Linköping Studies in Science and Technology

Dissertations. No. 1680

Hardware/Software Codesign of Embedded Systems with Reconfigurable and Heterogeneous Platforms

by

Adrian Alin Lifa

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2015

Copyright © 2015 Adrian Alin Lifa

ISBN 978-91-7519-040-2

ISSN 0345-7524

Printed by LiU Tryck 2015

URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-117637>

To my family

ABSTRACT

MODERN applications running on today's embedded systems have very high requirements. Most often, these requirements have many dimensions: the applications need high performance as well as flexibility, energy-efficiency as well as real-time properties, fault tolerance as well as low cost. In order to meet these demands, the industry is adopting architectures that are more and more heterogeneous and that have reconfiguration capabilities. Unfortunately, this adds to the complexity of designing streamlined applications that can leverage the advantages of such architectures.

In this context, it is very important to have appropriate tools and design methodologies for the optimization of such systems. This thesis addresses the topic of hardware/software codesign and optimization of adaptive real-time systems implemented on reconfigurable and heterogeneous platforms. We focus on performance enhancement for dynamically reconfigurable FPGA-based systems, energy minimization in multi-mode real-time systems implemented on heterogeneous platforms, and codesign techniques for fault-tolerant systems.

The solutions proposed in this thesis have been validated by extensive experiments, ranging from computer simulations to proof of concept implementations on real-life platforms. The results have confirmed the importance of the addressed aspects and the applicability of our techniques for design optimization of modern embedded systems.

POPULÄRVETENSKAPLIG SAMMANFATTNING

IDAG är inbyggda system vanligt förekommande och deras antal fortsätter att öka. De används inom en mängd olika domäner, t.ex. hemelektronik, fordonsindustri, flygelektronik, medicin, etc., och de hittas nästan överallt omkring oss: från våra telefoner, bärbara datorer och tvättmaskiner till våra bilar. De applikationer som körs på dessa system har flera olika ökande krav: högprestanda, energieffektivitet, flexibilitet, feltolerans, realtidsegenskaper, och naturligtvis låg kostnad. För att kunna uppfylla dessa krav har industrin anammat arkitekturer som är mer och mer heterogena och som har omkonfigureringsmöjligheter. Olyckligtvis ökar detta komplexiteten när man ska utforma effektiva inbyggda system. I detta sammanhang blir det av yttersta betydelse att utveckla effektiva optimeringsmetoder och verktyg.

Dagens applikationer består av en blandning av mjukvarukomponenter som har mycket olika energi- och prestandaegenskaper beroende på vilka hårdvaruenheter där de körs på, vilket gör dem lämpliga för heterogena plattformar. En heterogen plattform består av olika typer av hårdvaruenheter, var och en med sina egenskaper, som riktar sig till vissa tillämpningsområden. Under det senaste decenniet har utvecklingen av rekonfigurerbara hårdvarutekniker accelererat, vilket bidrar till den ökande populariteten för fältprogrammerbara grindmatriser (FPGA). Idag, ger hårdvarutillverkare stöd för partiell dynamisk omkonfigurering; detta innebär att delar av en FPGA kan konfigureras vid run-time, medan andra delar förblir fullt fungerande. Dessa tekniska framsteg har uppenbara fördelar, men kan också göra arbetet med att utforma inbyggda system mer komplext. Forskarvärlden har ansvar för att utveckla effektiva verktyg och föreslå konstruktionsmetoder och tekniker som gör det möjligt för designers att utveckla högpresterande, energieffektiva och säkra inbyggda system som i slutändan kommer att kunna förbättra vår vardag.

Bidragen i denna avhandling består av en mängd verktyg och konstruktionsmetoder för adaptiva system som gör det möjligt för designers att an-

vända de tillgängliga begränsade resurserna så effektivt som möjligt för att nå de mål som ålagts. Vi fokuserar på prestandaförbättringar för dynamiskt omkonfigurerbara system, energiminimering i multi-mode realtidssystem för heterogena plattformar och codesigntekniker för feltoleranta system.

ACKNOWLEDGEMENTS

It was a long journey, but it was the best part of my life so far. And what made it the best were the people.

First and foremost I want to thank my advisers, Petru Eles and Zebo Peng. They complement each other perfectly, in a charming way, and they made me not only a better researcher, but most importantly a better person.

Petru became my friend, he is one of the most intelligent, passionate and caring persons I know, one that I look up to with admiration and respect, and that will be a role model for the rest of my life. From the first email we exchanged (when I applied for the position in ESLAB), to the moment I finalized this thesis, he was constantly supportive and making sure I gave the best out of me. For everything, I am thankful.

Zebo was the best lab leader I could have ever wished for. Besides his valuable research-related support, he was also the one to encourage me to take the position on the SaS board, which was a positive experience. Furthermore, he was the one to organize the numerous badminton sessions that made our weeks more relaxing and more productive. Thank you Zebo.

The administrative staff at IDA was simply impeccable. Anne Moe was the most helpful, patient and supportive coordinator of graduate studies we could have had. Inger Norén was always available and made things happen. Inger Emanuelsson and Marie Johansson took care of the administrative issues with professionalism. Mariam Kamkar, our head of department, made sure that IDA is a great place to conduct research at.

My deepest appreciation to Eva Pelayo Danils and Åsa Kärrman, who made our life painless by solving administrative issues at the blink of an eye.

I also take the opportunity to thank Nahid Shahmehri, for showing interest in my work and well-being. She has my respect and my appreciation.

The people I met in ESLAB and IDA during my PhD years had a positive influence on me, and I want to collectively thank all of them for that. From the serious discussions up to the crazy lunch topics, the foosball, the badminton, everything made my time here enjoyable and helped me grow.

Sergiu was the one who took care of me in the beginning, gave me valuable advice and showed me his friendship. Slava, with his unique and contagious way of being, made my transition to the PhD life smooth. Dimitar was a good friend, and an awesome (fanatic, like me) squash partner. Breeta constantly reminded me how important it is to keep my childish side. By contrast, Nima taught me that sometimes it is important to be serious, and Farrokh taught me patience and persistence. Ivan was my confident in Russian matters, and one whose discipline I admired and envied. Arian was a good friend, one who listened and comforted me when I needed it, and one whose company I will always enjoy.

My thanks to Erik, for proofreading my “populärvetenskaplig sammanfattning” and to Urban, for helping me with my job search.

Special thanks go to Soheil, for being a good friend, for his valuable support in my job search process, and for proofreading my Swedish texts.

With Bogdan I have shared all the ups and downs, and it was great to have his friendship. Besides the countless interesting discussions on all possible topics, and all the funny jokes, he was also the one who introduced me to chess, and I am thankful for all that. I will also remember his favorite quote: “Everything will be ok in the end. If it’s not ok, it’s not the end.”

I want to thank Amir, another good friend, for making me a better person. I will miss all the intense discussions, all the late night dinners, all the jokes we shared, all the amazing biking and canoeing trips we made.

Adi crossed my path only recently; but the time I knew him was sufficient to befriend him and to think that he is cool enough.

Then there are my two good office neighbors: Ke and Unmesh. They were the ones I would turn to first, when times got harder. They were good friends, providing both research-related support, and personal one.

I would like to thank Sarah, for proofreading my “populärvetenskaplig sammanfattning”, and for teaching me to *always* question *everything*.

I thank Iulia, for encouraging me to come to ESLAB in the first place, and for her constant and unconditional support during our years together.

To all my friends, wherever in the world they are, goes my love and my most sincere thanks, for all the energy they voluntarily donated to me.

My sister. I thank her for loving me like nobody else, and for knowing how to always make me feel smart, no matter how stubborn I was. Confidence is very important when doing a PhD, and she gave me that.

Last but not least, my family. I could not thank them enough for always providing me everything that I needed, for the sacrifices they made for me, for their altruistic desire to see me happy and for their unlimited and unconditional love. I love you back!

ADRIAN ALIN LIFA
Linköping
15 Sep. 2015

CONTENTS

1	Introduction	1
1.1	Motivation	1
1.1.1	Performance Enhancement and Flexibility	2
1.1.2	Multi-Mode Behavior and Energy Minimization	2
1.1.3	Fault Tolerance and Error Detection Optimization	3
1.2	Summary of Contributions	4
1.3	List of Publications	5
1.4	Thesis Overview	6
2	Background and Related Work	7
2.1	Field Programmable Gate Arrays	7
2.1.1	Partial Dynamic Reconfiguration	7
2.1.2	Static and Hybrid Configuration Prefetching	8
2.1.3	Dynamic Configuration Prefetching	10
2.2	Heterogeneous Architectures	11
2.2.1	Multiprocessor Systems	11
2.2.2	Real-Time Computing on GPUs	12
2.3	Fault-Tolerant and Multi-Mode Systems	13
2.3.1	Fault-Tolerant Safety-Critical Applications	13
2.3.1.1	Fault-Tolerant Scheduling	14
2.3.1.2	Error Detection	14
2.3.2	Multi-Mode Behavior	15
3	FPGA Configuration Prefetching	17
3.1	System Model	18
3.1.1	Hardware Platform	18
3.1.1.1	Reconfigurable Slots	20
3.1.1.2	Custom Reconfiguration Controller	21
3.1.1.3	Piecewise Linear Predictor	22
3.1.2	Middleware and Reconfiguration API	23

3.1.2.1	Partial Reconfiguration	23
3.1.2.2	Reconfigurable Slots	23
3.1.2.3	Piecewise Linear Predictor	23
3.1.3	Application Model	24
3.2	Static FPGA Configuration Prefetching	25
3.2.1	Problem Formulation	25
3.2.2	Motivational Example	25
3.2.3	Speculative Prefetching	28
3.2.3.1	Prefetch Priority Function	30
3.2.3.2	Expected Execution Time Gain	31
3.2.3.3	Execution Time Distribution	34
3.2.4	Experimental Evaluation	38
3.2.4.1	Monte Carlo Simulator	38
3.2.4.2	Simulation Results	39
3.2.4.3	Case Study – GSM Encoder	42
3.2.4.4	Case Study – Floating Point Benchmark	44
3.3	Dynamic FPGA Configuration Prefetching	46
3.3.1	Problem Formulation	46
3.3.2	Motivational Example	46
3.3.3	Dynamic Prefetching	49
3.3.3.1	The Piecewise Linear Predictor	49
3.3.3.2	Expected Execution Time Gain	53
3.3.3.3	Prefetch Priority Function	55
3.3.3.4	Run-Time Strategy	57
3.3.4	Hardware Organization of the Predictor	59
3.3.5	Experimental Evaluation	61
3.3.5.1	Proof of Concept	61
3.3.5.2	Performance Improvement	67
3.4	Summary	69
4	Multi-Mode Systems	71
4.1	System Model	72
4.1.1	Architecture Model	72
4.1.2	Application Model	72
4.1.3	Resource Management	73
4.1.4	Scheduling	74
4.2	Problem Formulation	75
4.3	Motivational Example	75
4.4	Optimization for One Mode	81
4.4.1	ILP Formulation	81
4.4.2	Fast Heuristic	83
4.5	Multi-Mode Optimization	86
4.5.1	Caching of Solutions	87
4.5.2	Estimating the Mode Duration	87
4.5.3	Run-Time Policies	87

4.5.3.1	Heuristic + ILP	88
4.5.3.2	Heuristic-Only	88
4.6	Experimental Evaluation	88
4.6.1	Real-Life Measurements	88
4.6.2	Simulation Results	90
4.6.2.1	Energy Consumption	91
4.6.2.2	Rejection Rate	92
4.7	Summary	92
5	Fault-Tolerant Systems	93
5.1	Preliminaries: Error Detection Technique	93
5.2	Real-Time Distributed Embedded Systems	96
5.2.1	Optimization Framework	96
5.2.2	System Model	97
5.2.3	Problem Formulation	98
5.2.4	Motivational Examples	99
5.2.5	Static FPGA Reconfiguration	102
5.2.5.1	Tabu Search Moves	103
5.2.5.2	Neighborhood Restriction	105
5.2.5.3	Move Selection	106
5.2.5.4	Diversification	107
5.2.6	Partial Dynamic Reconfiguration	107
5.2.6.1	Revised System Model	108
5.2.6.2	Revised Problem Formulation	108
5.2.6.3	Revised Scheduler	108
5.2.6.4	Anti-fragmentation Policy	109
5.2.6.5	Illustrative Example	111
5.2.6.6	Tuning the Scheduler	112
5.2.7	Experimental Evaluation	113
5.2.7.1	Static FPGA Reconfiguration	114
5.2.7.2	Partial Dynamic Reconfiguration	116
5.2.7.3	Case Study – Adaptive Cruise Controller	119
5.3	Average Execution Time Minimization	121
5.3.1	Basic Concepts	122
5.3.2	System Model	124
5.3.3	Problem Formulation	124
5.3.4	Motivational Example	125
5.3.5	Speculative Reconfiguration	128
5.3.5.1	Reconfiguration Schedule Generation	128
5.3.5.2	FPGA Area Management	132
5.3.6	Experimental Evaluation	133
5.3.6.1	Experimental Results	134
5.3.6.2	Case Study – GSM Encoder	135
5.4	Summary	137

6	Conclusions and Future Work	139
6.1	Conclusions	139
6.1.1	FPGA Configuration Prefetching	139
6.1.2	Multi-Mode Systems	140
6.1.3	Fault-Tolerant Systems	140
6.2	Future Work	141

LIST OF FIGURES

3.1	A general architecture model for partial dynamic reconfiguration of FPGAs	19
3.2	The internal architecture of the reconfigurable slot RS_2 . . .	21
3.3	The custom reconfiguration controller	22
3.4	Motivational example for static FPGA configuration prefetching	26
3.5	Computing the gain probability distribution step by step . . .	33
3.6	Forward control dependence tree	34
3.7	Comparison with state-of-the-art [S ⁺ 10] for the synthetic benchmarks	41
3.8	Control flow graphs for the case studies	43
3.9	Comparison with state-of-the-art [S ⁺ 10] for the GSM encoder case study	44
3.10	Comparison with state-of-the-art [S ⁺ 10] for the floating point benchmark case study	45
3.11	Motivational example for dynamic FPGA configuration prefetching	47
3.12	The 3D weight array (Ω) of the predictor	50
3.13	The internal architecture of the predictor	59
3.14	The hardware organization of the predictor	60
3.15	The hardware implementation of corner detection	63
3.16	Average execution time reduction for SUSAN	65
3.17	Performance improvement for the simulation experiments . .	68
4.1	A heterogeneous architecture model for multi-mode systems .	72
4.2	Motivational example for on-the-fly energy minimization for multi-mode real-time systems	77
4.3	Experimental evaluation for the simulation experiments . . .	91
5.1	Code fragment with error detectors	94
5.2	Optimization framework overview	97

5.3	System model for fault-tolerant distributed embedded systems	97
5.4	Motivational examples for the optimization of error detection implementation	100
5.5	Architecture and application for motivational example 2 . . .	101
5.6	Swap move example for Tabu Search	104
5.7	Restricting the neighborhood for Tabu Search	105
5.8	Architecture and application example for neighborhood re- striction	105
5.9	Using the fragmentation metric to place modules on the FPGA according to the anti-fragmentation policy	110
5.10	Modified scheduler for systems with partial dynamic recon- figuration capabilities	111
5.11	The ranges for the random generation of EDI overheads . . .	113
5.12	Experiments' space	114
5.13	Comparison with theoretical optimum for statically reconfig- urable FPGAs	115
5.14	Impact of varying the hardware fraction for statically recon- figurable FPGAs	116
5.15	Impact of varying the number of tasks/processor for statically reconfigurable FPGAs	117
5.16	Impact of varying the hardware fraction for partially dynam- ically reconfigurable FPGAs	118
5.17	Impact of varying the number of tasks/processor for partially dynamically reconfigurable FPGAs	119
5.18	Average running times for the optimization heuristics	119
5.19	Task graph for the adaptive cruise controller case study . . .	120
5.20	Results for the adaptive cruise controller case study	120
5.21	Motivational example for speculative reconfiguration of error detection components	125
5.22	Performance improvement	134
5.23	Reconfiguration schedule table size	135
5.24	Results for the GSM encoder case study	137

LIST OF TABLES

3.1	Resource utilization for the partial dynamic reconfiguration framework	20
3.2	Hardware candidates' characteristics	48
4.1	Task parameters for the motivational example	76
4.2	Board measurements – CPU and GPU	89
4.3	Board measurements – FPGA	90
5.1	Worst-case execution times and error detection overheads for the motivational example	99
5.2	Task characteristics for the illustrative example	112
5.3	Time and area overheads for the adaptive cruise controller case study	121
5.4	Error detection overheads for the motivational example . . .	126
5.5	Conditional reconfiguration schedule tables for the motivational example	127
5.6	Time and area overheads of checkers for the GSM encoder case study	136

LIST OF ABBREVIATIONS

ACC	Adaptive Cruise Controller
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BB	Branch and Bound
BRAM	Block Random Access Memory
CDG	Control Dependence Graph
CE	Checking Expression
CFG	Control Flow Graph
CLB	Configurable Logic Block
CP	Critical Path
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
ECFG	Extended Control Flow Graph
EDI	Error Detection Implementation
EDT	Error Detection Technique
EST	Earliest Start Time
FCC	Fragmentation Contribution of a Cell
FCDG	Forward Control Dependence Graph
FCDT	Forward Control Dependence Tree
FF	Flip-Flop
FOD	Fetch-On-Demand
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HW	HardWare
ILP	Integer Linear Program
LUT	Look-Up Table
PCP	Partial Critical Path
PDR	Partial Dynamic Reconfiguration
PI	Performance Improvement

pmf	probability mass function
SUSAN	Smallest Univalve Segment Assimilating Nucleus
SW	SoftWare
TF	Total Fragmentation
USAN	Univalve Segment Assimilating Nucleus
WCET	Worst-Case Execution Time
WCP	Waiting (tasks on) Critical Path
WCSL	Worst-Case Schedule Length
WCTT	Worst-Case Transmission Time
2D	2-Dimensional
3D	3-Dimensional

INTRODUCTION

THE TOPIC of this thesis is hardware/software codesign and optimization of adaptive real-time systems implemented on reconfigurable and heterogeneous platforms. The main contributions include performance optimizations for dynamically reconfigurable FPGA-based systems, codesign methodologies for fault-tolerant systems that leverage the advantages of the aforementioned platform, and optimization algorithms for the minimization of energy consumption in multi-mode real-time systems implemented on heterogeneous platforms composed of CPUs, GPUs and FPGAs. In this chapter we shall introduce and motivate these research topics, as well as the existing challenges. Last we shall summarize the contribution of this thesis and outline its organization.

1.1 Motivation

Today's embedded systems have ever increasing requirements in many different dimensions: high performance, flexibility, energy-efficiency, real-time properties, fault tolerance, and, of course, low cost [Mar10], [Kop11]. Furthermore, the applications running on these systems have a high level of complexity, often exhibiting dynamic and non-stationary behavior, or having multi-mode characteristics [SSC03]. In this context, the use of reconfigurable and heterogeneous architectures attempts to address these stringent requirements [PTW10]. However, in order to leverage the advantages of such architectures, careful optimization is needed. The contribution of this thesis is a set of optimization tools and design methodologies for adaptive real-time systems that enable the designer to use the available limited resources as efficiently as possible in order to achieve the goals imposed.

1.1.1 Performance Enhancement and Flexibility

The development of reconfigurable hardware technologies, as well as the advances made in the area of design methodologies and tools, have contributed to the increasing popularity of dynamically reconfigurable hardware platforms [PTW10], like field programmable gate arrays (FPGAs) [Koc13]. Beside the obvious advantages compared to application specific integrated circuits (ASICs), like field reprogrammability and faster time-to-market, FPGAs also present the potential to implement fast and efficient systems, with high performance gains over equivalent software applications.

Many modern systems that require both hardware acceleration and high flexibility are suitable for FPGA implementation. Today, manufacturers provide support for partial dynamic reconfiguration (PDR) [Koc13], i.e. parts of the FPGA may be reconfigured at run-time, while other parts remain fully functional [Xil12]. This extra flexibility comes with one major drawback: the time overhead to perform partial dynamic reconfiguration. One technique that addresses this problem is configuration prefetching, which tries to preload future configurations and overlap as much as possible of the reconfiguration overhead with useful computations.

Chapter 3 presents our contributions to static and dynamic FPGA configuration prefetching¹, together with a complete framework for partial dynamic reconfiguration of FPGAs.

1.1.2 Multi-Mode Behavior and Energy Minimization

Many modern applications exhibit a multi-mode behavior, their computational requirements varying over time: certain tasks are active during a certain period of time, determining the load during that period and defining a *mode*. The application enters a new mode if the set of active tasks changes. For systems with dynamic and non-stationary behavior, the information regarding mode changes, or even the modes themselves, are usually unavailable at design-time. Thus, on-line optimization algorithms coupled with adaptive platforms are necessary to ensure energy-efficiency while meeting the real-time constraints.

In many of today's embedded systems, the typical workloads consist of a mix of tasks which have very different performance and power consumption characteristics depending on the processing element to which they are mapped, which makes them suitable for heterogeneous architectures. Platforms composed of CPUs, GPUs and FPGAs are becoming more and more widespread, partly due to the energy, performance and flexibility requirements of modern applications. The industry is making strong efforts to facilitate the adoption of heterogeneous architectures by streamlining the process of designing and programming applications for them [Fou15].

¹Configuration compression and caching [Li02] are complementary techniques that can be used in conjunction with our approaches.

Heterogeneous architectures have the advantage that each type of processing element provides substantial improvement (in terms of energy consumption, performance, flexibility, etc.) within its target domain [CLS⁺08]. GPUs have been shown to be efficient, e.g., for medical imaging [MLH⁺12], network packet processing [SGO⁺09], molecular dynamics [ALT08] and, of course, graphics (multimedia), image processing and other massively parallel applications. While the suitability of GPUs for throughput-oriented applications is accepted, their use in a real-time context is an open research issue. There are strong motivations for utilizing GPUs in real-time systems [EA11], [KLRI11]; their use can increase the performance orders of magnitude (thus, possibly improving system responsiveness, important in real-time systems), at a fraction of the power needed by traditional CPUs. FPGAs have been shown to be suitable for data mining [BP05], bioinformatics [HJL⁺07], query acceleration for databases [BBZT14], digital signal processing [ASG13], [BGG13], pattern matching and image processing [SBHT13], [dRGLH06], etc. Furthermore, as mentioned in Section 1.1.1, most FPGA manufacturers provide support for partial dynamic reconfiguration [Xil12]. This flexibility is highly beneficial in the context of multi-mode systems [WAST12], [WRZT13].

Chapter 4 presents our contribution to the optimization of multi-mode real-time systems implemented on heterogeneous platforms.

1.1.3 Fault Tolerance and Error Detection Optimization

Safety-critical applications must function correctly even in the presence of faults. Such faults might be transient, intermittent or permanent. Modern electronic systems are experiencing an increase in the rate of transient and intermittent faults [Con03], [MBT04], [HMA⁺01]. From the fault tolerance point of view, transient and intermittent faults manifest themselves very similarly: they have a short duration and then disappear without causing permanent damage. Thus, we will further refer to both types as transient faults².

Error detection is crucial for meeting the required reliability of the system. Unfortunately, it is also a major source of time overhead. To reduce this overhead, the error detection mechanisms could be implemented in hardware, but this increases the overall cost. Because error detection incurs high overheads, optimizing it early in the design phase of a system can result in a big gain. The advantages of partial dynamic reconfiguration of FPGAs, discussed in Section 1.1.1, can be used to optimize error detection such that the system's cost is kept at a minimum while the real-time constraints are met.

Chapter 5 presents our contributions to the optimization of error detection implementation for fault-tolerant systems.

²Permanent faults are not addressed in this thesis.

1.2 Summary of Contributions

This thesis addresses the area of system-level design optimization and code-sign of adaptive real-time embedded systems implemented on reconfigurable and heterogeneous platforms. The main contributions can be divided into three major parts which are treated in Chapters 3-5, respectively.

In Chapter 3 we start by introducing a reconfigurable framework for performance enhancement using partial dynamic reconfiguration of FPGAs [LEP15a]. We propose a hardware implementation based on commercial tools, together with a comprehensive API that enables designers to integrate their applications into our framework with minimal effort. The main challenge for dynamically reconfigurable systems is their high reconfiguration overhead. In order to address this issue we propose two configuration prefetching approaches. The first approach (static) schedules prefetches speculatively at design-time and simultaneously performs hardware/software partitioning in order to minimize the expected execution time of an application [LEP12b], [LEP12c]. The second approach targets applications that exhibit a dynamic and non-stationary phase behavior. The optimization technique dynamically schedules prefetches at run-time based on a piecewise linear predictor [LEP13]. Our methods achieve high degrees of adaptability and answer the performance requirements of cost-constrained systems.

In Chapter 4 we address the problem of energy efficiency in the context of multi-mode real-time systems implemented on heterogeneous platforms composed of CPUs, GPUs and FPGAs [LEP15b]. We consider applications that change their computational requirements over time and have tight timing constraints; thus, intelligent on-line resource management is essential. We propose a resource manager that implements run-time policies to decide on-the-fly task admission and the mapping of active tasks to resources, such that the energy consumption of the system is minimized and all deadlines are met.

In Chapter 5 we present system-level optimizations for error detection implementation in the context of fault-tolerant real-time distributed embedded systems used for safety-critical applications. We address the problem from two angles: inter-task and intra-task optimization. In the first case, we propose hardware/software codesign techniques that leverage the advantages of partial dynamic reconfiguration of FPGAs in order to minimize the global worst-case schedule length of an application, while meeting the imposed hardware cost constraints and tolerating multiple transient faults [LEP10]. In the latter case, we propose a technique to minimize the average execution time of a program by speculatively prefetching on the FPGA those error detection components that will provide the highest performance improvement [LEP11].

1.3 List of Publications

Parts of this thesis have been presented in the following publications:

- Adrian Lifa, Petru Eles, Zebo Peng and Viacheslav Izosimov. “Hardware/Software Optimization of Error Detection Implementation for Real-Time Embedded Systems.” *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2010)*, part of the *Embedded Systems Week*, Scottsdale, AZ, USA, October 24-29, 2010 [LEPI10].
- Adrian Lifa, Petru Eles and Zebo Peng. “Performance Optimization of Error Detection Based on Speculative Reconfiguration.” *48th Design Automation Conference (DAC 2011)*, San Diego, CA, USA, June 5-10, 2011 [LEP11].
- Adrian Lifa, Petru Eles and Zebo Peng. “Minimization of Average Execution Time Based on Speculative FPGA Configuration Prefetch.” *International Conference on ReConFigurable Computing and FPGAs (ReConFig 2012)*, Cancun, Mexico, December 5-7, 2012 [LEP12b].
- Adrian Lifa, Petru Eles and Zebo Peng. “Dynamic Configuration Prefetching Based on Piecewise Linear Prediction.” *Design, Automation & Test in Europe (DATE 2013)*, Grenoble, France, March 18-22, 2013 [LEP13].
- Adrian Lifa, Petru Eles and Zebo Peng. “A Reconfigurable Framework for Performance Enhancement with Dynamic FPGA Configuration Prefetching.” Accepted for publication in the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems Journal*, 2015 [LEP15a].
- Adrian Lifa, Petru Eles and Zebo Peng. “On-the-fly Energy Minimization for Multi-Mode Real-Time Systems on Heterogeneous Platforms.” *IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia 2015)*, part of the *Embedded Systems Week*, Amsterdam, The Netherlands, October 4-9, 2015 [LEP15b].

The following publications are not included in this thesis but are directly related to the field of reconfigurable real-time embedded systems:

- Adrian Lifa, Petru Eles and Zebo Peng. “Context-Aware Speculative Prefetch for Soft Real-Time Applications.” *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2012)*, Seoul, Korea, August 19-22, 2012 [LEP12a].
- Ke Jiang, Adrian Lifa, Petru Eles, Zebo Peng and Wei Jiang. “Energy-Aware Design of Secure Multi-Mode Real-Time Embedded Systems

with FPGA Co-Processors.” *21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, Sophia Antipolis, France, October 16-18, 2013 [JLE⁺13].

1.4 Thesis Overview

This thesis is organized in six chapters. In **Chapter 2** we discuss related research results in the area of real-time embedded systems implemented on reconfigurable and heterogeneous platforms. We shall present the current problems and challenges, highlighting the contributions of this thesis related to the state-of-the-art in the development and optimization of adaptive embedded systems.

In **Chapter 3** we present a reconfigurable framework for performance enhancement using partial dynamic reconfiguration of FPGAs. We propose two design optimizations for FPGA configuration prefetching: one static, that prepares prefetches at design-time, and another one dynamic, that adaptively decides the prefetches at run-time, suitable for applications with dynamic and non-stationary behavior.

In **Chapter 4** we address the problem of energy minimization for multi-mode real-time systems implemented on heterogeneous platforms composed of CPUs, GPUs and FPGAs. We propose an on-the-fly optimization that performs resource management such that the platform is used in an energy-efficient manner while the timing constraints are met.

In **Chapter 5** we focus on fault-tolerant real-time distributed embedded systems used for safety-critical applications. We propose design approaches that leverage the advantages of partial dynamic reconfiguration of FPGAs in order to optimize the error detection implementation for cost- and time-constrained systems.

In **Chapter 6** we conclude this thesis and we outline several directions of future research that build on the contribution presented here.

BACKGROUND AND RELATED WORK

THE PURPOSE of this chapter is to review the research efforts made in the area of hardware/software codesign and optimization of adaptive real-time systems implemented on reconfigurable and heterogeneous platforms. Section 2.1 covers the background and state-of-the-art related to partial dynamic reconfiguration of FPGAs and algorithms for static, hybrid and dynamic configuration prefetching. Section 2.2 presents the related work, advantages and challenges in the area of heterogeneous systems composed of CPUs, GPUs and FPGAs. Finally, Section 2.3 presents the necessary background and the related work concerning the topics of fault-tolerant and multi-mode real-time applications, respectively.

2.1 Field Programmable Gate Arrays

In recent years, dynamically reconfigurable hardware platforms [PTW10], especially those based on field programmable gate arrays (FPGAs), have been employed for a large class of applications because of their advantages: field reprogrammability, flexibility, faster time-to-market, and the potential to deliver high performance gains over equivalent software implementations.

2.1.1 Partial Dynamic Reconfiguration

Many modern applications that require both hardware acceleration and high flexibility are suitable for FPGA implementation. Today, manufacturers provide support for partial dynamic reconfiguration (PDR) [Koc13], which means that parts of the FPGA may be reconfigured at run-time, while other parts remain fully functional [Xil12].

There has been a lot of work related to dynamically reconfigurable systems [PTW10]. Especially relevant for this thesis are those approaches that make use of self-reconfiguring FPGA platforms [Koc13]. To the best of our knowledge, one of the first references to self-reconfiguration using an FPGA

device was presented in [ML99]. The authors describe in detail the design and physical implementation of the architecture, engineered on a Xilinx custom development platform, XC6216, one of the first FPGA families with partial dynamic reconfiguration (PDR) support.

Virtex-II and Virtex-II Pro were the successor families with PDR support, that included improved features, which are now present in most modern FPGAs. One example is the internal configuration access port (ICAP), which is a functional configuration interface accessible from inside the FPGA. In [BJRK⁺03], the authors present a platform in which the FPGA is reconfiguring itself through the ICAP under the control of an embedded CPU (Xilinx MicroBlaze). This implementation requires no external circuitry to control the reconfiguration process.

