcup_{s \in Succ(p)} RMOD(s) \tag{5.1}$$

An hardware operation op may be performed by calling procedure p if op is associated with procedure p (i.e. $op \in LRMOD(p)$) or if it can be performed by a procedure that is called from procedure p. For an efficient computation, the RMOD values should be computed in reverse topological order (i.e. reverse invocation order) when the call graph does not contain cycles (see step 2 from Table 5.1). The RMOD values for the example presented in Figure 5.1 are shown in Figure 5.2. For the basic blocks where LRMOD values are missing, they are implicitly assumed as $\emptyset$. We notice that by calling $putseq$

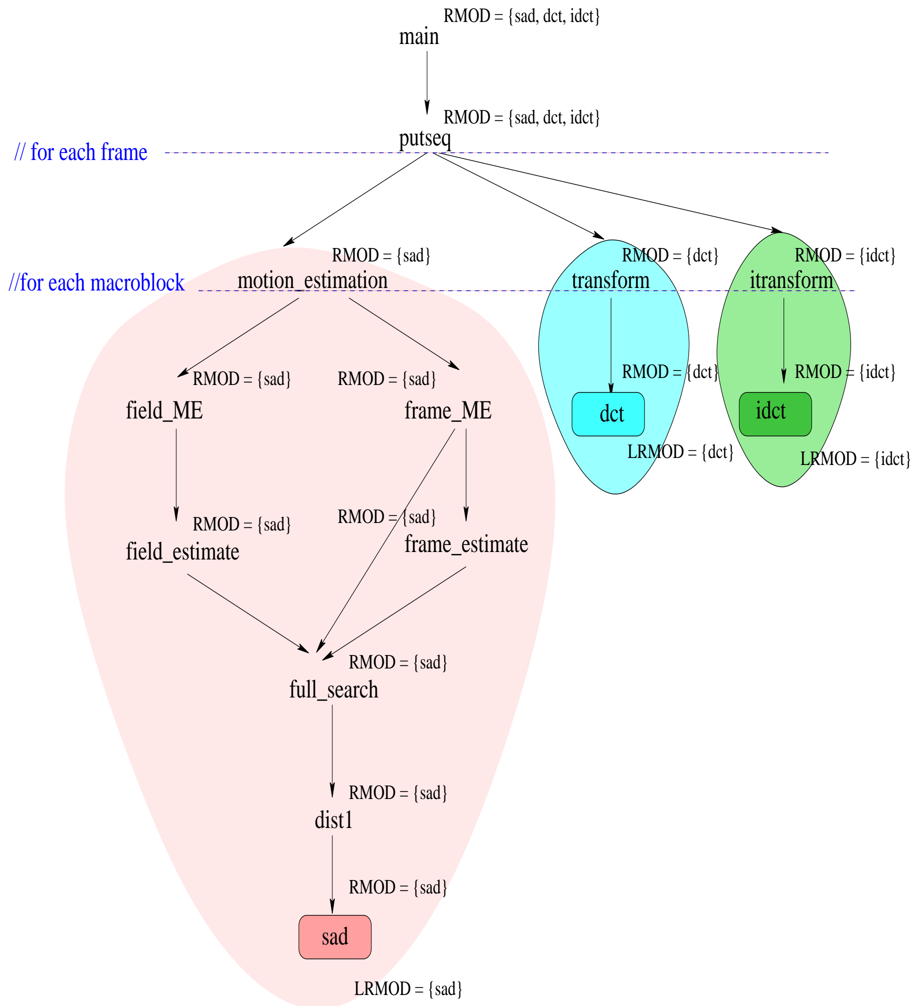

Figure 5.2: Interprocedural data-flow analysis for MPEG2 encoder

procedures, all three hardware operations sad, dct and $idct$ may be performed on the reconfigurable hardware.

Due to the increasing complexity of the interprocedural data-flow analysis, this step is performed only when the call graph G satisfies the following criteria. We assume that there are no *indirect procedure calls* (using pointer to functions). These limitations can be eliminated by considering all candidate set of functions that have the same prototype. Another limitation concerns the data-flow equations for procedures with recursive procedure calls (when the call

graph contains cycles). In this case, the strongly connected components (scc) should be computed and the data-flow equation should be collapsed for each scc into a single equation. The proposed optimization is applied only when the call graph is a DAG.

5.2.3 Step 3: Placement of Hardware Configuration Instructions

In this step, the hardware configuration instructions are anticipated in the call graph taking into account the possible hardware conflicts discovered in the previous step. In the first phase, the set of conflicting operations $CF(p)$ is computed for each procedure included in the call graph based on the $RMOD$ values as follows:

$$CF(p) = \{op_1 \in RMOD(p) | \exists op_2 \in RMOD(p), op_1 \not\leftrightarrow op2\} \qquad (5.2)$$

Next, for each edge of the call graph $< p_i, p_j >$, if there is an hardware operation op which does not have conflicts in p_j ($op \notin CF(p_j)$) but it has conflicts in the calling function p_i ($op \in CF(p_i)$), then a SET op instruction is inserted at all call sites of p_j from p_i. Finally, for all non-conflicting operations of the entry node of the call graph G (i.e. $RMOD(r) - CF(r)$), the corresponding SET instructions are inserted at the beginning of the *r* procedure (see step 3 from Table 5.1).

The CF values for the example presented in Figure 5.1 are shown in Figure 5.3, for the case where all considered hardware operations conflict with each other. For the basic blocks where CF values are missing they are implicitly assumed as $\emptyset$. It can be noticed that the hardware configuration instructions cannot simultaneously propagate upwards of $putseq$ procedure due to the considered hardware conflicts.

In Figure 5.4, we present an example similar to the example from Figure 5.3, but with a different set of conflicting hardware operations. We notice that in this case, the hardware configurations for *sad* and *idct* cannot be anticipated further due to their FPGA area conflict, while the SET instruction for the *dct* hardware operation can be safely moved to the application entry point.

sad ↮ dct
sad ↮ idct
dct ↮ idct

CF = {sad, dct, idct}
RMOD = {sad, dct, idct}
main

CF = {sad, dct, idct}
RMOD = {sad, dct, idct}
putseq

// for each frame

RMOD = {sad}
motion_estimation

//for each macroblock

RMOD = {dct}
transform

RMOD = {idct}
itransform

RMOD = {sad}
field_ME

RMOD = {sad}
frame_ME

RMOD = {dct}
dct
LRMOD = {dct}

RMOD = {idct}
idct
LRMOD = {idct}

RMOD = {sad}
field_estimate

RMOD = {sad}
frame_estimate

RMOD = {sad}
full_search

RMOD = {sad}
dist1

RMOD = {sad}
sad
LRMOD = {sad}

Figure 5.3: Interprocedural optimization for MPEG2 encoder

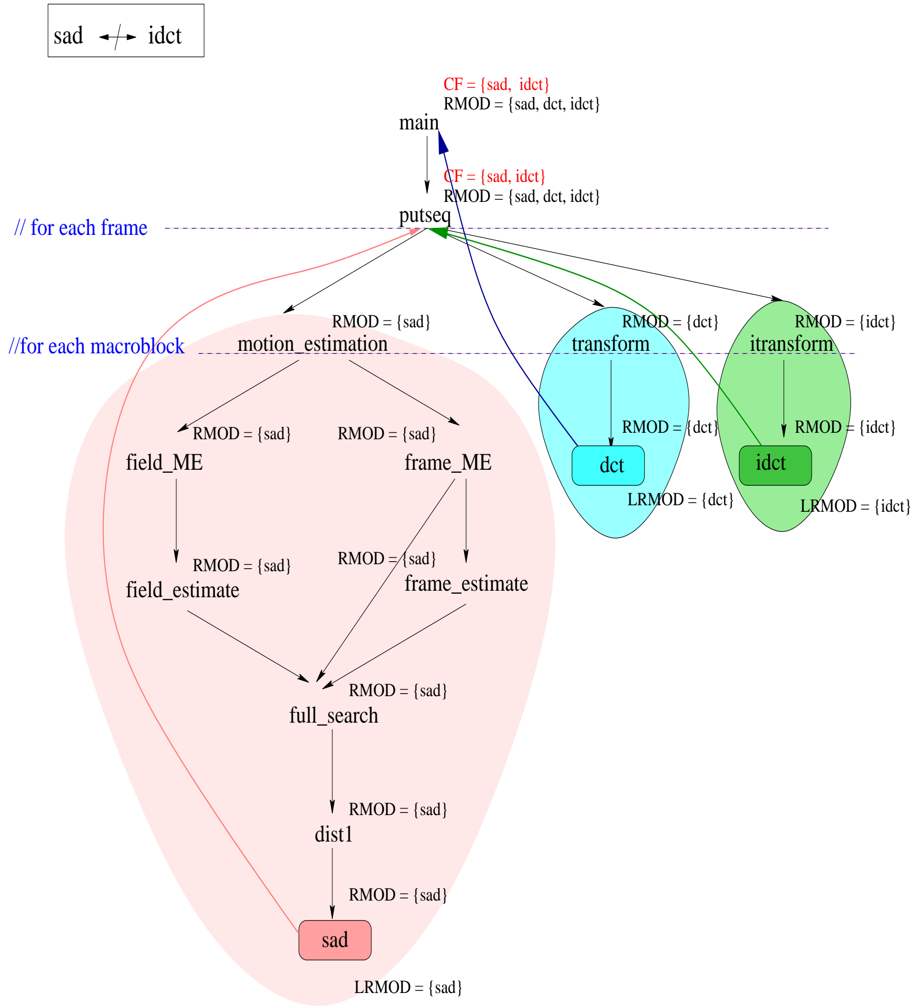

Figure 5.4: Interprocedural optimization for MPEG2 encoder

Name	# frames	Resolution
carphone	96	176x144
claire	168	360x288
container	300	352x288
football	125	352x240
foreman	300	352x288
garden	115	352x240
mobile	140	352x240
tennis	112	352x240

Table 5.2: MPEG test sequences

5.3 A MultiMedia Based Evaluation

In order to evaluate the impact on performance of the dynamic hardware configurations and the proposed optimization, we consider the MPEG2 encoder multimedia benchmarks and the test sequences presented in Table 5.2. Building on previous work [41][32], we look at the following time consuming functions that are implemented in reconfigurable hardware: SAD (sum of absolute-difference), 2D DCT (2 dimensional discrete cosine transform) and IDCT (inverse DCT). We consider a Molen machine organization [76] with an x86 as the GPP. More specifically, the compiler generates code for the x86 architecture while the measurements are performed on an AMD Athlon XP 1900+ at 1600 MHz.

We basically performed four experiments to assess the impact of hardware reconfiguration and proposed optimization. Firstly, we analyze the pure software approach and determine the theoretical (maximal) performance improvement that can be achieved by hardware acceleration of the considered functions. Secondly, we investigate the impact on performance of real reconfigurable hardware implementations for the considered functions, when a simple scheduling is used for the hardware configurations. Next, we determine the performance improvement achieved by hardware execution of each function when only one initial hardware configuration is required. Finally, we present the results of the proposed optimization when all considered hardware operations are executed on the reconfigurable hardware.

Video sequence	SAD (16x16)		DCT (8x8)		IDCT (8x8)	
	# Cycles	% Time	# Cycles	% Time	# Cycles	% Time
carphone	997	31.69	37796	28.19	2612	1.95
claire	1092	36.46	37796	26.44	2177	1.53
container	1008	34.44	37590	27.04	2208	1.59
football	1484	42.74	37537	22.93	2827	1.73
foreman	1298	39.93	37572	24.35	2193	1.42
garden	1311	40.21	37594	24.70	2463	1.62
mobile	1092	35.95	37536	26.30	2519	1.77
tennis	1344	41.23	37531	24.39	2221	1.44
Average	1203	37.83	37593	25.54	2402	1.63

Table 5.3: Profiling results for MPEG2 encoder

5.3.1 Scenario 1: MPEG 2 Profiling Results for Pure Software Execution

We first compute the number of cycles each function consumes for the input sequences given in Table 5.2 when executed on the target GPP without reconfigurable hardware acceleration. These profiling results for the MPEG2 encoder benchmark are presented in Table 5.3. The cumulated time spent by SAD, DCT and IDCT functions (Table 5.3, column 3,5 and 7) in the pure software approach represents about 65 % of the total MPEG2 execution time. In consequence, the hardware acceleration of these functions (as proposed in the Molen approach) can produce a significant speedup of the MPEG2 encoder up to 3x. The results from Table 5.3, column 3 suggest that the SAD function is the best candidate for hardware implementation as it can provide up to around 40 % performance improvement. Whereas for the encoding phase, IDCT cannot yield substantial performance improvement, in decoding phase, this function is heavily used and can then produce a significant performance increase.

5.3.2 Scenario 2: A Simple Hardware Reconfiguration Scheduling

We first present the target reconfigurable hardware platform and the features of the hardware implementations for the considered functions. Next, we introduce the formula we used for computing the performance of the Molen FCCM. Finally, we present the MPEG2 performance results for the reconfigurable

Op	Area	EXEC	HW Speedup
	[slices]	[cycles]	
SAD	831	133	9 x
DCT	4314	1184	31 x
IDCT	5436	1200	2 x

Table 5.4: FPGA area and hardware execution parameters for the considered kernels

Op	SET		SET_MAX		SET/SET_MAX
	[ms]	[cycles]	Mean	st.dev	
SAD	2	3200000	1070	167	2991
DCT	10	16000000	36409	80	439
IDCT	12	19200000	1202	225	15973

Table 5.5: Hardware configuration parameters for the considered kernels

hardware execution of the considered functions when a simple scheduling for hardware configurations is assumed.