Since then, several other self-reconfiguring platforms have been proposed. For example, the authors of [LKLJ09] investigate the performance of five different ICAP reconfiguration controllers. They all use the (now obsolete) Processor Local Bus (PLB). Another PLB design is presented in [DML11]. Its main focus is on reducing the reconfiguration time overhead, by using techniques such as ICAP overclocking or bitstream compression. The authors of [BPPC12] present a reconfiguration controller that supports dynamic frequency scaling in order to satisfy different power constraints.

Despite the advantages of PDR, one major barrier in its wide adoption in industrial applications seems to be the lack of mature high-level design tools and methodologies [TK11]; the few available ones are often cumbersome to use. Given this state of facts, in Chapter 3 we propose an IP-based architecture, together with an API, that is easy to deploy using the state-of-the-art, commercially available design suite from Xilinx¹. The framework hides away all the details related to PDR and lets the designer focus on application development.

The extra flexibility offered by PDR comes with another major drawback: the potentially high time overhead to perform partial reconfiguration. One technique that addresses this problem is configuration prefetching, which tries to preload future configurations and overlap as much as possible of the reconfiguration overhead with useful computations.

2.1.2 Static and Hybrid Configuration Prefetching

The literature contains a multitude of papers that approach, from different angles, the problem of partitioning and scheduling applications on reconfigurable architectures. Many of these works leverage the advantages of *static* FPGA configuration prefetching. The authors of [CRR⁺09] proposed a partitioning algorithm, as well as an ILP formulation and a heuristic approach to scheduling of task graphs. In [BBD05] the authors present an exact and a heuristic algorithm that simultaneously partitions and schedules task graphs

¹Note that the framework is general and can be implemented on any FPGA architecture that supports partial dynamic reconfiguration (e.g. [Alt12]).

on FPGAs. A similar problem is addressed in [CRM14], where the authors present an approach to manage reconfigurations while ensuring that the temporal constraints of hard real-time applications are met. The work presented in [CBR⁺14] introduces a mapper-scheduler for temporal constrained data flow diagrams that aims at reducing the reconfiguration overhead, using as few hardware resources as possible, while meeting the application deadline. Most of these works have the disadvantage that they do not consider the control flow. For a large class of applications, by ignoring the control flow, many prefetch opportunities are missed.

To our knowledge, the works most closely related to our own, presented in Section 3.2, are [S⁺10], [LH02] and [PBV05]. Panainte et al. proposed both an intra-procedural [PBV05] and an inter-procedural [PBV06] static prefetch scheduling algorithm that minimizes the number of executed FPGA reconfigurations taking into account FPGA area placement conflicts. In order to compute the locations in the control flow graph of an application where hardware reconfigurations can be anticipated, they first determine the regions of the graph not shared between any two conflicting hardware modules, and then insert prefetches at the beginning of each such region. This approach is too conservative and a more aggressive speculation could hide more reconfiguration overhead. Also, profiling information (such as branch probabilities and execution time distributions) could be used to prioritize between two non-conflicting modules.

Li et al. continued the pioneering work of Hauck [Hau98] in configuration prefetching. They compute the probabilities to reach any hardware module, based on profiling information [LH02]. This algorithm can be applied only after all the loops in the control flow graph of the application are identified and collapsed into dummy nodes. Then, the hardware modules are ranked at each basic block according to these probabilities and prefetches are issued. The main limitations of this work are that it removes all loops (which leads to loss of path information) and that it uses only probabilities to guide prefetch insertion (without taking into account execution time distributions, for example). Also, this approach was developed for FPGAs with relocation and defragmentation, and it does not account for placement conflicts between modules.

To our knowledge, the state-of-the-art in static configuration prefetching for partially reconfigurable FPGAs is the work of Sim et al. [S⁺10]. The authors present an algorithm that minimizes the reconfiguration overhead for an application, taking into account FPGA area placement conflicts. Using profiling information, the approach tries to predict the execution of hardware modules by computing ‘placement-aware’ probabilities (PAPs). They represent the probabilities to reach a hardware module from a certain basic block without encountering any conflicting hardware module on the way. These probabilities are then used in order to generate prefetch queues to be inserted by the compiler in the control flow graph of the application. The main limitation of this work is that it uses only the ‘placement-aware’

probabilities to guide prefetch insertion. As we will show in Section 3.2, it is possible to generate better prefetches (and, thus, further reduce the execution time of the application) if we also take into account the execution time distributions, correlated with the reconfiguration time of each hardware module.

The authors of [MSP⁺12], [R⁺05] and [RCG⁺08] present *hybrid* heuristics that identify a set of possible application configurations at design-time and then, at run-time, a resource manager chooses among them. In [Li02], the author also proposes a hybrid prefetch heuristic that performs part or all of the scheduling computations at run-time and also requires additional hardware. Although these approaches provide more flexibility than static ones, they are still limited by the fact that they rely on off-line information. For example, in the case of applications with highly non-stationary behavior it might be impossible to get accurate and complete profiling information. In such cases, a *dynamic* technique is more suitable.

2.1.3 Dynamic Configuration Prefetching

Several run-time resource managers for reconfigurable architectures have been developed. The authors of [JTY⁺99] describe a dynamically reconfigurable system that can support multiple applications running concurrently and implement a strategy to preload FPGA configurations in order to reduce the execution time. The authors of [HHSC10] propose a scheduling algorithm that can cope with dynamically relocating tasks from software to hardware, or vice versa. The authors of [PSA10] propose reconfiguration strategies for minimizing the number of reconfigurations.

In [HV09], the authors continue their work from [HV08], and propose an on-line algorithm that manages coprocessor loading by maintaining an aggregate gain table for all hardware candidates. For each run of a candidate, the performance gain resulted from a hardware execution over a software one is added to the corresponding entry in the table. When a coprocessor is considered for reconfiguration, the algorithm only loads it when the aggregate gain exceeds the reconfiguration overhead. One limitation of this work is that it does not perform prefetching, i.e it does not overlap the reconfiguration overhead with useful computations. Another difference between all the works discussed so far and ours, presented in Section 3.3, is that none of the above papers explicitly consider the control flow in their application model. Furthermore, they also ignore correlations.

The authors of [LH02] propose a dynamic prefetch heuristic that represents hardware modules as the vertices in a Markov graph. Transitions are updated based on the modules executed at run-time. Then, a weighted probability in which recent accesses are given higher weight is used as a metric for candidate selection and prefetch order. The main limitations of this work are that it uses only the weighted probabilities for issuing prefetches (ignoring other factors as, e.g., the execution time gain resulting from differ-

ent prefetch decisions), and that it uses a history of length 1 (i.e. it predicts only the next module to be executed based on the current module, consequently completely ignoring branch correlations). As we will show in Section 3.3, it is possible to obtain significant improvements by taking into account the branch history and capturing correlations, as well as by estimating the execution time gains associated with different prefetches.

2.2 Heterogeneous Architectures

Much work has been done in the area of heterogeneous systems. In some domains, using FPGAs (and their partial dynamic reconfiguration capabilities) alongside CPUs in order to accelerate tasks or increase energy efficiency is frequent practice (e.g. [SBH14], [PTW10]). However, the increasing use of GPUs for general processing tasks has determined researchers to look into heterogeneous architectures that consist of CPUs together with FPGA and GPU accelerators [NOS13], [CMHM10].

2.2.1 Multiprocessor Systems

The authors of [TL10] present Axel, a heterogeneous computer cluster, in which CPUs, FPGAs and GPUs run collaboratively. The architecture is described, together with a Map-Reduce framework to be used for implementing distributed applications. In [LHK09], the authors propose an adaptive mapping technique that automatically maps computations to processing elements on a CPU+GPU machine. They build performance models per CPU and accelerator, then use these models at run-time to balance the workload across processing elements and, thus, minimize the execution time. In [BBG13] the authors propose a workload balancing scheme for the execution of loops on heterogeneous multiprocessor systems composed of CPUs+GPUs or CPUs+FPGAs. Their algorithm dynamically learns the computational power of each processing element and then maps the workload accordingly. The authors of [BLBY11] present a machine learning algorithm to perform dynamic task scheduling and load balancing. In [HLS11], the authors present an utilization balancing approach to scheduling of functionally heterogeneous systems.

In [PRP⁺15] the authors present a design flow for customizing OpenCL applications in order to maximize their performance on heterogeneous platforms with multiple accelerators. They take into account device-specific constraints in a task tuning phase and, after that, they improve the task-level parallelism in a mapping phase. For the validation, the authors have used two heterogeneous platforms, one composed of CPU+GPU, and another one composed of four quad-core CPUs. In [ZSJ10] the authors present a Pareto efficient optimization approach that optimizes buffer requirements and hardware/software implementation cost for streaming applications on

a CPU+FPGA platform. Neither one of the above mentioned articles considers energy minimization as an objective, and neither one of them targets real-time computations.

The authors of [KFP⁺15] address the problem of scheduling dynamically-arriving tasks in a high performance heterogeneous computing system that is energy-constrained. They propose energy-aware resource allocation heuristics whose goal is to maximize the total utility of the system based on each task's completion time. This problem is different from the one that we address in Chapter 4, since we consider tasks with hard deadlines. Furthermore, we target multi-mode systems implemented on heterogeneous platforms with CPUs, GPUs and FPGAs.

Mapping optimizations for many-core systems are extensively addressed in the existing literature, targeting different application domains (see, e.g. [JPT⁺10], [hKYS⁺12], [QP13a]). The authors of [JLK⁺14] present a hybrid approach that performs compile-time scheduling and then chooses between the stored schedules at run-time. In [QP13b], the authors present a scenario-based task mapping algorithm for MPSoCs, based on statically derived mappings. The authors of [SBR⁺12] present a scenario-based design flow for mapping streaming applications on many-core systems. In [DSB⁺13], a DVFS technique for scenario-aware dataflow graphs is proposed, which assures timing guarantees while minimizing the energy consumption. The main limitation of these works is that they assume that the scenarios (analogue to our modes from Chapter 4) are known at design-time. There exist applications for which the scenarios are unknown, or their number is exponential. In such cases, adaptive on-the-fly optimizations are needed.

2.2.2 Real-Time Computing on GPUs

While GPUs have been traditionally used for graphics and throughput-oriented applications, in recent years the idea of using GPUs to perform real-time processing has been proposed [EA11], [EA12]. In [EA11], the authors explore possible applications for GPUs in real-time systems, summarizing the challenges and limitations, and discussing possible solutions to address them. In [EA12], the same authors present two analysis methods that permit the integration of GPUs in soft real-time multiprocessor systems. They follow up their work in [EWA13], where they propose and analyze GPUSync, a highly configurable real-time GPU management framework.

The authors of [VMS14] present a method that allows real-time applications to run in multi-GPU systems, by efficiently using the communication infrastructure and, at the same time, maintaining execution time predictability. They rely on executing batch operations from multiple command streams that can run in parallel. The authors of [MLH⁺12] investigate resource management and scheduling techniques for medical imaging

applications that employ GPU accelerators. The work proposes a scheduler capable to utilize multiple GPUs in order to minimize the response time of multiple applications with soft real-time requirements. Both works mentioned above do not address multi-mode systems and do not take energy minimization into consideration.

The recent work presented in [MBH⁺14] proposes an approach to scheduling hard real-time jobs from data parallel streams, such that the energy consumption of a GPU-based heterogeneous system is minimized. The authors developed a heuristic to generate a static cyclic schedule and to map jobs to computation resources. Although this work addresses the problem of energy minimization for real-time systems, its limitations lie in the simplified application model, and the fact that the approach is not designed to adapt to changing run-time conditions.

2.3 Fault-Tolerant and Multi-Mode Embedded Systems

This section will discuss the background and previous work related to two important topics in the embedded and real-time systems community: the first topic is that of fault-tolerant systems (which must function correctly even in the presence of faults), while the second topic addresses multi-mode systems (whose computational requirements vary over time).

2.3.1 Fault-Tolerant Safety-Critical Applications

Modern electronic systems are experiencing an increase in the rate of transient and intermittent faults. This happens because of several reasons, like smaller transistor sizes, higher operational frequencies, lower voltages [Con03], [MBT04], [HMA⁺01]. From the point of view of the fault tolerance techniques, transient and intermittent faults manifest themselves very similarly: they have a short duration and then disappear without causing permanent damage. Thus, we will further refer to both types as transient faults.

The topic of fault-tolerant systems has been addressed extensively in the literature, from many different angles: starting with general design optimization methods [IPEP05], [GGPM11], [SPM10], continuing with optimizations of control systems [SBE⁺12], [MEP05], and ending with considerations related to fault-tolerant communications [TBEP10], [TBEP11]. The considerable amount of research in this area highlights the importance of the topic.

2.3.1.1 Fault-Tolerant Scheduling

In this section we limit the discussion to transient faults². One line of previous work mainly focused on optimizing different fault tolerance techniques, while considering error detection as a black box: [KHM03], [IPEP05], [IPEP06], [IPEP08], [IPP⁺09]. In [KHM03], the authors present an approach to construct fault-tolerant schedules with sufficient slack to accommodate recovery and re-execution of at most one faulty task in a system period. In [IPEP05], a design optimization approach is presented that decides both the mapping of tasks to processors and the assignment of fault-tolerant policies (re-execution or replication) to processes. In [IPEP06], a more advanced scheduling algorithm is proposed, that makes use of the fault-occurrence information to reduce the overhead due to fault tolerance. In [IPP⁺09] a design optimization approach is presented, which combines hardware and software fault tolerance techniques, in order to achieve the required levels of reliability with low system costs. All the approaches discussed so far consider error detection as a black box. Thus, no optimization is done in order to reduce the overhead (or cost) of the error detection techniques.

2.3.1.2 Error Detection

Error detection is crucial for meeting the required reliability of the system. Unfortunately, it is also a major source of time overhead. To reduce this overhead, the error detection mechanisms could be implemented in hardware, but this increases the overall cost. Because error detection incurs high overheads, optimizing it early in the design phase of a system can result in a big gain.

An important body of previous work refers to various error detection techniques, both software-based and hardware-based: [BGFM06], [BMR⁺08], [HLD⁺09], [RCV⁺05], [PKI11]. In [BGFM06], the authors present two low-cost soft error protection techniques implemented in hardware. The first one uses a cache of live register values to protect the register file. The second technique augments a subset of flip-flops in the processor core with time-delayed shadow latches for fault detection. This is a pure hardware-only technique, and thus does not allow any trade-off between area and time overhead. In [BMR⁺08], the authors develop a solution that uses both software and hardware approaches to achieve high fault coverage in generic IP processors. The software part consists of instruction replication and self checking block signatures. Partial hardware replication is applied on top, considering a subset of processor registers. In [HLD⁺09], a technique to duplicate instructions at compile time for error detection in VLIW datapaths is presented. The need of comparison instructions is eliminated by using a hardware enhancement for result verification. In [RCV⁺05], the authors present SWIFT (software implemented fault tolerance), inserting redundant

²Permanent faults are not addressed in this thesis.

code to recompute all register values and using validation instructions before control-flow and memory operations. The paper also presents CRAFT (compiler-assisted fault tolerance), which is a hybrid hardware/software technique, that extends SWIFT with microarchitectural enhancements (an augmented store buffer that commits entries to memory only after they are validated, and a load value queue that achieves redundant load execution).

Recently, the concept of application-aware reliability has been introduced as an alternative to traditional one-size-fits-all approaches (like many of those presented above). Application-aware techniques [PKI11] use the knowledge about the application's characteristics to create customized solutions. Recent research has proved that it is possible to obtain high error coverage, with low percentage of benign errors detected, at reasonable cost and performance overheads [PKI11]. As a result, in Chapter 5 we will focus our attention on this technique and present approaches to optimize its hardware/software implementation.

2.3.2 Multi-Mode Behavior

Another important topic, addressed by this thesis in Chapter 4, is that of multi-mode applications; in this section we will present the background and previous work related to it.

Many modern applications exhibit a multi-mode behavior, their computational requirements varying over time: certain tasks are active during a certain period of time, determining the load during that period and defining a *mode*. A new mode is entered by the application if the set of active tasks changes. For systems with dynamic and non-stationary behavior [SSC03], [SBSH12], the information regarding mode changes, or even the modes themselves, are usually unavailable at design-time. Such applications are too dynamic to be implemented as fixed designs; instead, it is very important to have adaptive hardware platforms and flexible on-line optimization algorithms that can ensure that the application's constraints are met.

The authors of [WAST12] address the problem of modeling and dynamically placing, at run-time, multi-mode streaming applications on FPGA-based platforms. This allows for resource sharing between tasks which run mutually exclusive in different modes. The paper does not address energy minimization or real-time applications.

The authors of [SEPC09] address the synthesis of multi-mode embedded control systems. Since a control system can switch at run-time between alternative functional modes, the approach tries to exploit the available resources in each particular mode in order to optimize its control performance. The authors of [WPQ⁺12] propose a formal visual modeling framework which can be used to specify and analyze periodic control systems that exhibit a multi-mode behavior. Both these works are particular to control systems.

In [JLE⁺13] the authors follow up their work from [JEP13] and present a design framework that maximizes the security protection of a multi-mode real-time system in an energy-efficient manner, while meeting all the deadlines. The approach pre-computes off-line solutions for a subset of all the possible functional modes of the system. At run-time, if the solution for a certain mode has been pre-computed it will be applied; otherwise, a solution is derived from the existing ones. The limitation of this work is that for systems with dynamic and non-stationary behavior the information regarding the modes is, in most cases, unavailable at design-time. Thus, it is impossible to prepare solutions off-line.

The authors of [SAHE05] present a co-synthesis methodology that takes into account mode execution probabilities in order to statically decide the implementation of a multi-mode application such that the energy consumption is minimized and the timing constraints are satisfied. While taking into account mode execution probabilities is useful, this information is not always available (e.g. for applications with non-stationary behavior) and better results could be obtained with run-time management of resources.

PERFORMANCE ENHANCEMENT BASED ON FPGA CONFIGURATION PREFETCHING

IN RECENT years, FPGA-based reconfigurable computing systems have gained popularity because they promise to satisfy the simultaneous needs of high performance and flexibility [PTW10]. Modern FPGAs provide support for partial dynamic reconfiguration [Xil12], which means that parts of the device may be reconfigured at run-time, while the other parts remain fully functional. This feature offers high flexibility, but does not come without challenges: one major impediment is the high reconfiguration overhead. Configuration prefetching is one method to reduce this penalty by overlapping FPGA reconfigurations with useful computations.

Despite the potential advantages of partial dynamic reconfiguration of FPGAs [Koc13], the challenges faced by designers trying to set-up a functioning system are still significant, mainly because of the still immature design tools and limited device drivers [TK11]. In this chapter we first describe a complete framework, based on Xilinx's commercial design suite, that enables an application designer to leverage the advantages of partial dynamic reconfiguration with minimal effort. Our IP-based architecture, together with the comprehensive API, can be employed to accelerate an application by dynamically scheduling hardware prefetches. Based on this framework, we further propose two approaches to configuration prefetching for performance enhancement:

1. The first one is a speculative approach that schedules prefetches at design-time and simultaneously performs hardware/software partitioning, in order to minimize the expected execution time of an application. Our method prefetches and executes in hardware those configurations that provide the highest performance improvement. The algorithm takes into consideration profiling information (such as branch proba-

bilities and execution time distributions), correlated with the application characteristics.

2. The second approach addresses modern applications that exhibit a dynamic and non-stationary behavior, with certain characteristics in one phase of their execution, which change as the application enters new phases, in a manner unpredictable at design-time. In order to meet the demands of such applications, it is important to have adaptive and self-reconfiguring hardware platforms, coupled with intelligent on-line optimization algorithms, that together can adjust to the run-time requirements. Thus, we propose an optimization technique that minimizes the expected execution time of an application by dynamically scheduling hardware prefetches based on a piecewise linear predictor that captures correlations and predicts the hardware modules that will be reached.

The remainder of this chapter is organized as follows. Section 3.1 introduces our framework for partial dynamic reconfiguration of FPGAs, detailing the hardware platform together with the middleware and API, and discussing the application model assumed in this chapter. Our approaches to static and dynamic FPGA configuration prefetching are presented in Sections 3.2 and 3.3, respectively. The contribution of the chapter is summarized in Section 3.4.

3.1 System Model

3.1.1 Hardware Platform

Many dynamically reconfigurable systems are implemented using FPGA platforms [PTW10]. One common architecture choice is to partition the FPGA into a static region, and a partially dynamically reconfigurable (PDR) region (used as a coprocessor for hardware acceleration) [Koc13]. A host CPU resides in the static region, together with a reconfiguration controller and other hardware modules that need not change at run-time. In the PDR region, the application hardware modules can be dynamically loaded at run-time [SBB⁺06]. The host CPU executes the software part of the application and is also responsible for initiating the reconfiguration of the PDR region of the FPGA. The reconfiguration controller will configure this region by loading the bitstreams from the memory, upon CPU requests. While one reconfiguration is going on, the execution of the other (non-overlapping) modules on the FPGA is not affected.

Figure 3.1 presents our overall architecture, which is described next. The entire framework has been implemented and tested on a ML605 board from Xilinx¹, featuring an XC6VLX240T Virtex6 FPGA, which was one of

¹Note that the framework is general and can be implemented on any FPGA architecture that supports partial dynamic reconfiguration (e.g. [Alt12]).

Figure 3.1: A general architecture model for partial dynamic reconfiguration of FPGAs

Table 3.1: Resource utilization for the partial dynamic reconfiguration framework

	LUTs	FFs	BRAMs (36 Kb)
AXI4 Interconnect	886	1016	1
AXI4-Lite Interconnect	190	455	0
MicroBlaze	2114	2186	6
Interrupt Controller	78	115	0
DDR3 Controller	3264	3750	0
AXI Central DMA	1442	1263	1
AXI DMA Engine	1061	905	3
Reconfiguration Controller	202	272	0
Predictor	419	151	0
Reconfigurable Slot	133	192	0
BRAM Controller	281	446	0
BRAM Block	0	0	32
Total	10070	10751	43

the most advanced platforms at the time this research started. Table 3.1 presents the resource utilization for the different parts of our framework.

3.1.1.1 Reconfigurable Slots

We will first introduce some terminology. A *reconfigurable slot* is a dedicated region on the FPGA that contains a *reconfigurable partition*, which basically represents a placeholder that can be reconfigured dynamically with arbitrary functionality. A *reconfigurable module* is a netlist that resides in a reconfigurable partition. Multiple reconfigurable modules are mapped to the same reconfigurable partition (similar ideas and designs are proposed by the RecoBlock SoC platform [NSO13] and by the Erlangen Slot Machine [BMA⁺05]), but all of them must have the same fixed interface. At run-time, in order to configure a reconfigurable partition with a certain reconfigurable module, the partial bitstream for that module needs to be written to the internal configuration access port (ICAP).

Figure 3.1 includes two possible types of reconfigurable slots, RS_1 and RS_2 . Please note that this is just an illustrative example, and more instances of the same type of slot could coexist, as well as there could be more types of slots. As can be seen, RS_1 has a slave interface to the AXI4-Lite interconnect, which can be accessed by the embedded CPU (MicroBlaze in our case). The slot has an interrupt port that can be used to signal the CPU. For data-intensive applications, a fast local memory might be useful. For this purpose, RS_1 has direct access to a dual port BRAM block. Data can be transferred there with minimal intervention from the CPU, using the AXI Central DMA.

Figure 3.2: The internal architecture of the reconfigurable slot RS_2

Reconfigurable slot RS_2 differs from RS_1 by not having access to a shared BRAM block but, instead, benefiting from a master interface to the AXI4 interconnect. Figure 3.2 presents the internal organization of such a reconfigurable slot. The slot is connected to the AXI4 and AXI4-Lite interconnects through the corresponding AXI IP Interfaces (IPIF), master and slave, which simplify the implementation of the user logic (the reconfigurable partition). Of course, other types of interfaces could also be implemented, depending on the application needs.

The main advantage of such a design is the generality of the reconfigurable slots. The designer needs to implement only the logic which is particular to the actual application (user logic in Figure 3.2). Note that only this area (the reconfigurable partition) will be partially reconfigured at run-time, and everything else is static logic that will remain unchanged. An enable signal can be used to isolate the user logic until it is completely reconfigured, and all output signals should be registered on the static side. The local reset should be asserted in the user logic after reconfiguration has completed to ensure a known valid initial state.

3.1.1.2 Custom Reconfiguration Controller

All the bitstreams are stored in the DDR3 memory ($Bitstream_1$ to $Bitstream_n$ in Figure 3.1). When a reconfiguration is requested, the bitstream is transferred to the ICAP using DMA burst transfers. Thus, the CPU (MicroBlaze) is almost completely out of the loop. After setting up the DMA transfer, MicroBlaze can perform useful computations in parallel with the reconfiguration. We have implemented a custom reconfiguration controller, which is described below.

In Figure 3.1 the AXI DMA Engine streams the bitstream data through the Xilinx AXI4-Stream (AXI4S) interface (a high throughput point-to-point connection) to our reconfiguration controller, which forwards it to the ICAP. Similar solutions have been proposed in the literature (e.g. [LKLJ09],

Figure 3.3: The custom reconfiguration controller

[DML11]). The controller's architecture is depicted in Figure 3.3. On completion of every DMA transfer, the CPU (MicroBlaze) is notified via an interrupt. The CPU will use this opportunity to check if the reconfiguration has finished without errors. The main part of the controller is implemented as a finite state machine (FSM), that is clocked by the AXI4-Stream clock (100 MHz in our case) and has the main duty to deliver the reconfiguration stream to the ICAP. A bit ordering module takes care that the data has the format expected by ICAP. The AXI4-Lite IPIF (IP interface) is used by the MicroBlaze to read the reconfiguration status registers.

3.1.1.3 Piecewise Linear Predictor

We have performed a hybrid hardware/software implementation of the piecewise linear prediction algorithm (described in detail in Section 3.3.3.1), which is used to predict the hardware candidates that should be speculatively prefetched on the PDR region (i.e. the modules that should be loaded into the reconfigurable slots). Part of its functionality is implemented as a set of API functions (see Section 3.1.2.3), and part of it as a hardware module (Section 3.3.4 presents its architecture). The predictor has a slave interface, which is connected to the AXI4-Lite interconnect (see Figure 3.1), and is used by the embedded CPU (MicroBlaze) to transfer commands and relevant data for the prefetch prediction. Note that the predictor is used only by our dynamic configuration prefetching approach described in Section 3.3.

3.1.2 Middleware and Reconfiguration API

Our extensive API can be divided into several sets of utility functions, as described below.

3.1.2.1 Partial Reconfiguration

These functions hide most of the details from the application programmer and allow for:

- Initializing the reconfiguration components (the AXI DMA engine and the custom reconfiguration controller).
- Starting partial reconfigurations by setting up the DMA transfers. After that, the embedded CPU (MicroBlaze) is free to perform other useful computations.
- Treating the interrupts when partial reconfigurations finish. This includes reading the status registers of the DMA engine and of the reconfiguration controller, and dealing with eventual errors in case of failure.
- Managing the reconfigurable modules in the system: keeping track of which modules are currently configured and their status, as well as resetting them.

3.1.2.2 Reconfigurable Slots

These functions are the interface between the MicroBlaze and the hardware modules placed in the reconfigurable slots. They permit:

- Reading the status registers and writing the control registers (if any) of the reconfigurable modules, via the AXI4-Lite Interconnect.
- Sending and receiving data, either by reading/writing software mapped registers, or by setting up DMA transfers to shared BRAM blocks.
- Providing interrupt subroutines for the programmer, which deal with eventual interrupts coming from the reconfigurable modules.

3.1.2.3 Piecewise Linear Predictor

Our framework permits dynamic configuration prefetching by implementing a piecewise linear prediction algorithm (described in detail in Section 3.3.3.1). The application programmer will instrument the program using functions for:

- Initializing all the data structures needed by the predictor (see Section 3.3.3.1).

- Updating all the predictor data, e.g. execution frequencies for hardware modules, timestamps, the weights and all the branch histories used by the predictor.
- Performing the actual prediction, whose result will determine the modules to be prefetched.

3.1.3 Application Model

The main goal of our approaches (presented in Sections 3.2 and 3.3 respectively) is to minimize the expected execution time of a sequential program executed on the hardware platform described above. We model the application as a control flow graph $\mathcal{G}(\mathcal{N}, \mathcal{E})$, where nodes in $\mathcal{N}$ correspond to computational tasks (either basic blocks, or hardware modules), and edges in $\mathcal{E}$ model the flow of control within the application ($\mathcal{G}$ captures all potential execution paths). We distinguish several types of nodes: *root* and *sink*, correspond to the entry and the exit of the program; *control nodes* represent conditional instructions in the program; *loop header* nodes represent the entry points of loops, and they are the target of a *back edge* (i.e. an edge that points to a block that has already been met during a depth-first traversal of the graph). All the other nodes in $\mathcal{N}$ are either regular basic blocks (that will be executed only in software), or hardware modules (described below).

We denote with $\mathcal{H} \subseteq \mathcal{N}$ the set of tasks (also referred to as modules) considered for hardware implementation. We assume that all modules in $\mathcal{H}$ have both a hardware and a corresponding software implementation. Since sometimes it might be impossible to hide enough of the reconfiguration overhead for all candidates in $\mathcal{H}$, our techniques decide which are the most profitable modules to perform prefetches for (at a certain point in the application). Thus, for some candidates, it might be better to execute the module in software, instead of performing a prefetch too close to its location (because waiting for the reconfiguration to finish and then executing the module on the FPGA is slower than executing it in software).

Note that there exist many established methodologies (e.g. [SHTT13], [BPCF13], [NSO⁺12], [BSY⁺10], [LCD⁺00]) to perform the hardware/software partitioning step (i.e. determine the set $\mathcal{H}$ automatically), that will decide which are the hardware candidates to be (potentially) implemented on the FPGA. Alternatively, this decision can be taken by the designer, who has in-depth knowledge about the application domain. Intuitively, the hardware candidates in $\mathcal{H}$ will represent those parts of the application which are computationally intensive, and at the same time amenable for FPGA acceleration. Please note that it is not necessary that all candidate modules from $\mathcal{H}$ will end up on the FPGA. Given a particular choice for $\mathcal{H}$, our techniques will try to use the limited hardware resources as efficiently as possible in order to maximize the performance of the application.

At run-time, once a candidate $m \in \mathcal{H}$ is reached, there are two possibilities:

1. m is already fully loaded in a reconfigurable slot on the FPGA, and thus it will be used and executed there;
2. m is not fully loaded. Then, we face two scenarios:
 - (a) if starting/continuing the reconfiguration of m , waiting for it to finish, and then executing the module on FPGA results in an earlier finishing time than the software execution, then the application will do so;
 - (b) otherwise, m will be executed in software.

For each hardware candidate $m \in \mathcal{H}$, we assume that we know its software execution time, $sw : \mathcal{H} \rightarrow \mathbb{R}^+$, its hardware execution time (including any additional communication overhead between the CPU and the reconfigurable partition hosting m), $hw : \mathcal{H} \rightarrow \mathbb{R}^+$, and the time to reconfigure m on the FPGA, $rec : \mathcal{H} \rightarrow \mathbb{R}^+$.

3.2 Static FPGA Configuration Prefetching

In this section we will present our contribution to static configuration prefetching. We use profiling information (e.g. the function $prob : \mathcal{E} \rightarrow [0, 1]$ represents the probability of each edge in the control flow graph to be taken, and for each loop header n , $iter_prob_n : \mathbb{N} \rightarrow [0, 1]$ represents the probability mass function of the discrete distribution of loop iterations) to decide at design-time what prefetches to apply in order to obtain the best performance improvement for the application.