Reconfigurable Hardware Platform Before discussing the performance improvement that can be achieved by using the reconfigurable hardware, we present the target FPGA platform included in our experiments. We target the Xilinx Virtex II Pro, XC2VP20 chip and the 2D DCT and 2D IDCT cores available as IPs in the Xilinx Core Generator Tool ([71], [77]) as well as the SAD implementation presented in [32]. The area required by each function is given in Table 5.4, column 2. We present the estimated hardware execution time of each function in terms of the target Athlon processor cycles, given in Table 5.4, column 3. Based on the characteristics of the XC2VP20 chip, for which a complete configuration of 9280 slices takes about 20 ms, we estimate the reconfiguration time for the considered functions as presented in Table 5.5, column 2.

Performance Estimation for Molen FCCM Execution As the presented Molen FCCM does not (currently) support dynamic hardware configuration, we determine the performance of the Molen FCCM based on the measured profiling results for the GPP included in the MOLEN FCCM as follows:

$$n_{Molen} \simeq n_{X86} - n_f + n_{call} \cdot cost \tag{5.3}$$

$$cost = x_{SET} + y_{EXEC} \tag{5.4}$$

where

- n_{Molen}: the total number of GPP cycles spent in the considered application by the Molen processor;
- n_{X86}: the total number of GPP cycles when the considered application is performed only on the GPP;
- n_f: the total number of GPP cycles spent in function f when f is executed exclusively on the GPP;
- n_{call}: the number of calls to function f in the considered application;
- $cost$: the number of cycles for one execution of function f on FPGA;
- x_{SET}: the number of GPP cycles required for one configuration of the FPGA for function f;
- y_{EXEC}: the number of GPP cycles required for one execution on the FPGA of function f; for the considered hardware implementation, the execution time is not dependent on the input data.

In our experiments, we have measured the values for n_{X86}, n_f and n_{call} included in Formula 5.3. To this purpose, we used the *Halt* library[78] available in Machine SUIF. This library is an instrumentation package that allows the compiler to change the code of the application being compiled in order to collect information about the application's own behavior (at run-time). In order to minimize the impact of external factors on the measurements, we run the applications in single mode and with the highest priority in Linux.

MPEG 2 Performance Results for A Simple Hardware Reconfiguration Scheduling On the basis of the hardware execution times from Table 5.4 and the average software execution time given in Table 5.3 column 2,4,6, we determine that the hardware acceleration of the considered kernels (Table 5.4, column 4) is up to 31x. However, a simple scheduling where the corresponding SET and EXECUTE instructions for hardware configuration and hardware execution are consecutively executed for each operation can completely waste the hardware speedup. Due to the huge reconfiguration latency and repetitive hardware configurations, the use of reconfigurable hardware will result in a slowdown of the MPEG2 benchmark (computed as n_{Molen}/n_{X86} using Formula 5.3) by 2-3 orders of magnitude (Table 5.6, row 2) when compared to pure software execution on the GPP alone.

Based on the profiling result (Table 5.3), the reconfigurable hardware execution times (Table 5.4) and Formula 5.3, we determine the upper boundary for a SET instruction latency that ensures that the Molen FCCM is not slower than the pure software approach ($n_{Molen} \simeq n_{X86}$). We refer to this boundary as SET_MAX and is described by:

$$SET_MAX \simeq \frac{n_f}{n_{call}} - y_{EXEC} \tag{5.5}$$

The mean SET_MAX values and standard deviations are presented in Table 5.5, (columns 4-5). We notice that the complete hardware configuration of the currently available FPGA platforms (SET) accounts for 3-4 orders of magnitude (see Table 5.4, last column) more reconfiguration time than SET_MAX and produces for the MPEG 2 benchmark a performance decreasing of 2-3 order of magnitude (Table 5.6, row 2). In consequence, without an appropriate scheduling of the SET instructions, the overall performance is decreased due to the huge reconfiguration latency in spite of the faster hardware execution time. In Appendix A, we present in more details the design space exploration for multimedia applications, with an extended set of architectural parameters.

5.3.3 Scenario 3: Single Hardware Reconfiguration

In order to avoid the above presented limitation and to exploit the performance improvement achieved by the hardware execution of the considered functions, we analyze the case when only one function is executed on the reconfigurable hardware while the other functions are switched to the software execution (on GPP). In this case, only one hardware configuration is required for each function. We estimate the effect of this transformation on performance using the ceteris paribus approach meaning that the performance improvement of each function is individually estimated while assuming that none of the other functions are implemented in reconfigurable hardware.

The MPEG2 performance results for this scenario are presented in Table 5.6, row 3. It can be observed that, by removing the repetitive SET instructions for each function, a significant performance improvement (computed as $\frac{n_f - n_{call} * y_{EXEC} - x_{SET}}{n_{X86}}$) can be observed. The performance efficiency (the rapport between estimated performance improvement for the real hardware implementations and the theoretical performace improvement) is presented in Table 5.6, row 4 and it emphasizes that the individual improvement of each function is close to the maximum possible improvement.

	SAD	DCT	IDCT
Simple Scheduling Slowdown	1012 x	108 x	131 x
Single Hardware Reconfiguration Perf	33.61 %	24.68 %	0.75 %
Theor. Maximal Performance Impr	37.83 %	25.54 %	1.63 %
Performance Efficiency	89 %	97 %	46 %

Table 5.6: MPEG 2 encoder performance results with and without anticipated hardware configuration

5.3.4 Scenario 4: Interprocedural Optimization Results

After having assessed the performance contribution of each function, we investigate the impact of the proposed interprocedural optimization which allows the reconfigurable hardware execution for all considered functions. The described optimization algorithm has been implemented in the Molen compiler, more specifically in the SUIF compiler frontend. The optimization is applied on the call graph for MPEG2 encoder with 111 nodes (i.e. the applications contains 111 procedures).

The aim of the proposed optimization is to significantly reduce the number of the executed SET instructions for each hardware operation. In the results presented in the rest of this section, we compare the number of executed hardware configurations with and without our optimization (denoted as SET_OPT and respectively NO_SET_OP cases).

The number of hardware configurations for the considered functions in the MPEG2 encoder benchmark is presented in Table 5.7. When measuring the effects of the proposed optimization (Table 5.7, columns 4-8), we consider different possible conflicts between SAD, DCT, and IDCT; in the best case there is no conflict (column 4), while in the worst case all hardware operations are in conflict with each other (column 8). One important observation is the 3-5 order of magnitude decrease in the number of hardware configurations produced by our optimization algorithm. The main cause of this decrease is the particular feature of the MPEG2 algorithm where the SAD, DCT and IDCT hardware configurations can be anticipated at the frame level rather than macroblock level (see Figure 5.3). In consequence, due to our optimization algorithm, the

		Initial	With interprocedural SET optimization				
Sequence	HW op	[# SETs]	No conflict	SAD DCT cf	SAD IDCT cf	DCT IDCT cf	All cf
carphone	SAD	7932972	1	96	96	1	96
	DCT	63360	1	96	1	96	96
	IDCT	63360	1	1	96	96	96
claire	SAD	54779496	1	168	168	1	168
	DCT	399168	1	168	1	168	168
	IDCT	399168	1	1	168	168	168
container	SAD	98044520	1	300	300	1	300
	DCT	712800	1	300	1	300	300
	IDCT	712800	1	1	300	300	300
football	SAD	36219280	1	125	125	1	125
	DCT	264000	1	125	1	125	125
	IDCT	264000	1	1	125	125	125
foreman	SAD	98044520	1	300	300	1	300
	DCT	712800	1	300	1	300	300
	IDCT	712800	1	1	300	300	300
garden	SAD	32997680	1	115	115	1	115
	DCT	242880	1	115	1	115	115
	IDCT	242880	1	1	115	115	115
mobile	SAD	40435160	1	140	140	1	140
	DCT	295680	1	140	1	140	140
	IDCT	295680	1	1	140	140	140
tennis	SAD	32494000	1	112	112	1	112
	DCT	236544	1	112	1	112	112
	IDCT	236544	1	1	112	112	112

Table 5.7: The impact of the interprocedural optimization on the number of required hardware configurations in MPEG2 encoder

hardware configuration is transformed from a major bottleneck in a negligible factor on performance. A second observation is that, for the *no conflict* case, our optimization algorithm eliminates all hardware configurations and introduces at the application entry point only one hardware configuration for each hardware operation; thus, all the hardware configurations but one from the initial application (Table 5.7, column 2) are redundant.

In order to conclude this section, four points should be noticed regarding the presented results and optimization. Firstly, the reduction of the number of hardware configurations depends on the characteristics of the target applications. As previously presented, the impact of our optimizations for MPEG2 encoder is substantial, while for other applications (e.g. M-JPEG) it depends on the possible hardware conflicts between operations. Second, it should be

mentioned that this optimization can also increase the number of hardware configurations, e.g. when the considered procedure associated to the hardware operations have multiple call sites and conflicting operations. Flow-sensitive data-flow analysis and profile information can be used to prevent this situation. Nevertheless, taking into account that the hardware configuration can be performed in parallel with the execution of other instructions on the GPP, the reconfiguration latency may be (partially) hidden. The final observation is that a significant reduction of the number of executed hardware configurations is directly reflected in a significant reduction in power consumption, as the FPGA reconfigurations is a main source of power consumption (see [79]).

5.4 Conclusions

In this chapter, we extended the Molen compiler developed to support the Molen Programming Paradigm to study the conditions under which substantial performance improvements can be obtained with hardware acceleration using FCCMs. Based on profiling results, we showed that potential speedups can be completely outweighed by inappropriate scheduling of the reconfiguration instruction. When theoretically a performance improvement of up to 65 % is achievable, the slowdown caused by improper scheduling can be as large as a factor 1000 (e.g. for SAD). We also showed that given a suitable scheduling up to 97 % of the maximal performance improvement can be obtained. The optimization presented in this chapter reduces the number of executed hardware configurations by a factor of 3-5 order of magnitude when compared to the simple scheduling.

In the previous and current chapters, we assume a predefined FPGA area allocation for the operations executed on the reconfigurable hardware. In the next chapter, we will address the FPGA area allocation problem in order to minimize the conflicting operations and the reconfiguration overhead.

Chapter 6

Compiler-driven FPGA-area Allocation

Although the new generations of FPGAs provide support for partial and dynamic configuration, the huge reconfiguration latency is still a major shortcoming of the current FCCMs (see [80]). In this chapter, we propose two FPGA-area allocation algorithms for the tasks executed on the reconfigurable hardware. The goal is to minimize the FPGA-area which is reconfigured at runtime and improve the overall performance, taking into account the application runtime features. More specifically, we use the reconfiguration frequency for the target application to guide the allocation algorithms. Two scenarios are discussed: the first one corresponds to the case when all hardware operations must be placed/executed on the target FPGA while in the second scenario, a hardware operation can be switched to its pure software execution on the core processor in order to reduce the pressure/competition for the FPGA area. Both FPGA-area allocation problems are formulated as 0-1 integer linear programming (LP) problems and efficient LP solvers are used for finding the optimal solutions.

The chapter is organized in six sections. The background and related work is presented in the following section. Next, we discuss some motivational examples and define the FPGA-area allocation problem addressed in this chapter. The proposed allocation algorithms are detailed in section 6.3. Finally, we provide the evaluation of the proposed algorithms and present conclusions and future work.

6.1 Related Work

Previous approaches for FPGA-area allocation are mainly focused on cases where the whole application is decomposed in tasks which are entirely executed on the FPGA. In [81], the authors propose an optimal module placement based on packing classes. A solution based on backtracking with bounding heuristics is presented in [82]. In [83], the authors address the allocation and space-time instruction scheduling based on maps of probabilities that are used to represent the allocations of hardware resources and the time slots. The probabilities reflect the confidence of the allocation and can be adjusted by the tools involved in the scheduling. Other approaches [63] [84] [85] addresses the HW/SW partitioning problem and the reconfiguration latency minimization problem in the context of configuration prefetching and/or multi-context devices. The proposed solutions require detailed information (such as data flow graphs, dependency graphs of tasks) about the application's features and regular application behavior.

A similar approach targeting the Xputer architecture [86] is presented in [87], where the optimization of overall execution time does not rely on ILP solvers - as in our approach, but it is based on a simulated annealing algorithm [88]. Another approach (see [89] [90] [91] [64] [92]) is the task allocation at run-time in an operating system for reconfigurable computing. This general approach is not suitable for the single application execution cases - as assumed in the Molen Programming Paradigm, due to the increased overhead introduced by the operating system. Additionally, information about specific application behavior is not used by the operating systems in order to guide this allocation, thus optimization opportunities can be lost.

The work presented in [93] [94] addresses compiler optimization for reducing the number of redundant FPGA configurations based on a predefined FPGA-area allocation. In consequence, the compiler optimization will benefit from an efficient FPGA-area allocation that minimizes the FPGA-area overlaps for a target application. In this chapter, we propose two FPGA-area allocation algorithms that reduce furthermore the number of FPGA configurations by minimizing the total reconfigured area for a given trace of execution.

6.2 Problem Overview and Definition

6.2.1 Motivational Example

In order to intuitively introduce the FPGA-area allocation problem, we use a motivational example (Figure 6.1(a)) which sketches an FPGA device and the area requirements for three operations implemented on the FPGA. In this chapter, we assume FPGAs with column-based reconfiguration (the reconfiguration may only be performed for a full column of CLBs of the chip) such as the well-known Xilinx Virtex devices. For one application that uses the three hardware operations, a simple FPGA area allocation (presented in 6.1(b)) places all operations starting with the first column. Due to the FPGA area overlaps, such allocation requires the FPGA reconfiguration before each execution of the considered operations. As shown in [80], FPGA reconfiguration is slow and thus, repetitive FPGA reconfigurations can produce a significant performance decrease. In consequence, a better FPGA-area allocation is required in order to reduce the reconfiguration overhead. An allocation strategy is possible only when the placement of the hardware operations is not predefined.