3.2.1 Problem Formulation

Given an application (as described in Section 3.1.3) intended to run on the reconfigurable architecture described in Section 3.1.1, our goal is to determine, at each node $n \in \mathcal{N}$, the candidate modules to be prefetched (stored in $loadQ(n)$) by the middleware described in Section 3.1.2 such that the expected execution time of the application is minimized. This will implicitly also determine the hardware/software partitioning of the candidate modules from $\mathcal{H}$.

3.2.2 Motivational Example

Let us consider the control flow graph (CFG) in Figure 3.4a, where candidate hardware modules are represented with squares, and software nodes with circles. The discrete probability distribution for the iterations of the loop $a - b$, the software and hardware execution times for the nodes, as well as the edge probabilities, are illustrated on the graph. The reconfiguration times are: $rec(M_1) = 37$, $rec(M_2) = 20$, $rec(M_3) = 46$. We also consider

Figure 3.4: Motivational example for static FPGA configuration prefetching

that hardware modules M_1 and M_2 are conflicting due to their placement (denoted with $M_1 \bowtie M_2$).

Let us try to schedule the configuration prefetches for the three hardware modules on the given CFG. If we use the method developed by Panainte et al. [PBV05], the result is shown in Figure 3.4b. As we can see, the *load* for M_3 can be propagated upwards in the CFG from node M_3 up to r . For nodes M_1 and M_2 it is not possible (according to this approach) to propagate their *load* calls to their ancestors, because they are in placement conflict. The data-flow analysis performed by the authors is too conservative, and the propagation of prefetches is stopped whenever two *load* calls targeting conflicting modules meet at a common ancestor (e.g. node f for M_1 and M_2). As a result, since the method fails to prefetch modules earlier, the reconfiguration overhead for neither M_1 , nor M_2 , can be hidden at all. Only module M_3 will not generate any waiting time, since the minimum time to reach it from r is $92 > \text{rec}(M_3) = 46$. Using this approach, the application must stall (waiting for the reconfigurations to finish) $W_1 = 90\% \cdot \text{rec}(M_1) + \text{rec}(M_2) = 90\% \cdot 37 + 20 = 53.3$ time units on average (because M_1 is executed with a probability of 90%, and M_2 is always executed).

Figure 3.4c shows the resulting prefetches after using the method proposed by Li et al. [LH02]. As we can see, the prefetch queue generated by this approach at node r is $\text{loadQ}(r) : M_2, M_3, M_1$, because the probabilities to reach the hardware modules from r are 100%, 95% and 90% respectively. Please note that this method is developed for FPGAs with relocation and defragmentation and it ignores placement conflicts. Also, the load queues are generated considering only the probability to reach a module (and ignoring other factors, such as the execution time distribution from the prefetch point up to the prefetched module). Thus, if applied to our example, the method performs poorly: in 90% of the cases, module M_1 will replace module M_2 (initially prefetched at r) on the FPGA. In this cases, none of the reconfiguration overhead for M_1 can be hidden, and in addition, the initial prefetch for M_2 is wasted. The average waiting time for this scenario is $W_2 = 90\% \cdot \text{rec}(M_1) + (100\% - 10\%) \cdot \text{rec}(M_2) = 90\% \cdot 37 + 90\% \cdot 20 = 51.3$ time units (the reconfiguration overhead is hidden in 10% of the cases for M_2 , and always for M_3).

For this example, although the approach proposed by Sim et al. [S⁺10] tries to avoid some of the previous problems, it ends up with similar waiting time as Li et al. [LH02]. The method uses ‘placement-aware’ probabilities (PAPs). For any node $n \in \mathcal{N}$ and any hardware module $m \in \mathcal{H}$, $\text{PAP}(n, m)$ represents the probability to reach module m from node n , without encountering any conflicting hardware module on the way. Thus, the prefetch order for M_1 and M_2 is correctly inverted since $\text{PAP}(r, M_1) = 90\%$, as in the previous case, but $\text{PAP}(r, M_2) = 10\%$, instead of 100% (because in 90% of the cases, M_2 is reached via the conflicting module M_1). Unfortunately, since the method uses only PAPs to generate prefetches, and $\text{PAP}(r, M_3) = 95\%$ (since it is not conflicting with neither M_1 , nor M_2), M_3 is prefetched before

M_1 at node r , although its prefetch could be safely postponed. The result is illustrated in Figure 3.4d (M_2 is removed from the load queue of node r because it conflicts with M_1 , which has a higher PAP). These prefetches will determine that no reconfiguration overhead can be hidden for M_1 or M_2 (since the long reconfiguration of M_3 postpones their own one until the last moment). The average waiting time for the application will be $W_3 = 90\% \cdot \text{rec}(M_1) + \text{rec}(M_2) = 90\% \cdot 37 + 20 = 53.3$ time units.

If we examine the example carefully, we can see that taking into account only the ‘placement-aware’ probabilities is not enough. The prefetch generation mechanism should also consider the distance from the current decision point to the hardware modules candidate for prefetching, correlated with the reconfiguration time of each module. Our approach, presented in this section, is to estimate the performance gain associated with starting the reconfiguration of a certain module at a certain node in the CFG. We do this by considering both the execution time gain resulting from the hardware execution of that module (including any stalling cycles spent waiting for the reconfiguration to finish) compared to the software execution, and by investigating how this prefetch influences the execution time of the other reachable modules. For the example presented here, it is not a good idea to prefetch M_3 at node r , because this results in a long waiting time for M_1 (similar reasoning applies for prefetching M_2 at r). The resulting prefetches are illustrated in Figure 3.4e. As we can see, the best choice of prefetch order is M_1, M_3 at node r (M_2 is removed from the load queue because it conflicts with M_1), and this will hide most of the reconfiguration overhead for M_1 , and all for M_3 . The overall average waiting time is $W = 90\% \cdot \overline{W}_{rM_1} + \text{rec}(M_2) = 90\% \cdot 4.56 + 20 \approx 24.1$, less than half of the penalties generated by the previous methods (Section 3.2.3.2 and Figure 3.5 explain the computation of the average waiting time generated by M_1 , $\overline{W}_{rM_1} = 4.56$ time units).

3.2.3 Speculative Prefetching

Our goal is to determine, at each node $n \in \mathcal{N}$ in the CFG, a queue of modules to be prefetched at run-time when that point in the program is reached. We will denote this load queue with $\text{load}Q(n)$. Our overall strategy is shown in Algorithm 1. The main idea is to intelligently assign priorities to the candidate prefetches and determine the $\text{load}Q(n)$ for every node $n \in \mathcal{N}$ (line 8). We try to use all the available knowledge from the profiling in order to take the best possible decisions and speculatively prefetch the hardware modules with the highest potential to reduce the expected execution time of the application. The intelligence of our algorithm resides in computing the priority function C_{nm} (line 5), which tries to estimate at design-time what is the impact of reconfiguring a certain module on the average execution time (see Section 3.2.3.1). We consider for prefetch only the modules for which it is profitable to start a prefetch at the current point (line 4): either

Algorithm 1 Generating the prefetch queues**Input:** $\mathcal{N}, \mathcal{H}, PAP(n, m)$, profiling information**Output:** $loadQ(n)$ = prefetches to apply at node n

```

1: procedure GENERATEPREFETCHQ
2:   for all  $n \in \mathcal{N}$  do
3:     for all  $\{m \in \mathcal{H} | PAP(n, m) \neq 0\}$  do
4:       if  $\overline{G}_{nm} > 0 \vee m$  in loop then
5:         compute priority function  $C_{nm}$ 
6:       end if
7:     end for
8:      $loadQ(n) \leftarrow$  modules in decreasing order of  $C_{nm}$ 
9:     remove all lower priority modules that have area conflicts with
       higher priority modules in  $loadQ(n)$ 
10:   end for
11:   eliminate redundant prefetches
12: end procedure

```

the average execution time gain $\overline{G}_{nm}$ (over the software execution of the candidate) obtained if its reconfiguration starts at this node is greater than 0, or the module is inside a loop. In the latter case, even if the reconfiguration is not finished in the first few loop iterations and we execute the module in software, we will gain from executing the module in hardware in future loop iterations. Since in most of the cases the software execution time of a module has the same order of magnitude as the reconfiguration overhead for the module, the reconfiguration will be finished after the first loop iteration. More exactly, while the module in discussion is executed in software in the first loop iteration, its hardware version is being reconfigured on the FPGA. As a result, in the next loop iterations the module will be executed in hardware.

The next step is to sort the prefetch candidates in decreasing order of their priority function (line 8), and in case of equality we give higher priority to modules placed in loops. After the $loadQ(n)$ has been generated for a node n , we remove all the lower priority modules that have area conflicts with the higher priority modules in the queue (line 9). Once all the queues have been generated, we eliminate redundant prefetches (all consecutive candidates at a child node that are a starting sub-sequence at all its parents in the CFG), as in [LH02] or [S⁺10].

The actual hardware module to be prefetched will be determined at run-time (by the middleware, using the reconfiguration API), since it depends on the run-time conditions. If the module with the highest priority (the head of $loadQ(n)$) is not yet loaded and is not being currently reconfigured, it will be loaded at that particular node n . If the head of $loadQ(n)$ is already on FPGA, the module with the next priority that is not yet on the FPGA will be loaded, but only in case the reconfiguration controller is idle. Finally, if

a reconfiguration is ongoing, it will be preempted only in case a hardware module with a priority higher than that of the module being reconfigured is found in the current list of prefetch candidates ($loadQ(n)$). As explained in Section 3.1.3, once a hardware module $m \in \mathcal{H}$ is reached at run-time, the middleware checks whether m is already fully loaded on the FPGA, and in this case it will be executed there. Thus, previously reconfigured modules are reused. Otherwise, if m is currently reconfiguring, the application will wait for the reconfiguration to finish and then execute the module on FPGA, but only if this generates a shorter execution time than the software execution. If none of the above are true, the software version of m will be executed.

3.2.3.1 Prefetch Priority Function

Our prefetch function represents the priorities assigned to the hardware modules reachable from a certain node $n \in \mathcal{N}$ in the CFG and, thus, determines the $loadQ(n)$ at that location. Considering that the processor must stall if the reconfiguration overhead cannot be completely hidden and that some candidates will provide a higher performance gain than others, our priority function will try to estimate the overall impact on the average execution time that results from different prefetches being issued at a particular node in the CFG. In order to accurately predict the next configuration to prefetch, several factors have to be considered.

The first one is represented by the ‘placement-aware’ probabilities (PAPs), computed with the method from [S⁺10]. The second factor that influences the decision of prefetch scheduling is represented by the execution time gain distributions (that will be discussed in detail in Section 3.2.3.2). The gain distributions reflect the reduction of execution time resulting from prefetching a certain candidate and executing it in hardware, compared to executing it in software. They are directly impacted by the waiting time distributions (which capture the relation between the reconfiguration time for a certain hardware module and the execution time distribution between the prefetch node in the CFG and the node corresponding to that module).

We denote the set of hardware modules for which it is profitable to compute the priority function at node n with $Reach(n) = \{m \in \mathcal{H} \mid PAP(n, m) \neq 0 \wedge (\bar{G}_{nm} > 0 \vee m \text{ in loop})\}$. For our example in Figure 3.4a, $Reach(r) = \{M_1, M_2, M_3\}$, but $Reach(M_3) = \emptyset$, because it does not make sense to reconfigure M_3 anymore (although $PAP(M_3, M_3) = 100\%$, we have the average waiting time $\bar{W}_{M_3M_3} = rec(M_3)$ and $rec(M_3) + hw(M_3) = 46 + 12 > 50 = sw(M_3)$). Thus, we do not gain anything by starting the reconfiguration of M_3 right before it is reached, i.e. $\bar{G}_{M_3M_3} = 0$. Considering the above discussion, our priority function expressing the reconfiguration gain generated

by prefetching module $m \in Reach(n)$ at node n is defined as:

$$\begin{aligned}
 C_{nm} = & PAP(n, m) \cdot \bar{G}_{nm} \\
 & + \sum_{k \in MutEx(m)} PAP(n, k) \cdot \bar{G}_{sk} \\
 & + \sum_{k \notin MutEx(m)} PAP(n, k) \cdot \bar{G}_{nm}^k
 \end{aligned} \tag{3.1}$$

In the above equation, $\bar{G}_{nm}$ denotes the average execution time gain generated by prefetching module m at node n (see Section 3.2.3.2), $MutEx(m)$ denotes the set of hardware modules that are executed mutually exclusive with m , the index s in $\bar{G}_{sk}$ represents the node where the paths leading from n to m and k split, and $\bar{G}_{nm}^k$ represents the expected gain generated by k , given that its reconfiguration is started immediately after the one for m .

The first term of the priority function represents the contribution (in terms of average execution time gain) of the candidate module m . The second term tries to capture the impact that the reconfiguration of m will produce on other modules that are executed mutually exclusive with it. In this case, the earliest time we can start the reconfiguration of module k , which is competing with m for the reconfiguration controller, is at node s , where the paths reaching m and k split. The third term captures the impact on the execution time of modules that are not mutually exclusive with m (and might be executed after m). The intuition behind the third term is the following: if from node n we can reach both module m and k (i.e. they are not mutually exclusive), then we want to see what is the impact of reconfiguring m first, and only then k .

In Figure 3.4a, modules M_1 , M_2 and M_3 are not mutually exclusive². Let us calculate the priority function for the three hardware modules from Figure 3.4a at node r (considering their areas proportional with their reconfiguration time). $C_{rM_1} = 90\% \cdot 40.4 + 0 + 10\% \cdot 36.9 + 95\% \cdot 38 \approx 76.1$. Note that $MutEx(M_1) = \emptyset$, $PAP(r, M_1) = 90\%$, $PAP(r, M_2) = 10\%$, $PAP(r, M_3) = 95\%$, $\bar{G}_{rM_1} = 40.4$, $\bar{G}_{rM_2} = 36.9$, $\bar{G}_{rM_3} = 38$. Similarly we compute $C_{rM_2} = 10\% \cdot 40 + 90\% \cdot 22.5 + 95\% \cdot 38 \approx 60.3$ and $C_{rM_3} = 95\% \cdot 38 + 90\% \cdot 1.7 + 10\% \cdot 30.9 \approx 40.7$ (the computation of execution time gains is discussed in Section 3.2.3.2). As we can see, since $C_{rM_1} > C_{rM_2} > C_{rM_3}$, the correct $loadQ(r) = M_1, M_3$ is generated at node r . Note that M_2 is removed from $loadQ(r)$ because it is in placement conflict with M_1 , which is the head of the queue (see line 9 in Algorithm 1).

3.2.3.2 Expected Execution Time Gain

Let us consider a node $n \in \mathcal{N}$ from the CFG and a hardware module $m \in \mathcal{H}$, reachable from n . Given that the reconfiguration of module m starts at

²Two mutually exclusive nodes are, for example, d and e in Figure 3.4a, and the paths reaching them split at c .

node n , we define the average execution time gain $\overline{G}_{nm}$ as the expected execution time that is saved by executing m in hardware (including any stalling cycles when the application is waiting for the reconfiguration of m to be completed), compared to the software execution of m . In order to compute it, we start with the distance (in time) from n to m . Let X_{nm} be the random variable associated with this distance. The waiting time is given by the random variable:

$$W_{nm} = \max(0, \text{rec}(m) - X_{nm}) \quad (3.2)$$

Note that the waiting time cannot be negative (if a module is already present on FPGA when we reach it, it does not matter how long ago its reconfiguration finished). The execution time gain is given by the distribution of the random variable:

$$G_{nm} = \max(0, \text{sw}(m) - (W_{nm} + \text{hw}(m))) \quad (3.3)$$

In case the software execution time of a candidate is shorter than waiting for its reconfiguration to finish and executing it in hardware, then the module will be executed in software by the middleware, and the gain is zero. If we denote the probability mass function (pmf) of G_{nm} with g_{nm} , then the average gain $\overline{G}_{nm}$ will be computed as:

$$\overline{G}_{nm} = \sum_{x=0}^{\infty} (x \cdot g_{nm}(x)) \quad (3.4)$$

The discussion is illustrated graphically in Figure 3.5, considering the nodes $n = r$ and $m = M_1$ from Figure 3.4a. The probability mass function (pmf) for X_{rM_1} (distance in time from r to M_1) is represented in Figure 3.5a and the pmf for the waiting time W_{rM_1} in Figure 3.5b. Note that the negative part of the distribution (depicted with dotted line) generates no waiting time. In Figure 3.5c we add the hardware execution time to the potential waiting time incurred. Finally, Figure 3.5d represents the discrete probability distribution of the gain G_{rM_1} . The resulting average gain is $\overline{G}_{rM_1} = 34 \cdot 18\% + 39 \cdot 42\% + 44 \cdot 6\% + 45 \cdot 34\% = 40.44$ time units.

Before presenting our algorithm for computing the gain distribution and the average gain, let us first introduce a few concepts. Given a control flow graph $\mathcal{G}_{cf}(\mathcal{N}_{cf}, \mathcal{E}_{cf})$, we introduce the following definitions [ALSU06]:

Definition 3.1 A node $n \in \mathcal{N}_{cf}$ is *post-dominated* by a node $m \in \mathcal{N}_{cf}$ in the control flow graph $\mathcal{G}_{cf}$ if every directed path from n to *sink* (excluding n) contains m .

Definition 3.2 Given a control flow graph $\mathcal{G}_{cf}$, a node $m \in \mathcal{N}_{cf}$ is *control dependent* upon a node $n \in \mathcal{N}_{cf}$ via a control flow edge $e \in \mathcal{E}_{cf}$ if the following two hold:

Figure 3.5: Computing the gain probability distribution step by step

Figure 3.6: Forward control dependence tree

- There exists a directed path P from n to m in $\mathcal{G}_{cf}$, starting with e , with all nodes in P (except m and n) post-dominated by m ;
- m does not post-dominate n in $\mathcal{G}_{cf}$.

In other words, there is some control edge from n that definitely causes m to execute, and there is some path from n to *sink* that avoids executing m .

Definition 3.3 A *control dependence graph* (CDG) $\mathcal{G}_{cd}(\mathcal{N}_{cd}, \mathcal{E}_{cd})$ corresponding to a control flow graph $\mathcal{G}_{cf}(\mathcal{N}_{cf}, \mathcal{E}_{cf})$ is defined as: $\mathcal{N}_{cd} = \mathcal{N}_{cf}$ and $\mathcal{E}_{cd} = \{((n, m), e) \mid m \text{ is control dependent upon } n \text{ via edge } e\}$. If we ignore all the backward edges in the CDG we obtain a *forward control dependence tree* (FCDT) [ALSU06].

Figure 3.6 shows the FCDT corresponding to the CFG in Figure 3.4a (note that the pseudo-edge $r \rightarrow s$ was introduced in the CFG in order to get all nodes to be directly, or indirectly, control dependent on r).

Algorithm 2 presents our method for computing the average gain. Let us consider a node $n \in \mathcal{N}$ and a hardware module $m \in \mathcal{H}$. Given that the reconfiguration of module m starts at node n , our algorithm estimates the average execution time gain over the software execution, that results from executing m in hardware (after waiting for its reconfiguration to finish if needed). The first steps are to construct the subgraph with all the nodes between n and m and to build its FCDT (lines 2-3), according to Definition 3.3. Then we compute the execution time distribution of the subgraph constructed earlier, representing the distance (in time) between n and m (line 4). Next we compute the waiting time distribution (line 5) and the gain distribution (line 6). Finally we compute the average gain with formula (3.4) (lines 7-8). In the following section we will present our algorithm for computing the execution time distribution (used at line 4).

3.2.3.3 Execution Time Distribution

Algorithm 3 details our method for computing the execution time distribution between node n and module m . We remind the reader that all the

Algorithm 2 Computing the average execution time gain**Input:** $\mathcal{G}(\mathcal{N}, \mathcal{E})$, FCDDT**Output:** $\bar{G}_{nm}$ = average gain if reconfiguration of m starts at n

```

1: procedure AVGEEXECUTETIMEGAIN( $n, m$ )
2:   construct subgraph with nodes between  $n$  and  $m$ 
3:   build its FCDDT
4:    $X_{nm} \leftarrow \text{EXECUTIMEDIST}(n, m)$  ▷ see Algorithm 3
5:    $W_{nm} \leftarrow \max(0, \text{rec}(m) - X_{nm})$ 
6:    $G_{nm} \leftarrow \max(0, \text{sw}(m) - (W_{nm} + \text{hw}(m)))$ 
7:   for all  $y \in \{t \mid g_{nm}(t) \neq 0\}$  do
8:      $\bar{G}_{nm} \leftarrow \bar{G}_{nm} + g_{nm}(y) \cdot y$ 
9:   end for
10: end procedure

```

computation is done considering the subgraph containing only the nodes between n and m and its forward control dependence tree. Before applying the algorithm we transform all post-test loops into pre-test ones (this transformation is done on the CFG representation, for analysis purposes only). Note that in this section we assume that branches are independent. In Section 3.3 we will present a method for dynamic FPGA configuration prefetching that takes into consideration branch correlations.

Our approach is to compute the execution time distribution of node n and all its children in the FCDDT, using the recursive function EXECUTIMEDIST. If n has no children in the FCDDT (i.e. no nodes control dependent on it), then we simply return its own execution time (line 4). For the root node we convolute³ its execution time with the execution time distribution of all its children in the FCDDT (line 6).

For a control node, the approach is to compute the execution time distribution for all its children in the FCDG that are control dependent on the ‘true’ branch, convolute this with the execution time of n and scale the distribution with the probability of the ‘true’ branch, t (line 9). Similarly, we compute the distribution for the ‘false’ branch as well (line 10) and then we superpose the two distributions to get the final one (line 11). For example, for the branch node c in Figure 3.4a, we have $ex_t(2+3) = 30\%$, $ex_f(2+8) = 70\%$ and, thus, the pmf for the execution time of the entire if-then-else structure is $x(5) = 30\%$ and $x(10) = 70\%$.

Finally, for a loop header, we first compute the distribution of all its children in the FCDDT (which represents the execution time of all the nodes inside the loop body) and then we convolute this with the execution time of the header (line 13). The result will be the execution time distribution of one iteration through the loop (ex_{li}). Then we use the distribution of loop iterations (available from profiling) to convolute ex_{li} with itself ($(*)^i$)

³This is done because the probability distribution of a sum of two random variables is obtained as the convolution of the individual probability distributions.

Algorithm 3 Computing the execution time distribution**Input:** $\mathcal{G}(\mathcal{N}, \mathcal{E})$ **Output:** x = execution time distribution between n and m

```

1: function EXECIMEDIST( $n, m$ )
2:    $ex_n \leftarrow \text{EXEC}(n)$ 
3:   if  $n$  has 0 children in FCDT then
4:      $x(ex_n) \leftarrow 100\%$ 
5:   else if  $n.type = \text{root}$  then
6:      $x \leftarrow ex_n * \text{FCDTCHILDRENDIST}(n, m, l)$ 
7:   else if  $n.type == \text{control}$  then ▷ ‘if’ blocks
8:      $(t, f) \leftarrow \text{GetLabels}(n)$  ▷ branch frequencies
9:      $ex_t \leftarrow t \times (ex_n * \text{FCDTCHILDRENDIST}(n, m, t))$ 
10:     $ex_f \leftarrow f \times (ex_n * \text{FCDTCHILDRENDIST}(n, m, f))$ 
11:     $x \leftarrow ex_t + ex_f$ 
12:   else if  $n.type == \text{loopheader}$  then
13:      $ex_{li} \leftarrow ex_n * \text{FCDTCHILDRENDIST}(n, m, l)$ 
14:      $\text{TRUNCATE}(ex_{li}, \text{rec}(m))$ 
15:     for all  $i \in \text{Iterations}(n)$  do
16:        $ex_i \leftarrow \text{iter\_prob}_n(i) \times [(*)^i ex_{li}]$ 
17:        $\text{TRUNCATE}(ex_i, \text{rec}(m))$ 
18:        $ex_{lb} \leftarrow ex_{lb} + ex_i$  ▷ the loop body
19:       if  $\min\{y \mid ex_i(y) \neq 0\} \geq \text{rec}(m)$  then
20:          $\text{break}$  ▷ no point to continue
21:       end if
22:     end for
23:      $x \leftarrow ex_n * ex_{lb}$  ▷ header executed last time
24:   end if
25:    $\text{TRUNCATE}(x, \text{rec}(m))$ 
26:   return  $x$ 
27: end function

28: function FCDTCHILDRENDIST( $n, m, \text{label}$ )
29:   for all  $c \in \text{FCDTChildren}(n, \text{label})$  do
30:      $\text{dist} \leftarrow \text{dist} * \text{EXECIMEDIST}(n, m)$ 
31:      $\text{TRUNCATE}(\text{dist}, \text{rec}(m))$ 
32:     if  $\min\{y \mid \text{dist}(y) \neq 0\} \geq \text{rec}(m)$  then
33:        $\text{break}$  ▷ no point to continue
34:     end if
35:   end for
36:   return  $\text{dist}$ 
37: end function

38: function TRUNCATE( $\text{dist}, \text{rec}$ )
39:    $y_{\min} \leftarrow \min\{y \mid \text{dist}(y) \neq 0\}$ 
   (continues on next page)

```

 (continued from previous page)

```

40:   if  $y_{min} \geq rec$  then
41:        $trunc(y_{min}) \leftarrow dist(y_{min})$ 
42:   else
43:        $trunc(y) \leftarrow \begin{cases} dist(y) & : y < rec \\ 0 & : y \geq rec \end{cases}$ 
44:   end if
45:   return  $trunc$ 
46: end function

47: function EXEC( $m$ )
48:   if  $m.type == hardware$  then
49:       return  $hw(m) + \alpha_m \cdot (sw(m) - hw(m))$ 
50:   else
51:       return  $sw(m)$ 
52:   end if
53: end function

```

denotes the operation of convolution with itself i times). The result is scaled (line 16) with the probability of i iterations to occur ($iter_prob_n(i)$) and then superposed (line 18) with the distribution of the loop body computed so far (ex_{lb}).

Let us illustrate the computation for the loop composed of nodes $a - b$ in our CFG example from Figure 3.4a using its corresponding FCDT from Figure 3.6. Since in this particular case b is the only node inside the loop body (note that a is the loop header), $ex_{li} = ex_a * ex_b$ gives the probability mass function (pmf) $ex_{li}(1 + 4) = 100\%$. Then we convolute ex_{li} with itself two times, and we scale the result with the probability to iterate twice through the loop, $iter_prob_a(2) = 60\%$. Thus, we obtain $ex_2(10) = 60\%$. Similarly, $ex_4(20) = 20\%$ and $ex_5(25) = 20\%$. By superposing ex_2 , ex_4 and ex_5 we get the pmf for the loop body, ex_{lb} , which we finally have to convolute with ex_a to get the pmf of the entire loop: $x(10 + 1) = 60\%$, $x(20 + 1) = 20\%$ and $x(25 + 1) = 20\%$. This distribution can be further used in the computation of $\overline{G}_{rM_1}$, for example.

The function FCDTCHILDRENDIST (line 28) simply convolutes the distributions of all children of n in the FCDT that are control dependent on the parameter edge *label*. In order to speed-up the computation, we exploit the following observation: when computing the pmf for the execution time distributions, we discard any values that are greater than or equal to the reconfiguration time, because those components of the distribution will generate no waiting time (one example in lines 31-33).

Function TRUNCATE works as follows: if the smallest execution time is already greater than the reconfiguration overhead, we keep only this smallest value in the distribution (line 41). This is done because the distribution in question might be involved in convolutions or superpositions (in function

EXECTIMEDIST), and keeping only this minimal value is enough for computing the part of the execution time distribution of interest (that might generate waiting time). Otherwise, we simply truncate the distribution at *rec* (line 43).

One observation related to the computation of the execution time of a node $m \in \mathcal{N}$ (function EXEC in Algorithm 3, line 47) is that, if m is a hardware candidate ($m \in \mathcal{H}$) we need to approximate its execution time, since the prefetches for it might be yet undecided and, thus, it is not known if the module will be executed in software or on the FPGA. In order to estimate the execution time, we make use of a coefficient $\alpha_m \in [0, 1]$. The execution time for a hardware module m will be computed as (line 49): $exec(m) = hw(m) + \alpha_m \cdot (sw(m) - hw(m))$. Our experiments have proven that good results are obtained by setting the value of α_m , for each hardware module, to the ratio between its own hardware area and the total area needed for all modules in $\mathcal{H}$: $\alpha_m = \frac{area(m)}{\sum_{k \in \mathcal{H}} area(k)}$. The intuition behind using α_m is the following: the smaller the area of a module, the faster it is to reconfigure it and the easier it is to accommodate it on the FPGA. As a result, we can expect that module to be executed in hardware often. For modules with big area, it will be harder to accommodate them on the FPGA. Thus, they will be executed sometimes in software too.

3.2.4 Experimental Evaluation

3.2.4.1 Monte Carlo Simulator

Sampling

In order to evaluate the quality of our prefetch solutions we have used an in-house developed Monte Carlo simulator that produces the execution time distribution of an application considering the architectural assumptions described in Section 3.1.1 and 3.1.2. Each simulation generates a trace through the control flow graph, starting at the *root* node, and ending at the *sink* node (and we record the length of these traces). Whenever a branch node is encountered, we perform a Bernoulli draw (based on the probabilities of the outgoing edges) to decide if the branch is taken or not. At loop header nodes we perform random sampling from the discrete distribution of loop iterations ($iter_prob_n$) to decide how many times to loop. Note that, as mentioned earlier, we consider branches independent. In Section 3.3 we will address the case when branches are correlated and will present a dynamic FPGA configuration prefetching method that will take advantage of this.

Accuracy Analysis

We stop the Monte Carlo simulation once we reach a satisfactory accuracy for the mean of the execution time distribution. We describe the desired accuracy in the following way: “*The mean of the output distribution*

should be accurate to within $\pm\epsilon$ with confidence κ ". The accuracy can be arbitrarily precise at the expense of longer simulation times. We will next present an analysis based on confidence intervals [HWC14], to determine the number of samples to run in order to achieve the required accuracy.

Let us assume that μ is the actual mean of the true output distribution and $\hat{\mu}$ is the estimate mean computed by the Monte Carlo simulation. Since each simulation result is an independent sample from the same distribution, using the Central Limit Theorem we have that the distribution of the estimate of the true mean is (asymptotically) given by:

$$\hat{\mu} = \text{Normal}\left(\mu, \frac{\sigma}{\sqrt{N}}\right)$$

where σ is the true standard deviation of the output execution time distribution and N represents the number of samples. The above equation can be rewritten as follows:

$$\mu = \text{Normal}\left(\hat{\mu}, \frac{\sigma}{\sqrt{N}}\right)$$

By considering the required accuracy for our mean estimate and performing a transformation to the standard Normal distribution (i.e. with mean 0 and standard deviation 1), we can obtain the following relationship [HWC14]:

$$\epsilon = \frac{\sigma}{\sqrt{N}} \Phi^{-1}\left(\frac{1+\kappa}{2}\right)$$

where the function $\Phi^{-1}(\bullet)$ is the inverse of the standard Normal cumulative distribution function. By rearranging the terms and considering that we want to achieve at least this accuracy we obtain the minimum value for the number of samples N :

$$N > \left(\frac{\sigma \cdot \Phi^{-1}\left(\frac{1+\kappa}{2}\right)}{\epsilon} \right)^2$$

Please note that we do not know the true standard deviation σ , but for our purpose we can estimate it by taking the standard deviation of the first few samples (e.g. the first 40).

3.2.4.2 Simulation Results

In order to evaluate the effectiveness of our algorithm we first performed experiments on synthetic examples. We generated two sets of control flow graphs: *Set*₁ contains 20 CFGs with ~ 100 nodes on average (between 67 and 126) and *Set*₂ contains 20 CFGs with ~ 200 nodes on average (between 142 and 268).

The software execution time for each node was randomly generated in the range of 10 to 100 time units. A fraction of all the nodes (between

15% and 25%) were then chosen to become hardware candidates, and their software execution time was generated β times larger than their hardware execution time. The coefficient β was chosen from the uniform distribution on the interval [3, 7], in order to model the variability of hardware speedups over software. We also generated the size of the hardware modules, which in turn determined their reconfiguration time.

The size of the PDR region available for placement of hardware modules for each application was varied as follows: we summed up the areas of all those hardware nodes for which our heuristic performed at least one prefetch: $\text{MAX_HW} = \sum_{m \in \mathcal{H}} \text{area}(m)$. Then we generated problem instances by considering the size of the available reconfigurable region corresponding to different fractions of MAX_HW : 15%, 25%, 35%, 45%, and 55%. As a result, we obtained a total of $2 \times 20 \times 5 = 200$ experimental settings. All experiments were run on a PC with CPU frequency 2.83 GHz, 8 GB of RAM, and running Windows Vista.

For each experimental setting, we first generated a placement for all the hardware modules, which determined the area conflict relationship between them. Then, for each application we inserted the configuration prefetches in the control flow graph. Finally, we have evaluated the result using the simulator described in Section 3.2.4.1 that produces the average execution time of the application considering the architectural assumptions described in Section 3.1.1 and 3.1.2. We have determined the result with an accuracy of $\pm 1\%$ with confidence 99%.

As a baseline we have considered the average execution time of the application (denoted as *baseline*) in case all the hardware candidates are placed on FPGA from the beginning and, thus, no prefetch is necessary. Please note that this is an absolute lower bound on the execution time; this ideal value might be unachievable even by the optimal static prefetch, because it might happen that it is impossible to hide all the reconfiguration overhead for a particular application.

First of all we were interested to see how our approach compares to the current state-of-the-art [S⁺10]. Thus, we have simulated each application using the prefetch queues generated by our approach and those generated by [S⁺10]. Let us denote the average execution times obtained with ex_G for our approach, and ex_{PAP} for [S⁺10]. Then we computed the performance loss over the baseline for our approach, $PL_G = \frac{ex_G - \text{baseline}}{\text{baseline}}$; similarly we calculate PL_{PAP} . Figures 3.7a and 3.7c show the results obtained (averaged over all CFGs in Set_1 and in Set_2). As can be seen, for all FPGA sizes, our approach achieves better results compared to [S⁺10]: for Set_1 , the performance loss over ideal is between 10% and 11.5% for our method, while for [S⁺10] it is between 15.5% and 20% (Figure 3.7a). In other words, we are between 27% and 42.6% closer to the ideal baseline than [S⁺10]. For Set_2 , we also manage to get from 28% to 41% closer to the ideal baseline than [S⁺10] (Figure 3.7c).

One other metric suited to evaluate prefetch policies is the total time

Figure 3.7: Comparison with state-of-the-art [S+10] for the synthetic benchmarks

spent by the application waiting for FPGA reconfigurations to finish (in case the reconfiguration overhead was not entirely hidden). One major difference between the approach proposed in this report and that in [S+10] is that we also execute candidates from $\mathcal{H}$ in software (if this is more profitable than reconfiguring and executing on FPGA), while under the assumptions in [S+10] all candidates from $\mathcal{H}$ are executed only on FPGA. Considering this, for each execution in software of a candidate $m \in \mathcal{H}$, we have no waiting time, but we do not execute m on FPGA either. In these cases, in order to make the comparison to [S+10] fair, we penalize our approach with $sw(m) - hw(m)$. Let us define the reconfiguration penalty (RP): for [S+10] RP_{PAP} is the sum of all waiting times incurred during simulation, and for our approach RP_G is the sum of all waiting times plus the sum of penalties $sw(m) - hw(m)$ whenever a module $m \in \mathcal{H}$ is executed in software during simulation. Figures 3.7b and 3.7d show the reconfiguration penalty reduction $RPR = \frac{RP_{PAP} - RP_G}{RP_{PAP}}$, averaged over all CFGs in Set_1 and in Set_2 . As we can see, by intelligently generating the prefetches we manage to significantly reduce the penalty (with up to 40%, for both experimental sets), compared to [S+10].

Concerning the running times of the heuristics, our approach took longer time than [S+10] to generate the prefetches: from just $1.6\times$ longer in the best case, up to $12\times$ longer in the worst case, incurring on average $4.5\times$

more optimization time. For example, for the 15% FPGA fraction, for the biggest CFG in *Set₂* (with 268 nodes), the running time of our approach was 3832 seconds, compared to 813 seconds for $[S^+10]$; for CFGs with a smaller size and a less complex structure we generated a solution in as low as 6 seconds (vs 2 seconds for $[S^+10]$).

3.2.4.3 Case Study – GSM Encoder

We also tested our approach on a GSM encoder, which implements the European GSM 06.10 provisional standard for full-rate speech transcoding. This application can be decomposed into 10 functions executed in a sequential order: Init, GetAudioInput, Preprocess, LPC_Analysis, ShortTermAnalysisFilter, LongTermPredictor, RPE_Encoding, Add, Encode, Output. The execution times were derived using the MPARM cycle accurate simulator, considering an ARM processor with an operational frequency of 60 MHz. We have identified through profiling the most computation intensive parts of the application, and then these parts were synthesized as hardware modules for an XC6VLX240T Virtex6 device, using the Xilinx ISE design suite. The resulting overall CFG of the application contains 30 nodes. The reconfiguration times were estimated considering a 100 MHz configuration clock frequency and the ICAP 32-bit width configuration interface (see our reconfiguration controller described in Section 3.1.1.2).

The CFG for the GSM Encoder, as well as the profiling information, was generated using the LLVM suite [LA04] as follows: *llvm-gcc* was first used to generate LLVM bytecode from the C files. The *opt* tool was then used to instrument the bytecode with edge and basic block profiling instructions. The bytecode was next run using *lli*, and then the execution profile was generated using *llvm-prof*. Finally, *opt-analyze* was used to print the CFGs to .dot files. Profiling was run considering several audio files (.au) as input.

Figure 3.8a shows the detailed control flow graph (CFG) for our GSM encoder case study. Nodes are labeled with their ID, their execution time (for example, for node with $id = 1$, the execution time is 10 time units) and their type: root, sink, control nodes, loop header nodes, basic nodes and hardware candidates (denoted with HW and represented with shaded boxes). We have used two scenarios: one considering 5 nodes as hardware candidates (namely modules with IDs 6, 9, 12, 15 and 22), and another scenario considering 9 nodes as hardware candidates (depicted in Figure 3.8a).

We have used the same methodology as for the synthetic examples and compared the results using the same metrics defined above in Section 3.2.4.2, i.e. performance loss over ideal and reconfiguration penalty reduction (presented in Figure 3.9). As can be seen, for the scenario with 5 candidate modules, the performance loss over ideal is between 10.5% and 14.8% for our approach, while for $[S^+10]$ it is between 25.5% and 32.9% (Figure 3.9a). Thus, we were from 50% up to 65% closer to the ideal baseline than $[S^+10]$. The reconfiguration penalty reduction obtained is as high as 58.9% (Figure

Figure 3.8: Control flow graphs for the case studies

Figure 3.9: Comparison with state-of-the-art [S+10] for the GSM encoder case study

3.9b). For the setting with 9 hardware candidates, the performance loss over ideal is between 50% and 56% for our approach, while for [S+10] it is between 117% and 135% (Figure 3.9c). Thus we manage to get from 52% up to 59% closer to the ideal baseline than [S+10]. This is also reflected in the reconfiguration penalty reduction of up to 60% (Figure 3.9d). The prefetches were generated in 27 seconds by our approach and in 11 seconds by [S+10].

3.2.4.4 Case Study – Floating Point Benchmark

Our second case study was a SPECfp benchmark (FP-5 from [CBP04]), characteristic for scientific and computation-intensive applications. Modern FPGAs, coupled with floating-point tools and IP, provide performance levels much higher than software-only solutions for such applications [Alt09]. In order to obtain the inputs needed for our experiments, we used the framework and traces provided for the first Championship Branch Prediction competition [CBP04]. The given instruction trace consists of 30 million instructions, obtained by profiling the program with representative inputs.

We have used the framework provided in [CBP04] to reconstruct the control flow graph (CFG) of the FP-5 application based on the given trace. We have obtained a CFG with 65 nodes, after inlining the functions and

Figure 3.10: Comparison with state-of-the-art $[S+10]$ for the floating point benchmark case study

pruning all control flow edges with a probability lower than 10^{-5} . Then we used the traces to identify the parts of the CFG that have a high execution time (mainly loops).

Figure 3.8b shows the detailed control flow graph (CFG) for our second case study: the floating point benchmark from SPECfp. Nodes are represented as discussed in Section 3.2.4.3. The software execution times for the basic blocks were obtained by considering the following cycles per instruction (CPI) values for each instruction: for calls, returns and floating point instructions $CPI = 3$, for load, store and branch instructions $CPI = 2$, and for other instructions $CPI = 1$. Similar to the previous experimental sections, we considered the hardware execution time β times smaller than the software one, where β was chosen from the uniform distribution on the interval $[3, 7]$.

We have used two scenarios: one considering as hardware candidates the top 9 nodes with the highest software execution times (namely modules with IDs 5, 11, 17, 21, 22, 32, 42, 45 and 56), and another scenario considering the top 25 nodes (depicted in Figure 3.8b). Following the same methodology as described in Section 3.2.4.2, we compared our approach with $[S+10]$. The results are presented in Figure 3.10. For example, for the scenario with 9 candidate modules, the performance loss over ideal is between 77% and 90% for our approach, while for $[S+10]$ it is between 107% and 118% (Figure

3.10a). Thus, we are from 20% up to 28% closer to the ideal baseline than [S⁺10]. The reconfiguration penalty reduction is as high as 28% (Figure 3.10b). For the setting with 25 hardware candidates the reconfiguration penalty reduction increases, up to 31.1% (Figure 3.10d). As we can see, for both case studies our approach produces significant improvements compared to the state-of-the-art. The prefetches for FP-5 were generated in 53 seconds by our approach and in 32 seconds by [S⁺10].

3.3 Dynamic FPGA Configuration Prefetching

As we have shown above, static configuration prefetching algorithms can significantly improve the performance of an application. Unfortunately, they suffer from one important limitation: in case of non-stationary behavior they are unable to adapt since the prefetch schedule is fixed based on average profiling information. In such cases a dynamic configuration prefetching technique is desirable, as the one that we will present in this chapter.

We assume that the application exhibits a non-stationary branch behavior. More exactly, we consider that the application passes through an unpredictable number of stationary phases [SSC03]. Branch probabilities and correlations are stable in one phase and then change to another stable phase, in a manner unpredictable at design-time. This model can be applied to a large class of applications [SBSH12].

3.3.1 Problem Formulation

Given an application (as described in Section 3.1.3) intended to run on the reconfigurable architecture described in Section 3.1.1, our goal is to determine dynamically, at run-time, at each node $m \in \mathcal{H}$, the prefetches to be issued such that the expected execution time of the application is minimized.

3.3.2 Motivational Example

Figure 3.11 illustrates two of the stationary phases, each one with a different branch behavior, among those exhibited by an application. Hardware candidates are represented with squares (i.e. $\mathcal{H} = \{M_0, M_1, M_2, M_3, M_4\}$), and software-only nodes with circles (i.e. $\{B_0, B_1, B_2, B_3, n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$). For each phase, the edge probabilities of interest are illustrated on the graph, and they remain unchanged for that phase. We represent branch correlations with different line patterns: in phase 1, conditional branch B_0 is positively correlated with B_2 and B_1 is positively correlated with B_3 ; in phase 2, B_0 and B_2 , as well as B_1 and B_3 , become negatively correlated and the probabilities of B_0 and B_2 also change, from 50%-50%

Figure 3.11: Motivational example for dynamic FPGA configuration prefetching

initially, to 34%-66%. For the simplicity of illustration we assume that the correlation in this example is perfect. However, our prediction algorithm will capture the tendency of branches to be correlated even if it is not perfect correlation. The software and hardware execution times for the candidates, as well as their reconfiguration times are given in Table 3.2.

Let us assume that the PDR region of the FPGA is big enough to host only one module at a time, i.e. we have only one reconfigurable slot available. Hardware modules M_0 , M_1 , M_2 , M_3 and M_4 cannot coexist since they are all mapped to this unique slot, thus having a so-called *placement conflict*. Let us consider a prediction point in the application, module M_1 , currently placed on the FPGA and executing. After its execution we want to issue a prefetch for a new module, M_2 , M_3 or M_4 , in order to improve the performance of the application. Note that, by issuing a prefetch after M_1 the reconfiguration overhead will be hidden (at least partly). In the case of M_2 or M_3 , the overhead is overlapped with the computation of nodes n_4 , B_2 and B_3 , while in the case of M_4 , its reconfiguration is overlapped with n_4 and B_2 .

Let us see how a static prefetch approach works, like the one we pre-

Table 3.2: Hardware candidates' characteristics

M_i	sw	hw	rec
M_0	40	11	90
M_1	70	3	80
M_2	90	5	110
M_3	80	4	100
M_4	50	10	160

sented in Section 3.2 or the one from [S⁺10]. Since it considers profiling information, regardless of the application's phase, it will issue the prefetch based on the average branch probabilities. In our example, considering the reconfiguration overheads and the execution times of the modules, [S⁺10] would always prefetch M_4 because it has the highest average probability to be reached from M_1 , i.e. $\frac{50\%+34\%}{2} = 42\%$; our own method from Section 3.2 would always prefetch M_2 because it generates the biggest expected performance improvement of $\frac{50\% \cdot 45\% + 66\% \cdot 55\%}{2} \cdot (sw(M_2) - hw(M_2)) = 29.4\% \cdot 85 = 24.99$ time units. The static approaches are performing poorly, because in more than 50% of the cases (when module M_4 is not reached for [S⁺10], or M_2 is not reached for our own static prefetching technique), the prefetches are wasted and the other modules (depending on the path followed) will be executed in software.

Let us now see how the dynamic technique proposed in [LH02] works: using a Markov model, the approach estimates the probability to reach the next module from M_1 , and based on this it issues the prefetches. Assuming that a phase is long enough for the model to learn the branch probabilities, the priorities associated with modules M_2 , M_3 and M_4 are equal to their probabilities to be reached, i.e 22.5%, 27.5% and 50% in phase 1, and 36.3%, 29.7% and 34% in phase 2 respectively. Based on this technique, the generated prefetches at M_1 will be: M_4 in phase 1 and M_2 in phase 2. Although this is a dynamic approach, many prefetch opportunities are still wasted (e.g. when M_4 is not reached in phase 1, in $100\% - 50\% = 50\%$ of the cases or, similarly, when M_2 is not reached in phase 2).

The dynamic technique presented in [HV09], based on an aggregate gain table, will prefetch M_3 in phase 1, and M_2 in phase 2, because they have the highest normalized long run aggregate gains in each phase, i.e. $50\% \cdot 55\% \cdot (sw(M_3) - hw(M_3)) = 27.5\% \cdot 76 = 20.9$ time units for M_3 and $66\% \cdot 55\% \cdot (sw(M_2) - hw(M_2)) = 36.3\% \cdot 85 = 30.85$ time units for M_2 respectively (note that the other modules have smaller normalized long run aggregate gains).

If we could also exploit the branch correlation information, then we could issue prefetches better than both the static and the dynamic approaches discussed above. In phase 1, we should prefetch M_4 whenever path $B_0 - n_1$

is followed (because of the positive correlation of B_0 with B_2), M_2 whenever path $B_0 - n_0 : B_1 - n_2$ is followed and M_3 whenever path $B_0 - n_0 : B_1 - n_3$ is followed (for similar considerations). Similar reasoning can be used for phase 2.

The big limitation of static prefetching is its lack of robustness and flexibility: the prefetches are fixed based on average profiling information and they cannot adapt to the run-time conditions. For our example, the static approaches always prefetch M_4 , which is a very restrictive decision. Although the dynamic approaches provide extra flexibility, they still miss prefetch opportunities. One limitation exhibited by [LH02] and [HV09] is that the approaches rely only on the hardware modules' history and do not exploit the path information (like branch outcomes together with their correlations). As a result, for [LH02] and [HV09], the prefetch opportunities are wasted in more than 50% of the cases. We will present an approach that tries to overcome the mentioned limitations.

3.3.3 Dynamic Prefetching

We next describe a piecewise linear prediction algorithm used for FPGA configuration prefetching and its hardware organization. The main idea is to assign priorities to all the hardware candidates and then issue prefetches based on them, at certain points in the application (a natural choice for the prefetch points is after the execution of a candidate $m \in \mathcal{H}$).

3.3.3.1 The Piecewise Linear Predictor

The concept of piecewise linear prediction was successfully applied in the context of branch prediction [Jim05] and it is based on the idea to use the history of a branch in order to predict its outcome (*taken* or *not taken*). We extend the concept for the case of FPGA configuration prefetching. We associate one piecewise linear predictor with each hardware candidate and try to capture the correlation between a certain branch history leading to a certain prediction point, and a candidate being reached in the future. The branch history for a prefetch point $m \in \mathcal{H}$ is represented by the dynamic sequence of conditional branches (program path) leading to m . The predictor keeps track of the positive or negative correlation of the outcome of every branch in the history with the predicted hardware candidate being reached or not.

The output of a predictor (associated with a hardware candidate) is a single number, obtained by aggregating the correlations of all branches in the current history, using a linear function (PREDICT in Algorithm 4). This function induces a hyperplane in the space of all outcomes for the current branch history, used to estimate what is the likelihood to reach the hardware candidate in discussion. We interpret the output y of the predictor as the probability to reach the candidate, since the distance of y from the hyperplane (on the positive side) is proportional to the degree of certainty

Figure 3.12: The 3D weight array (Ω) of the predictor

that the candidate will be reached. Considering that there are many paths leading to a prediction point $m \in \mathcal{H}$, there are also many linear functions used for prediction. Together, they form a piecewise linear surface that separates the paths for which the hardware candidate will be reached in the future, from those paths for which it will not be reached.

Data Structures

The predictor uses the following data structures:

- A – the branch address history register. At run-time, when a conditional branch is executed, its address is shifted into the first position of this register.
- H – the branch outcome history register, containing the *taken* or *not taken* outcomes for branches. As for the address register, the outcomes are shifted into the first position. Together, A and H characterize the path history.
- h – the length⁴ of both the history registers A and H .
- HW – the hardware history register, containing the last q hardware candidates reached.
- q – the length⁴ of the hardware history register HW .
- Ω – a 3D array of weights (shown in Figure 3.12 for the example from Figure 3.11, with five hardware candidates, M_0 to M_4). The indexes are: the ID of a hardware candidate, the ID of a branch and its position (index) in the path history. We can view Ω as a collection of matrices, one for each hardware candidate. The entries in Ω keep track of the

⁴The values of h and q reflect the trade-off between the space budget available for the predictor and its accuracy. We obtained good experimental results for relatively small values: $h \in [4, 16]$ and $q \in [2, 8]$.

Algorithm 4 Prediction function**Input:** $\mathcal{H}$, history registers H and A **Output:** $\tilde{L}_{mk}$ = probability to reach k from m

```

1: procedure LIKELIHOODS( $m$ )
2:   for all  $k \in \mathcal{H} \setminus \{m\}$  do
3:      $\lambda_{mk} \leftarrow \text{PREDICT}(k, H, A)$ 
4:   end for
5:    $\lambda_{min} \leftarrow \min_k \lambda_{mk}$ 
6:    $\lambda_{max} \leftarrow \max_k \lambda_{mk}$ 
7:   for all  $k \in \mathcal{H} \setminus \{m\}$  do
8:      $\tilde{L}_{mk} \leftarrow \frac{\lambda_{mk} - \lambda_{min}}{\lambda_{max} - \lambda_{min}}$ 
9:   end for
10: end procedure

11: function PREDICT( $k, H[1..h], A[1..h]$ )
12:    $output \leftarrow 0$ 
13:   for all  $i = 1..h$  do
14:     if  $H[i] == \text{taken}$  then
15:        $output \leftarrow output + \Omega[k, A[i], i]$ 
16:     else
17:        $output \leftarrow output - \Omega[k, A[i], i]$ 
18:     end if
19:   end for
20:   return  $output$ 
21: end function

```

correlations between branches and hardware candidates. For example, $\Omega[M_i, B_j, p]$, denoted ω_{jp}^i , represents the weight of the correlation between branch B_j occurring at index p in the history A and module M_i being reached. Please note that addition and subtraction on the weights ω_{jp}^i saturate⁵ at $\pm t$. The more positive the weight ω_{jp}^i is, the stronger the positive correlation (i.e., if B_j , present at index p in A is *taken*, then it is likely to reach M_i later), the more negative the weight is, the stronger the negative correlation, and for ω_{jp}^i close to zero, there is no or only weak correlation.

Prediction Function

Algorithm 4 details our approach for computing the likelihoods to reach the hardware candidates from a prediction point $m \in \mathcal{H}$. We use the function PREDICT to compute the *output* that reflects the correlation between the

⁵The threshold t is chosen based on the application characteristics: in case the phases are short and switch often, then a smaller value of t is preferred (e.g. $t \in [8, 16]$), for quick adaptation. Otherwise, larger values can be used (e.g. $t \in [32, 128]$).

branch history leading to module m and candidate k being reached in the future. For all the entries in the current history (line 13), if the branch on position i was *taken* then we add to the *output* the weight $\Omega[k, A[i], i]$ (line 15); otherwise we subtract it (line 17). Once the outputs were calculated for all candidates (lines 2-4), we normalize the results (line 8). The result $\tilde{L}_{mk} \in [0, 1]$ represents the estimated likelihood to reach hardware candidate k from prefetch point m , and will be used in computing the prefetch priority function (see Section 3.3.3.3). Consider the example in Figure 3.11, with $A = [B_0 \ B_1]$ and $H = [taken \ taken]$ at prediction point M_1 . Then, in phase 1, the estimated probability to reach M_2 is $\tilde{L}_{M_1 M_2} = 100\%$ and the probabilities to reach M_3 or M_4 are $\tilde{L}_{M_1 M_3} = \tilde{L}_{M_1 M_4} = 0$. This is because the predictor weights are $\omega_{00}^2 = \omega_{11}^2 = +t$ and $\omega_{00}^3 = \omega_{11}^3 = \omega_{00}^4 = \omega_{11}^4 = -t$ (where t is the saturation threshold).

Lazy Update Function

After making a prediction, we need to train the predictor based on the real outcome (i.e. which modules were actually reached). Since this information becomes available only later, we opted for a lazy update: we save the context (history registers A and H) based on which the current prediction is made, and we update the predictor as late as possible, i.e. only when the same prediction point is reached again, and before making a new prediction. The next q candidates that will be reached after the prediction point are accumulated in the hardware history register HW .

Algorithm 5 presents our lazy update. It takes as parameters the module m where the update is done, the saved branch outcome register (H_S^m), the saved branch address register (A_S^m) and the saved hardware history register (HW_R^m), containing the first q hardware modules executed after the prediction point m . The path information was saved when the last prediction was made at m , and the register HW was saved when the q^{th} module after m was reached and m was evicted from the HW register. For all history positions (line 2), for all the q modules reached after m (line 3), we update the corresponding weights in Ω (saturating at $\pm t$), depending on the outcome of the branch on position i : if the branch was *taken*, then we increment the corresponding weight (line 5); otherwise we decrement it (line 7). For all the modules not reached after m we do the opposite: decrement the weights in Ω for *taken* branches, and increment them for *not taken* ones (lines 10-16).

Next, we save the current path that has led to m (lines 18-19). Then we pop the q^{th} hardware module (*top*) from the history register HW (line 20) and we push the current module m on the first position of HW , but only if m is not already there (line 21). If m is executed in a loop, we want to avoid having repetitions in the HW history register; instead of pushing m , we only update its timestamp, used for computing the potential execution time gain (see Section 3.3.3.2). Finally we save the hardware history register containing the q modules executed after *top* (line 24).

Algorithm 5 Lazy update function

Input: $\mathcal{H}$, history registers $H_S^m[1..h]$, $A_S^m[1..h]$ and $HW_R^m[1..q]$
Output: Ω is updated

```

1: procedure UPDATE( $m$ ,  $H_S^m[1..h]$ ,  $A_S^m[1..h]$ ,  $HW_R^m[1..q]$ )
2:   for all  $i = 1..h$  do
3:     for all  $j = 1..q$  do
4:       if  $H_S^m[i] == \text{taken}$  then
5:          $\Omega[HW_R^m[j], A_S^m[i], i] \leftarrow \max(t, \Omega[HW_R^m[j], A_S^m[i], i] + 1)$ 
6:       else
7:          $\Omega[HW_R^m[j], A_S^m[i], i] \leftarrow \min(-t, \Omega[HW_R^m[j], A_S^m[i], i] - 1)$ 
8:       end if
9:     end for
10:    for all  $k \in \mathcal{H} \setminus HW_R^m$  do
11:      if  $H_S^m[i] == \text{taken}$  then
12:         $\Omega[k, A_S^m[i], i] \leftarrow \min(-t, \Omega[k, A_S^m[i], i] - 1)$ 
13:      else
14:         $\Omega[k, A_S^m[i], i] \leftarrow \max(t, \Omega[k, A_S^m[i], i] + 1)$ 
15:      end if
16:    end for
17:  end for
18:   $H_S^m \leftarrow H$ 
19:   $A_S^m \leftarrow A$ 
20:   $top \leftarrow HW[q]$ 
21:  if  $m \neq HW[1]$  then
22:    push  $m$  in history register  $HW$ 
23:  end if
24:   $HW_R^{top} \leftarrow HW$ 
25: end procedure

```

3.3.3.2 Expected Execution Time Gain

Let us consider a prediction point at the completion of $m \in \mathcal{H}$ (e.g. M_1 in Figure 3.11) and one hardware candidate $k \in \mathcal{H}$ (e.g. M_2 in Figure 3.11), reachable from m . Given that the reconfiguration of k starts immediately after m finishes executing, we define the execution time gain γ_{mk} as the time that is saved by executing k in hardware (including any stalling cycles when the application is waiting for the reconfiguration of k to be completed), compared to the software execution of k , given that module k will be reached in the current application run. Let χ_{mk} represent how much time it takes to reach k from the moment m finishes executing. The waiting time, corresponding to a particular run of the application is given by:

$$\varpi_{mk} = \max(0, \text{rec}(k) - \chi_{mk}) \quad (3.5)$$

This time cannot be negative (if a module is present on FPGA when it is reached, it does not matter how long ago its reconfiguration finished). The

execution time gain over the software execution is:

$$\gamma_{mk} = \max(0, sw(k) - (\varpi_{mk} + hw(k))) \quad (3.6)$$

If the software execution time of a candidate is shorter than waiting for its reconfiguration to finish plus executing it in hardware, then the module is executed in software, and the gain is zero.

In order to estimate χ_{mk} on-line, we use timestamps. At run-time, whenever a candidate k finishes executing, we save the value of the system clock and, at the same time, we compute χ_{mk} for all modules m present in the current history HW (relative to the timestamps when they finished executing).

As an example, assume the reconfiguration and hardware/software execution times for candidates given in Table 3.2. Let us consider that, for a particular application run, nodes n_4, B_2 and B_3 from Figure 3.11, have execution times of 100, 2 and 2 time units, respectively. Considering the prediction point M_1 , the time to reach the candidates from M_1 are $\chi_{M_1M_2} = \chi_{M_1M_3} = 100 + 2 + 2 = 104$ time units and $\chi_{M_1M_4} = 100 + 2 = 102$ time units. Note that these values are actually estimated on-line during each application run, using timestamps (and not *computed* as in this illustrative example). The waiting times experienced by each candidate if we start its reconfiguration after M_1 finishes executing are: $\varpi_{M_1M_2} = \max(0, 110 - 104) = 6$, $\varpi_{M_1M_3} = \max(0, 100 - 104) = 0$ and $\varpi_{M_1M_4} = \max(0, 160 - 102) = 58$. Thus, we can compute the potential gains $\gamma_{M_1M_2} = \max(0, 90 - (6 + 5)) = 79$, $\gamma_{M_1M_3} = \max(0, 80 - (0 + 4)) = 76$ and $\gamma_{M_1M_4} = \max(0, 50 - (58 + 10)) = 0$.

In the case of M_2 , the reconfiguration is not completely overlapped with the execution of n_4, B_2 and B_3 , but it is worth waiting for 6 time units and then execute the module in hardware for another 5, instead of executing the module in software for 90 time units. In the case of M_3 , the entire reconfiguration overhead is overlapped with the execution of n_4, B_2 and B_3 , so when we reach the module it is already on the FPGA, prepared for execution. Finally, in the case of M_4 , the waiting time is too large (which means that not enough of its reconfiguration overhead can be hidden), and it is actually better to execute the candidate in software. In other words, it is too late to start reconfiguring M_4 after module M_1 finishes executing.

In order to give higher precedence to candidates placed in loops, we adjust the value of ϖ_{mk} as follows: First, for every hardware candidate $k \in \mathcal{H}$ we record its frequency (number of times it is executed inside a loop), φ_k , during each run of the application. In the simple example from Figure 3.11, since there are no loops, $\varphi_i = 1, \forall i \in \{0, 1, \dots, 4\}$. Then we compute an estimate $\tilde{F}_k$ for the average frequency over the past runs, using an exponential smoothing formula in order to emphasize the recent history:

$$\tilde{F}_k^t = \alpha \cdot \varphi_k^t + (1 - \alpha) \cdot \tilde{F}_k^{t-1} \quad (3.7)$$

In equation 3.7, $\tilde{F}_k^t$ represents the estimate at time t of the expected frequency for module k , $\tilde{F}_k^{t-1}$ represents the previous estimate, φ_k^t represents k 's frequency in the current application run and α is the smoothing parameter. Given $\tilde{F}_k$, we adjust the waiting time $\tilde{\omega}_{mk} = \frac{\omega_{mk}}{F_k}$, and consequently the adjusted gain is:

$$\tilde{\gamma}_{mk} = \max(0, sw(k) - (\tilde{\omega}_{mk} + hw(k))) \quad (3.8)$$

This adjustment is done because, for modules in loops, even if the reconfiguration is not finished in the first few loop iterations and we execute the module in software first, we will gain from executing the module in hardware in future loop iterations. Note that for the simple illustrative example from Figure 3.11, since $\varphi_i = 1, \forall i \in \{0, 1, \dots, 4\}$, we have $\tilde{\gamma}_{mk} = \gamma_{mk}, \forall m, k \in \mathcal{H}$.

We are interested to estimate the potential performance gain $\tilde{\gamma}_{mk}$ over several application runs. We denote this estimate $\tilde{G}_{mk}$, and we use an exponential smoothing formula (similar to equation 3.7) to compute it, emphasizing recent history:

$$\tilde{G}_{mk}^t = \alpha \cdot \tilde{\gamma}_{mk}^t + (1 - \alpha) \cdot \tilde{G}_{mk}^{t-1} \quad (3.9)$$

In equation 3.9, $\tilde{G}_{mk}^t$ represents the estimate at time t of the potential gain obtained if we start reconfiguring module k immediately after m finishes executing, $\tilde{G}_{mk}^{t-1}$ represents the previous estimate, $\tilde{\gamma}_{mk}^t$ represents the adjusted gain computed considering the current application run and α is the smoothing parameter. The speed at which older observations are dampened is a function of α , which can be adjusted to reflect the application characteristics: if the stationary phases are short, then α should be larger (for quick dampening and fast adaptation); if the phases are long, then α should be smaller.