Two important observations can be made regarding the example from Figure 6.1. The first observation is that the three considered operations cannot fit together on the FPGA as the sum of the area of their hardware implementations exceeds the total available FPGA-area. The second observation concerns the simple allocation strategy, where there is unused FPGA-area while parts of the FPGA have to be reconfigured before each execution. For the considered example, even when the Rop2 and Rop3 do not have overlapping FPGA-area, the placement of Rop1 will introduce FPGA-area overlaps with one of the two operations.

In order to determine an efficient FPGA-area allocation, we propose an approach that divides the hardware operations in two categories: FIX and RW. An operation is called FIX if it has no overlapping area with any other hardware operations in the considered application. Such a FIX operation requires only one initial FPGA configuration (which can be preloaded and can be neglected). An operation is called RW (reconfigurable) if its area overlaps with other operations and it has to be configured before each execution.

Loosely stated, the main idea of our approach is to minimize the reconfigured FPGA-area based on the reconfiguration frequency of each operation. Using profiling information, we determine the execution order for the hardware operations (called trace) and compute the reconfiguration frequency in the trace.

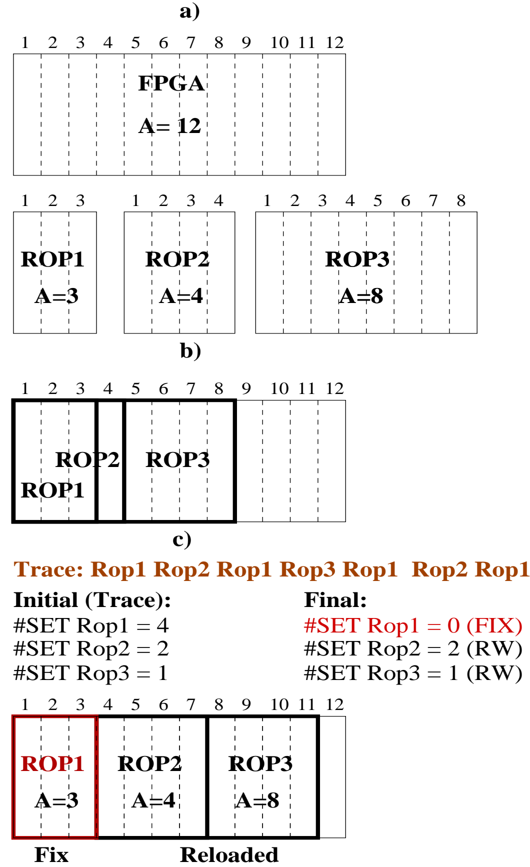

Figure 6.1: Example: a) Total FPGA-area and three Rops; b) a simple FPGA-area allocation c) optimal allocation based on the execution trace

The goal is to allocate the larger and frequently reconfigured operations as FIX operations. The example shown in Figure 6.1(c) presents the optimal FPGA-area allocation for a given execution trace. We can observe the elimination of hardware configurations for the operations allocated as FIX operations (Rop1 in this example). The selection of the FIX operations is based on 0-1 linear programming and is explained in Section 6.3. The used terminology and a formal description of the allocation problem is presented in the rest of this section.

6.2.2 Problem statement

We represent a set of n reconfigurable hardware operations (Rops) as $ROP = \{Rop_1, Rop_2, ..., Rop_i, ..., Rop_n\}$, where each operation Rop_i occupies for its hardware implementation an FPGA-area A_i. The total available area of the target FPGA device is S. Although in this chapter we address the case when the reconfiguration is column-based (the area is expressed as the number of columns), the extension to the 2D or 3D cases is straightforward.

An execution trace is a sequence of Rops that are executed for a set of representative input data for the target application and it is represented as $T : Rop_i, Rop_j, ..., Rop_k, ...$ A trace is normalized if it does not contain two identical consecutive Rops. This normalization represents the fact that consecutive hardware reconfigurations for the same Rop are redundant and can be eliminated by compiler optimization (see [80]) or hardware prefetching. For each $Rop_i \in ROP$ and a normalized trace T, the reconfiguration frequency $n(T)_i$ represents the number of occurrences of Rop_i in the trace T.

As previously explained, the idea of our approach is to divide the ROP set in two subsets FIX and RW, where

$ROP = FIX \bigcup RW$ and $FIX \bigcap RW = \emptyset$.

The Rops in the FIX set will have a dedicated area allocated on the FPGA that is not used by other Rops (they do not have area overlaps with other Rops). The advantage is that the FIX Rops will not require an FPGA reconfiguration before their executions. The total area occupied by the FIX Rops is

$$A_{FIX} = \sum_{Rop_j \in FIX} A_j.$$

The Rops in RW set are the operations that have area overlaps. The reconfiguration overhead is proportional to the FPGA-area which is reconfigured at runtime. The aim is to minimize the total reconfigured area $A_{RW} = \sum_{Rop_i \in RW} n(T)_i * A_i$ which is the sum of the area of the Rops from RW multiplied by their reconfiguration frequency. The minimization corresponds to the minimization of the reconfiguration overhead and implicitly, to the improvement of the overall performance gain. A formal description of this problem is as follows:

Problem Given a set $ROP = \{Rop_1, Rop_2, ..., Rop_i, ..., Rop_n\}$, a total available FPGA-area S, a normalized execution trace T, each Rop_i having an FPGA-area A_i and the reconfiguration frequency $n(T)_i$, find $RW \subseteq ROP$

that minimizes the reconfigured area $\sum_{Rop_i \in RW} n(T)_i * A_i$, under the following constraint:

- $\forall \, Rop_k \in RW$, $A_k + \sum_{Rop_j \in FIX} A_j \leq S$, where $FIX = ROP - RW$.

The constraint represents the requirement that any RW Rop must have enough available area to coexists on the FPGA at the execution time with all FIX Rops. Implicitly, as the FPGA-area is a positive number, the constraint expresses also the requirement that all FIX Rops should fit together on the target FPGA. Once the RW set has been determined for the above mentioned problem, an effective FPGA-area allocation is straightforward. Assuming that A_i represents the number of required columns, an FPGA-area allocation associates with each Rops, the number of the first column where A_i is placed. In the first step, the FIX Rops are consecutively allocated on the FPGA: $for\ each\ Rop_i \in FIX, C_i = C_{FIX}$ and $C_{FIX} = C_i + A_i$ with the initial $C_F IX = 1$. C_i represents the column number of the first column allocated for Rop_i. In the second step, the RW Rops are all allocated at the end of the FPGA-area allocated for the FIX Rops: $\forall Rop_k \in RW, C_k = C_{FIX}$.

6.3 FPGA-area Allocation Algorithms

For the problem defined in the previous section, we propose its formulation as an integer linear pseudo-Boolean (0-1) programming problem and consequently, the solutions can be determined using efficient solvers (see [95]). More specifically, we propose two scenarios. The first case (associated with the FIX/RW Algorithm) corresponds to the above mentioned problem, where the Rops are placed in the FIX or in the RW part on the FPGA. In the second case (corresponding to the FIX/RW/SW Algorithm), we assume than an Rop can have three options for execution: on the FIX or RW part or additionally, it can be switched to its software execution (on GPP). The last options can be preferred for those Rops where the huge reconfigurations latency consumes the gain produced by the fast execution on the FPGA. In the rest of this section, we introduce in detail the two FPGA-area allocation algorithms.

6.3.1 FIX/RW Algorithm

As previously presented, we translate the FPGA-area allocation problem in a 0-1 linear programming problem to produce an optimal solution using efficient solvers.

0-1 Selection In the considered case, any Rop can be executed on the FIX or RW part of the FPGA. In consequence, we associate with any Rop_i a variable x_i such that

$$x_i = \begin{cases} 0 & \text{if} \quad Rop_i \in FIX \\ 1 & \text{if} \quad Rop_i \in RW \end{cases}.$$

Finding the optimal partition of ROP in FIX and RW is reduced to finding the optimal 0-1 values for all x_i.

Objective function In the problem definition in Section 6.2, the minimization of the reconfigured area

$\sum_{Rop_i \in RW} n(T)_i * A_i$ can be expressed as the following objective function

$\sum_{Rop_i \in ROP} n(T)_i * A_i * x_i$. If Rop_i is a FIX Rop, then $x_i = 0$ and it does not increase the reconfigured area as it does not need any configuration. In consequence, only the contribution of the RW Rops is included in the minimization objective function.

Linear Pseudo-Boolean Inequalities The system of linear pseudo-Boolean inequalities of the linear programming problem formulation corresponds to the constraints included the initial problem. The constraint that

$$\forall\, Rop_k \in RW,\, A_k + \sum_{Rop_j \in FIX} A_j \leq S$$

can be expressed as follows:

min:	+2*39*x1	+ 3*13*x2	+ 3*16*x3;	
C1:		+ 13*$\overline{x}$2	+ 16*$\overline{x}$3	$\leq$ 58 - 39
C2:	+ 39*$\overline{x}$1		+ 16*$\overline{x}$3	$\leq$ 58 - 13
C3:	+ 39*$\overline{x}$1	+ 13*$\overline{x}$2		$\leq$ 58 - 16

Figure 6.2: LP problem for the MPEG2 example in Section 6.4 and FIX/RW Algorithm

$$\begin{cases} A_1 * x_1 + \sum\limits_{Rop_j \in ROP} A_j * \overline{x}_j \leq S \\ A_2 * x_2 + \sum\limits_{Rop_j \in ROP} A_j * \overline{x}_j \leq S \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ A_i * x_i + \sum\limits_{Rop_j \in ROP} A_j * \overline{x}_j \leq S \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ A_n * x_n + \sum\limits_{Rop_j \in ROP} A_j * \overline{x}_j \leq S \end{cases}$$

This system of inequalities should be interpreted as follows:

(1) The term $\sum\limits_{Rop_j \in ROP} A_j * \overline{x}_j$ represents the permanently configured FPGA-area occupied by FIX Rops:

$\sum\limits_{Rop_j \in ROP} A_j * \overline{x}_j = \sum\limits_{Rop_j \in FIX} A_j * \overline{x}_j.$

(2)The second observation regards the first term in the inequalities, namely $A_i * x_i$. For the cases when $Rop_i \in FIX \Longrightarrow x_i = 0$, the term $A_i * x_i$ can be eliminated. The ith inequality is transformed in

$\sum\limits_{Rop_j \in ROP} A_j * \overline{x}_j \leq S$ which represents the constraint that the total area allocated for FIX Rops should be smaller or equal than the total available FPGA-area S. Similarly, for the cases when $Rop_i \in RW \Longrightarrow x_i = 1$, the inequality is transformed in

$A_i * x_i + \sum\limits_{Rop_j \in ROP} A_j * \overline{x}_j \leq S$ which represents the constraint that an RW

Rop has to fit on the FPGA together with all FIX Rops.

In our model implementation, each ith inequality should not contain both x_i and $\overline{x}_i$; thus it can be reduced as follows:

$$A_i * x_i + \sum_{j=1}^{n} A_j * \overline{x}_j \leq S \quad \Longleftrightarrow$$

$$A_i * x_i + A_i * \overline{x}_i + \sum_{j=1}^{i-1} A_j * \overline{x}_j + \sum_{j=i+1}^{n} A_j * \overline{x}_j \leq S \quad \Longleftrightarrow$$

$$A_i * (x_i + \overline{x}_i) + \sum_{j=1}^{i-1} A_j * \overline{x}_j + \sum_{j=i+1}^{n} A_j * \overline{x}_j \leq S \quad \Longleftrightarrow$$

$$\sum_{j=1}^{i-1} A_j * \overline{x}_j + \sum_{j=i+1}^{n} A_j * \overline{x}_j \leq S - A_i$$

Example A real example (discussed in details in Section 6.4) is presented in Figure 6.2, for three Rops with $A_1 = 39, A_2 = 13, A_3 = 16, n(T)_1 = 2, n(T)_2 = 3, n(T)_3 = 3$ and $S = 58$. The solution to this problem is $\{x_1 = 0, x_2 = 1; x_3 = 1\}$, which corresponds to $FIX = \{Rop_1\}$ and $RW = \{Rop2, \quad Rop3\}$.

6.3.2 FIX/RW/SW Algorithm

The FIX/RW algorithm previously presented has two important limitations: i) it cannot find a viable FPGA allocation if there is an Rop_i with $A_i > S$ because the constraint set is unsatisfiable; and ii) although the FPGA execution is (usually) faster than the software execution for any Rop, the reconfiguration overhead can significantly increase the overall execution time. In order to eliminate these limitations, we propose the FIX/RW/SW algorithm where the Rops can additionally be switched to software execution. The FPGA-area allocation problem can again be formulated as 0-1 LP problem including the following components.

0-1 Selection In this case, a Rop has three options for execution: on the FIX or RW part on the FPGA or additionally in software (SW). The allocation problem involves the division of ROP in three subsets FIX, RW and SW, such that

$$ROP = FIX \bigcup RW \bigcup SW$$

and

$FIX \bigcap RW = \emptyset$, $FIX \bigcap SW = \emptyset$, $RW \bigcap SW = \emptyset$.

These options can be expressed using three boolean variables for each Rop_i, namely $xfix_i, xrw_i$ and xsw_i, where $xfix_i = \begin{cases} 1 & \text{if} \quad Rop_i \in FIX \\ 0 & \text{if} \quad Rop_i \notin FIX \end{cases}$ and similar for xrw_i and xsw_i.