3.3.3.3 Prefetch Priority Function

At each node $m \in \mathcal{H}$ we assign priorities to all the hardware candidates in $\mathcal{K}^m = \{k \in \mathcal{H} | \tilde{L}_{mk} > 0\}$, thus deciding a prefetch order for all the candidates reachable from m with nonzero likelihood (the computation of $\tilde{L}_{mk}$ is described in Algorithm 4). Our priority function estimates the overall impact on the average execution time that results from different prefetches being issued after module m finishes executing. Three factors are considered:

1. The estimated likelihood $\tilde{L}_{mk}$ to reach a candidate $k \in \mathcal{K}^m$ from m , obtained from the piecewise linear prediction algorithm (see Section 3.3.3.1);
2. The estimated performance gain $\tilde{G}_{mk}$ resulting if k is prefetched at m (see Section 3.3.3.2);

3. The estimated frequencies for candidates, $\tilde{F}_k$ (see equation 3.7), used to give higher precedence to modules executed many times inside a loop:

$$\Gamma_{mk} = \tilde{L}_{mk}(1 + \log_2 \tilde{F}_k) \tilde{G}_{mk} + \sum_{h \in \mathcal{K}^m \setminus \{k\}} \tilde{L}_{mh}(1 + \log_2 \tilde{F}_h) \tilde{G}_{kh} \quad (3.10)$$

The first term in equation 3.10 represents the contribution (in terms of execution time gain) of module k : the larger the probability to reach it from m ($\tilde{L}_{mk}$), the higher its estimated frequency ($\tilde{F}_k$) and the bigger the performance gain ($\tilde{G}_{mk}$) it generates if its reconfiguration is started immediately after m finishes executing, the better. Note that the potential execution time gain of module k ($\tilde{G}_{mk}$) is weighted with the estimated probability to reach it ($\tilde{L}_{mk}$) in the current application run. The second term captures the impact that k 's reconfiguration will produce on other modules competing with it for the reconfiguration controller. For all these other reachable modules ($h \in \mathcal{K}^m \setminus \{k\}$), we look at the gain ($\tilde{G}_{kh}$) obtained if we will reconfigure them only after k has finished executing. If for example, k and h are mutually exclusive, then $\tilde{G}_{kh} = 0$. Otherwise, if $\tilde{G}_{kh} \neq 0$, it means that we can afford to start the reconfiguration of h after k and, thus, by reconfiguring k first we still have potential gain generated by h .

Let us compare Γ_{mk} from equation 3.10 (used for dynamic configuration prefetching) with C_{nm} from equation 3.1 (used for static configuration prefetching). $\tilde{L}_{mk}$ is the analogue of $PAP(n, m)$ and reflects the probability to reach the hardware module in discussion from the current prediction point. $\tilde{G}_{mk}$ is the analogue of $\bar{G}_{nm}$ and represents the execution time gain generated by the hardware module in discussion if we prefetch it at the current prediction point. Note that the frequency with which module k is executed, $\tilde{F}_k$, is taken into consideration in the case of dynamic configuration prefetching, as a way to improve over the priority function used for static configuration prefetching. Another difference between the two priority functions is that, in the case of dynamic prefetching (equation 3.10), it does not make sense to discriminate the modules that are mutually exclusive with the one in discussion. This is because in the dynamic prefetching case we only issue prefetches after hardware candidates, not after every node in the control flow graph (as in the static prefetching case). To summarize, the first term in equation 3.10 is analogue to the first term in equation 3.1, and the second term in 3.10 is analogue to the second and third terms in 3.1.

Let us consider the example from Figure 3.11, and the prediction point immediately after M_0 finishes executing. In this case, we have 4 reachable candidates, M_1 to M_4 , and we want to start the reconfiguration of one of them. Assume the reconfiguration and execution times for candidates given in Table 3.2; nodes B_1, n_2 and n_3 have execution times of 2, 100 and 100 time units, respectively. For these values, the time to reach M_1 from M_0 is

$\chi_{M_0M_1} = 2 + 100 = 102$ time units, the waiting time experienced by M_1 if we start its reconfiguration immediately after M_0 finishes executing is $\varpi_{M_0M_1} = \max(0, 80 - 102) = 0$ and the gain $\gamma_{M_0M_1} = \max(0, 70 - (0 + 3)) = 67$ time units. The potential gains for M_2, M_3 and M_4 , as computed in Section 3.3.3.2, are $\gamma_{M_1M_2} = 79, \gamma_{M_1M_3} = 76$ and $\gamma_{M_1M_4} = 0$ (similarly, we can compute all the other needed gains, $\gamma_{M_0M_2} = 85, \gamma_{M_0M_3} = 76, \gamma_{M_0M_4} = 40$ and $\gamma_{M_2M_1} = \gamma_{M_2M_3} = \gamma_{M_2M_4} = \gamma_{M_3M_1} = \gamma_{M_3M_2} = \gamma_{M_3M_4} = \gamma_{M_4M_1} = \gamma_{M_4M_2} = \gamma_{M_4M_3} = 0$).

Let us now consider phase 1 from Figure 3.11a, and assume that B_0 was *taken*, i.e. n_0 was executed. Under these assumptions, we have the following probabilities to reach the four hardware candidates: $\tilde{L}_{M_0M_1} = 100\%, \tilde{L}_{M_0M_2} = 45\%, \tilde{L}_{M_0M_3} = 55\%$ and $\tilde{L}_{M_0M_4} = 0$. The priority functions for the four candidates will be: $\Gamma_{M_0M_1} = 100\% \cdot 1 \cdot 67 + 45\% \cdot 1 \cdot 79 + 55\% \cdot 1 \cdot 76 + 0 \cdot 1 \cdot 0 = 144.35, \Gamma_{M_0M_2} = 45\% \cdot 1 \cdot 85 + 100\% \cdot 1 \cdot 0 + 55\% \cdot 1 \cdot 0 + 0 \cdot 1 \cdot 0 = 38.25, \Gamma_{M_0M_3} = 55\% \cdot 1 \cdot 76 + 100\% \cdot 1 \cdot 0 + 45\% \cdot 1 \cdot 0 + 0 \cdot 1 \cdot 0 = 41.8$ and $\Gamma_{M_0M_4} = 0 \cdot 1 \cdot 40 + 100\% \cdot 1 \cdot 0 + 45\% \cdot 1 \cdot 0 + 55\% \cdot 1 \cdot 0 = 0$ time units. Thus, we can see that the highest execution time gain is obtained if we start reconfiguring M_1 after M_0 , because once M_1 will finish executing, we can replace it with one of the other candidates, and we gain by executing both M_1 and the other module in hardware. In contrast, reconfiguring M_2 or M_3 after M_0 will make sure that their gain is obtained, but it will prevent obtaining any gain from the execution of M_1 , which will be in software.

3.3.3.4 Run-Time Strategy

Algorithm 6 presents our overall run-time strategy for configuration prefetching. Once a module $m \in \mathcal{H}$ is reached, we increment its frequency counter (line 2) that will be used later to update the frequency estimates with equation 3.7. Next we perform the update of the piecewise linear predictor (line 3). We use the path history (registers H_S^m and A_S^m) saved when the last prediction was made at m , and the q candidates reached after m , saved in HW_R^m . After the update, we save the current path history (to be used at the next update), as described in Algorithm 5. Next we compute the likelihoods to reach other candidates from m (line 4), as illustrated in Algorithm 4. Then, for all candidates reachable with nonzero likelihood (line 5), we compute the priority Γ_{mk} (line 6) with equation 3.10. Once all Γ_{mk} have been computed, we find the candidate from $\mathcal{K}^m = \{k \in \mathcal{H} | \tilde{L}_{mk} > 0\}$ with the highest priority (line 7), giving precedence to modules placed in loops in case of equality. If the top candidate found is already fully configured on the FPGA, it will be reused and there is no need to reconfigure it. In this case, we find the candidate with the next highest priority that does not have a placement conflict (i.e. it is not mapped to the same reconfigurable slot) with the higher priority module (because we do not want to replace it

Algorithm 6 Overall run-time strategy for prefetching

-
- 1: {candidate m is reached
 - 2: increment frequency counter φ_m
 - 3: UPDATE(m, H_S^m, A_S^m, HW_R^m) ▷ Update the predictor using the path history saved when the last prediction was made, and then save the current path history
 - 4: LIKELIHOODS(m) ▷ Compute $\tilde{L}_{mk}$
 - 5: $\mathcal{K}^m \leftarrow \{k \in \mathcal{H} | \tilde{L}_{mk} > 0\}$ ▷ Compute the set of candidates reachable from m with nonzero likelihood
 - 6: compute priority function $\Gamma_{mk}, \forall k \in \mathcal{K}^m$, based on the current estimates for likelihoods ($\tilde{L}_{mk}$), frequencies ($\tilde{F}_k$) and performance gains ($\tilde{G}_{mk}$), using equation 3.10
 - 7: pick the module with the highest priority Γ_{mk} that is not yet placed on the FPGA
 - 8: stop (pause) any ongoing reconfiguration
 - 9: start reconfiguring top candidate }

The part above executes in parallel with the one below

- 10: **if** m fully loaded on FPGA **then**
 - 11: execute m on FPGA
 - 12: **else if** remaining reconfiguration + $hw(m) < sw(m)$ **then**
 - 13: continue reconfiguration and execute m on FPGA
 - 14: **else**
 - 15: execute m in SW
 - 16: **end if**
 - 17: save timestamp of m
 - ▷ Compute estimated gains based on m 's finishing time
 - 18: **for all** $k \in HW$ **do** ▷ Candidates reached before m
 - 19: compute performance gain $\tilde{G}_{km}$ with equation 3.9, using the timestamps of k and m to get χ_{km} , $\tilde{\omega}_{km}$ and $\tilde{\gamma}_{km}$ }
 - 20: **end for**
-

on the FPGA). Finally, we pause⁶ any ongoing reconfiguration (line 8) and we start prefetching the top candidate (line 9) identified (line 7).

The execution of the predictor update and the mechanism of generating the prefetches (described above) take place in parallel with the execution of m . This observation, coupled with the facts that our algorithm has a worst-case complexity of $O(|\mathcal{H}| \log |\mathcal{H}|)$ and that part of it runs as a dedicated hardware module (see Section 3.3.4), makes the on-line approach feasible. The execution of m is done in hardware or in software, depending on the run-time conditions: if m is already loaded then it can be executed on the FPGA (lines 10-11); if reconfiguring m and then executing it on the FPGA results

⁶Note that, if not overwritten, these modules might have their reconfiguration resumed later.

(a) Function PREDICT

(b) Function UPDATE

Figure 3.13: The internal architecture of the predictor

in a shorter delay than its software execution, then it is worth to wait for the reconfiguration to finish and then execute the module on the FPGA (lines 12-13); if none of the above holds, then the candidate is executed in software (lines 14-15). Once the execution of m finishes, we save its timestamp and then we compute the estimated performance gains $\tilde{G}_{km}$ for all the modules currently recorded in the hardware history register HW (lines 17-19). After this, the execution of the application continues as normal.

3.3.4 Hardware Organization of the Predictor

This section will discuss the hardware organization of a HW/SW prototype implementation of the predictor described above, on a Xilinx ML605 board. Note that the part specific to the Xilinx architecture is minimal, referring

Figure 3.14: The hardware organization of the predictor

only to the communication between the embedded CPU and the predictor. Everything else concerning the hardware organization is independent from any FPGA vendor. The SW part is implemented as an API for the MicroBlaze embedded CPU (see Section 3.1.2.3), while the HW part is implemented as a slave module connected to the AXI4-Lite bus (see the *Predictor* block in Figure 3.1). Figure 3.14 details the organization of our prototype⁷, considering an example from Figure 3.11 with five hardware candidates (M_0 to M_4), a branch history length of 2 (i.e. $h = 2$), and a hardware history length of 2 (i.e. $q = 2$).

A set of SW mapped registers (SW_0 to SW_7) are used to transfer data necessary for prediction, between the MicroBlaze and the predictor module. Each time an instrumented conditional branch is encountered in the program, the branch ID and its outcome (*taken/not taken*) are sent to the predictor via the software accessible register SW_0 . By asserting the *branch* bit in the control register (SW_7), the branch ID and its outcome are pushed into the FIFO history registers A and H (see Section 3.3.3.1). Similarly, the ID of every HW candidate reached by the application is pushed into the HW history register, via SW_6 , on assertion of the *candidate* bit. The register P has one bit for every HW candidate: this bit is asserted for all the modules currently present in the HW history register, and is used during

⁷Some of the trivial details have been omitted for clarity of the illustration.

the update of the predictor. Instead of explicitly saving all the IDs of the hardware modules reached after a prediction point, we save the register P . The *Control Logic* block takes care of generating the internal control signals.

The architecture presented here implements the functions PREDICT (from Algorithm 4) and UPDATE (from Algorithm 5). For clarity, we will describe them separately, although they work with the same data structures (same colors/shades are used in Figures 3.14, 3.13a and 3.13b to illustrate the relations between the three schematics). Figure 3.13a presents the internal organization of the prediction function (exemplified for candidate M_0). Note that the address history register A currently contains the IDs of 2 branches, B_x and B_y . Each one of these is used to index into a table containing the weights (Ω), which are added or subtracted (depending on the content of the branch outcome history register H) to compute the prediction output for M_0 . Similarly, all the outputs are computed for all the hardware candidates, using the corresponding weights from Ω . The results are written to the software accessible registers (SW_1 to SW_5), from where the MicroBlaze can read them.

Figure 3.13b illustrates the update functionality. Before proceeding, let us mention that, each time an update is made (after reaching a hardware candidate m), the registers A , H and P are saved in A_S^m , H_S^m and $P[HW_R^{top}]$ (where *top* is the module at the top of the HW history register). These saved values are used when we perform the lazy update (see Algorithm 5). Returning to Figure 3.13b, note that the weights in Ω are incremented for taken branches, and decremented for not taken ones, if the corresponding module was reached (as indicated by $P[HW_R^m]$), and vice versa for not reached modules (note that we encode *taken/not taken* branches, as well as *reached/not reached* modules with ± 1). As mentioned above, the control signals are generated by the *Control Logic* block in Figure 3.14, and, for clarity, are not depicted.

Given the hardware implementation of the predictor, all the updates to Ω , as well as the prediction, can be made in parallel, thus reducing the time overhead incurred, especially for big designs. This comes at the cost of a small area overhead (see Table 3.1), which is justifiable given the performance improvement obtained in return.

3.3.5 Experimental Evaluation

3.3.5.1 Proof of Concept

We have implemented the entire framework described in Section 3.1.1 on a Xilinx ML605 board, featuring an XC6VLX240T Virtex6 FPGA⁸. In order to demonstrate the efficiency and practicality of the piecewise linear predictor we used a case study application: the SUSAN image processing [SB97]. This algorithm was developed for recognizing corners and edges in MRIs of

⁸Note, however, that the framework could be instantiated on any other FPGA architecture that supports PDR (e.g. [Alt12])

the brain, but it is typical of any application which would be employed in vision based systems that benefit corner and edge detection. Such examples include machine vision (industrial robot guidance, automatic inspection), autonomous cars (lane keeping, collision avoidance), face recognition, image search, etc. Most of these applications require high performance, and we will show that our framework can deliver that with minimal energy consumption overheads. Without exception, cost is important in all these systems, so keeping the resource usage as low as possible is essential.

We will not go into details regarding the SUSAN corner and edge detection algorithms, but will only briefly explain them. The main idea is to use non-linear filtering in order to define which parts of an image are closely related to each pixel. A local region that has similar brightness with each pixel is determined, and the feature (corner/edge) detectors are based on minimizing this local image region.

The algorithm uses a mask, with its center pixel referred to as the *nucleus*. For each possible position of the mask in an image, the area of the mask that has similar brightness as the nucleus is found. This area is known as *USAN* (Univalue Segment Assimilating Nucleus). The most interesting property of an USAN is its area, since it conveys information about the structure of the image around the nucleus. The USAN area attains its maximum when the nucleus lies in a flat region of the image, being approximately halved as we approach an edge, and falling even further (to roughly a quarter of the maximum) when inside a corner. Based on these observations, the SUSAN (Smallest USAN) algorithm was developed [SB97].

We have ported the software implementation to the MicroBlaze embedded CPU and we have developed hardware modules to perform SUSAN corner and edge detection. Figure 3.15 presents the general design of the corner detector module⁹. Note that the blocks shown in Figure 3.15 represent the user logic to be contained in reconfigurable slot RS_1 from Figure 3.1. The edge detector module is mapped to the same slot RS_1 . Note that we have considered that we have space on the FPGA for only one reconfigurable slot (RS_2 from Figure 3.1 is not used in our implementation, and was presented earlier only for illustrative purposes). The FSM implements the functionality of the corner detection module: it interfaces with the shared BRAM block, from which it reads the image pixels. For faster processing, a pipeline is used (the mask is illustrated with light shading and the nucleus with black). A FIFO buffer is used between the FSM and the pipeline. For each nucleus, the USAN is computed, and written to an internal memory buffer. Then, a 5×5 mask is used to search for local minima, and thus identify the corners in the image. Once a corner is found, its coordinates are delivered to the FSM, and the corner is marked at the appropriate position in the shared BRAM block containing the image. After the whole image is processed, an interrupt signal is generated and the embedded CPU can read the result from the shared BRAM block.

⁹The architecture of the edge detector is very similar.

Figure 3.15: The hardware implementation of corner detection

In order to evaluate our framework, we have assumed the following scenario: two different sources generate images that need to be processed with either corner or edge detection. Each source has a buffer allocated in the DDR3 memory to store its corresponding images, one at a time (see Figure 3.1). The application runs periodically, and in each period, one of the two feature detectors are applied to one of the two image sources. One switch controls which feature detection should be performed and another switch determines which image source should be used. One could imagine such a scenario to be useful in machine vision or automotive vision applications, like, e.g. lane recognition.

The software implementations of both corner and edge detection, running on the MicroBlaze, were extremely slow. For a test image of roughly 128 KB, the SUSAN corner detection took 9.367 seconds to finish, while the SUSAN edge detection took as long as 28.804 seconds. Obviously, these running times were prohibitive for the applications we suggested above. Thus, a hardware solution was necessary. The FPGA implementation of corner detection ran in only 2.15 milliseconds, while edge detection ran in 2.2 milliseconds, orders of magnitude faster than the corresponding software implementations. Note that these times do not include the time to transfer the image from DDR3 to the shared BRAM block, using the AXI central DMA. It took 341 microseconds, to write/read the image to/from the BRAM memory. The partial reconfiguration of either one of the two modules took 3.07 milliseconds, the size of the bitstreams being 1.12 MB,

and the effective throughput of our custom reconfiguration controller with DMA reaching 375 MB/s.

We have experimented with two different phases, characterized by different correlations: in phase 1, images from source 1 require edge detection while those from source 2 require corner detection, and vice versa in phase 2. Given this, we considered a number of scenarios, varying the phase length, as well as the frequency at which images were generated by the two sources within a phase. All these parameters determined the interleaving of performing edge detection or corner detection, as well as the long term probabilities to perform one or the other.

We have considered four scenarios:

1. Phase length 100 iterations, and the two image sources produce images with the same frequency (i.e. we always alternate between corner and edge detection);
2. Phase length 100 iterations, but the second image source produces images with half the frequency of the first;
3. Phase length 20 iterations, and the two image sources produce images with the same frequency;
4. Phase length 20 iterations, and the second image source produces images with half the frequency of the first.

We compared our approach with two alternative approaches:

1. The straightforward fetch-on-demand (FOD) technique. In this case, no reconfiguration overhead is hidden, since the hardware modules are reconfigured only on demand, when they need to run, but are not already on the FPGA.
2. The static configuration prefetching approach presented in Section 3.2. In this case, the hardware modules are prefetched based on probabilities obtained via profiling¹⁰.

We have run 1000 iterations of the application, and we have measured how long one iteration takes on average.

Figure 3.16 presents the improvements that we obtained for the different scenarios: given that the average execution time of one iteration of the program is denoted with AVG , we computed the improvement produced by the prediction (P) approach over FOD as $I_{FOD}^P = \frac{AVG_{FOD} - AVG_P}{AVG_{FOD}}$ and over the static (S) prefetching (Section 3.2) as $I_S^P = \frac{AVG_S - AVG_P}{AVG_S}$. Note

¹⁰Note that this information can be obtained for the simple scenarios we have assumed, but in more complex cases it might be impossible to obtain such profiling information, or it might be inaccurate. Furthermore, the profiling information is average, and the static technique completely ignores the phase behavior. Thus, in many cases, the prefetches generated for the average behavior might be completely irrelevant for certain phases.

Figure 3.16: Average execution time reduction for SUSAN

that we consider that the shorter the average execution time, the better the performance of the application. One could imagine, for example, that by having shorter execution times more images could be processed, and this would translate into better performance. The improvements obtained range from 23% up to 32% compared to FOD, and from 19% up to 20% compared to static prefetching. The improvement obtained is better when comparing with the fetch-on-demand (FOD) technique. This is expected, since FOD does not perform any prefetching: when a hardware candidate is reached, if it is not present on the FPGA, then it will be reconfigured first, and only after that executed in hardware. When the two sources produce images with the same frequency (scenarios 1 and 3), the program will always alternate between corner and edge detection and, as a result, the reconfiguration overhead is incurred in every program iteration for FOD. As opposed to that, our prediction approach will exploit the correlations in order to adapt to the run-time requirements and prefetch the corresponding candidate. The performance of FOD is slightly better for scenarios 2 and 4, because in this case, since one source produces images with double the frequency of the other, FOD benefits from the fact that one of the modules is reused on the FPGA, before being replaced by the other one. Note that the impact of the phase length is not so significant, since our predictor adapts quickly to the phase changes, and FOD is unaffected by phases.

For the case of static prefetching, the improvements are slightly lower, but still significant (from 19% up to 20%). As opposed to the case of FOD, static prefetching does not perform better when one source produce images with double the frequency of the other. Considering that only correlations change from phase 1 to phase 2, the long term probabilities associated with reaching corner or edge detection do not depend on the frequency with which sources generate images: in phase 1, edge detection is executed twice as often as corner detection, and in phase 2, vice versa. Since static prefetching fixes the prefetch decisions at design-time, the same hardware module (edge detection in this case) will always be prefetched, and for the second module (i.e. corner detection), the reconfiguration overhead will never be overlapped with useful computations. Also, the corner detection will not be reused either (even if it is needed in two consecutive program iterations), since it

will always be overwritten by the prefetch of the edge detection module. As in the case of FOD, the effect of phase length is marginal.

Energy measurements

We have also evaluated the energy consumption of our predictor framework, implemented on the Xilinx ML605 board. We have used a shunt resistor ($R = 100\text{m}\Omega$) in series with the voltage source of the FPGA ($V_{CC} = 12\text{V}$) in order to measure the current drawn by the board (I). The voltage drop on the shunt resistor (U) was amplified ($10\times$) and monitored using the PicoScope USB oscilloscope. The current drawn by the board is $I = \frac{U}{R} = \frac{U_{osc}V}{10 \times 0.1\Omega} = U_{osc}A$, where U_{osc} is the value measured by the oscilloscope. Thus, by integrating the oscilloscope traces on a time interval $\Delta t = [t_0, t_1]$ and then multiplying with the supply voltage $V_{CC} = 12\text{V}$ we obtained the energy consumption of the board in the interval Δt .

We have run 1000 iterations of the image processing application for scenario 1) from above, we have measured the energy consumption, and then computed the average energy consumed by the application in one iteration. We have considered three implementations of the image processing application:

1. SW-only: both corner and edge detection were run as software sub-routines;
2. static-HW: both corner and edge detection were implemented as modules on the FPGA, each one with its dedicated area (i.e. no partial dynamic reconfiguration was used);
3. with prediction: the hardware modules for corner and edge detection were mapped to the same area on the FPGA, and our predictor was used to prefetch the modules using partial dynamic reconfiguration.

For the SW-only implementation, since the execution time of the image processing application was very big (as reported above, 9.367s for corner detection and 28.804s for edge detection), the average energy consumption was also high: $E_{\text{SW-only}} = 304.60\text{J}$. For the static-HW implementation, which is orders of magnitude faster (2.49ms for corner detection and 2.54ms for edge detection), the average energy consumption amounts to $E_{\text{static-HW}} = 89.59\text{mJ}$. Finally, when running the predictor, we obtain an average energy consumption of $E_{\text{pred}} = 98.64\text{mJ}$. This represents an increase of as little as 10.1% over the static-HW implementation, but only half the FPGA area was needed for the implementation of the image processing application, and the average execution time with the predictor was only 10% bigger than the static-HW execution time.

Note that the resource usage of the framework itself is equal in the case of FOD and static prefetching. Our technique needs the “Predictor” module (depicted in Figure 3.1), so an overhead of 419 LUTs and 151 FFs is incurred

(as little as 4% out of the total framework LUTs, and 1.5% out of the total framework FFs; see Table 3.1). We consider this small area overhead justified, given the improvements obtained. The measurements show that by using our prediction framework we can obtain good performance (short execution times) with limited hardware area and with only a minor energy consumption overhead.

3.3.5.2 Performance Improvement

We have also performed simulation experiments with generated examples, in order to evaluate how the piecewise linear predictor behaves compared to other prefetching approaches. We generated 2 sets containing 25 control flow graphs each: Set_1 with small graphs (between 48 and 166 nodes, ~ 100 on average), and Set_2 with bigger graphs (between 209 and 830 nodes, ~ 350 on average). The software execution time for each node was randomly generated in the range of 50 to 1000 time units. Between 15% and 40% of the nodes were selected as hardware candidates (those with the highest software execution times), and their hardware execution time was generated β times smaller than their software one (the coefficient β models the hardware speedups). We considered two possible situations: in the first one, β was chosen from the uniform distribution on $[10^3, 3 \cdot 10^3]$ (*fast hardware*: applications exhibiting high hardware acceleration), in order to reflect the results obtained from our board measurements (see Section 3.3.5.1); in the second situation, we generated much lower speedups, from the uniform distribution on $[3, 7]$ (*slow hardware*: applications exhibiting lower hardware acceleration), similar to the assumptions from Section 3.2.4.2 and [S⁺10]. We also generated the size of the candidates in concordance with our real-life implementation of the SUSAN image processing algorithm. The reconfiguration time for the modules was determined based on their size, considering the average throughput of our DMA reconfiguration controller (375 MB/s). The placement was decided using existing techniques (like [HMZB12] or [BHS⁺13]) that minimize the number of placement conflicts. We generated problem instances where the size of the reconfigurable region is a fraction (15% or 25%) of the total area required by all candidates, $\text{MAX_HW} = \sum_{m \in \mathcal{H}} \text{area}(m)$.

All experiments were run on a PC with CPU frequency 2.83 GHz, 8 GB of RAM, and running Windows Vista. For each application we have evaluated different prefetch techniques using an in-house Monte Carlo simulator (based on the one described in Section 3.2.4.1) that considers the architectural assumptions described in Section 3.1.1. Recall that in the case of static prefetching (see Section 3.2) we ignored correlations between branches. Thus, we had to modify the simulator such that, for control nodes, correlations between two or more branches were captured through joint probability tables. In such a case, whenever we performed a draw from the marginal Bernoulli distribution for a branch, we computed the conditional probabilities for all the branches correlated with it, based on the joint probability

Figure 3.17: Performance improvement for the simulation experiments

table. Later in the simulation, when the correlated branches were reached, we did not sample their marginal distribution, but instead we sampled their conditional distribution based on the outcome of the first branch. We have determined the results with an accuracy of $\pm 1\%$ with confidence 99%.

We were interested to see how our approach compares to the current state-of-the-art, both in static (Section 3.2) and dynamic prefetching [LH02], [HV09]. Thus, we simulated each application using the prefetch queues generated by our approach and those generated by the static prefetching technique presented in Section 3.2, [LH02] and [HV09]. The parameters for our predictor (see Section 3.3.3.1) were chosen, depending on the application size, from the following ranges: $\alpha = [0.4, 0.6]$, $q = [2, 8]$, $h \in [4, 16]$.

The appropriate metric to evaluate prefetch policies is the total time spent executing the hardware candidates, plus the time waiting for reconfigurations to finish (in case the reconfiguration overhead was not entirely overlapped with useful computations). If this value is small, it means that the prefetch policy had accurate predictions (many candidates were executed in hardware), and the prefetch was done early enough to have short waiting times. We do not include comparisons with the SW-only and static-HW implementations: As our case study suggested, the SW-only version of an application has prohibitive execution times and can only be worse than any prefetching technique; The static-HW version corresponds to an ideal and unrealistic scenario, when we have unlimited resources (i.e. enough HW area to place *all* the application modules *at the same time* on the FPGA).

We denote the average time spent executing hardware candidates, plus waiting for reconfigurations to finish, with EX_P for our dynamic approach, with EX_S for the static approach from Section 3.2, with EX_M for the dynamic Markov approach [LH02] and with EX_A for the dynamic aggregate gains approach [HV09]. We compute the performance improvement of our approach over the static, $PI_S^P = \frac{EX_S - EX_P}{EX_S}$; similarly we calculate PI_M^P and PI_A^P , the improvements of our approach over [LH02] and [HV09], respectively.

Figures 3.17a and 3.17b show the results obtained (averaged over all graphs in Set_1 and Set_2) for the scenarios with fast and slow hardware, re-

spectively. The improvements obtained for fast hardware are slightly higher than those obtained for slower hardware. It is important to note that the baseline approaches also benefit from faster hardware. The improvements over the static approach (presented in Section 3.2) are higher because static prefetch lacks flexibility. The improvements over both dynamic approaches ([LH02], [HV09]) are also significant, ranging from 15% up to 30% on average for fast hardware and from 14% up to 29% on average for slow hardware.

3.4 Summary

This chapter proposed a complete framework for partial dynamic reconfiguration of FPGAs, together with optimization approaches to configuration prefetching for performance enhancement. We first presented a static prefetching algorithm. Based on profiling information, and taking into account the placement of hardware modules on the FPGA, we statically schedule the appropriate prefetches (and implicitly perform HW/SW partitioning of the candidate hardware modules) such that the expected execution time of the application is minimized. For applications with inaccurate or unavailable profiling information, and for those that exhibit non-stationary behavior, it is important to have a mechanism that adapts to changes. Thus, we proposed an approach for dynamic prefetching of FPGA configurations, with the goal to minimize the expected execution time of an application. We used a piecewise linear predictor, coupled with an on-line mechanism, based on timestamps, in order to generate prefetches at run-time. The efficiency and practicality of our FPGA configuration prefetching platform was demonstrated with a proof of concept implementation of a real-life application (the SUSAN image processing algorithm [SB97]), complemented by extensive simulations.

ON-THE-FLY ENERGY MINIMIZATION FOR MULTI-MODE REAL-TIME SYSTEMS

MODERN heterogeneous architectures bring together multiple general-purpose CPUs and multiple GPUs and FPGAs, in an attempt to answer the performance, energy-efficiency and flexibility requirements of today's complex applications [Fou15]. However, in order to leverage the advantages of such architectures, careful optimization is essential. Real-time multi-mode systems are a good model for a wide range of applications that dynamically change their computational requirements over time [SSC03], [SBSH12]. In this context, intelligent on-line resource management is needed, such that the heterogeneous resources are used in an energy-efficient manner, while meeting the real-time constraints. In this chapter we propose a resource manager that implements run-time policies to decide on-the-fly task admission and the mapping of active tasks to resources, such that the energy consumption of the system is minimized and all task deadlines are met.

The remainder of this chapter is organized as follows. The system model assumed and the statement of the problem addressed are presented in Sections 4.1 and 4.2, respectively. Section 4.3 describes the challenges we are facing by means of a motivational example. Details of the design optimization approach for one mode and the integrated multi-mode optimization for energy minimization, respectively, are described in Sections 4.4 and 4.5. We present the experimental validation of the proposed approaches in Section 4.6, which includes both real-life measurements and simulation results. The contribution of the chapter is summarized in Section 4.7.

Figure 4.1: A heterogeneous architecture model for multi-mode systems

4.1 System Model

4.1.1 Architecture Model

We consider a heterogeneous platform (see Figure 4.1) consisting of m CPUs, q GPUs, and an FPGA (divided into r reconfigurable partitions). Note that resources of a certain type need not be identical (i.e. we could have different CPUs and different GPUs). We assume that the platform is under the control of a resource manager, whose role is described in Section 4.1.3. The FPGA supports partial dynamic reconfiguration, which means that parts of it may be reconfigured at run-time, while other parts remain fully functional. The FPGA is organized into identical reconfigurable partitions, where tasks can be mapped and reconfigured dynamically at run-time. We assume that the resource manager can issue non-blocking reconfiguration commands to a reconfiguration controller which is responsible for downloading the bitstreams to the FPGA partitions. A detailed implementation of such a reconfiguration controller on a Xilinx board is presented in Section 3.1.1.2.

4.1.2 Application Model

We denote the set of active tasks (i.e. releasing periodic jobs) with $T = \{\tau_i | i = 1, 2, \dots, n\}$. Each task τ_i is described by several parameters:

- Period p_i , which gives the time duration between two consecutive task activations (referred to as *jobs* in the rest of this chapter);
- Deadline $d_i = p_i$ (implicit-deadline assumption¹);
- Worst-case execution time (WCET), c_{ij} , of task τ_i on resource j . Note that resource j can be a CPU, a GPU, or the FPGA;
- The energy consumption, e_{ij} , of task τ_i on resource j ;

¹Generalization to $d_i \leq p_i$ is straight-forward.

- The time overhead c_i^r and the energy overhead e_i^r of reconfiguring task τ_i on the FPGA;
- Given the above parameters, we can compute for every task τ_i its maximal utilization on every resource: $u_{ij} = \frac{c_{ij}}{p_i}$.

We assume that all tasks from the active task set have implementations for at least one computational resource². The executable code for the CPUs, the kernel implementations for the GPUs, and the bitstreams specifying the FPGA implementations are all stored in a memory to which the resource manager has access. We denote with $map(\tau_i) = j$ the fact that task τ_i is mapped to resource j .

Let us next define a *mode*: we characterize a functional mode o by the composition of the active task set T_o containing the tasks that are currently releasing periodic jobs. For the duration of a certain mode, T_o does not change. When an existing task leaves the system (i.e. stops releasing jobs), or a new task enters the system, we say that a transition to a new mode has occurred. Every such transition (if allowed by the resource manager described in Section 4.1.3) has to take place within a specified *set-up time*, t_{set} . This time is used to perform the mapping optimization and enforce any remapping decisions, including FPGA reconfigurations and loading of the tasks' software into memory.

4.1.3 Resource Management

The entire system is under the control of a *resource manager*, whose main responsibility is to decide at run-time task admission and the mapping of tasks to resources such that the energy consumption is minimized while all the deadlines are met. We consider that the manager is running on a dedicated resource, and it could be implemented either in software or in hardware. The manager needs to know the task parameters mentioned in Section 4.1.2 and to be aware of the current mode (i.e. which is the currently active task set). Thus, any task that is currently active but wants to leave the system (i.e. stop releasing jobs) must notify the manager of this. Also, any new task that wants to become active in the system (i.e. wants to start releasing jobs) must first register with the resource manager (i.e. communicate its task parameters), and wait for permission to become active. In case the manager cannot find a mapping for the new task such that all deadlines will be satisfied, then the new task will not be granted permission to activate. This situation might occur in two cases: either the system would become overloaded if the new task is accepted (thus, no feasible mapping exists), or a feasible mapping exists, but the manager is unable to find

²Of course, the more implementations a task has the better the optimization opportunities. However, it is not needed that tasks have implementations for all processing elements; if no implementation is available for task τ_i on resource j then the corresponding mapping option is ignored by setting $c_{ij} = e_{ij} = \infty$.

it. Section 4.6.2.2 evaluates our proposed run-time policies (presented in Section 4.5.3) from this point of view. The resource manager performs *admission management* by deciding if a new task can enter the system or not (i.e. deciding if transition to a new mode is allowed or not), depending on the current load. Moreover, if admission is granted, a task mapping is also decided. In Section 4.5.3 we propose several run-time policies to perform these decisions.

4.1.4 Scheduling

One approach to multiprocessor scheduling is partitioning, which means that tasks are assigned to resources and then each resource schedules its assigned tasks using a uniprocessor scheduling algorithm. We have to address the scheduling problem for each of the three types of processing elements (CPUs, GPUs, FPGA) assumed in our architecture.

Let us first note that for the FPGA, there is no scheduling problem, because each FPGA partition can be seen as a dedicated processor, running one single task. Thus, we need to make sure that at any moment in time there are no more than r tasks mapped to the FPGA (because we have r reconfigurable partitions):

$$\sum_{\substack{i \\ \text{map}(\tau_i)=j}} 1 \leq r, j = \text{FPGA index} \quad (4.1)$$

Note that, in order to ensure the feasibility of mapping a certain task to the FPGA, we also need to make sure that its execution time is shorter than its deadline:

$$c_{ij} \leq d_i, \forall \tau_i \in T : \text{map}(\tau_i) = j, j = \text{FPGA index} \quad (4.2)$$

As opposed to the FPGA, the situation is different for the CPUs and the GPUs, as we will explain next. We will apply the Earliest Deadline First (EDF) scheduling policy for the tasks mapped to the CPUs. One of the advantages of EDF is that there exists a simple necessary and sufficient condition which ensures that all task deadlines are met [But04]:

$$\sum_{\substack{i \\ \text{map}(\tau_i)=j}} \frac{c_{ij}}{p_i} \leq 1, \forall \text{CPU } j \quad (4.3)$$

As far as the GPUs are concerned, execution of tasks is non-preemptive. Thus, we chose to schedule the tasks mapped to the GPUs using uniprocessor FIFO scheduling. The schedulability conditions in this case are:

$$\sum_{\substack{i \\ \text{map}(\tau_i)=j}} \frac{c_{ij}}{p_i} \leq 1, \forall \text{GPU } j \quad (4.4)$$

and

$$\sum_{\substack{i \\ \text{map}(\tau_i)=j}} c_{ij} \leq d_k, \forall \text{GPU } j, \forall \tau_k \in T : \text{map}(\tau_k) = j \quad (4.5)$$

Inequation (4.4) guarantees that no GPU is overloaded, while (4.5) ensures that no deadline is missed for any task mapped to a GPU. These conditions are sufficient to ensure the schedulability of GPU tasks [EA12]. Although these conditions are not necessary, they have the advantage that they are simple and easy to integrate in the optimization. Such conditions are required to make sure that we meet the hard deadlines of all tasks admitted to the system by the resource manager (described in Section 4.1.3). Note that in the case of non-preemptive FIFO scheduling, considering the synchronous implicit-deadline periodic task model, in the worst case the execution of a task could be delayed by all the other tasks in the task set that are mapped to the same resource.

4.2 Problem Formulation

Given a heterogeneous architecture (as described in Section 4.1.1), and an application (as described in Section 4.1.2), our goal is to determine a run-time policy to enable the resource manager to perform task admission and map tasks to resources such that the energy consumption is minimized and all deadlines are met.

4.3 Motivational Example

We will use a motivational example to illustrate the challenges posed by solving the problem formulated above. Let us consider a heterogeneous architecture as the one illustrated in Figure 4.1, composed of 2 CPUs, 1 GPU, 1 FPGA with 2 reconfigurable partitions, and a resource manager. Let us consider the 12 tasks whose parameters are specified in Table 4.1 (we remind the reader that our framework does not assume that the task set is known at design-time). For simplicity of the illustration, we consider the 2 CPUs identical and thus $c_{i1} = c_{i2}$, $e_{i1} = e_{i2}$ and $u_{i1} = u_{i2}, \forall \tau_i$.

Let us assume the following multi-mode behavior for the system (illustrated graphically in Figure 4.2): it starts in a mode o_1 with the active task set $T_{o_1} = \{\tau_1, \tau_2, \dots, \tau_{11}\}$. This mode will be resident for $t_{o_1} = 100$ time units, and then the system will transition into a new mode o_2 where task τ_{11} becomes inactive and stops releasing jobs, the active task set becoming $T_{o_2} = T_{o_1} \setminus \{\tau_{11}\} = \{\tau_1, \tau_2, \dots, \tau_{10}\}$. After another 100 time units, τ_{12} wants to become active; unfortunately, the manager fails to find a feasible mapping for τ_{12} . Thus, the potential transition to mode o_3 , with $T_{o_3} = T_{o_2} \cup \{\tau_{12}\} = \{\tau_1, \tau_2, \dots, \tau_{10}, \tau_{12}\}$, is prohibited by the resource manager, this decision ensuring the correct and uninterrupted functioning of the

Table 4.1: Task parameters for the motivational example

τ_i	p_i	$d_i = p_i$	c_{i1}	c_{i2}	c_{i3}	c_{i4}	e_{i1}	e_{i2}	e_{i3}	e_{i4}	u_{i1}	u_{i2}	u_{i3}	u_{i4}
τ_1	1	1	0.1	0.1	0.07	0.03	0.3	0.3	0.04	0.05	0.1	0.1	0.07	0.03
τ_2	0.1	0.1	0.03	0.03	0.005	0.002	0.1	0.1	0.04	0.03	0.3	0.3	0.05	0.02
τ_3	0.5	0.5	0.1	0.1	0.03	0.05	0.1	0.1	0.04	0.09	0.2	0.2	0.06	0.1
τ_4	1	1	0.4	0.4	0.17	0.13	0.3	0.3	0.04	0.05	0.4	0.4	0.17	0.13
τ_5	0.2	0.2	0.12	0.12	0.05	0.08	0.4	0.4	0.12	0.1	0.6	0.6	0.25	0.4
τ_6	2	2	1.2	1.2	0.02	0.02	0.4	0.4	0.01	0.008	0.6	0.6	0.01	0.01
τ_7	1.2	1.2	0.2	0.2	0.06	0.05	0.25	0.25	0.03	0.025	0.16	0.16	0.05	0.04
τ_8	0.4	0.4	0.1	0.1	0.05	0.02	0.1	0.1	0.05	0.04	0.25	0.25	0.125	0.05
τ_9	0.2	0.2	0.12	0.12	0.05	0.08	0.4	0.4	0.12	0.1	0.6	0.6	0.25	0.4
τ_{10}	1	1	0.25	0.25	0.7	0.06	0.3	0.3	0.04	0.05	0.25	0.25	0.7	0.06
τ_{11}	0.4	0.4	0.25	0.25	0.06	0.04	0.1	0.1	0.09	0.03	0.62	0.62	0.15	0.1
τ_{12}	0.1	0.1	0.035	0.035	0.066	0.02	0.1	0.1	0.04	0.03	0.35	0.35	0.66	0.2

Figure 4.2: Motivational example for on-the-fly energy minimization for multi-mode real-time systems

current mode o_2 . Assume that mode o_2 stays resident for 50 more time units (thus, $t_{o_2} = 150$) and then the system performs a transition back to the initial mode o_1 . Further, the events and transitions illustrated in Figure 4.2 happen, and they are discussed below.

Our goal is to minimize the energy consumption of the system during its lifetime, while ensuring that all the active tasks' deadlines are met. Recall that any mode transition has to take place within a set-up time, which in this example we consider to be at most $t_{set} = 5$ time units. Also, note that we handle task arrivals and departures one at a time. In Figure 4.2 we present a timeline to illustrate the above multi-mode scenario. At $t = 0$ the system starts and all the tasks from the set T_{o_1} register their parameters with the resource manager and express their intent to become active and start releasing jobs. The manager has to decide if the task set T_{o_1} is schedulable on the given architecture and what is the best mapping of tasks to processing elements that will generate the minimum energy consumption. Since the set-up time is only 5 time units, the manager cannot afford to wait for the result of an exact optimization (which would take longer). Instead, a fast heuristic solution is found and enforced. The mapping vector $y_i = \text{map}(\tau_i)$ obtained with our heuristic presented in Section 4.4.2 is specified in Figure 4.2; note that a schedulable solution is found and no task is rejected. The solution is obtained and applied (i.e. FPGA reconfigured and processing elements initialized) in 2 time units.

Thus, at time $t = 2$ the system starts running in mode o_1 , with all tasks active and releasing jobs. According to the obtained mapping, tasks τ_1, τ_7 and τ_{11} will be assigned to CPU_1 , τ_2 and τ_{10} to CPU_2 , τ_3, τ_4, τ_6 and τ_8 to the GPU , and τ_5 and τ_9 to the $FPGA$. Although this mapping is suboptimal we cannot do better at this moment. However, we could run an exact optimization for this mode, cache the result and apply it next time when the mode is visited. We solve our ILP presented in Section 4.4.1 in parallel with the execution of mode o_1 . The optimization takes 20 time units, after which the result is cached. At time 102, task τ_{11} becomes inactive, thus freeing some computational resources. Since we do not have a solution cached for this mode yet, we are now faced with several options:

1. Run the fast remapping heuristic. However, since the difference compared to the previous mode is only that task τ_{11} became inactive, we cannot expect a significantly improved solution. If we look at the heuristic solutions for modes o_1 and o_2 in Figure 4.2, we can see that the mapping for tasks τ_1 to τ_{10} is identical. Instead of wasting time and energy to re-run the heuristic, we would better just keep the mapping from the previous mode o_1 . In this case, the system would function without interruption. However, we can do better as we discuss next.
2. Another viable option is to use the exact solution (cached) from mode o_1 for all tasks except τ_{11} which is not active anymore. As can be seen from Figure 4.2, tasks τ_1 to τ_{10} have identical mapping in modes

o_1 and o_2 for the ILP too. Thus, we will use a set-up time of 1.5 time units to replace the heuristic mapping from o_1 with the mapping adapted for o_2 from the cached ILP solution for o_1 (which turns out to be optimal for all the tasks active in o_2).

Let us next discuss the time step $t = 203.5$, when task τ_{12} wants to become active attempting to switch the system to mode o_3 . In this case, the resource manager is again faced with several options, in order to decide where to map τ_{12} :

1. The most straight-forward solution would be to map τ_{12} to a processing element that still has enough free capacity to accommodate τ_{12} and that is the most energy efficient for it. Unfortunately, there is no single resource with enough computational capacity to accommodate τ_{12} (and this is often the case in situations where the system load is high). Thus, we need a better approach.
2. A second option is to run the fast mapping heuristic, and if this results in a feasible solution, enforce it. Otherwise, reject task τ_{12} . In our example it happens that the heuristic does not find a feasible solution for mapping the task set T_{o_3} even though one exists (see the ILP solution in Figure 4.2). As a result, τ_{12} is rejected at 204.5.

In order to mitigate the problem above, we resort to caching: if a cached solution for mode o_3 existed, it would not be needed to re-run any optimization. With this in mind, we run the ILP in parallel with the execution of the system, and cache the result. A feasible solution that permits the activation of τ_{12} exists and at a future instantiation of mode o_3 we will be able to accommodate it. In the scenario we discussed above, mode o_2 continues its execution until $t = 253.5$, and then task τ_{11} becomes active again and the system transitions back to mode o_1 with $T_{o_1} = \{\tau_1, \tau_2, \dots, \tau_{11}\}$. Note that this time we already have an optimal solution for o_1 (cached from its previous instantiation) and we can apply it directly in a very short set-up time of 0.1 time units.

The system goes next through the following mode transitions: $o_1 \rightarrow o_2$ (τ_{11} leaving the system at 353.6); no cached solution exists for o_2 , but the ILP solution for o_1 was cached at 22. Thus, the resource manager can adapt this mapping (as it did the first time when o_2 was entered, at 103.5). Next we encounter $o_2 \rightarrow o_3$ (τ_{12} arriving at 453.6 and being allowed to activate at 353.7); the instantiation of mode o_3 is now possible as opposed to the first time (when τ_{12} arrived at 203.5), because an optimal solution for o_3 was cached at 224.5. At time $t = 503.7$, when τ_{11} arrives, the system will try to perform the transition $o_3 \rightarrow o_4$, to the new mode o_4 with $T_{o_4} = \{\tau_1, \tau_2, \dots, \tau_{12}\}$. This situation is not schedulable (see Figure 4.2, no feasible mapping found for o_4), so the resource manager will not allow τ_{11} to activate. It is important to note that we consider a very simple scenario in this example, only to illustrate some of the challenges. New tasks might

appear and generate new modes. We need not know which tasks will run in the system at design-time, the only requirement is that they register with the resource manager when they want to become active.

So far we did not mention the energy consumption of the system. Let us consider the time horizon up to 553.7 time units. We have simulated the scenario described above (applying heuristic solutions together with ILP ones as described), and we obtained a total energy consumption of $E = 268$ J; this value includes the energy overheads for running the heuristic and the ILP, as well as the overheads of reconfiguring the FPGA partitions. For comparison purposes, we have simulated a *golden run* of the system, with the same mode changes, but every time we assumed that we have the ILP solution already cached. For the golden run we did not consider any overheads for running the optimization, but only those for reconfiguring the FPGA, and we obtained an energy consumption of $E_{gold} = 209$ J. The described approach is only 28% away from the golden run. Note that the golden run is impossible to achieve in practice, and the fact that we considered only a restricted time horizon. Depending on the multi-mode behavior of the system, the cached solutions might be reused multiple times in the future, thus obtaining low energy consumption without re-running the optimizations.

Let us note that we have discussed three ways to obtain the mapping for a new mode:

1. Optimal, by running the ILP from Section 4.4.1;
2. Heuristic:
 - (a) by running the algorithm from Section 4.4.2;
 - (b) by adapting the exact cached solution of a previous mode.

Let us summarize the run-time behavior of the resource manager. Whenever a mode change occurs, the following actions are taken:

1. If a cached (thus optimal) solution for the new mode exists, it is enforced;
2. Otherwise, if a cached solution for a super-mode (i.e. a mode whose task set is a superset of the current active task set) exists, it is adapted to the current mode by ignoring all the tasks that are not active (like we did in the example from Figure 4.2 at time steps 103.5 and 353.6 when for mode o_2 we adapted the optimal solution generated for mode o_1);
3. If none of the above is successful, then the fast heuristic (see Section 4.4.2) is run. If a feasible mapping is found, it is enforced; otherwise, the new task is not allowed to activate and the system remains in the previous mode;
4. Once a decision regarding the new mode has been taken, in parallel with the system's execution the resource manager will run the ILP;

- (a) If the ILP returns a feasible solution, it is cached for future use (e.g. the ILP solution for o_1 is used 3 times in Figure 4.2);
- (b) Otherwise, the mode is marked as infeasible (e.g. mode o_4 in Figure 4.2).

Since running the ILP is a time and energy consuming process, we would like to do this as rarely as possible. We try to achieve this by using caching. For example, let us look at time step 253.5 in Figure 4.2: when τ_{11} arrives the ILP is running, in an attempt to find a feasible solution for mode o_3 (for which the heuristic was unable to obtain a schedulable mapping). If a solution for o_1 would not be cached, at 253.5 the resource manager would need to interrupt the ILP and run the heuristic for the new mode instead. Note also that, the longer the residence time of a mode the more pay-off we get from having an optimal solution prepared for it.

The solutions generated by the fast heuristic are not cached, because we want to store as many exact (ILP) solutions as possible in the limited cache space. Due to the low overhead of running the heuristic, we can afford to do this. Note that, since the cache for storing solutions is limited, a replacement policy is needed (this is discussed in Section 4.5.1). In the following sections we will propose solutions to the challenges identified above.

4.4 Optimization for One Mode

4.4.1 ILP Formulation

We will first formulate an integer linear program to solve the problem of mapping the active tasks in mode o , i.e. T_o , to the available resources in our heterogeneous architecture. In what follows, we will use the following conventions:

- CPUs are indexed from 1 to m ;
- GPUs are indexed from $m + 1$ to $m + q$;
- The FPGA has index $m + q + 1$;
- Recall that the FPGA has r reconfigurable partitions;
- Size of the task set in the current mode is n ;
- The mapping variables are denoted with x_{ij} , where:

$$x_{ij} = \begin{cases} 1 & \text{if task } \tau_i \text{ is mapped to resource } j; \\ 0 & \text{otherwise} \end{cases}$$

Let us assume that the length (residence time duration) of a certain mode o is t_o time units. A particular task $\tau_i \in T_o$ (with period p_i) that is active during mode o would release $\left\lceil \frac{t_o}{p_i} \right\rceil$ jobs³. We are interested in min-

³We assume that jobs are immediately ready at the beginning of a mode.

imizing the total energy consumption of the system during mode o , which is expressed as $\sum_{i=1}^n \sum_{j=1}^{m+q+1} x_{ij} \left\lceil \frac{t_o}{p_i} \right\rceil e_{ij}$. Since we assume that mode durations are significantly longer than task periods (i.e. $t_o \gg p_i, \forall \tau_i \in T_o$), we can approximate the number of job releases for each task with $\left\lceil \frac{t_o}{p_i} \right\rceil \approx \frac{t_o}{p_i}$. As a result, we can factor out the mode duration t_o , obtaining the following ILP formulation:

$$\begin{aligned} \underset{x_{ij}}{\text{minimize}} \quad & \sum_{i=1}^n \sum_{j=1}^{m+q+1} x_{ij} \frac{e_{ij}}{p_i} \\ \text{subject to} \quad & \sum_{i=1}^n x_{ij} u_{ij} \leq 1, \quad j = 1, 2, \dots, m \end{aligned} \quad (4.6)$$

$$\sum_{i=1}^n x_{ij} u_{ij} \leq 1, \quad j = m+1, m+2, \dots, m+q \quad (4.7)$$

$$\text{if } x_{kj} = 1 : \quad \sum_{i=1}^n x_{ij} c_{ij} \leq d_k, \quad j = m+1, m+2, \dots, m+q \quad (4.8)$$

$$\sum_{i=1}^n x_{ij} \leq r, \quad j = m+q+1 \quad (4.9)$$

$$x_{ij} c_{ij} \leq d_i, \quad j = m+q+1 \quad (4.10)$$

$$\sum_{j=1}^{m+q+1} x_{ij} = 1, \quad i = 1, 2, \dots, n \quad (4.11)$$

Constraints (4.6) model the necessary and sufficient conditions for EDF scheduling (if neither one of the CPUs are overloaded, then the tasks meet their deadlines). The next two constraints capture the schedulability conditions for the GPUs. Constraints (4.7) ensure that no GPU is overloaded. Constraints (4.8) correspond to schedulability conditions (4.5) and require further clarifications. As can be seen, the variable x_{kj} controls whether the linear relationship takes effect or not; in other words, if task τ_k is mapped to GPU j , only then we need to check whether the task will meet its deadline under non-preemptive FIFO scheduling on GPU j . One standard way to deal with such constraints is to use the so-called big-M formulation⁴:

$$\sum_{i=1}^n x_{ij} c_{ij} \leq x_{kj} d_k + (1 - x_{kj}) M \quad (4.12)$$

where M is sufficiently large (one option is to set $M = \sum_{\tau_i \in T_o} \max_{j=m+1, \dots, m+q} c_{ij}$).

⁴Note that, already starting with CPLEX 10, indicator constraints have been introduced as an alternative to big-M formulations, in order to avoid numerical instability in cases when M is too large [IBM09].

Constraint (4.9) models the FPGA area restriction; recall that the FPGA is divided into r identical reconfigurable partitions, thus at most r tasks can be mapped to the FPGA at a time. Constraint (4.10) simply states that any FPGA task should finish before its deadline (each one of the r FPGA partitions could be considered a dedicated processing element for the task mapped to it). Finally, constraints (4.11), also referred to as the *coupling constraints*, enforce that each task is mapped to one and only one resource.

Please note that the constraint matrix has block-angular structure; if we delete the coupling constraints we are left with one block for each resource. As a result, Dantzig-Wolfe decomposition can be applied to improve the tractability of large-scale linear programs of this type. The original problem is reformulated into a master program and several subprograms (one for each block in the constraint matrix: in our case, $m + q + 1$, as many resources we have). A solution to a sub-problem represents a column in the master problem, which then enforces that the coupling constraints are satisfied. This particular structure of the constraint matrix makes it possible to solve the problem in reasonable time.

4.4.2 Fast Heuristic

Although the ILP formulation can be solved in reasonable time, it is still too expensive to apply at run-time for every mode change. Thus, we need a heuristic that provides high quality results and has affordable execution times to apply. We decided to extend the polynomial-time algorithm described in Chapter 7.4 from [MT90]. Note that this heuristic solves the generalized assignment problem, and our formulation contains the extra scheduling constraints (4.8) and (4.10).

Algorithm 7 describes our optimization approach. Lines 2-19 perform the initialization of the algorithm. Note that $y_i = \text{map}(\tau_i)$ is the mapping vector, specifying for each task τ_i the index j of the resource where it is mapped; in case no feasible mapping is found for particular tasks, their y_i will be zero. These tasks will not be allowed to become active by the resource manager. Let f_{ij} be a measure of the “desirability” of assigning task τ_i to resource j (the smaller the f_{ij} the better). For the CPUs and the GPUs, we consider $f_{ij} = \frac{e_{ij}}{p_i}, \forall j \in \{1, 2, \dots, m + q\}$ (line 8). However, in the case of the FPGA, reconfiguring a new task incurs a reconfiguration energy overhead. As a result, we would like to discourage task migrations (unless the expected energy reduction is greater than the reconfiguration overhead), or in other words it might be desirable to keep tasks that are already on the FPGA from a previous mode. Thus, for the FPGA we define $f_{ij} = \frac{e_{ij}}{p_i} + \frac{e_i^r}{t}, j = m + q + 1$ if a task is not already on the FPGA from a previous mode (line 14), where t is the predicted mode length. Otherwise, we keep $f_{ij} = \frac{e_{ij}}{p_i}, j = m + q + 1$ (line 12).

We iteratively consider all the tasks that were not yet mapped to any resource (line 20), and determine the task τ_{i^*} that has the maximum difference

Algorithm 7 Mapping heuristic**Input:** $m, q, r, n, p_i, c_{ij}, e_{ij}$ **Output:** $y_i = \text{map}(\tau_i)$

```

1: procedure MAPPING
2:    $M \leftarrow \{1 \dots m + q + 1\}$  ▷ indexes of resources
3:    $U \leftarrow \{1 \dots n\}$  ▷ indexes of active tasks
4:   for  $i = 1 \rightarrow n$  do
5:      $y_i \leftarrow 0$  ▷ no task mapped yet
6:     for  $j = 1 \rightarrow m + q$  do
7:        $u_{ij} \leftarrow \frac{c_{ij}}{p_i}$ 
8:        $f_{ij} \leftarrow \frac{e_{ij}}{p_i}$ 
9:     end for
10:     $j \leftarrow m + q + 1$  ▷ the index for the FPGA
11:     $u_{ij} \leftarrow 1$  ▷ each task occupies 1 partition
12:     $f_{ij} \leftarrow \frac{e_{ij}}{p_i}$ 
13:     $t \leftarrow \text{PRED\_LENGTH}$ 
14:     $f_{ij} \leftarrow f_{ij} + \frac{e_i^r}{t}, \forall \tau_i : \text{map}(\tau_i) \neq m + q + 1$ 
15:  end for
16:  for  $i = 1 \rightarrow m + q$  do
17:     $\bar{K}_i \leftarrow 1$  ▷ capacity of CPUs and GPUs is 1
18:  end for
19:   $\bar{K}_{m+q+1} \leftarrow r$  ▷ FPGA has  $r$  partitions
20:  while  $U \neq \emptyset$  do
21:     $d^* \leftarrow -\infty$ 
22:    for all  $i \in U$  do
23:       $F_i \leftarrow \{j \in M \mid u_{ij} \leq \bar{K}_j\}$ 
24:      if  $F_i \neq \emptyset$  then
25:         $j^* \leftarrow \text{argmin}\{f_{ij} \mid j \in F_i\}$ 
26:        if  $F_i \setminus \{j^*\} = \emptyset$  then  $d \leftarrow +\infty$ 
27:      else
28:         $j' \leftarrow \text{argmin}\{f_{ij} \mid j \in F_i \setminus \{j^*\}\}$ 
29:         $d \leftarrow f_{ij'} - f_{ij^*}$ 
30:        if  $d > d^*$  then
31:           $d^* \leftarrow d$ 
32:           $i^* \leftarrow i$ 
33:           $F_{i^*} \leftarrow F_i$ 
34:        end if
35:      end if
36:    end if
37:  end for
38:  while  $y_{i^*} == 0$  do
39:    if  $\text{SCHEDULABLE}(i^*, j^*)$  then
    (continues on next page)

```

 (continued from previous page)

```

40:          $y_{i^*} \leftarrow j^*$ 
41:          $\bar{K}_{j^*} \leftarrow \bar{K}_{j^*} - u_{i^*j^*}$ 
42:          $U \leftarrow U \setminus \{i^*\}$ 
43:     else ▷  $\tau_{i^*}$  can't be scheduled on resource  $j^*$ 
44:          $F_{i^*} \leftarrow F_{i^*} \setminus \{j^*\}$ 
45:         if  $F_{i^*} == \emptyset$  then ▷ no more resources
46:             break
47:         else ▷ pick next best resource
48:              $j^* \leftarrow \operatorname{argmin}\{f_{i^*j} | j \in F_{i^*}\}$ 
49:         end if
50:     end if
51: end while
52: end while
53: end procedure
  
```

between the smallest and the second smallest $f_{i^*j}, j = 1, 2, \dots, m+q+1$ (lines 22-37). Intuitively, we want to find that task which, if it is not mapped on its most desirable resource, will generate the biggest penalty in terms of energy consumption. Once task τ_{i^*} was identified, we need to decide where to map it (considering the schedulability constraints). We try to map the task to resource j^* for which $f_{i^*j^*}$ is minimum (lines 39-42). If the scheduling constraint for resource j^* is violated (considering the tasks mapped to it so far), we try to map τ_{i^*} to the resource j^* for which $f_{i^*j^*}$ is the next smallest (lines 43-50). We continue until we manage to map τ_{i^*} to a resource (fulfilling the scheduling constraints), or until all resources have been considered and no feasible mapping for τ_{i^*} has been found (line 46). In the latter case task τ_{i^*} will not be allowed to become active (i.e. release jobs). Thus, for certain modes, the resource manager will decide not to admit certain new tasks in order to guarantee the timing constraints of all the other tasks. In Section 4.6 we will evaluate how well our heuristic performs from this point of view, i.e. how many tasks are rejected compared to the ideal case.