Moreover, a Rop must be included in only one subset; this constraint can be expressed as:

$xfix_i + xrw_i + xsw_i = 1$.

Finding the optimal partition of ROP in FIX, RW and SW is reduced to finding the optimal 0-1 values for all $xfix_i, xrw_i$ and xsw_i.

Objective function In the problem definition of the previous FIX/RW Algorithm, the goal of the objective function is the minimization of the total reconfigured area. This objective function cannot be used in the current scenario as all Rops can be switched to their software execution. In the FIX/RW/SW algorithm, the goal is the performance gain and the new objective function is the minimization of the execution time for the considered Rops and is expressed as

$$\sum_{i=1}^{n} cost_fix_i * xfix_i + \sum_{i=1}^{n} cost_rw_i * xrw_i + \sum_{i=1}^{n} cost_sw_i * xsw_i,$$

where $cost_fix_i/cost_rw_i/cost_sw_i$ represent the total execution time for Rop_i in FIX/RW/SW respectively and their values can be determined using profiling information and estimations.

Linear Pseudo-Boolean Inequalities The system of linear pseudo-Boolean inequalities of the linear programming problem formulation is similar to the previous FIX/RW system:

$$\begin{cases} A_1 * xrw_1 + \sum_{j=1}^{n} A_j * xfix_j \leq S \\ A_2 * xrw_2 + \sum_{j=1}^{n} A_j * xfix_j \leq S \\ \dots\dots\dots\dots\dots\dots \\ A_i * xrw_i + \sum_{j=1}^{n} A_j * xfix_j \leq S \\ \dots\dots\dots\dots\dots\dots \\ A_n * xrw_n + \sum_{j=1}^{n} A_j * xfix_j \leq S \end{cases}$$

$$
\begin{array}{llll}
\text{min:} & +cost_fix_1 * xfix_1 & +cost_fix_2 * xfix_2 & +cost_fix_3 * xfix_3 + \\
 & +cost_rw_1 * xrw_1 & +cost_rw_2 * xrw_2 & +cost_rw_3 * xrw_3 + \\
 & +cost_sw_1 * xsw_1 & +cost_sw_2 * xsw_2 & +cost_sw_3 * xsw_3
\end{array}
$$

$$
\begin{array}{llllll}
\text{C1:} & xfix_1 & +xrw_1 & +xsw_1 & = 1 & \\
\text{C2:} & xfix_2 & +xrw_2 & +xsw_2 & = 1 & \\
\text{C3:} & xfix_3 & +xrw_3 & +xsw_3 & = 1 & \\
\text{C4:} & 39\ xrw_1 & +39\ xfix_1 & +13\ xfix_2 & +16\ xfix_3 & \leq 58 \\
\text{C5:} & 13\ xrw_2 & +39\ xfix_1 & +13\ xfix_2 & +16\ xfix_3 & \leq 58 \\
\text{C4:} & 16\ xrw_3 & +39\ xfix_1 & +13\ xfix_2 & +16\ xfix_3 & \leq 58
\end{array}
$$

Figure 6.3: The linear problem description for the MPEG2 example presented in Section 6.4 and FIX/RW/SW Algorithm

The main idea is the same as in the previous algorithm: each RW Rop must have allocated enough FPGA-area to fit with all FIX Rops on the FPGA.

Example One linear model for the three Rops presented in Section 6.4 and FIX/RW/SW Algorithm is presented in Figure 6.3. For the estimated costs, the solution to this linear problem is $\{xfix_1 = 1, xfix_2 = 1, xsw3 = 1\}$, while the other boolean variables are zero.

As a final observation for both algorithms, we notice that the generated FPGA-area allocations will preserve the application semantics even when the input execution trace T is not a representative trace. In such cases, some performance gain may be lost, but the application has the correct behavior.

Additionally, we notice that in the translation to the linear programming problem we do not take into account the Rops order in the normalized trace. The reason is that the order information transforms our problem in a non-linear problem. Our future work will address the searching of the optimal solutions as the naive backtracking solution is expensive for a significant number of Rops and large traces.

6.4 Results

In this section, we present the compiler extensions of the Molen compiler (see Section 3) regarding the two FPGA-area allocation algorithms discussed above and the evaluation of the performance achieved by the proposed algorithms in the MPEG2 and MJPEG case study.

Compiler Extension for FPGA-area Allocation The presented FPGA-area allocation algorithms are integrated in the Molen compiler as two Machine-SUIF (see [78]) passes and the user is allowed to choose the allocation to be used. The compiler extensions involve the following:

- extraction of **profile information** for guiding the FPGA-area allocation algorithm: we use code instrumentation techniques in order to determine: the *execution trace* T, the costs (measured in processor cycles) for the software executions $cost_sw$ and the *execution frequency* of the considered Rops.
- **linear programming solver integration**: we use an efficient *LP solver* implementation based on Davis-Putman enumeration methods presented in [95] and publically available as a software package.
- **elimination of FPGA reconfiguration instructions for FIX Rops**: the SET instructions for the FIX Rops are all eliminated from the application code and one SET instruction is added for each FIX Rop at the application entry point.
- **SW switching** for SW Rops in FIX/RW/SW Algorithm: all SET instructions of SW Rops are eliminated from the application code and the EXEC instructions associated to the SW Rops are transformed in standard function calls.

Target Applications, Rops and FPGA The target C applications considered in this section are the well-known multimedia benchmarks MPEG2 and MJPEG encoders. The input sequence for the MPEG2 is the set of three frames that comes with the benchmark, while for MJPEG we use 30 color frames from ”tennis” in YUV format with a resolution of 256x256 pixels.

The Rops candidate for execution on the FPGA are

- *for MPEG2* - **SAD** (sum of absolute-difference), **2D DCT** (2 dimensional discrete cosine transform) and **IDCT** (2D inverse DCT) with the real FPGA implementations presented in [31], and
- *for MJPEG* - **DCT**, **Quantization** and **VLC** (Variable Length Coding) with the the real FPGA implementations for Quantization and VLC presented in [96].

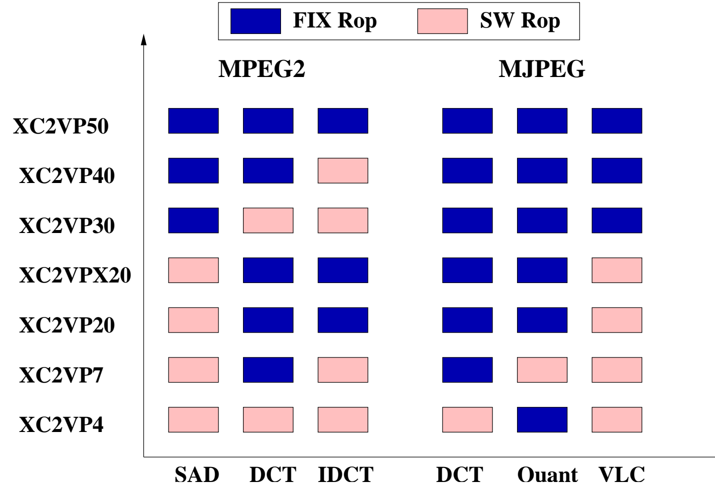

Figure 6.4: FPGA allocation for the FIX/RW/SW algorithm

The target reconfigurable platforms are Xilinx Virtex II Pro devices (see [50]) with CLB array sizes varying from 40 x 22 for XC2VP4 up to 88 x 70 for XC2VP50 and also including one PowerPC processor. The required FPGA-area (expressed in slices) and FPGA execution time (converted in PowerPC at 300 MHz cycles) for the considered Rops are presented in Table 6.1, columns 2-3. We estimate the FPGA reconfiguration time per CLB based on the total configuration time: 47.55 ms for the whole XC2VP50 chip (CLB array of 88x70) using SelectMAP at 50MHz (as presented in [49]); thus, the reconfiguration overhead (converted in PowerPC cycles) is 2315 cycles per CLB. The basic configuration time for the considered Rops is presented in Table 6.1, columns 4. For the software execution, the profiling results for computing $cost_sw$ for each Rop are based on simulations using the PowerPC simulator from Simics [74]. The time spent for the software execution for the considered Rops reported to the total software execution time is presented in Table 6.1, last column.

FPGA-area Allocation Algorithms Evaluation A comparison between the estimated performance for the MPEG2 / MJPEG encoder applications and the two FPGA-area allocation algorithms is presented in Figure 6.5. The reference unit of this comparison (SW) is the pure software execution when all Rops are executed on the GPP. We also include in this comparison the performance estimated for the naive FPGA-area allocation presented in Section 6.2 and denoted as NAlloc for MPEG2. The performance for the proposed algorithms are rep-

Rop Name	Area[Slices]	EXEC[cycles]	SET[Kcycles]	SW [%]
MPEG2				
SAD	13613	49	7880	62 %
DCT	4314	306	2498	15%
IDCT	5436	315	3146	1 %
MJPEG				
DCT	4314	306	2498	80%
Quant	1179	104	683	3%
VLC	6422	110	3718	12.5 %

Table 6.1: HW/SW features for the Rops that candidate for FPGA execution

resented as FIX/RW Alg and FIX/RW/SW Alg. The corresponding solutions for the FIX/RW/SW algorithm are graphically represented in Figure 6.4.

In all cases, we considered that only one FPGA reconfiguration is performed before a sequence of consecutive Rop executions. Otherwise, in the case when an FPGA configuration is performed before each Rop execution, the overall performance is decreased by several orders of magnitude (see [80]). For both algorithms, we use an efficient LP solver implementation based on Davis-Putman enumeration methods presented in [95] and publically available as a software package.

From Figure 6.5, we notice that the FIX/RW algorithm does not generate solutions for the FPGAs with relatively small CLB arrays (as explained in Section 6.3.2), while FIX/RW/SW algorithm guarantees that a better (or equal, in the worst case) solution compared to SW is selected. However, for the FPGA devices with large CLB arrays both algorithms select the best solution - all Rops allocated as FIX Rops - which corresponds to an overall performance improvement of 61 % for MPEG2 and 94 % for MJPEG. In an example scenario using the FIX/RW algorithm for the MPEG2 application and XC2VP40 device where the partial and dynamic hardware configuration is needed, it can be observed that the reconfiguration overhead is reduced by 47 %. For the MJPEG application, the reconfiguration overhead is reduced in all cases by at least 36 %.

In Figure 6.4, we notice that FIX/RW/SW algorithm does not select RW Rops, but SW or FIX Rops are preferred. This observation is explained by the huge reconfiguration latency of the considered devices. Additionally, we present in Figure 6.6, the influence of the reconfiguration overhead on the solutions gen-

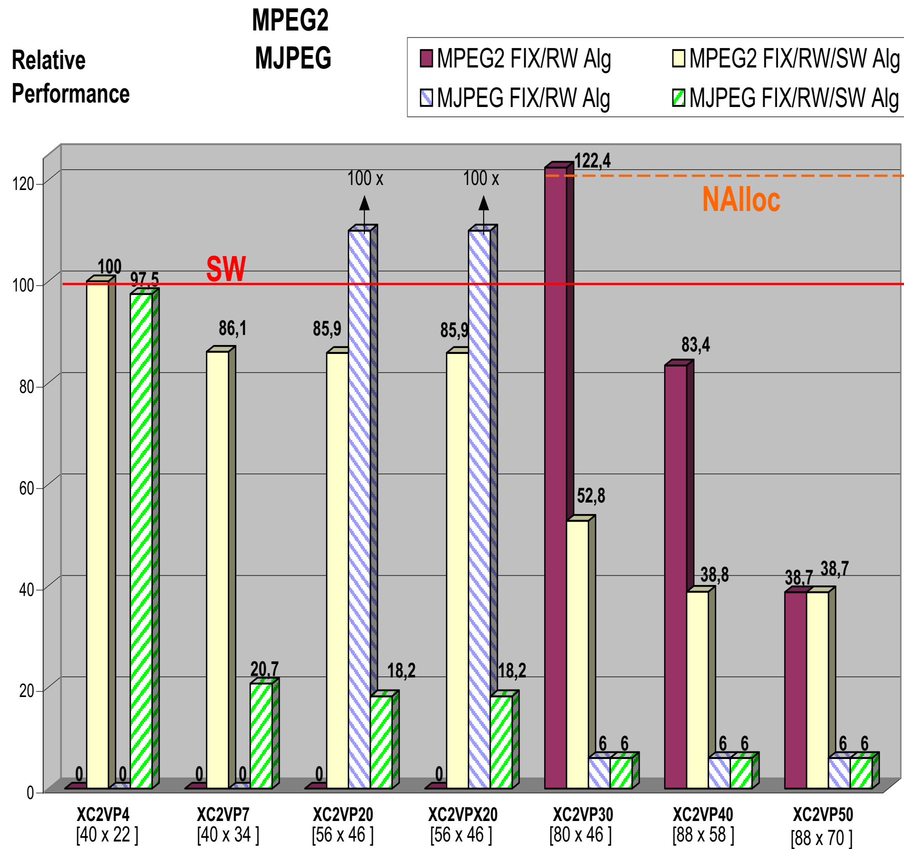

Figure 6.5: Performance comparison for the FPGA-area allocation algorithms

erated by FIX/RW/SW algorithm for the MPEG2 application and XC2VP30 device. An important observation is that the RW Rops are used only when the reconfiguration latency is at least 10 times smaller than the current values. In consequence, the FPGA reconfiguration must be at least one order of magnitude faster for an efficient dynamic FPGA usage.

The proposed allocation algorithms can be easily integrated with the scheduling algorithms presented in Chapter 4 and Chapter 5. After the elimination of the hardware reconfiguration for the reconfigurable operations allocated in the FIX set by the two allocation algorithms, the scheduling algorithms will address only the SET instructions for the reconfigurable operations allocated in the RW set. As a final conclusion, we note that the scheduling algorithms will cooperate with the allocation algorithms to further decrease the reconfiguration

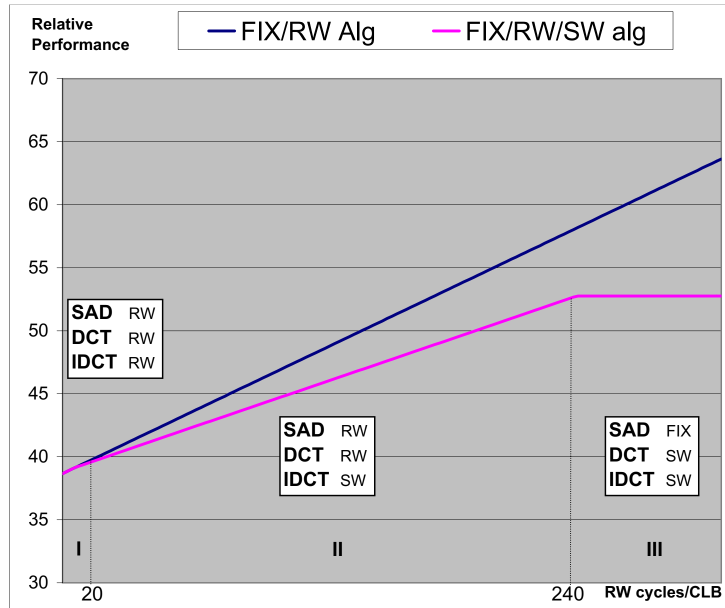

Figure 6.6: The influence of the RW overhead over the FIX/RW/SW algorithm for MPEG2 and XC2VP30 device

overhead.

6.5 Conclusions

In this chapter, we have presented two FPGA-area allocation algorithms for minimizing the huge reconfiguration overhead of the current FPGAs. Two scenarios have been proposed: the traditional placement problem when all Rops are executed on the FPGA and the goal is the minimization of the total reconfigured area and additionally, the case when any Rop can be switched to its software execution and the objective function is to maximize the overall performance gain.

The algorithms incorporate advanced 0-1 LP solvers and use profiling information such as the reconfiguration frequency and software execution time as well as hardware information such as configuration time and hardware execution time for finding the optimal Rops allocations. The presented results show that a performance gain of up to 61 % for MPEG2 and 94 % for MJPEG is to be expected when the proposed allocation algorithms are used. Addition-

ally, the proposed allocation algorithms can be integrated with the scheduling algorithms proposed in the previous chapters.

Chapter 7

Conclusions

In this thesis, we addressed the design and implementation of the Molen compiler for reconfigurable architectures under the Molen Programming Paradigm. More specifically, we first presented the basic compiler extensions required for code generation for reconfigurable architectures. Additionally, we have implemented a PowerPC compiler backend and presented as a proof of concept a real experiment with a multimedia benchmark compiled for and executed on the Molen Polymorphic Media Processor. Given these basic compiler extensions, we subsequently proposed a set of advanced compiler optimizations that address one main shortcoming of the current FPGAs, namely the reconfiguration overhead. Assuming a predefined FPGA area allocation, we proposed two compiler optimizations - at intra and interprocedural level, that aim to anticipate the hardware reconfiguration instructions and to reduce the total number of required reconfigurations. Finally, we proposed two efficient FPGA area allocation algorithms based on profiling results and advanced LP solvers that further reduce the reconfiguration overhead.

In this chapter we present the conclusions of this thesis which are organized as follows. Section 7.1 presents a summary of the thesis. Next, we present the major contributions of this thesis and in Section 7.3, we propose future research directions.

7.1 Summary

In this dissertation we investigated compiler optimizations for reconfigurable architectures that specifically address the reconfiguration overhead. The work

presented in this thesis can be summarized as follows.

In Chapter 2, we presented background information for reconfigurable architectures, with a classification and a set of representative examples for different approaches of reconfigurable architectures. Consequently, we identified the major shortcomings of these approaches, which can be summarized as:

- limited number of new instructions for the reconfigurable hardware and constraints over the instruction operands, due to the instruction encoding formats.
- technology dependent hardware implementation for the operations executed on the reconfigurable hardware
- lack of support for parallel execution of the reconfigurable operations.

In order to address these shortcomings, we presented the Molen machine organization and Programming Paradigm that provide a compact ISA extension for a virtually infinite number of new reconfigurable operations, modularity and parallel execution on the reconfigurable hardware. Finally, we described the DelftWorkBench project that aims to provide the semi-automatic tools that aid the designer in mapping and execution of the input application on the target reconfigurable architecture under the Molen Programming Paradigm.

In Chapter 3, we presented the Molen compiler for reconfigurable architectures under the Molen Programming Paradigm. The compiler is a key component of the DelftworkBench design tool chain as it produces code tailored for the software and hardware features of the target application and architecture. We first presented the compiler framework based on the SUIF/MACHINESUIF infrastructure and the general extensions for the Molen Programming Paradigm which involves ISA extension, Register file extensions as well as hardware/software co-design information. Next, we discussed the specific extensions which have been implemented for the Molen Polymorphic Media Processor which includes a PowerPC backend. Finally, we described an experiment with a multimedia application compiled by the Molen compiler and executed on the Molen Polymorphic Processor with an average speedup of 2.5 compared to the pure software execution.

In Chapter 4 we first presented a formal problem statement for SET instruction scheduling where the goal is the minimization of the reconfiguration overhead based on the reduction of the number of total executed hardware reconfigurations. Next, we presented an intraprocedural compiler optimization that solves the presented problem using advanced data-flow analyses and graph

algorithms. More specifically, we introduced a modified data-flow analysis for partial anticipability that reflects the spatial constraints of the reconfigurable hardware. Additionally, we used the data flow analysis for availability to determine the redundant hardware configurations, when the FPGA is already configured for a specific operation by a previous SET instruction and a new SET instruction is not longer required. In order to minimize the number of executed reconfigurations, we used the minimum s-t cut algorithm on a reduced control flow graph and identify the less frequently executed edges where the SET instructions can be conservatively moved, preserving the application semantics. Using profile information and software/hardware estimation, the proposed optimization selected the software/hardware execution for the candidate operations for execution on the reconfigurable hardware based on the performance improvement criterion. Finally, we estimated that the proposed optimization have a significant impact on performance for the current FPGAs and additionally, the proposed SET scheduling is usefull even for future faster FPGAs.

In Chapter 5 we investigated the impact of the reconfiguration overhead of the current FPGAs on the overall performance and determine that the basic code generation without specific optimizations regarding the reconfiguration overhead can significantly *decrease* the overall performance compared to pure software execution. Additionally, we estimated that the FPGA execution of the considered hardware operations can provide an *acceleration* of several order of magnitudes compared to their execution on the GPP. Thus, in order to exploit the faster reconfigurable hardware execution and to reduce the reconfiguration overhead, we proposed a compiler optimization that is an extension of the compiler optimization proposed in Chapter 4 and it anticipates the hardware reconfiguration instructions at the interprocedural level, in the application call graph. The interprocedural SET scheduling is based on a simplified interprocedural dataflow analysis and it takes into account the spatial constraints of the target reconfigurable hardware.

In Chapter 6 we proposed two compiler-driven FPGA-area allocation algorithms that aim to minimize the overall reconfiguration overhead and to maximize the overall performance improvement compared to the pure software execution. In both algorithms, the FPGA allocation problem was formulated as a 0-1 LP problem and efficient LP solvers were used to find efficient solutions. The main idea was to divide the FPGA area in two parts: one fixed, which is not modified at execution time and one reconfigurable, which is reconfigured at execution time. Next, the “promising” operations were allocated in the fixed part based on profile information and software/hardware estimations. In

the first algorithm, the objective function involved in the 0-1 LP problem is the minimization of the total reconfigured area, which is proportional to the overall reconfiguration overhead. However, for some hardware operations the reconfiguration overhead cannot be hidden and it will significantly reduce the overall performance. In order to address such situations, the second algorithm introduced the switching to software execution of such operations and the objective function was to maximize the overall performance improvement. For the considered multimedia benchmarks, the proposed allocation algorithms can provide significant performance improvements for medium size FPGAs. Additionally, we determined that the dynamic hardware reconfiguration is efficiently used when the reconfiguration latency is at least 10 times smaller than the reconfiguration latency of the current FPGAs.

7.2 Contributions

The main contributions of this thesis can be summarize by the following:

- We have implemented in the Molen compiler the general extensions for code generation for reconfigurable architectures under the Molen Programming Paradigm and specific extensions for the Molen Polymorphic Media Processor. The presented experiment shows that an average 2.5 x overall speedup is achieved with only one hardware operation executed on the FPGA.
- We have performed a design space exploration for reconfigurable architectures under the Molen Programming Paradigm and identify that the reconfiguration overhead can have a major negative impact on the overall performance, when dynamic reconfiguration is required.
- We have proposed a modified dataflow analysis for partial anticipability that reflects the characteristics of the SET instructions. More specifically, we proposed the conditional reunion operator to limit the anticipation of the SET instructions for conflicting hardware operations.
- We have proposed an intraprocedural compiler optimization for hiding and reducing the reconfiguration overhead, that combines data flow analysis for partial anticipability and availability, a graph algorithm for minimum s-t cut and hardware/software selection. The algorithm contributes to 94 % overall performance improvement for the considered benchmarks.

- We have presented an interprocedural compiler optimization that performs the anticipation of the hardware reconfiguration instructions at interprocedural level, based on a simplified interprocedural data flow analysis.
- We have proposed an FPGA-area allocation algorithm for reducing the total reconfiguration overhead. The allocation problem is translated in a 0-1 LP problem and the operations are placed as fixed hardware operations or reconfigurable operations. The algorithm contributes to up to 61.3 % performance improvement for MPEG2 encoder benchmark and 94 % for MJPEG benchmarks.
- We have proposed an FPGA area allocation algorithm for the maximization of the overall performance improvement, where each operation can be allocated as fixed, reconfigurable or switched to its pure software execution.

7.3 Future Research Directions

For the research presented in this thesis, we suggest the following directions for future improvements:

- Hardware/software selection, SET instruction scheduling algorithms and FPGA-area allocation algorithms should be tightly coupled. Such integration is similar to the well known problem of coupling code generation, register allocation and instruction scheduling and its complexity should be analyzed.
- New heuristics should be investigated for the extension of the conditional reunion operator used by the intraprocedural SET instruction scheduling, in order to allow the anticipation of "promising" SET instructions above the conflict points.
- The interprocedural SET instruction scheduling algorithm should be extended to take into account profile informations such as the execution frequency for the basic blocks where the SET instructions should be placed. Such extension can prevent the increasing of the number of executed SET instructions.
- FPGA-area allocation algorithms should be extended to consider dynamic placement of the hardware operations on the target FPGA. In the

thesis, we considered that a hardware operation is synthesized for an unique placement for the entire applications.

- FPGA-area allocation algorithms should be extended to take into account also the order of the hardware reconfigurations, not only the reconfiguration frequency. However, this extension will transform the allocation problem into a non-linear problem.

- Compiler analyses, transformations and scheduling algorithms should be proposed for efficient parallel execution of EXEC instructions. We notice that altough standard compiler techniques can be used, the parallelism that can be exploited by reconfigurable architectures under the Molen Programming Paradigm may differ from the loop level parallelism or instruction level parallelism which is usually targeted by traditional compiler techniques.

Appendix A

Multimedia Design Space Exploration

In this appendix we examine the potential of the Molen approach in terms of execution time for the well-known multimedia applications MPEG2 and JPEG encoders and decoders. The multimedia benchmarks are particularly suitable for the Molen approach as they usually involve intensive computation for highly regular operations, intensive I/O or memory accesses and require real-time processing capabilities. More specifically, we perform a design space exploration study and quantitatively analyze:

- **performance boundaries**: we first determine the maximal performance gains for each operation implemented on the reconfigurable hardware. We also compute for each operation the latency range of the valid hardware designs whose execution on the reconfigurable hardware is faster then the pure software execution. Consequently we show that for real operation implementations the MPEG2 encoder executed on the Molen processor achieves 53 % performance improvement compared to the pure software execution.

- **parameter exchange**: we investigate the effects on performance of the parameter passing between the general purpose processor (GPP) and reconfigurable hardware and we show that the overhead is negligible.

- **memory bottlenecks**: we examine the effect of the data communication between the reconfigurable hardware and memory on performance and show that for DCT a high IO bandwidth (512 bytes/cycle) is required when a fast execution time of around 20-30 cycles is imposed. For SAD and IDCT, the data communication bandwidth is not a constraint.

Name	# frames	Resolution
carphone	96	176x144
claire	168	360x288
container	300	352x288
football	125	352x240
foreman	300	352x288
garden	115	352x240
mobile	140	352x240
standard	3	128x128
tennis	112	352x240

Table A.1: MPEG test sequences in YUV format

On the basis of our design space exploration, the hardware designer can compute in advance for each hardware implementation the global performance improvement and the influence of memory or parameter passing latencies on the overall performance. For example, when a specific speed-up is imposed, the designer is aided to choose the operations that can achieve the required speed-up and the IO bandwidth that eliminates the bottlenecks in the system.

A.1 The MPEG2 and JPEG Case Study

In this section we explore the hardware constraints for implementing on an FPGA a set of well-known time-consuming multimedia operations. The main goal is to determine the parameters that have a substantial impact on the system performance and their range of values in order for the Molen processor to outperform the standalone GPP.