The function SCHEDULABLE presented in Algorithm 8 checks the schedulability conditions. In particular, given a certain partial mapping specified by $\text{map}(\tau_k)$, the function tries to see if it is possible to map task τ_i to resource j without violating any of the constraints. There are three cases:

1. If resource j is a CPU, then we only need to check the necessary and sufficient condition specified by inequation (4.3), which ensures that the processor is not overloaded. This is done in line 3.
2. If resource j is a GPU, then we first need to check that the GPU is not overloaded (line 9), corresponding to inequation (4.4). On top of this, we need to make sure that by mapping task τ_i to GPU j , the deadlines of all the tasks mapped to the same GPU are still met (lines

Algorithm 8 Schedulability check

Input: $m, q, r, n, p_i, c_{ij}, u_{ij}$

```

1: function SCHEDULABLE( $i, j$ )
2:   if  $j \in \{1 \dots m\}$  then ▷ CPU
3:     if  $\sum_{map(\tau_k)=j} u_{kj} + u_{ij} \leq 1$  then
4:       return true
5:     else
6:       return false
7:     end if
8:   else if  $j \in \{m + 1 \dots m + q\}$  then ▷ GPU
9:     if  $\sum_{map(\tau_k)=j} u_{kj} + u_{ij} \leq 1$  then
10:      for all  $k \in \{l | map(\tau_l) = j\}$  do
11:        if  $\sum_{map(\tau_h)=j} c_{hj} + c_{ij} > p_k$  then
12:          return false
13:        end if
14:      end for
15:      return true
16:     else
17:       return false
18:     end if
19:   else ▷ FPGA
20:     if  $\sum_{map(\tau_k)=j} u_{kj} + u_{ij} \leq r$  and  $c_{ij} \leq p_i$  then
21:       return true
22:     else
23:       return false
24:     end if
25:   end if
26: end function

```

10-15), corresponding to inequation (4.5).

3. If resource j is the FPGA, then we need to ensure that the FPGA area is sufficient, as specified by inequation (4.1), i.e. we do not map more tasks on the FPGA than the r available reconfigurable partitions (note that we consider the utilization of an FPGA task to be 1, since it occupies one reconfigurable partition — see line 11 in Algorithm 7). We also need to check that τ_i will finish before its deadline if mapped to the FPGA, corresponding to inequation (4.2). If both conditions are satisfied, the task is schedulable on the FPGA (lines 20-21).

4.5 Multi-Mode Optimization

Section 4.4 described how to decide the mapping of tasks to resources in a given mode such that the energy consumption of the system is minimized

and all timing constraints are satisfied. However, our ultimate goal is to minimize the energy consumption of the system during its lifetime. Next we present the run-time strategy to achieve our goal.

4.5.1 Caching of Solutions

Since finding the solution for a particular mode is a time- and energy-consuming optimization, we try to minimize the number of times the mapping algorithm is run. As a result, we decided to apply caching of solutions. Every time the solution for a particular mode is obtained, it is saved in a cache. If the same mode⁵ is visited again later, a mapping solution might reside in the cache and the resource manager could immediately use it. When the cache is full, we use the least frequently used replacement policy to choose the solution that will be replaced.

In case no cached solution for the current mode exists, we return the solution (if present in the cache) for the most recently used super-mode (whose task set is a superset of the current active task set). If the difference between the size of this superset and the size of the current mode's task set is smaller than a threshold⁶ tr , then we adapt the solution to the current mode by ignoring the tasks that are not anymore active.

4.5.2 Estimating the Mode Duration

As we discussed previously, it is desirable to estimate how long a certain mode will be resident (referred to as the *residence time*). We use this information to compute the desirability factors f_{ij} for the heuristic (see line 14 in Algorithm 7).

We use timestamps to estimate the residence time t_o of a mode o at run-time. We assume that the residence time of a mode is similar to its residence times in the recent past. Let t_o denote the current measured residence time of mode o . We update the estimate $\tilde{t}_o$ for the average residence time over the past mode occurrences using an exponential smoothing formula in order to emphasize the recent history:

$$\tilde{t}_o = \alpha t_o + (1 - \alpha)\tilde{t}_o \quad (4.13)$$

This is the value returned by the function `PRED_LENGTH` on line 13 from Algorithm 7.

4.5.3 Run-Time Policies

We describe next two possible run-time policies for the resource manager.

⁵Recall that a mode is defined by the active task set.

⁶We obtained good experimental results for $tr \in [3, 5]$.

4.5.3.1 Heuristic + ILP

In case it is feasible to run the ILP on the platform assumed⁷, then the following procedure can be applied. At the beginning of every new mode, the cache is checked. If a solution for the current mode is found, then it is applied. Otherwise, if there exists in the cache a solution for a super-mode (whose task set size differs from the current mode's task set size within a threshold tr), then that solution is adapted to the current mode (as explained in Section 4.5.1). If no solution is found in the cache, the fast mapping heuristic is applied first. If a feasible solution is obtained, the mode starts running as soon as the mapping is enforced (during the set-up time). Otherwise, it means that some tasks could not be mapped such that all the deadlines are met; those tasks will not receive permission to activate from the resource manager. Once the heuristic is done, the manager will start running the ILP. The exact solution, if found, will be cached and used later, the next time the mode activates. If the ILP fails to find a solution it means that the mode is not feasible and it will be marked as such. In case a new mode arrives before the ILP has finished and we need to run the mapping optimization for the new mode, the execution of the ILP is interrupted.

4.5.3.2 Heuristic-Only

If it is impossible to run the ILP on-the-fly, then we propose to always use the heuristic to decide the mapping of tasks to resources. Of course, caching can still be applied to save the energy consumption overhead of re-running the heuristic. As we will show in the experimental evaluation, good results are still obtained and the number of rejected tasks stays within reasonable limits.

4.6 Experimental Evaluation

We performed simulation experiments in order to evaluate how our policies for on-line resource management behave. First of all, we performed measurements on real platforms (with an actual CPU, GPU, and FPGA). Based on the results obtained for execution time and energy consumption, we generated experimental settings whose parameters had similar ranges.

4.6.1 Real-Life Measurements

We selected the Samsung Exynos Dual Arndale platform (popular in the industry) running at 5 volts. The board contains two Cortex A15 CPU cores running at 1.7 GHz and a Mali T604 GPU with 4 cores running at 533 MHz. We selected 6 applications: convolution, Rijndael (AES), pattern matching,

⁷Note that it is not necessary to run the ILP on the platform itself. Since the ILP is run only seldom, one might imagine that this optimization happens in the cloud.

Table 4.2: Board measurements – CPU and GPU

Application	GPU (ms)	CPU (ms)	Time impr.	GPU (J)	CPU (J)	Enrg. impr.
Convolution	797.2	13743	17	1.77	19.43	11
AES	114.8	345.13	3	0.29	0.52	1.7
Pattern matching	14.04	62.9	4.5	0.04	0.08	2
Histogram	7086.8	25671	3.6	17.2	34.5	2
Bit count	3629.2	28516	7.9	5.1	35.6	7
Genetic program.	1336.7	9154.1	6.8	1.06	10.4	9.7

a simple histogram program, a bit counting one, and genetic programming⁸. For the convolution kernel we used as input an image of 30 MB, for AES a text file of 3 MB, and for pattern matching a text of 10 MB; the histogram and the bit counting programs were repeated in a loop 100 times, and for the genetic programming application we ran 100 loop iterations.

We have used a shunt resistor ($R = 100\text{m}\Omega$) in series with the voltage source of the board in order to measure the average current drawn by the Arndale board when running each application, as well as when being idle. Using these values we computed the energy consumption. We performed the measurements both for the GPU implementations and for the CPU ones and the results are shown in Table 4.2. Note that the GPU measurements include the transfer of input data, the kernel computation and reading the results. The speedup varies between $3\times$ to $17\times$, while the energy reduction varies between $2\times$ to $11\times$. We have used these ranges when we generated task parameters for our simulations.

Since the Arndale board consists of CPU and GPU cores but it has no FPGA, we performed the FPGA measurements on another board (the one used in Section 3.3.5.1), namely the Xilinx ML605, featuring an XC6VLX240T Virtex6 FPGA, with supply voltage 12 volts. The execution times and energy measurements for the SUSAN image processing application [SB97] (corner and edge detection), implemented as hardware modules on the FPGA, were already presented in Section 3.3.5.1, but we repeat them here. For the software versions we measured the execution times and energy consumed by the ARM CPU. The results obtained for an input image of 128 KB are presented in Table 4.3.

Note that we have also measured the time and energy overheads to reconfigure the modules on the FPGA. The reconfiguration of either one of the two modules took 3.07 ms, the size of the bitstreams being 1.12 MB, and the effective throughput of our custom reconfiguration controller with

⁸Note that we selected a broad spectrum of applications, some more suitable for GPU implementation than others, because we wanted to compare the GPU/CPU performance and energy consumption tradeoffs for different real-life applications.

Table 4.3: Board measurements – FPGA

Application	FPGA (ms)	CPU (ms)	Time impr.	FPGA (J)	CPU (J)	Enrg impr.
Corner Detection	2.49	552	221	88.5	179.1	2.02
Edge Detection	2.54	974	383	91.1	298.7	3.27

DMA reaching 375 MB/s. Given these values, we tried to generate the task parameters to resemble them in our simulation experiments.

4.6.2 Simulation Results

In order to test our optimization approaches we have generated test scenarios using the following methodology: first of all, we generated 100 different task sets, each task set representing one “simulation universe”. The parameters for each task were generated randomly: periods $p_i \in [10, 100]$, WCETs for the CPUs $c_{ij}, \forall \text{CPU } j$ were generated such that the utilization was in the interval $u_i \in [0.05, 0.4]$, the energy consumptions for the CPUs $e_{ij}, \forall \text{CPU } j$ were generated by choosing values such that the ratio e_{ij}/c_{ij} would randomly correspond to one of the applications measured in Section 4.6.1. Then we generated the WCETs for the GPUs and for the FPGA (as well as the reconfiguration overheads) to reflect our measurements from Section 4.6.1.

Once the task set for each “universe” was decided, we proceeded to model the multi-mode behavior of our applications. We first chose the active task set for the initial mode by selecting tasks from the 100 in the “universe” until the utilization of the active task set was ca. $75\%(m+q)$, where m and q represent the platform’s CPUs and GPUs respectively (we also made sure that this initial mode is schedulable). Then we generated transitions to new modes by randomly removing or adding a task to the active task set. For each new mode, we also generated a probability to visit that mode from the current mode and a probability to come back from the newly generated mode to the current one. We continued this process until 30 modes were generated; what resulted was a graph of modes, that we interpreted as a Markov model (i.e. the probability to transition to a new mode depended only on the current mode). For each mode o we associated an average residence time $t_o \in [500, 5000]$, and during simulations we drew concrete residence times from the normal distribution with mean $\mu = t_o$ and standard deviation $\sigma = 10$.

We have then simulated the system, on a PC with CPU frequency 2.83 GHz, 8 GB of RAM, and running Windows Vista. Each scenario was simulated for 200 mode changes considering an architecture composed of 4 CPUs, 1 GPU and 1 FPGA with 5 reconfigurable partitions. For deciding the mapping and performing task admission management we ran both approaches

Figure 4.3: Experimental evaluation for the simulation experiments

described in Section 4.5.3, i.e. heuristic+ILP (denoted with H+), and the heuristic-only policy (denoted with H). The cache size for our algorithms was considered big enough to hold at most 10 solutions at a time; the cache replacement policy was least frequently used. As a baseline, we have used the concept of a *golden run* (denoted with G): for every mode encountered during the simulation, we considered that we had available in the cache the optimal mapping. This scenario is not achievable in practice, but it was useful to evaluate the quality of our approaches, both in terms of energy consumption and in terms of number of tasks that were rejected (although they could have been accommodated with an optimal mapping) by our algorithms.

4.6.2.1 Energy Consumption

Let us denote with E_G , E_{H+} and E_H the total energy consumption recorded during the simulation for the golden run, heuristic+ILP and heuristic-only approach respectively (see Section 4.5.3). Note that E_{H+} and E_H include the energy overheads for running our optimization algorithms, while the golden run does not include any such overheads. The values were estimated using a typical average energy consumption of 0.13J/100ms of computation, as it resulted from our measurements on the Cortex A15 CPU of the Arndale board (see Section 4.6.1). Another important observation is that we did not activate in the golden run the tasks that were not accepted by the heuristics. This was done because we wanted a fair comparison: the reported energy consumption corresponds to the same tasks, but with different mappings⁹. Of course, in all cases, the time and energy overheads of reconfiguring the FPGA were considered.

For each of the 100 scenarios simulated we normalized E_{H+} and E_H relative to E_G and then we averaged the results over all the scenarios. The

⁹The problem of rejecting tasks by our approaches, although the golden run could accommodate them, is discussed in Section 4.6.2.2.

results are presented in Figure 4.3a. As can be seen, H+ manages to get results that are 23.5% away from the golden run, while H is 31.4% away. It is important to note that the golden run is not achievable in practice, since it is impossible to always have the optimal solution prepared for every mode, without any time or energy overheads. If it is feasible to run the ILP on the platform, our H+ method can be used to get results close to the golden run. The energy consumed by running the ILP is amortized to some extent by using caching. Our H method obtains good results too, but the number of rejected tasks is higher compared to H+ (see Section 4.6.2.2). It is interesting to note that the energy consumption of the H+ method (which includes the energy overheads of running the ILP), is lower than the energy consumption of H. The optimal solutions found with the ILP reduce the energy consumption of the tasks and this more than compensates for the consumption of the ILP.

4.6.2.2 Rejection Rate

Let us denote with R_G , R_{H+} and R_H the rejection rates (i.e. the percentage of tasks that were rejected out of the total tasks that wanted to become active during the simulation) for the golden run, heuristic+ILP and heuristic-only approach respectively. The rejects of the golden run correspond to unschedulable modes. Thus, we subtracted R_G from R_{H+} and R_H in order to see what percentage of tasks were rejected by our heuristics although they could have been accommodated by an optimal mapping (false negatives). The results averaged over the 100 scenarios simulated are presented in Figure 4.3b. We notice that H+ rejects on average 9% of the incoming tasks; these rejections originate from those cases when the ILP solution for a mode is not cached, and the fast heuristic cannot find a feasible mapping to accommodate a new task. For H the average number of rejections is higher (i.e. 22%), because for modes with high utilization the fast heuristic cannot find solutions and we never run the ILP.

4.7 Summary

This chapter proposed a framework for the optimization of multi-mode real-time systems implemented on heterogeneous platforms with CPUs, GPUs and FPGAs. We evaluated a resource manager that implements run-time policies in order to decide on-the-fly task admission and the mapping of active tasks to resources, such that the energy consumption is minimized while all the task deadlines are met. Our policies combine exact solutions obtained from an ILP and heuristic solutions obtained from an efficient mapping algorithm, together with solution caching strategies and mode length estimation in order to get the best results.

HARDWARE/SOFTWARE OPTIMIZATION OF ERROR DETECTION IMPLEMENTATION

SAFETY-CRITICAL systems need to function correctly even in the presence of faults. To provide resiliency against transient faults, efficient error detection and recovery techniques have to be employed. Unfortunately, these mechanisms incur high penalties, either from the cost perspective, or from the performance point of view. Since both cost and performance are important issues for today's embedded systems, this chapter presents system-level approaches to optimize the hardware/software implementation of error detection in the context of fault-tolerant real-time embedded systems.

The remainder of this chapter is organized as follows. We start by presenting in Section 5.1 the preliminaries related to the error detection techniques assumed in our work. Section 5.2 presents two approaches to minimize the global worst-case schedule length of a distributed application, while meeting the imposed hardware cost constraints and tolerating multiple transient faults. Both statically reconfigurable FPGAs and partially dynamically reconfigurable FPGAs are considered. Section 5.3 discusses an approach to minimize the average execution time of an application by optimizing the hardware/software implementation of error detection. Finally, Section 5.4 summarizes the contributions of this chapter.

5.1 Preliminaries: Error Detection Technique

We next present the application-aware error detection technique that we use in this chapter. The idea of this technique ([PKI11], [LCP⁺09]) is to identify, based on specific metrics [PKI05], *critical variables* in a program. A critical variable is defined as “a program variable exhibiting high sensitivity to random data errors” [PKI11]. The backward program slice for each acyclic control path is extracted for the identified critical variables. The backward program slice is defined as “the set of all program statements/instructions

Figure 5.1: Code fragment with error detectors

that can affect the value of the variable at a program location” [PKI11]. Each slice is optimized at compile time, resulting in a series of checking expressions. Since the backward program slices run across basic blocks, the optimizations performed cannot be done by regular compilers. The slices do not contain any conditional statements (they consist of straight-line code); thus, aggressive optimization is possible. The checking expressions are inserted in the original code before the use of a critical variable. Finally, the original program is instrumented with instructions to keep track of the control paths followed at run-time and with checking instructions to choose the corresponding checking expression and, then, compare the results.

We present an example of how this error detection technique works. We use the C program in Figure 5.1 (adapted from [PKI11]). The original program code is presented on the left (no shading), the checking code added is presented on the right (light shading) and the path tracking instrumentation is shown with dark shading. Assuming that x is a critical variable, it needs to be checked before its use.

We identify two paths in the program slice of x , corresponding to the two branches. To compute the corresponding backward program slices, we start with the instruction computing the critical variable (x) and traverse the program backward, placing in each slice the instructions that can affect the value of x . The resulting slices are:

- for the first path (if $s = 0$):

$y = \text{read_int}();$

```

p = y;
r = -y;
s = y mod 3;
t = s*y;
u = p - r;
v = y + t;
x = s*y + u - v;

```

- for the second path (if $s \neq 0$):

```

y = read_int();
p = y;
r = -y;
s = y mod 3;
t = s*y;
u = 2*p + r;
v = y - t;
x = s*y + u - v;

```

The instructions on each path are optimized, resulting in a concise expression that checks the correctness of the variable's value. For the first path the expression is $x' = y'$, and for the second one, it is $x' = 2 * s' * y'$ (values are assigned to the temporary variable x'). Variables y' and s' are copies of the corresponding variables from the original program. Although the backward program slices run across several basic blocks, aggressive compile time optimization is possible, because the slices are specialized for each acyclic control path (do not contain conditional statements). At run-time, when control reaches a point that uses x , one of the checking expressions is chosen based on the *path* variable (updated via the instrumentation code added). The value of x (computed by the original program) is compared with the value of x' (recomputed by the checking expression). In case of a mismatch, an error flag is raised and a recovery action should be taken.

This technique has two sources of performance overhead: path tracking and variable checking. Both can be implemented either in software, potentially incurring high performance overheads, or in hardware, which leads to costs sometimes exceeding the available resources.

Pattabiraman et al. [PKI11] have proposed a software-only approach, where both path tracking and variable checking are implemented in software and executed together with the application. The path tracking alone incurs a time overhead of up to 400%, while the overhead of variable checking is up to 80% [PKI11]. Complete hardware implementations are proposed in [PKI11] and [LCP⁺09]. Between the extreme solutions of implementing all error detection in software, on the one side, and performing it in hardware,

on the other side, there is a wide range of alternatives characterized by the implementation decision taken for each task in the application. This decision depends on various factors: time criticality, amount and cost of available hardware resources and their nature (FPGAs with static or partial dynamic reconfiguration). We focus on efficiently implementing error detection in the context mentioned above.

The error detection technique described above detects any transient errors that corrupt the architectural state, provided that they corrupt one or more variables in the backward slice of a critical variable. To achieve maximal error coverage, we assume that some complementary, generic error detection techniques (like a watchdog processor and/or error-correcting codes) are used in conjunction with the application-aware one. Hardware redundancy techniques might also be used to deal with the remaining, not covered, faults [BMR⁺08]. We concentrate on the optimization of the application-aware error detection component.

5.2 Real-Time Distributed Embedded Systems

In this section we focus on the hardware/software implementation of the application-aware error detection components for a safety-critical application implemented on a distributed embedded platform. We present approaches to the optimization of error detection implementation in a system-level design context, minimizing the global worst-case schedule length, while meeting the imposed hardware cost constraints.

Our fault model assumes that a maximum number k of transient faults can affect the system during one period. To provide resiliency against these faults re-execution is used. Once a fault is detected, the initial state is restored and the task is re-executed. We will use the scheduling technique presented in [IPEP06], which considers error detection as a black box. The authors proposed to generate fault-tolerant schedules for hard real-time systems such that multiple transient faults are tolerated. The algorithm produces, as output, schedule tables that capture alternative execution scenarios corresponding to possible fault occurrences. Among all fault scenarios, one corresponds to the worst-case schedule length (WCSL). We are interested in minimizing this WCSL by accelerating error detection in reconfigurable hardware, so that we meet the time and cost constraints imposed.

5.2.1 Optimization Framework

Figure 5.2 illustrates our framework. The applications, available as C code, are represented as a set of task graphs. The code is processed through the error detection instrumentation framework [LCP⁺09], which outputs the initial code with the embedded error detectors, as well as VHDL code needed to synthesize error detector modules on FPGA.

Figure 5.2: Optimization framework overview

Figure 5.3: System model for fault-tolerant distributed embedded systems

This information, together with the system architecture and the mapping of tasks to computation nodes, is used by the optimization tool to find a close to optimal error detection implementation (EDI). The cost function is the WCSL generated by the fault-tolerant schedule synthesis tool [IPEP06].

5.2.2 System Model

We consider a set of real-time applications $\mathcal{A}_i$, modeled as directed acyclic graphs $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i)$, executed with period T_i . The graphs $\mathcal{G}_i$ are merged into a single graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, having the period T equal with the least common multiple of all T_i . Each vertex $P_j \in \mathcal{V}$ represents a task, and each edge $e_{jk} \in \mathcal{E}$ indicates that P_j 's output is P_k 's input. Tasks are non-preemptable and all data dependencies have to be satisfied before a task can start executing.

We assume a distributed architecture composed of computation nodes, connected to a bus (see Figure 5.3b). The task mapping to these nodes is given (illustrated with shading in Figure 5.3a). The bus is assumed to be fault-tolerant (we use a protocol such as TTP [KB03]). Each node consists of a central processing unit, a communication controller, a memory subsystem, and a reconfigurable device (FPGA). Since SRAM-based FPGAs are susceptible to single event upsets [WJR⁺03], we assume that suitable mitigation techniques are employed (e.g. [LCR03], [NOS14]) in order to provide

sufficient reliability of the hardware used for error detection.

For each task we consider three alternative error detection implementations (EDIs):

1. SW-only, when the checking code (light shading in Figure 5.1) and the path tracking instrumentation (dark shading in Figure 5.1) are implemented in software and interleaved with application code.
2. mixed-HW/SW, when the path tracking is moved to hardware and done concurrently with the application's execution, while the checking expressions remain in software, interleaved with the initial code. This is a reasonable refinement, since the path tracking's overhead is significant.
3. HW-only, which further reduces the time overhead by also moving the execution of the checking expressions to hardware.

For each task P_i and each of the three possible EDIs, we know the worst-case execution time $WCET_i^\square$ [WEE⁺08], the corresponding HW area $h_i^\square$, and reconfiguration time $\rho_i^\square$ needed to implement error detection ($\square \in \{\text{SW-only, mixed-HW/SW, HW-only}\}$).

For all messages sent over the bus, their worst-case transmission time $WCTT_{ij}$ is given. This transmission is modeled as a communication task inserted on the edge $P_i \rightarrow P_j$ (see m_{24} in Figure 5.3a). For tasks mapped on the same node, the communication time is considered to be part of the task's WCET and is not modeled explicitly.

5.2.3 Problem Formulation

Input

- A set of applications $\mathcal{A}_i$ modeled together as a task graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ (see Section 5.2.2).
- The WCETs for each alternative EDI of every task $P_i \in \mathcal{V}$, given by $W : \mathcal{V} \times \mathcal{H} \rightarrow \mathbb{Z}^+$, where $\mathcal{H} = \{\text{SW-only, mixed-HW/SW, HW-only}\}$.
- The hardware costs for each alternative EDI of every task $P_i \in \mathcal{V}$, given by $C : \mathcal{V} \times \mathcal{H} \rightarrow \mathbb{Z}^+ \times \mathbb{Z}^+$, i.e. we know the size ($h_i = \text{rows}_i \times \text{columns}_i$) of the rectangle needed on FPGA¹.
- The reconfiguration time (ρ_i) for each alternative EDI of every task $P_i \in \mathcal{V}$, given by $R : \mathcal{V} \times \mathcal{H} \rightarrow \mathbb{Z}^+$.
- The worst-case transmission time ($WCTT_{ij}$) for each message m_{ij} , from P_i to P_j .

¹Function C covers both 1D and 2D reconfiguration scenarios. Note also that, being more generic, these assumptions cover the ones used in the previous chapters.

Table 5.1: Worst-case execution times and error detection overheads for the motivational example

P_i	$WCET_i^U$	Error Detection Implementation (EDI)								
		SW-only			mixed-HW/SW			HW-only		
		$WCET_i$	h_i	ρ_i	$WCET_i$	h_i	ρ_i	$WCET_i$	h_i	ρ_i
P_1	60	240	0	0	100	15	20	80	40	45
P_2	50	140	0	0	80	15	20	60	40	45
P_3	40	150	0	0	60	10	15	50	30	35
P_4	30	100	0	0	60	15	20	40	40	45

- The parameter k , denoting the number of transient faults to be tolerated during one period, T . This k is used for the generation of fault-tolerant schedules with the synthesis tool presented in [IPEP06].
- The set of computation nodes $\mathcal{N}$ connected by a bus $\mathcal{B}$ on which the application is implemented. For each $N_j \in \mathcal{N}$, the available hardware area ($HW_j = rows_j \times columns_j$) which we can use to implement the error detection is known.
- The mapping of tasks to computation nodes, given by $M : \mathcal{V} \rightarrow \mathcal{N}$.

Output

- An EDI assignment $S : \mathcal{V} \rightarrow \mathcal{H}$, such that k transient faults are tolerated and the worst-case schedule length (WCSL) is minimal, while the hardware cost constraints are met.

5.2.4 Motivational Examples

Figure 5.3a presents an application with four tasks, $P_1 - P_4$, mapped on an architecture with two computation nodes, N_1 and N_2 , connected by a bus (Figure 5.3b). Tasks P_1 and P_2 are mapped on N_1 , while P_3 and P_4 are mapped on N_2 . The WCET of tasks for each of the three alternative EDIs are listed in Table 5.1. The WCET of the un-instrumented task ($WCET_i^U$) is also given in Table 5.1, only to emphasize the error detection time overheads. For example, the error detection overhead incurred by the SW-only EDI for P_1 is $240 - 60 = 180$ time units. The HW area (h_i) and the re-configuration times (ρ_i) incurred by each EDI², are also presented in Table 5.1. The WCTT of messages is 20 time units. The recovery overhead for all tasks is 10 time units. Our application has to tolerate $k = 1$ fault within its period.

²For simplicity we consider the 1D case here, but note that the model supports 2D placement and FPGA reconfiguration.

(a) Motivational example 1

(b) Motivational example 2

Figure 5.4: Motivational examples for the optimization of error detection implementation

In the Gantt charts we represent task execution with white boxes, recovery overheads with dark shading and FPGA reconfiguration overheads with a checkerboard pattern.

The Gantt chart in Figure 5.4a-(i) shows the worst-case scenario for the SW-only solution. We implement error detection in software for all tasks

Figure 5.5: Architecture and application for motivational example 2

and then we generate fault-tolerant schedules with the tool from [IPEP06]. We obtain $WCSL = 750$ time units (the scenario when P_1 experiences a fault), but we do not use any additional hardware. Considering that we have unlimited reconfigurable hardware, the shortest possible $WCSL$ corresponds to the HW-only solution (Figure 5.4a-(v)): 290 time units. Since we use HW-only EDI for all tasks, we need at least $HW_1 = 80$ plus $HW_2 = 70$ area units for $FPGA_1$ and $FPGA_2$, respectively. These are the extreme cases corresponding to the longest $WCSL$, but no additional hardware cost, and to the shortest $WCSL$, with maximal hardware cost, respectively. We will try to obtain the minimal $WCSL$, subject to the available hardware area.

Considering that we have statically reconfigurable FPGAs of size 20 on each node, we can only place into hardware the mixed-HW/SW error detection module for one task/node. A naive approach that places the error detection module corresponding to the longest task on FPGA, is shown in Figure 5.4a-(ii). By placing error detection for P_1 and P_3 into hardware, we reduced the $WCET$ from 240 to 100 for P_1 and from 150 to 60 for P_3 and, thus, obtained $WCSL = 510$ time units. Nevertheless, for N_2 it is actually better to place the error detection for task P_4 into HW (Figure 5.4a-(iii)). Although we only shorten its $WCET$ with 40 time units (compared to 90 for P_3), we finally obtain a shorter $WCSL = 470$ time units. Thus, we improved the SW-only solution by 37%. Because of the slack following P_3 , shortening its $WCET$ does not impact the end-to-end delay, and the FPGA can be used more efficiently with P_4 .

Let us now assume that the FPGAs of size 20 we use have partial dynamic reconfiguration (PDR) capabilities (i.e. parts of the device may be reconfigured at run-time, while other parts remain functional). Figure 5.4a-(iv) illustrates the shortest $WCSL$ we could obtain in this case: 390 time units. After P_1 finishes execution, its mixed-HW/SW EDI is replaced, using PDR, with the mixed-HW/SW EDI for P_2 , reusing the FPGA area. Comparing the minimal $WCSL$ from Figure 5.4a-(iv) (i.e. 390 time units) with the minimal $WCSL$ obtained using static FPGAs of the same size (i.e. 470 time units), we see that by exploiting PDR it is possible to shorten the $WCSL$ with an extra 17%. Remember that we used FPGAs a quarter of the maximum size needed to implement the HW-only solution.

Let us now consider the application from Figure 5.5a mapped on one computation node (Figure 5.5b). Figure 5.4b-(i) presents the Gantt chart for the SW-only solution, with $WCSL = 880$ time units. Assuming an FPGA of size 50, without PDR capabilities, the shortest worst-case schedule obtained is illustrated in Figure 5.4b-(ii) (450 time units). This results by assigning mixed-HW/SW EDI for $P_1 - P_3$. We next consider an FPGA of 25 area units, but having PDR capabilities (Figure 5.4b-(iii)). We initially place the mixed-HW/SW implementations for P_1 and P_3 on the FPGA. As soon as P_1 finishes, we reuse the FPGA area corresponding to its detector module and reconfigure in advance the mixed-HW/SW EDI for P_2 . This is done in parallel with P_3 's execution and, thus, P_2 is scheduled immediately after P_3 . Unfortunately, for P_4 we cannot reconfigure in parallel with P_2 's execution, since we only have 10 area units available. We have to wait until P_2 ends, then reconfigure the FPGA with P_4 's mixed-HW/SW detector module and after that schedule P_4 . Although P_4 's reconfiguration could not be masked, this solution is preferred compared to the SW-only alternative for P_4 , because we gain 20 time units $= WCET_4^{SW-only} - (\rho_4^{mixed-HW/SW} + WCET_4^{mixed-HW/SW})$. Comparing Figure 5.4b-(iii) ($WCSL = 430$ time units) with Figure 5.4b-(ii) ($WCSL = 450$ time units), we see that, by exploiting PDR capabilities, we get even better performance than using static FPGAs double the size. The improvement relative to the SW-only solution (Figure 5.4b-(i): $WCSL = 880$ time units) is 51%.

5.2.5 Static FPGA Reconfiguration

In this section we are going to address the case in which FPGA reconfiguration is done statically. This means that, for a certain application, the modules that are placed on the FPGA before start-up cannot be exchanged with other modules during run-time. Looking at Figure 5.4a-(ii) and 5.4a-(iii), for example, we can see that no reconfiguration of the FPGA is performed at run-time. By contrast, in Figure 5.4a-(iv), P_2 is reconfigured on the FPGA after P_1 finishes executing. This is done by means of partial dynamic reconfiguration (which is addressed later, in Section 5.2.6).