Target Architecture and Applications We consider a Molen machine organization with an x86 as the Core Processor. More specifically, the compiler generates code for the x86 architecture while the measurements are performed on an AMD Athlon XP 1900+ at 1600 MHz. The considered applications are a set of multimedia benchmarks consisting of the Berkeley MPEG2 encoder and decoder and the SPEC95 JPEG encoder and decoder. The time-consuming operations candidate for hardware execution are SAD (sum of absolute-difference), 2D DCT (2 dimensional discrete cosine transform), IDCT (inverse DCT), VLC (variable length coding) and VLD (variable length decoding). The input data are representative series of test images and scenes of various sizes, presented

Name	Resolution
boat.ppm	512x512
clegg.ppm	814x880
frymire.ppm	1118x1105
lena.ppm	512x512
mandrill.ppm	507x509
monarch.ppm	768x512
peppers.ppm	512x512
sail.ppm	768x512
serrano.ppm	629x794
tulips.ppm	768x512
penguin.ppm	1024x739
specmun.ppm	1024x768
vigo.ppm	1024x768

Table A.2: JPEG test images

in Tables A.1 and A.2.

In this thesis we assume that the GPP and FPGA do not run concurrently and that the execution of an operation on the FPGA is each time preceded by the FPGA configuration (even if the previous configuration is the same). Moreover, we assume that the FPGA performs only one operation at the same time. In order to evaluate the performance of the Molen processor for one application where a function f is executed on the FPGA, we compute the number of GPP cycles for the Molen processor using:

$$n_{Molen} \simeq n_{X86} - n_f + n_{call} \cdot cost \quad \text{(A.1)}$$

$$cost = x_{SET} + y_{EXEC} + n_{par} * z_{MOV_XR} + c$$

where

- n_{Molen}: the total number of GPP cycles spent in the considered application by the Molen processor;
- n_{X86}: the total number of GPP cycles when the considered application is executed exclusively on the GPP;
- n_f: the total number of GPP cycles spent in all executions of function f on the GPP;

- n_{call}: the number of calls to function f in the considered application;
- $cost$: the number of GPP cycles for one execution of function f on FPGA; the time for FPGA configuration and execution is converted in GPP cycles.
- x_{SET}: the number of GPP cycles required for one configuration of the FPGA for function f;
- y_{EXEC}: the number of GPP cycles required for one execution on the FPGA of function f; it may depend on the input data. In order to be constant for the chosen set of input data we consider the largest values;
- n_{par}: the number of instructions for sending the parameters from GPR to XR and returning the results;
- z_{MOV_XR}: the number of GPP cycles for one MOV_XR instruction (movtx or movfx)
- c: quantifies the calling convention differences in number of GPP cycles. As c is small (< 10 cycles) for the considered applications, we neglect it.

In our design space exploration, we first analyze the pure software execution and extract the relevant profile information for the considered applications and functions. Based on the profile information and Formula A.1, we examine the performance and the hardware parameters for the target Molen FCCM. The profile information is extracted using *Halt Library* [78] for code instrumentation. Additionally, we develop a set of analysis routines to measure the number of cycles executed in a specific function (using RDTSC - Read Time Stamp Counter instruction) and the number of function calls. More specifically, we have measured the values for n_{X86}, n_f and n_{call} included in Formula A.1. In order to minimize the impact of external factors on the measurements, we run the applications in single mode and with the highest priority in Linux.

As illustrated in Formula A.1, the cost per function call for a reconfigurable execution is determined by the cost of the FPGA configuration (x_{SET}), FPGA execution (y_{EXEC}) and transfer of parameters (z_{MOV_XR}). The influence of these factors on the overall performance and their optimal ranges are explored in the rest of this section.

Cost Range The purpose of the GPP extension with reconfigurable hardware is to achieve a performance improvement over the GPP alone, meaning

	MPEG encoder				
Input	SAD	DCT	IDCT	VLC I	VLC II
carphone	**997**	37796	2612	2631	2196
claire	1092	37595	**2177**	1710	1524
container	1008	37590	2208	1842	**1476**
football	1484	37537	2827	2795	2318
foreman	1298	37572	2193	**1577**	1494
garden	1311	37594	2463	2046	1524
mobile	1092	37536	2519	2123	1564
standard	1199	37549	3423	2930	2239
tennis	1344	**37531**	2221	1702	1578

	MPEG decoder		
Input	IDCT	VLD I	VLD II
carphone	2513	1763	1347
claire	**2056**	745	659
container	2087	880	**586**
football	2678	1940	1499
foreman	2071	**568**	606
garden	2332	1091	662
mobile	2398	1177	722
standard	3295	-	-
tennis	2099	718	713

Table A.3: Software cost (bold) expressed in GPP cycles for the functions included in MPEG2 application

$n_{Molen} < n_{X86}$ which holds when

$$cost < n_f/n_{call}. \tag{A.2}$$

The values for limit cost (n_f/n_{call}) in Formula A.2 are presented in Tables A.3 and A.4. They represent the cut-off points for the hardware execution from where an implementation provides a performance improvement. We refer to the minimal values of each operation as the software cost (presented in bold in Tables A.3 and A.4). For an implementation that executes the operation in a number of cycles less than the software cost, a performance improvement is guaranteed to hold for all input data in the study.

Performance boundaries For each operation, we determine the number of

	JPEG encoder		JPEG decoder	
Input	VLC	DCT	IDCT	VLD
boat	**2272**	**2746**	**2905**	**7322**
clegg	3631	2748	3701	13950
frymire	3371	2758	3313	12989
lena	2469	2759	3461	8080
mandrill	3463	2759	3686	12967
monarch	2403	2758	3360	7836
penguin	2488	2762	3453	8014
peppers	2505	2764	3463	8369
sail	3110	2758	3545	11276
serrano	2955	2766	3698	10753
specmun	2375	2776	3389	7389
tulips	2882	2760	3571	10177
vigo	2550	2755	3424	8444

Table A.4: Software cost (bold) expressed in GPP cycles for the functions included in JPEG application

cycles it consumes in the pure software approach from the overall application (n_f/n_{X86}) as presented in Table A.5 (second column). These values represent the maximal improvements of the overall performance that can be achieved by hardware acceleration of the considered functions. We notice that implementing the SAD function on the FPGA can improve the overall performance up to 38 % while the overall improvement for VLC is very low (0.2 %). When taking into account all the functions for MPEG2 encoder (Fig. A.1), the maximal reduction of the number of cycles is 65 % compared to the pure software implementation.

We are also interested in determining the boundaries between which any real implementation should be situated. The upper boundary is when there is no improvement, meaning $n_{Molen} = n_{X86}$ and the lower (theoretical) boundary corresponds to an infinite hardware acceleration ($cost$ = 0) of each function. These boundaries are presented in Fig. A.1 and they limit the design space where a hardware designer should place a particular implementation. When all operations are designed to execute at the software cost, then an overall performance improvement is still guaranteed (6 % in Fig. A.1). This improvement is due to the safe choice of minimal value for the software cost in order to guarantee no performance decreasing even for the worst case input data.

For the real, non-optimized FPGA implementations described in [32], we also plotted in Fig. A.1 the performance for the same operations assuming that

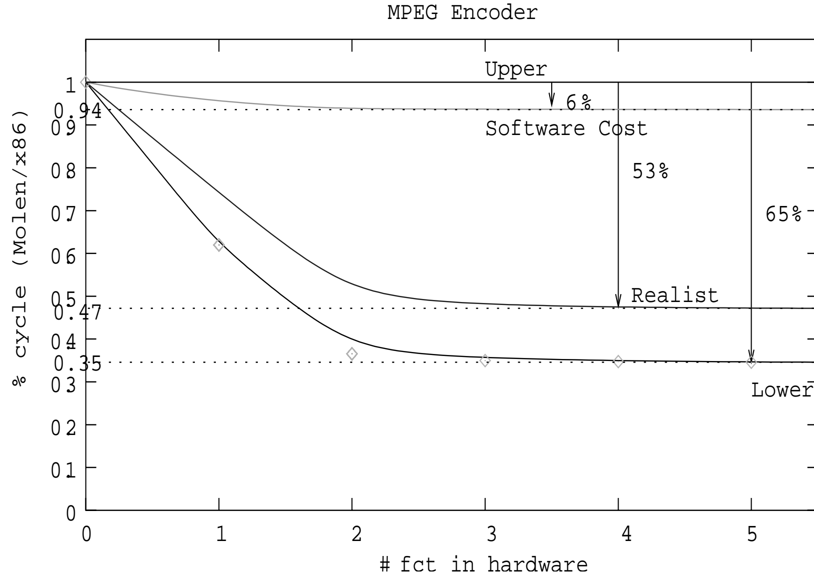

Figure A.1: Relative performance boundaries and a real implementation analysis

compiler optimizations hide the configuration latency ($x_{SET} = 0$) as presented in [94] [93]. After converting the reported number of cycles to our target processor, we obtain 53% performance improvement.

Parameter Passing Impact In order to understand the impact of the Molen parameter passing mechanism, we assume a scenario in which the cost from Formula A.1 is exclusively spent for passing parameters ($cost = n_{par} * z_{MOV_XR}$ and $x_{SET} = y_{EXEC} = 0$). Under this theoretical assumption we compute the maximal number of cycles for z_{MOV_XR} as $z_{MOV_XR} = software_cost/n_{par}$ given in Table A.5 (last column). In order to interpret the results, it is important to realize that MOV_XR instructions resemble the move general purpose register instructions which usually require a small number ($\sim$ 3) of cycles. Our computations show that for the SAD function, a MOV_XR instruction can be executed in up to 166 cycles before the maximal performance (38 % in Table A.5, second column) is consumed. In the case of DCT, the MOV_XR can take up to 37531 cycles before the penalty is higher than the maximal performance gain of 25.4 %.

In order to analyze the communication overhead between GPP and FPGA, we assumed an exaggerated scenario in which the cost for the hardware configu-

Function	MAX Improv	MOV_XR max
MPEG2 encoder		
SAD	38.0 %	166
DCT	25.4 %	37531
IDCT	1.6 %	2177
VLC I	0.2 %	225
VLC II	0.1 %	369
MPEG2 decoder		
IDCT	38.3 %	2056
VLD I	3.1 %	284
VLD II	2.0 %	586
JPEG encoder		
VLC	14.7 %	568
DCT	14.0 %	2746
JPEG decoder		
IDCT	49.5 %	581
VLD	24.3 %	732

Table A.5: Marginal improvement for each function and the maximal cost for MOV_XR (cycles)

ration and execution is half of the software cost. The impact of z_{MOV_XR} is presented in Fig. A.2 showing that the MPEG2 encoder is the only application whose performance may be negatively influenced by z_{MOV_XR}. This is explained by the low software cost for SAD (compared to DCT, Table A.3) and the large number (8) of parameters. In conclusion, we consider that for the operations under considerations, transferring parameters and returning the results is not a bottleneck in the system.

Communication FPGA - Memory Finally, we investigate the FPGA-memory data communication bandwidth as some parameters passed to the FPGA in XRs are pointers to blocks of data placed in external memory. In this context, we assume that access to memory is sequential and symmetrical (number of cycles to read and write one block of data are equal). In order to determine the amount of data transferred to/from external memory, we introduce a special pass in the compiler to annotate each basic block of the considered functions with the number of read and write memory instructions, as well as the corresponding number of bytes. Stack operations are not considered as read/write (R/W) operations, as the FPGA implementation most probably will not use a

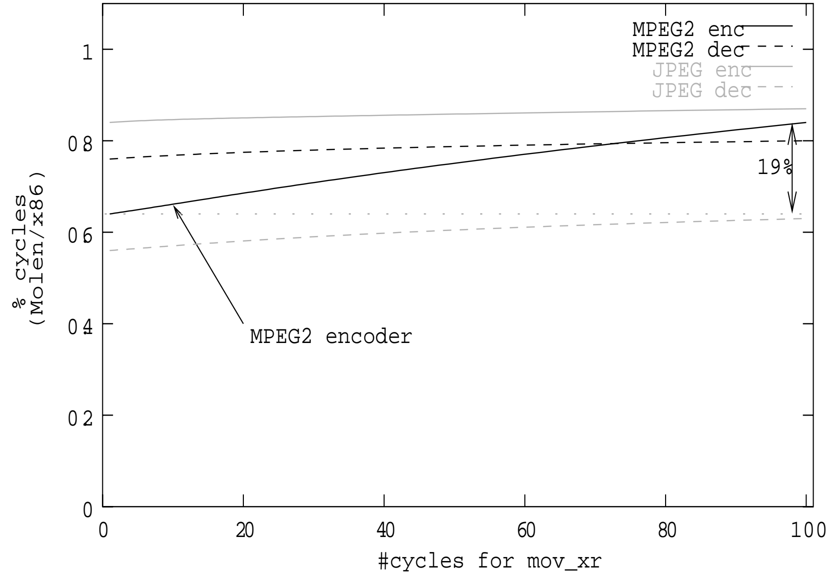

Figure A.2: z_{MOV_XR} impact when passing parameters requires software cost/2

stack. In Table A.6, we present the number of read and write memory operations together with the corresponding number of bytes (R_B and W_B). In column R_W_B, the total number of read and written bytes (R_B + W_B) is given.

When we assume automatic transformation and FPGA mapping performed by tools such as Compaan [54], that preserve the memory accesses performed in software, we can analyze the memory bandwidth. As far as the results for DCT are concerned, our calculations are done using the Berkeley implementation of the MPEG2 encoder benchmark including forward DCT-double precision. Figure A.3 shows the computed bandwidth requirements for different execution times. Our calculations indicate that DCT is the most demanding function. The reasons of this high bandwidth requirement are: (i) the use of doubles (8 bytes) to minimize information loss during compression, (ii) temporary results are also stored in memory and (iii) the parameters are each time read from memory. If a fast DCT design of around 20-30 cycles is required then around 512 bytes need to be transferred per cycle to fully utilize the DCT unit. Fast SAD and IDCT implementations are less demanding as far as IO is concerned. If SAD is going to be implemented in around 5 cycles, as de-

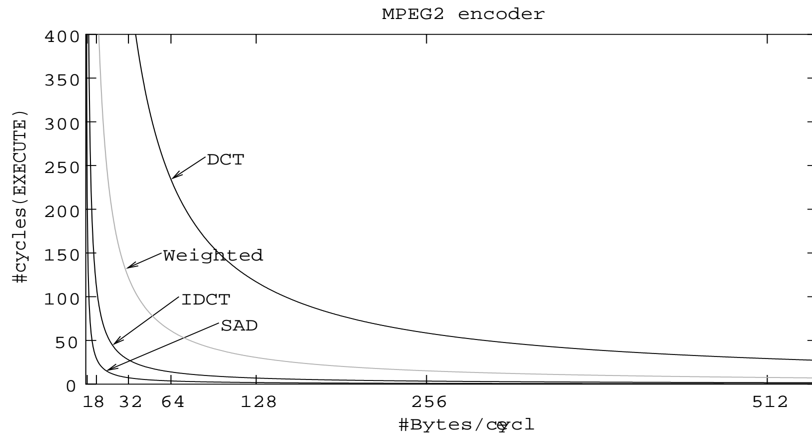

Figure A.3: Execution time for different bandwidth

Function	Read	R_B	Write	W_B	R_W_B
MPEG2 encoder					
SAD	235	235	0	0	235
DCT	2112	13824	192	1152	14976
IDCT	254	636	128	256	892
VLC I	129	197	1	4	201
VLC II	128	192	0	0	192
MPEG2 decoder					
IDCT	254	636	128	256	892
VLD I	288	962	72	270	1232
VLD II	225	749	55	207	956
JPEG encoder					
VLC	184	547	118	472	1019
DCT	256	1024	128	512	1536
JPEG decoder					
IDCT	344	995	128	320	1315
VLD	1849	6752	539	1926	8678

Table A.6: Number of loads/stores performed in the pure software approach

scribed in [97], then a bandwidth of 47 bytes per cycle is enough to have a performance gain of 37 % (which is close to the maximal 38 % improvement). When a bandwidth of 128 bytes per cycle is assumed, then a SAD operation can be performed without starvation even in 2 cycles. Similar conclusions can be drawn for IDCT. We finally also computed the bandwidth requirements taking into account the weighted execution times for each function. This curve reflects the requirements of a possible real implementation and suggests that a fast execution time of around 50 cycles for all operations requires a bandwidth of 83 bytes per cycle.

We emphasize that the presented results are based on the assumption that y_{EXEC} is constant (requiring the maximal possible delay) for a specific function, even though it can vary according to the specific input data (e.g. for VLD function). Therefore, the actual performance improvements may be higher that presented in this appendix.

Bibliography

[1] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," in *Proceedings of Design, Automation and Test in Europe*, (Munich, Germany), pp. 642–649, March 2001.

[2] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architecture design of reconfigurable pipelined datapaths," in *Advanced Research in VLSI*, pp. 23–40, 1999.

[3] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD - reconfigurable pipelined datapath," in *6th International Workshop on Field Programmable Logic and Applications (FPL 96)*, vol. 1142, (Darmstadt, Germany), pp. 126–135, Springer-Verlag Lecture Notes in Computer Science (LNCS), September 1996.

[4] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams," in *Proceedings of International Symposium on Computer Architecture*, June 2004.

[5] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. L. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proceedings 7th ACM International Symposium on Field-Programmable Gate Arrays (FPGA 99)*, pp. 135–143, February 1999.

[6] B. Kastrup, J. van Meerbergen, and K. Nowak, "Seeking (the right) problems for the solutions of reconfigurable computing," in *9th International Workshop on Field-Programmable Logic and Applications (FPL 99)*, (Glasgow, Scotland), pp. 520–525, Springer-Verlag Lecture Notes in Computer Science (LNCS), September 1999.

[7] R. Wittig and P. Chow, "Onechip: An FPGA processor with reconfigurable logic," in *4th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 96)*, (Napa Valley, California), pp. 126–135, April 1996.

[8] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. C. Filho, "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive application," in *IEEE Transactions on Computers*, vol. 49(5), pp. 465–481, May 2000.

[9] J. Jacob and P. Chow, "Memory interfacing and instruction specification for reconfigurable processors," in *7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 99)*, (Monterey, California), pp. 145–154, February 1999.

[10] P. Athanas and H. Silverman, "Processor reconfiguration through instruction-set metamorphosis," in *IEEE Computer*, vol. 26(3), pp. 11–18, March 1993.

[11] S. Trimberger, "Reprogrammable instruction set accelerator," in *U.S. Patent No. 5,737,631*, April 1998.

[12] T. Miyamori and K. Olukotun, "A quantitative analysis of reconfigurable coprocessors for multimedia applications," in *6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 98)*, (Napa Valley, California), pp. 2–22, Springer-Verlag Lecture Notes in Computer Science (LNCS), April 1998.

[13] S. Sawitzki, A. Gratz, and R. Spallek, "Increasing microprocessor performance with tightly-coupled reconfigurable logic arrays," in *8th International Workshop on Field-Programmable Logic and Applications (FPL 98)*, (Tallin, Estonia), pp. 411–415, September 1998.

[14] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, "A dag-based design approach for reconfigurable VLIW processors," in *IEEE Design and Test Conference in Europe*, (Munich, Germany), pp. 778–780, March 1999.

[15] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: A reconfigurable architecture and compiler," *IEEE Computer*, vol. 33(4), pp. 70–77, April 2000.

[16] R. Razdan and M. Smith, "A high performance microarchitecture with hardware-programmable functional units," in *27th Annual International*

Symposium on Microarchitecture MICRO-27, (San Jose, California), pp. 172–180, November 1994.

[17] B. Kastrup, A. Bink, and J. Hoogerbrugge, "Concise: A compiler-driven cpld-based instruction set accelerator," in *Proceedings of FCCM'99*, (Napa Valley CA), pp. 92–100, April 1999.

[18] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera reconfigurable functional unit," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, (Napa, California), pp. 87–96, 1997.

[19] M. B. Gokhale and J. M. Stone, "Napa C: Compiling for a Hybrid RISC/FPGA Architecture," in *Proceedings of FCCM'98*, (Napa Valley, CA), pp. 126–137, April 1998.

[20] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, "The napa adaptive processing architecture," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 28–37, April 1998.

[21] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *IEEE Computer*, vol. 33(4), pp. 62–69, April 2000.

[22] C. Hauser and J. Wawrzynek, "GARP: A MIPS processor with a reconfigurable coprocessor," in *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 12–21, April 1997.

[23] Z. A. Ye, N. Shenoy, and P. Banerjee, "A C compiler for a processor with a reconfigurable functional unit," in *ACM/SIGDA Symposium on FPGAs*, (Monterey, California, USA), pp. 95–100, Feb 2000.

[24] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," in *Proc. of the 5th IEEE Symposium on FPGAs for Custom Computing Machines*, (Los Alamitos, Caliornia), pp. 87–96, 1997.

[25] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer, "A coprocessor for streaming multimedia acceleration," in *Proc. of the 26th International Symposium on Computer Architecture*, (Georgia, USA), pp. 28–39, 1999.

[26] B. Mei, F. Veredas, and B. Masschelein, "Mapping an h.264/avc decoder onto the adres reconfigurable architecture," in *Proceedings International*

Conference on Field Programmable Logic and Applications (FPL 2005), (Tampere, Finland), pp. 622–625, August 2005.

[27] B. Mei, S. Vernalde, D. Verkest, H. de Man, and R. Lauwereins, "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," in *FPT 2002*, (Hong Kong , China), pp. 166–173, December 2002.

[28] A. L. Rosa, L. Lavagno, and C. Passerone, "Hardware/software design space exploration for a reconfigurable processor," in *Proc. of DATE 2003*, (Munich, Germany), pp. 570–575, March 2003.

[29] F. Campi, M. Toma, A. Lodi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," in *In ISSCC Digest of Technical Papers*, pp. 250–251, Feb 2003.

[30] M. Sima, S. Vassiliadis, S.Cotofana, J. van Eijndhoven, and K. Vissers, "Field-programmable custom computing machines - a taxonomy," in *12th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2438, (Montpellier, France), pp. 79–88, Springer-Verlag Lecture Notes in Computer Science (LNCS), Sep 2002.

[31] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Moscu Panainte, "The Molen Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53(11), pp. 1363– 1375, November 2004.

[32] S. Vassiliadis, S. Wong, and S. Cotofana, "The molen $\rho\mu$-coded processor," in *11th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2147, (Belfast, UK), pp. 275–285, Springer-Verlag Lecture Notes in Computer Science (LNCS), Aug 2001.

[33] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, "The Molen Programming Paradigm," in *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, (Samos, Greece), pp. 1–7, July 2003.

[34] R. J. Meeuws, Y. D. Yankova, and K. Bertels, "Towards a quantitative model for hardware/software partitioning," in *RCosy Report*, p. 57, April 2006.

[35] C. Galuzzi, E. Moscu Panainte, Y. D. Yankova, K. Bertels, and S. Vassiliadis, "Automatic selection of application-specific instruction-set extensions," in *CODES+ISSS 2006 - Proceedings of the 4th international con-*

ference on Hardware/software codesign and system synthesis, pp. 160–165, October 2006.

[36] C. Galuzzi, K. Bertels, and S. Vassiliadis, “A linear complexity algorithm for the automatic generation of convex multiple input multiple output instructions,” in *Proceedings of ARC 2007*, pp. 130–141, March 2007.

[37] Y. D. Yankova, K. Bertels, S. Vassiliadis, R. J. Meeuws, and A. Virginia, “Automated hdl generation: Comparative evaluation,” in *Proceedings of International Symposium on Circuits and Systems (ISCAS2007)*, May 2007.

[38] Z. Guo and W. Najjar, “A compiler intermediate representation for reconfigurable fabrics,” in *Proc. of the 16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, (Madrid, Spain), August 2006.

[39] J. Cardoso and H. Neto, “Towards an automatic path from JavaTM bytecodes to hardware through high-level synthesis,” in *IEEE International Conference on Electronics, Circuits and Systems*, vol. 1, (Lisboa, Portugal), pp. 85–88, 1998.

[40] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, “The molen media processor: Design and evaluation,” in *Proceedings of the International Workshop on Application Specific Processors, WASP 2005*, pp. 26–33, September 2005.

[41] E. Moscu Panainte, K. Bertels, and S. Vassiliadis, “Compiling for the Molen Programming Paradigm,” in *13th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2778, (Lisbon, Portugal), pp. 900–910, Springer-Verlag Lecture Notes in Computer Science (LNCS), Sep 2003.

[42] http://suif.stanford.edu/suif/suif2.

[43] http://www.eecs.harvard.edu/hube/software.

[44] “Iso/iec 9899,” (http://www.open-std.org/JTC1/SC22/WG14/www/standards).

[45] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, “The molen media processor: Design and evaluation,” in *Proceedings of the International Workshop on Application Specific Processors, WASP 2005*, pp. 26–33, September 2005.

[46] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, "The Virtex II Pro MOLEN processor," in *Proceedings of the 4th International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS 2004)*, pp. 192–202, July 2004.

[47] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, "The MOLEN processor prototype," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, pp. 296–299, April 2004.

[48] Xilinx Corporation, *PowerPC Processor Reference Guide*, September 2003.

[49] Xilinx Corporation, *Virtex-II Pro Platform FPGA Handbook v2.0*, October 2002.

[50] Xilinx Corporation, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Functional Description*, June 2004.

[51] S. Sobek and K. Burke, *PowerPC Embedded Application Binary Interface 32-Bit Implementation, Version 1.0*.

[52] G. Kuzmanov and S. Vassiliadis, "Arbitrating Instructions in an $\rho\mu$-coded CCM," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, vol. 2778, (Lisbon, Portugal), pp. 81–90, Springer-Verlag Lecture Notes in Computer Science (LNCS), September 2003.

[53] E. Moscu Panainte, K. Bertels, and S. Vassiliadis, "Multimedia reconfigurable hardware design space exploration," in *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, pp. 398–403, November 2004.

[54] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: Deriving process networks from matlab for embedded signal processing architectures," in *Proc. of CODES'2000*, (San Diego, CA), pp. 13–17, May 2000.

[55] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, " Laura: Leiden Architecture Research and Exploration Tool," in *13th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2778, (Lisbon, Portugal), pp. 911–920, Springer-Verlag Lecture Notes in Computer Science (LNCS), Sep 2003.

[56] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: The Compaan/Laura approach," in *Proc. of DATE 2004*, (Paris, France), pp. 340–345, Feb 2004.

[57] http://www.xilinx.com/ise_eval/index.htm.

[58] http://www.xilinx.com/ise/embedded/edk.htm.

[59] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proceedings of AFIPS Conference*, pp. 483–485, 1967.

[60] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing, 1986.

[61] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[62] E. Moscu Panainte, K. Bertels, and S. Vassiliadis, "Dynamic Hardware Reconfigurations: Performance Impact on MPEG2," in *Proceedings of SAMOS*, vol. 3133, (Samos, Greece), pp. 284–292, Springer-Verlag Lecture Notes in Computer Science (LNCS), July 2004.

[63] J. Noguera and R. Badia, "A hw/sw partitioning algorithm for dynamically reconfigurable architectures," in *Proceedings of Design, Automation and Test in Europe*, (Munich, Germany), March 2001.

[64] J. Noguera and R. Badia, "Run-time HW/SW codesign for discrete event systems using dynamically reconfigurable architectures," in *Proceedings of the 13th international symposium on System synthesis*, (Madrid, Spain), pp. 100–106, 2000.

[65] E. Moscu Panainte, K. Bertels, and S. Vassiliadis, "The PowerPC backend molen compiler," in *FPL*, vol. 3203, (Antwerp, Belgium), pp. 434–443, Springer-Verlag Lecture Notes in Computer Science (LNCS), September 2004.

[66] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," in *DATE 2005*, (Munich, Germany), pp. 106–111, March 2005.

[67] X. Tang, M. Aalsma, and R. Jou, "A compiler directed approach to hiding configuration latency in Chameleon processors," in *FPL*, vol. 1896, (Villach, Austria), pp. 29–38, Springer-Verlag Lecture Notes in Computer Science (LNCS), Aug 2000.

[68] Q. Cai and J. Xue, "Optimal and efficient speculation-based partial redundancy elimination," in *ACM CGO*, (San Francisco, California), pp. 91–102, 2003.

[69] J. Edmonds and R. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," in *Journal of the ACM*, vol. 19 (2), pp. 248–264, 1972.

[70] J.-F. Lalande, M. Syska, and Y. Verhoeven, "Mascopt - a network optimization library: Graph manipulation," Tech. Rep. RT-0293, INRIA Sophia Antipolis, 2004 route des lucioles - BP 93 - FR-06902 Sophia Antipolis, April 2004.

[71] L. Pillai, "Video compression using DCT," in *Application Note: Virtex-II Series*, (http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf).

[72] L. Pillai, "Quantization," in *Application Note: Virtex and Virtex-II Series*, (http://direct.xilinx.com/bvdocs/appnotes/xapp615.pdf).

[73] L. Pillai, "Variable length coding," in *Application Note: Virtex-II Series*, (http://direct.xilinx.com/bvdocs/appnotes/xapp621.pdf).

[74] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Transactions on Computers*, vol. 35(2), pp. 50–58, February 2002.

[75] B. Blodget, C. Bobda, M. Huebner, and A. Niyonkuru, "Partial and dynamic reconfiguration of Xilinx Virtex-II FPGAs," in *FPL*, vol. 3203, (Antwerp, Belgium), pp. 801–810, Springer-Verlag Lecture Notes in Computer Science (LNCS), September 2004.

[76] S. Vassiliadis, G. Kuzmanov, S. Wong, E. Moscu Panainte, G. N. Gaydadjiev, K. Bertels, and D. Cheresiz, "PISC: Polymorphic instruction set computers," in *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC 2006)*, pp. 274–286, March 2006.

[77] L. Pillai, "Video compression using IDCT," in *Application Note: Virtex-II Series*, (http://direct.xilinx.com/bvdocs/appnotes/xapp611.pdf).

[78] M.Mercaldi, M. D. Smith, and G. Holloway, “The halt library,” in *The Machine-SUIF Documentation Set*, (Hardvard University), 2002.

[79] R. Fischer, K. Buchenrieder, and U. Nageldinger, “Reducing the power consumption of FPGAs through retiming,” in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’05)*, pp. 89–94, 2005.

[80] E. Moscu Panainte, K. Bertels, and S. Vassiliadis, “Instruction scheduling for dynamic hardware configurations,” in *Proceedings of Design, Automation and Test in Europe (DATE 05)*, (Munich, Germany), pp. 100–105, March 2005.

[81] S. Fekete, E. Khler, and J. Teich, “Optimal FPGA module placement with temporal precedence constraints,” in *Proceedings of Design, Automation and Test in Europe 2005 (DATE 01)*, pp. 658–665, 2001.

[82] R. Maestre, F. J. Kurdahi, N. Bagherzadeh, H. Singh, R. Hermida, and M. Fernndez, “Kernel scheduling in reconfigurable computing,” in *Proceedings of Design, Automation and Test in Europe (DATE ’99)*, pp. 90–96, 1999.

[83] S. Swenson, “Spatial instruction scheduling for raw machines,” in *Master’s thesis, Massachusetts Institute of Technology, Feb 2002.*, 2002.

[84] S. Hauck, “Configuration prefetch for single context reconfigurable coprocessors,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 65–74, February 1998.

[85] K. Chatha and R. Vemuri, “Hardware-software codesign for dynamically reconfigurable architectures,” in *9th International Workshop on Field-Programmable Logic and Applications (FPL 99)*, (Glasgow, UK), pp. 175–184, Springer-Verlag Lecture Notes in Computer Science (LNCS), September 1999.

[86] R. H. A. Hirschbiel and M. Weber, “A novel paradigm of parallel computation and its use to implement simple high performance hardware,” in *Proceedings of the Joint International Conference on Vector and Parallel Processing*, pp. 51–62, March 1990.

[87] R. Hartenstein, J. Becker, and R. Kress, “Two-level partitioning of image processing algorithms for the parallel map-oriented machine,” in *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, pp. 77–84, March 1996.

[88] L. Lin, “High-level synthesis, introduction to chip and system design,” Kluwer Acad. Publ., Boston, London, 1992.

[89] M. A. George, M. Pink, D. Kearney, and G. Wigley, “Efficient allocation of FPGA area to multiple users in an operating system for reconfigurable computing,” in *Proceedings of Engineering of Reconfigurable Systems and Algorithms (ERSA02)*, pp. 238–242, 2002.

[90] H. Walder and M. Platzner, “Online scheduling for block-partitioned reconfigurable devices,” in *In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, (Munich, Germany), pp. 290–295, 2003.

[91] M. Dales, “Managing a reconfigurable processor in a general purpose workstation environment,” in *In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, (Munich, Germany), pp. 10980–10985, 2003.

[92] B. Jeong, S. Yoo, S. Lee, and K. Choi, “Hardware-software cosynthesis for runtime incrementally reconfigurable FPGAs,” in *Proceedings of Asia and South Pacific DAC*, pp. 169–174, January 2000.

[93] E. Moscu Panainte, K. Bertels, and S. Vassiliadis, “The Molen compiler for reconfigurable processors,” *ACM Transactions in Embedded Computing Systems (TECS)*, vol. 6(1), February 2007.

[94] E. Moscu Panainte, K. Bertels, and S. Vassiliadis, “Interprocedural compiler optimization for partial run-time reconfiguration,” *Journal of VLSI Signal Processing*, vol. 43(2), pp. 161–172, May 2006.

[95] P. Barth, “A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization,” Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[96] Sundance, “Fc-jpeg04 jpeg compression design specification,” (http://www.sundance.com/docs/FC-JPEG04 Sundance - 300504.pdf), pp. 1–4, 2004.

[97] S. Vassiliadis, E. A. Hakkennes, S. Wong, and G. G. Pechanek, “The sum-of-absolute-difference motion estimation accelerator,” in *Proceedings of the 24th Euromicro Conference*, (Vasteras, Sweden), pp. 559–566, Aug 1998.

List of Publications

International Journals

1. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **The Molen Compiler for Reconfigurable Processors**, ACM Transactions in Embedded Computing Systems (TECS), February 2007, Volume 6 , Issue 1
2. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **Interprocedural Compiler Optimization for Partial Run-Time Reconguration**, Journal of VLSI Signal Processing, pp. 161-172, May 2006, Volume 43, Number 2
3. S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, E. Moscu Panainte, **The Molen Polymorphic Processor**, IEEE Transactions on Computers, pp. 1363- 1375, November 2004, Volume 53, Issue 11

International Conferences

1. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **Compiler-driven FPGA-area Allocation for Reconfigurable Computing**, Proc. of Design, Automation and Test in Europe 2006 (DATE 06), pp. 369-374, March 2006
2. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **Instruction Scheduling for Dynamic Hardware Configurations**, Proc. of Design, Automation and Test in Europe 2005 (DATE 05), pp. 100-105, 2005
3. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **Interprocedural Optimization for Dynamic Hardware Configurations**, Proc. of the International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS 05), pp. 2-11, July 2005, Springer-Verlag Lecture Notes in Computer Science (LNCS)

4. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **The PowerPC Backend Molen Compiler**, in 14th International Conference on Field-Programmable Logic and Applications (FPL'04), pp. 434-443, September 2004, Springer-Verlag Lecture Notes in Computer Science (LNCS), vol. 3203

5. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **Dynamic Hardware Reconfigurations: Performance Impact on MPEG2**, Proc. of the International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS 04), pp. 284-292, July 2004, July 2003, Springer-Verlag Lecture Notes in Computer Science (LNCS), vol. 3133

6. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **Multimedia Reconfigurable Hardware Design Space Exploration**, Proc. of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), pp. 398-403, November 2004

7. E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, **Compiling for the Molen Programming Paradigm**, Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL'03), pp. 900-910, September 2003, Springer-Verlag Lecture Notes in Computer Science (LNCS), vol. 2778

8. S. Vassiliadis, G. N. Gaydadjiev, K.L.M. Bertels, E. Moscu Panainte, **The Molen Programming Paradigm**, Proc. of the Third International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS 03), pp. 1-10, July 2003, Springer-Verlag Lecture Notes in Computer Science (LNCS), vol. 3133

Publications not directly related to this dissertation

1. CG Galuzzi, E. Moscu Panainte, Y. D. Yankova, K.L.M. Bertels, S. Vassiliadis, **Automatic Selection of Application-Specific Instruction-Set Extensions**, Proc. of the 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS 2006), pp. 160-165, October 2006

2. E. Moscu Panainte, I. Athanasiu, S. D. Cotofana, **An Optimization Framework for Retargetable Compilers**, Proc.. 13th International Conference on Control Systems and Computer Sciences (CSCS-13), pp. 427-432, May 2001

Samenvatting

In **dit** proefschrift presenteren wij de opzet van de Molen compiler bedoeld voor herconfigureerbare architecturen die vallen onder het Molen Programeer Paradigma. In het bijzonder introduceren wij een pakket van compiler optimalisaties dat één van de belangrijkste tekortkomingen van herconfigureerbare architecturen, namelijk configuratie overhead, tegengaat. De voorgestelde optimalisaties zijn gebaseerd op inter- en intra procedurele dataflow analyse, met inachtneming van de concurrentie strijd om de beschikbare herconfigureerbare hardware en van de ruimte-tijd toewijzing. De hardware configuratie instructies zijn geplaatst voor de hardware operatie instructies, zodat gebruik wordt gemaakt van het aanwezige parallellisme tussen de hardware configuratie fase en de sequentiële uitvoer van operaties op de hoofd processor. De introprocedurele optimalisatie maakt gebruik van het 'min s-t cut' graaf algoritme met het doel het aantal hardware configuraties te verminderen door de overtollige configuraties te identificeren. Daarnaast presenteren wij twee algoritmen voor het toewijzen van de beschikbare herconfigureerbare hardware, die het totale te herconfigureren gebied minimaliseren en de totale prestatie verbetering maximaliseren. Gebaseerd op profilering en software/hardware schattingen, genereren de compiler optimalisaties en de toewijzing algoritmes geoptimaliseerde code voor de bedoelde herconfigureerbare architectuur en toepassing, zodanig dat deze voldoet aan de ruimte-tijd beperkingen. Tevens assisteren zij bij de keuze tussen hardware en software uitvoering van de operaties die geschikt zijn voor uitvoering op de herconfigureerbare hardware. Ten einde de Molen compiler te evalueren, presenteren wij, ten eerste, een experiment met een toepassing uit een multi-media benchmark, gecompileerd met de Molen compiler en uitgevoerd op de Molen polymorphic media processor. Het programma blijkt 2,5 keer sneller voltooid te zijn op de herconfigureerbare hardware dan het geval is bij een pure software aanpak. Vervolgens maken wij de inschatting dat de intraprocedurele compiler optimalisaties tot 94 % aan prestatie verbetering, vergeleken met een pure software benadering, bijdragen,

terwijl de intraprocedurele compiler optimilisaties en de toewijzing algoritmes het aantal herconfiguraties aanzienlijk vermindert voor de gebruikte benchmarks. Ten slotte stellen wij vast dat de belangrijke invloed, van onze compiler optimalisaties en toewijzing algoritmes, op de prestaties zullen toenemen voor toekomstige snellere FPGAs.

Curriculum Vitae

Elena Moscu Panainte was born on the 21st of January 1977 in Adjud, Romania. After finishing her secondary education at ”Liceul Teoretic Emil Botta, Adjud”, she studied Computer Science at the Faculty of Automatic Control and Computers of ”Politehnica” University Bucharest, where she graduated as M.Sc. in 2000 on a retargetable framework for compiler optimizations. The work for her thesis was carried out during a six-month scholarship at Computer Engineering group, TU Delft. Between 2000 and 2002, she was a teaching assistant at the Faculty of Automatic Control and Computers, Bucharest.

In 2002, she started her Ph.D studies at Computer Engineering group, TU Delft. She worked in DelftWorkBench project under the supervision of Prof. Stamatis Vassiladis and Prof. Koen Bertels. Her work was focused on compiler optimizations for reconfigurable architectures. The results of this research is presented in this thesis.

Her main research interests include: Compiler Design and Optimizations, Reconfigurable Computing, Hardware/Software co-Design, Embedded Systems, Computer Architecture and Image Processing.