The problem defined in Section 5.2.3, assuming static FPGA reconfiguration, is a combined mapping and scheduling problem, which is NP-complete [GJ90]. Thus, for big problem sizes finding an exact solution is unfeasible. The solution proposed here is a Tabu Search heuristic [RB93].

As illustrated by the motivational example, depending on the resource constraints, different EDIs represent area-latency trade-off points for a task. Algorithm 9 presents the pseudocode for our EDI assignment optimization algorithm. The exploration of the solution space starts from a random initial solution (line 2). Next, successive moves are performed based on a neighborhood search. The transition from one solution to another is the result of the selection (line 6) and application (line 8) of an appropriate move. At each iteration, in order to evaluate our cost function ($WCSL$), tasks

Algorithm 9 EDI optimization algorithm**Input:** $\mathcal{A} = \mathcal{G}(\mathcal{V}, \mathcal{E}), W, C, WCTT_{ij}, k, HW_j, M$ **Output:** $best_Sol$ = best solution found by Tabu Search

```

1: procedure EDI_OPTIMIZATION
2:    $best\_Sol = current\_Sol \leftarrow \text{Random\_Initial\_Solution}();$ 
3:    $best\_WCSL = current\_WCSL \leftarrow \text{WCSL}(current\_Sol);$ 
4:    $Tabu \leftarrow \emptyset;$ 
5:   while  $iteration\_count < max\_iterations$  do
6:      $best\_Move \leftarrow \text{SELECT\_BEST\_MOVE}(current\_Sol,$ 
                                      $current\_WCSL);$ 
7:      $Tabu \leftarrow Tabu \cup \{\text{reverse}(best\_Move)\};$ 
8:      $current\_Sol \leftarrow \text{Apply}(best\_Move, current\_Sol);$ 
9:      $current\_WCSL \leftarrow \text{WCSL}(current\_Sol);$ 
10:     $\text{Update}(best\_Sol);$ 
11:    if  $no\_improvement\_count > diversification\_count$  then
12:       $\text{Restart\_Diversification}();$ 
13:    end if
14:  end while
15:  return  $best\_Sol;$ 
16: end procedure

```

and messages are scheduled using the fault-tolerant scheduling technique (function WCSL in Algorithm 9) presented in [IPEP06]. To assure the proper breadth of the search process, diversification is employed (lines 11-12). The whole search is based on a recency memory (*Tabu* list) and a frequency memory (*Wait* counters).

5.2.5.1 Tabu Search Moves

We have considered two types of moves: *simple* ones and *swaps*. A simple move applied to a task P_i is defined as the transition from one error detection implementation to any of the adjacent ones from the ordered set $H = \{\text{SW-only, mixed-HW/SW, HW-only}\}$. Intuitively, EDI of a task is moved more into hardware (for example, from the SW-only alternative to the mixed-HW/SW EDI, or from the mixed-HW/SW to the HW-only EDI), or more into software (for example from the HW-only to the mixed-HW/SW EDI, or from the mixed-HW/SW to the SW-only EDI), but direct transitions between SW-only and HW-only EDIs are not allowed. The motivation behind restricting transitions only to adjacent EDIs was to limit the size of our neighborhood (defined as the set of solutions that can be directly reached from the current one, by applying a single move). A swap consists of two “opposite” simple moves, concerning two tasks mapped on the same computation node. The idea is that, in order to move the EDI of a task more into hardware, in the case we would not have enough resources, it is

Figure 5.6: Swap move example for Tabu Search

first needed to move the EDI of another task mapped on the same computation node, more into software, to make room on the FPGA device. The advantage of performing a swap is that, if possible, we manage to find a more efficient use of the currently occupied HW resources.

In Figure 5.6a we consider the case of tasks P_3 and P_4 from the motivational example 1 (Figures 5.3a and 5.3b), which are mapped on the same computation node (N_2). The two tasks have the HW cost – time overhead trade-off points shown in Figure 5.6a (see also Table 5.1). The point on the vertical axis represents the SW-only EDI, the middle point represents the mixed-HW/SW EDI and the third point illustrates the HW-only EDI, for a particular task. At a certain step during the design space exploration, P_3 and P_4 both have mixed-HW/SW EDI assigned, which implies a total $FPGA_2$ area of $10 + 15 = 25$ units. The worst-case schedule in this case is illustrated in Figure 5.6b-(i) (note that also P_1 and P_2 have their mixed-HW/SW EDI assigned). Assuming that $FPGA_2$ has a total size of 40 area units, we have 15 units free in the above scenario. In order to be able to move P_4 to the HW-only EDI, we need 25 extra units of area. Since we only have 15 units available, the solution is to move P_3 to its SW-only EDI, thus freeing 10 extra area units. After this simple move, we can proceed and apply the second simple move, occupying the 25 available area units, by moving P_4 to the HW-only solution. Please note that the two simple moves

Figure 5.7: Restricting the neighborhood for Tabu Search

Figure 5.8: Architecture and application example for neighborhood restriction

mentioned above are performed in the same iteration, thus forming a swap move. The swap in our example had a beneficial impact and we reduced the WCSL from 370 to 350 time units (Figure 5.6b-(ii)), thus getting closer to the minimum.

An important feature of Tabu Search is its capability to escape from local minima by allowing the selection of non-improving moves. After selecting such a move, it is important to avoid the cycling caused by selection of the reverse (improving) move leading back to the local optimum. This is solved by the use of tabus. Whenever we perform a simple move we declare tabu the move that would reverse its effect, i.e. assigning P_i its previous EDI (line 7 in Algorithm 9). The move is forbidden for a number of iterations equal to its *tabu tenure* (determined empirically). When performing a swap move, for each of its constituent simple moves, we declare tabu the corresponding reverse move (and record them individually in the Tabu list).

5.2.5.2 Neighborhood Restriction

In theory, the best move is selected (line 6 in Algorithm 9) by considering all possible moves (simple or swap) and evaluating the cost function for

Algorithm 10 Hierarchical neighborhood exploration**Input:** $\mathcal{A} = \mathcal{G}(\mathcal{V}, \mathcal{E}), W, C, WCTT_{ij}, k, HW_j, M$ **Output:** $trial_Move$ = best move found according to Section 5.2.5.3

```

1: function SELECT_BEST_MOVE( $current\_Sol, current\_WCSL$ )
2:    $CP \leftarrow \text{Select\_CP\_Tasks}(current\_Sol);$ 
3:    $trial\_Move \leftarrow \text{Try\_Simple\_Moves\_into\_HW}(CP);$ 
4:   if  $trial\_Move$  exists then
5:     return  $trial\_Move;$ 
6:   else
7:      $trial\_Move \leftarrow \text{Try\_Swap\_Moves}(CP);$ 
8:      $trial\_Sol \leftarrow \text{Apply}(trial\_Move, current\_Sol);$ 
9:      $trial\_WCSL \leftarrow \text{WCSL}(trial\_Sol);$ 
10:    if  $trial\_WCSL < current\_WCSL$  then
11:      return  $trial\_Move;$ 
12:    else
13:       $WCP \leftarrow \{p \in CP \mid \text{Wait}(p) > \text{waiting\_count}\};$ 
14:       $trial\_Move \leftarrow \text{Diversifying\_Non-improving\_Moves}(WCP);$ 
15:    return  $trial\_Move;$ 
16:  end if
17: end if
18: end function

```

each one. This, however, is inefficient from the computational point of view. Therefore, in each iteration, the selection of the best move is done by exploring only a subset of the possible moves, namely the ones affecting the tasks on the critical path (CP) of the worst-case schedule for the current solution. In Figure 5.7 we consider the application from Figure 5.8a mapped on the architecture in Figure 5.8b, with static FPGAs of size 20. We show how we move from the SW-only solution (Figure 5.7-(i): $WCSL = 750$) to the solution in Figure 5.7-(iii) ($WCSL = 510$), passing through two consecutive iterations. At each iteration, only the tasks on the CP (illustrated with dotted rectangles) are considered for new EDI assignment. From (i) to (ii), the best possible choice is to move P_1 to its mixed-HW/SW EDI. As a result, the CP changes (as can be seen from Figure 5.7-(ii)) and, thus, in the next iteration P_3 will also be considered (while P_1 is now excluded). The best choice is to move P_3 to its mixed-HW/SW EDI. The result is shown in Figure 5.7-(iii) ($WCSL = 510$).

5.2.5.3 Move Selection

Algorithm 10 presents our approach to selecting the best move in each iteration (line 6, Algorithm 9). As explained earlier, we first determine the set of tasks on the critical path (CP) of the current solution (line 2). Next, based on this set, we proceed and search for the best move in a hierarchical

manner (in order to reduce the number of evaluations of the cost function, done in each iteration). We first explore the simple moves into HW (line 3). If at least one such move exists and it is not tabu, then we select the move that generates the best improvement and we stop further exploring the rest of candidate moves (lines 4-5). Otherwise, we try to improve the current WCSL by searching for the best swap move (line 7). If we get closer to a minimum (line 10), we accept the move (line 11). Otherwise, we diversify the search (lines 13-15).

5.2.5.4 Diversification

In order to assure the proper breadth of the search process, we decided to employ a *continuous* diversification strategy (lines 13-15, Algorithm 10), complemented by a *restart* diversification strategy (lines 11-12, Algorithm 9). The continuous diversification uses an intermediate-term frequency memory (*Wait* counters), where we record how many iterations a task has waited since the last time it was involved in a move. The *Wait* counters are reset whenever a new minimal solution is reached. Whenever we need to escape local minimum points, the *Wait* memory is used to selectively filter candidate moves (and generate the set *WCP* – line 13 in Algorithm 10). So, if the waiting counter of a task is greater than a threshold (*waiting_count*), we consider that the task has waited a long time and should be selected for diversification (it is included in *WCP*). Our neighborhood exploration continues by selecting the non-improving move (simple or swap) that leads to the solution with the lowest cost, giving priority to diversifying moves (line 14 in Algorithm 10).

We complemented our continuous diversification strategy with a restart diversification. Whenever we do not get any improvement of the best known solution for more than a certain number, *diversification_count*, of iterations, we restart the search process (lines 11-12 in Algorithm 9). The search is restarted from a state corresponding to an EDI assignment to tasks that has not been visited, or has been visited rarely, during the previous searches.

The search process is stopped after a specified maximum number of iterations (line 5 in Algorithm 9). This value (*max_iterations*), as well as the counters used for diversification purposes (i.e. *waiting_count* and *diversification_count*), and the length of the tabu list, were determined empirically for each application size.

5.2.6 Partial Dynamic Reconfiguration

PDR enables the reuse of FPGA area corresponding to error detector modules of tasks that finished executing. Task execution is overlapped with reconfiguration of other detector modules, in order to mask the reconfiguration overhead. We reconfigure the detector modules on the FPGA as they are needed, at run-time. In this way we accelerate error detection in

hardware, while, at the same time, keeping system costs within reasonable limits.

5.2.6.1 Revised System Model

We model our PDR FPGA as a matrix of configurable logic blocks (CLBs). Each EDI occupies a rectangular area of this matrix. The model allows 2D or 1D placement and reconfiguration. The execution of an EDI can proceed in parallel with the reconfiguration of another EDI, but only one reconfiguration may be done at a time (i.e. we assume a single reconfiguration controller). If it is not possible to completely overlap reconfiguration with useful computations, then task execution is scheduled as soon as the reconfiguration of its EDI ends. Figure 5.4b-(iii) illustrates this: for P_2 the entire reconfiguration overhead was masked; Because of limited resources, this was not possible for P_4 , which had to wait for the reconfiguration of its EDI module to finish.

5.2.6.2 Revised Problem Formulation

Under PDR assumptions, our problem formulation (Section 5.2.3) becomes more complex. Besides generating the EDI assignment to tasks ($S : \mathcal{V} \rightarrow \mathcal{H}$) we also need to generate a placement and a schedule for EDI reconfigurations on FPGA. For all tasks with error detection in HW (mixed-HW/SW or HW-only), we find the function:

$$R : \{p \in \mathcal{V} \mid S(p) \neq \text{SW-only}\} \rightarrow \mathbb{Z}^+$$

which specifies the reconfiguration start time for the EDI modules, and the placement function:

$$P : \{p \in \mathcal{V} \mid S(p) \neq \text{SW-only}\} \rightarrow \mathbb{Z}^+ \times \mathbb{Z}^+$$

which specifies the position of the upper left corner of each EDI module on the FPGA.

5.2.6.3 Revised Scheduler

We simultaneously schedule tasks on the processor and place their corresponding EDIs on the FPGA: once a task is selected for scheduling from the list of ready tasks, its EDI is placed onto the FPGA, as soon as enough space is available for it. Thus, the generated static schedules are correct by construction.

In order to take into account the issues presented above, we extended the fault-tolerant schedule synthesis tool from [IPEP06] (used in Section 5.2.5). This tool is based on a list scheduling approach that uses a modified partial critical path (*PCP*) priority function [EDPP00] to decide the order of task execution. Since this priority function does not capture the

particular issues related to PDR, we changed it with another one, similar to [BBD05]. This priority function accounts for the application's characteristics (captured by *PCP*), the EDI characteristics (captured by *WCET* and EDI area for tasks) and the physical issues related to FPGA placement and reconfiguration (captured by the earliest start time – *EST* – of a task):

$$f(EST, WCET, area, PCP) = x \times EST + y \times WCET + z \times area + w \times PCP$$

Our tabu search will try to assign the best values to the weights x, y, z, w , such that the scheduling priority function generates the shortest schedule length for each particular application. The optimization is described in Section 5.2.6.6.

5.2.6.4 Anti-fragmentation Policy

When computing the *EST*, we find the earliest time slot when a task can be scheduled, subject to the various constraints. We first search for the earliest time instant when a feasible EDI placement on FPGA is available. The position of an EDI module on the FPGA is decided by using an anti-fragmentation policy.

The free FPGA space is managed by keeping a list of maximal empty rectangles. When placing a new error detection module on the FPGA, the location that generates the lowest fragmentation is chosen. The amount of FPGA fragmentation is quantified using the fragmentation metric from [HV04]. The idea is to compute the fragmentation contribution of each cell³ (*FCC*) in an empty area (rectangle that can fit the EDI module) as:

$$FCC_d(C) = \begin{cases} 1 - \frac{v_d}{2\bar{L}_d - 1} & \text{if } v_d \leq 2\bar{L}_d - 1 \\ 0 & \text{otherwise} \end{cases} \quad (5.1)$$

where v_d represents the number of empty cells in the vicinity of cell C and $\bar{L}_d$ is the average size in direction d (where d is either the horizontal or the vertical direction) of all the modules placed so far. We assume that if a rectangle can accommodate a module as large as twice the average size of the modules being placed, the area inside that rectangle is not fragmented.

The *FCC* represents “the amount of fragmentation in horizontal or vertical direction that an empty cell contributes toward total fragmentation of the FPGA area at a given time” [HV04]. The total fragmentation (*TF*) of an area can be computed as the normalized sum of fragmentation (*FCC*) of all the N cells in that area, in both directions:

$$TF(Area) = \frac{\sum_{C \in Area} (FCC_x(C) + FCC_y(C))}{N} \times 100 \quad (5.2)$$

A higher value of *TF* means more fragmentation of the FPGA.

³A cell represents a configurable logic block (CLB) in our model.

Figure 5.9: Using the fragmentation metric to place modules on the FPGA according to the anti-fragmentation policy

Whenever we choose a location for an error detection module, we calculate the TF of all maximal empty rectangles large enough to fit that module. We choose the rectangle with the highest TF , leaving less fragmented areas on the FPGA for placement of future modules. Once a rectangle is chosen for placement, we compute the TF of the remaining empty area when the module is placed in each of the four corners of the rectangle. We choose the location that generates the lowest TF , to keep the remaining FPGA area as less fragmented as possible.

Figure 5.9 exemplifies the fragmentation metric, assuming 2D placement of modules on the FPGA. Let us assume that at one point in time EDI module M_1 is placed on the FPGA and we need to decide a location for module M_2 . We identify two maximal empty rectangles on the FPGA, R_1 and R_2 (Figure 5.9a). Applying equations 5.1 and 5.2 to compute the TF for R_1 and R_2 we obtain $TF(R_1) = 0.92$ and $TF(R_2) = 0.95$. Thus, we choose R_2 for placing M_2 . Figures 5.9b and 5.9c show the two possible location of M_2 within R_2 . For the first case $TF(FPGA) = 0.67$, while for the second case $TF(FPGA) = 1.20$. So we choose the first location for M_2 . Assuming that the next module to be placed is M_3 , we observe that it can be accommodated in Figure 5.9b, but not in Figure 5.9c, because of the fragmentation.

After finding a location for the EDI module, its reconfiguration can be scheduled immediately if the reconfiguration controller is available. Otherwise, it has to wait until the controller becomes free. Once the reconfiguration component (corresponding to the EDI of a task) is scheduled, we check if the task could be scheduled immediately after that, subject to dependency constraints.

Figure 5.10: Modified scheduler for systems with partial dynamic reconfiguration capabilities

5.2.6.5 Illustrative Example

The Gantt charts in Figure 5.10c illustrate some of the issues related to PDR. Let us consider the application from Figure 5.10a, having to tolerate a number of $k = 1$ faults, mapped on an architecture with two computation nodes (Figure 5.10b), with $FPGA_1$ and $FPGA_2$ having sizes $HW_1 = HW_2 = 25$ area units. We assume that at a certain step during the optimization, each task has a particular EDI assigned. The WCETs of tasks, as well as the EDI areas and reconfiguration overheads corresponding to this current EDI⁴ are presented in Table 5.2. WCTT of all messages and

⁴Note that, although our model supports 2D placement and FPGA reconfiguration, for simplicity we illustrate the 1D case here.

Table 5.2: Task characteristics for the illustrative example

P_i	$WCET_i$	PCP_i	EDI area (h_i)	rec. time (ρ_i)
P_1	70	170	20	25
P_2	40	160	10	15
P_3	30	120	10	15
P_4	100	100	25	30
P_5	40	90	15	20
P_6	50	50	15	20

the recovery overheads for tasks are 10.

Figure 5.10c-(i) shows the schedule obtained using only the partial critical path (PCP_i in Table 5.2) for priority assignment, having WCSL = 525.

Let us assume that the priority function from Section 5.2.6.3 is used. We consider the weights $(x, y, z, w) = (0.5, -0.5, 0, 1)$, implying $f = 0.5 \times EST - 0.5 \times WCET + PCP$. Thus we give higher priority to tasks that have a big PCP and EST , while the $WCET$ should be small. The result is that P_2 is scheduled before P_1 , which in turn makes P_3 - P_5 - P_6 run earlier, thus reducing the WCSL to 475 in Figure 5.10c-(ii).

It is possible to choose better weights, adapted to the application's characteristics. It is preferable to give higher priority to tasks with a big PCP value, but a small $WCET$ and EDI area, and that can start earlier (smaller EST). Setting $(x, y, z, w) = (-0.5, -0.5, -1, 1)$ we obtain the schedule in Figure 5.10c-(iii), with WCSL = 360. Scheduling P_5 before P_4 has two advantages: firstly, P_6 runs earlier, in parallel with P_4 , and secondly P_3 and P_5 fit together on the FPGA and, thus, the run-time reconfiguration for P_5 's EDI is eliminated. As seen, it is important to choose proper weights for the scheduler.

5.2.6.6 Tuning the Scheduler

The scheduler weights x, y, z, w (Section 5.2.6.3) are dynamically tuned for each application, during our optimization. We kept the Tabu Search core used for the static reconfiguration approach (Section 5.2.5), with two modifications:

1. we use the modified list scheduler described above, as a cost function, instead of the scheduler used before (line 9, Algorithm 9 and line 9, Algorithm 10);
2. we add a new type of moves, that concern the weights x, y, z, w used in the scheduler's priority function.

In each iteration, before exploring different EDI assignments to tasks, we explore different values for the weights, which can take values between -1 and 1, with a 0.25 step. Every iteration we explore if modifying these

Figure 5.11: The ranges for the random generation of EDI overheads

weights results in a priority function that leads to a better scheduling, and consequently to a smaller WCSL. If this is not possible, we search for a better EDI assignment, exactly as we did before (Algorithm 10).

5.2.7 Experimental Evaluation

We first performed experiments on synthetic examples. We generated task graphs with 20, 40, 60, 80, 100 and 120 tasks each, mapped on architectures consisting of 3, 4, 5, 6, 7 and 8 nodes respectively. We generated 15 graphs for each application size. Tasks' WCETs were assigned randomly from the uniform distribution on the $[10, 250]$ time units range. All messages were assumed to have equal WCTT. We considered that our system has to tolerate $k = 2$ faults⁵.

In order to generate time and hardware cost overheads for each EDI, we proceeded as follows: we generated one class of experiments (*Testcase₁*), based on the estimation of overheads done by Pattabiraman et al. in [PKI11] and by Lyle et al in [LCP⁺09]. We also generated a second class of experiments (*Testcase₂*), for which we assumed slower hardware (in other words, in order to get the same time overheads as in *Testcase₁*, we need to use more hardware). In Figure 5.11 we show the ranges used for randomly generating the overheads. The point corresponding to 100% HW cost overhead represents the maximum HW area that the EDI for this task might occupy if mapped to FPGA. We assumed that this value is proportional to the task size.

Figure 5.11a depicts the ranges for *Testcase₁*: for the SW-only EDI, we considered a time overhead as big as 300% and as low as 80%, related to

⁵We also conducted experiments with $k \in [3, 8]$ and we concluded that the impact of different k is not significant for the quality of results produced by our heuristic.

Figure 5.12: Experiments' space

the worst-case execution time of the corresponding task; obviously, the HW cost overhead in this case is 0. As far as the mixed HW/SW implementation is concerned, the time overhead range is between 30% and 70%, and the HW cost overhead range is between 5% and 15%. Finally, the HW-only implementation would incur a time overhead between 5% and 25% and a HW cost overhead between 50% and 100%. Figure 5.11b depicts the ranges for *Testcase₂*: the time overhead ranges are the same, but we pushed the HW cost ranges more to the right. Also note that for *Testcase₂*, the centers of gravity of the considered areas are more uniformly distributed. The execution time overheads and the HW cost overheads for the tasks in our synthetic examples are distributed uniformly in the intervals depicted in Figure 5.11a (*test-case₁*) and Figure 5.11b (*Testcase₂*).

We varied the size of the FPGAs by summing up all the HW cost overheads corresponding to the HW-only implementation for all tasks:

$$\text{MAX_HW} = \sum_{i=1}^{\text{card}(\mathcal{V})} C(P_i, \text{HW-only})$$

and then generated problem instances with FPGA areas equal to 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 60%, 80%, 90% and 100% of MAX_HW, distributed evenly among computation nodes. The 100% is an extreme case, and represents the situation in which we have available all the HW that we need, so the optimization problem actually disappears. Figure 5.12 shows the resulting space of experiments. A total of $2 \times 6 \times 15 \times 12 = 2160$ settings were used for experimental evaluation.

5.2.7.1 Static FPGA Reconfiguration

Our baseline was the WCSL obtained by choosing the SW-only EDI for all tasks ($\text{WCSL}_{\text{baseline}}$). For the same task graphs, we then considered the various HW fractions assigned and for each case we calculated the worst-case schedule length, $\text{WCSL}_{\text{static}}$, after applying our heuristic. The performance

Figure 5.13: Comparison with theoretical optimum for statically reconfigurable FPGAs

improvement (PI) obtained for each case is:

$$PI = \left(\frac{WCSL_{baseline} - WCSL_{static}}{WCSL_{baseline}} \times 100 \right) \%$$

We compared our results with the theoretical optimum, $WCSL_{opt}$, obtained with a Branch and Bound (BB) search. We calculated the PI as above, using $WCSL_{opt}$ instead of $WCSL_{static}$. It was possible to obtain the optimal solution only for application sizes up to 20, with up to 40% HW fraction.

Figure 5.13 shows the average improvement over all test cases for our heuristic and for the optimal solution. The differences were up to 1% for *Testcase₁*, and up to 2.5% for *Testcase₂*, which shows the effectiveness of our approach.

Next, we evaluated the impact of the HW fraction assigned to FPGAs, on the PI . Figure 5.14 shows the average PI obtained. We shortened the WCSL with up to 64% (compared to the baseline – SW-only solution). We observe a saturation point, beyond which assigning more HW area produces only marginal improvement. At the saturation point, all tasks having an impact on the schedule length already have their best EDI assigned.

With only 15% HW fraction, we reduced the WCSL by more than half, for *Testcase₁*. For *Testcase₂*, in order to reduce the WCSL by half, we need ~40% HW fraction. This is due to our assumptions that the hardware is slower for *Testcase₂* and, thus, we need more HW in order to get the same performance improvement as for *Testcase₁*. This also influences the saturation point.

We also evaluated how the number of tasks/processor influences the results obtained. For that, we mapped an application of a specific size on

(a) *Testcase₁* (b) *Testcase₂*

Figure 5.14: Impact of varying the hardware fraction for statically reconfigurable FPGAs

architectures with different number of processors. We present the result corresponding to applications consisting of 60 tasks, from *Testcase₁*, which were mapped on 2, 3, 4 and 5 processors. Figure 5.15 shows the average improvement obtained. Note that the number of tasks/processor does not influence significantly the results.

5.2.7.2 Partial Dynamic Reconfiguration

We kept the same experimental setup as for the static case, and we considered as baseline the WCSL obtained considering static reconfiguration, $WCSL_{static}$. Then we computed:

$$PI^{PDR} = \left(\frac{WCSL_{static} - WCSL_{PDR}}{WCSL_{static}} \times 100 \right) \%$$

Figure 5.15: Impact of varying the number of tasks/processor for statically reconfigurable FPGAs

Figure 5.16 shows the average PI^{PDR} . By employing PDR, execution of some tasks was overlapped with the reconfiguration of the error detectors for others and, consequently, we shortened the schedule length with up to 37% (with a HW fraction of only 5%) for *Testcase₁* and with up to 36% (with a HW fraction of 20%) for *Testcase₂*.

For *Testcase₁*, after the peak of improvement gain, the improvement drops with increasing HW fraction, then increases slightly and finally drops again. The explanation resides in the ranges for EDI overheads in *Testcase₁*. The initial peak of improvement results from the fact that the algorithm is able to move EDIs to the mixed-HW/SW implementation. Then, assigning more HW does not help proportionally much, due to the large gap between the mixed-HW/SW and HW-only EDIs for *Testcase₁*. As the FPGAs get bigger, the HW-only EDIs are accommodated (second improvement increase). For big FPGA sizes, there is enough space from the beginning, so the static strategy can readily place all the needed EDIs.

For *Testcase₂*, the gap between mixed-HW/SW and HW-only EDIs is smaller and, thus, the maximum improvement (36%) corresponds to a HW fraction of ~20-25% (compared with 5% for *Testcase₁*).

Figure 5.17 shows the impact of considering different number of tasks/processor. The results correspond to the same applications as for the static case (60 tasks, *Testcase₁*), shown in Figure 5.15. For the PDR approach the number of tasks/processor influences the improvement obtained. The results are better with more tasks/processor (e.g. architectures with 2 processors) because the opportunities of using PDR are increased. For 10% and 15% hardware fractions we get an extra 14% improvement when mapping the application on 2 processors as opposed to 5 processors.

Figure 5.18 presents the running times of our optimization. Experiments were performed on a Windows Vista PC with CPU frequency 2.83 GHz and 8 GB RAM. The values correspond to the setting with 40% HW fraction

(a) *Testcase₁* (b) *Testcase₂*

Figure 5.16: Impact of varying the hardware fraction for partially dynamically reconfigurable FPGAs

(usually producing the longest running times). For small HW fractions, the algorithm does not have many alternative solutions to choose from, so a result is reached faster (e.g, for a 5% HW fraction and 80 tasks application size, the execution time is roughly 28% shorter for *Testcase₁* static, and 80% shorter for *Testcase₂* static, while for *Testcase₁* PDR is 18% shorter, and for *Testcase₂* PDR is 67% shorter than the corresponding values in Figure 5.18). For big HW fractions, the algorithms converge to good solutions relatively fast, since there is more freedom to place EDIs on FPGA. For example, for a 90% HW fraction and 60 tasks application size, for both approaches, the execution times were around 19% shorter for *Testcase₁* and for *Testcase₂* around 25% shorter than the corresponding values in Figure 5.18. Another aspect worth mentioning is that the PDR approach takes considerably more time, because in this case we also have to adjust the

Figure 5.17: Impact of varying the number of tasks/processor for partially dynamically reconfigurable FPGAs

Figure 5.18: Average running times for the optimization heuristics

weights for the scheduler (see Section 5.2.6.6).

5.2.7.3 Case Study – Adaptive Cruise Controller

We tested our approach on a real-life example, an adaptive cruise controller (ACC), similar to the one described in [AMHN05], whose task graph contains 13 tasks, depicted in Figure 5.19. The ACC keeps a desired speed and a safe distance to the preceding vehicle, has the possibility of autonomous changes of the maximum speed depending on speed limit regulations, and helps with the braking procedure in extreme situations. The functionality of the adaptive cruise controller is as follows: based on the driver specification and on the speed limit regulations, the SpeedLimit task computes the actual speed limit allowed in a certain situation. The Object Recognition task calculates the relative speed to the vehicle in front. This component

Figure 5.19: Task graph for the adaptive cruise controller case study

Figure 5.20: Results for the adaptive cruise controller case study

is also used to trigger ModeSwitch in case there is a need to use the brake assist functionality. ModeSwitch is used to trigger the execution of the ACC or of the BrakeAssist component. The ACC assembly (P₉ and P₁₀ in Figure 5.19) controls the throttle lever, while the BrakeAssist task is used to slam the brakes if there is an obstacle in front of the vehicle that might cause a collision.

We instrumented every task with error detectors, using the technique from Section 5.1. The execution times were derived using the MPARM cycle accurate simulator, considering an ARM processor operating at 40 MHz. The path trackers and checking modules were synthesized on an XC6VLX240T Virtex6 device, using the Xilinx ISE design suite. The reconfiguration times were estimated considering a 100 MHz configuration clock frequency and the ICAP 32-bit width configuration interface (see our reconfiguration controller described in Section 3.1.1.2). We used a methodology similar to the one in [SBB⁺06] to reduce the reconfiguration granularity. Table 5.3 presents the execution times and the hardware overheads obtained.

We mapped the application on two computation nodes. Considering a number of $k = 2$ faults, the results are presented in Figure 5.20. Using static

Table 5.3: Time and area overheads for the adaptive cruise controller case study

P_i	Error Detection Implementation (EDI)						
	SW-only	mixed-HW/SW			HW-only		
	$WCET_i$ (μs)	$WCET_i$ (μs)	h_i (slices)	ρ_i (μs)	$WCET_i$ (μs)	h_i (slices)	ρ_i (μs)
P_1	8.12	6.02	2	1.23	4.9	15	9.23
P_2	8.12	6.02	2	1.23	4.9	15	9.23
P_3	8.12	6.02	2	1.23	4.9	15	9.23
P_4	8.17	6.07	2	1.23	4.95	15	9.23
P_5	7.32	5.4	2	1.23	4.27	13	8
P_6	8.37	6.27	2	1.23	4.92	13	8
P_7	10.35	7.87	2	1.23	6.75	42	25.83
P_8	12.17	9.6	3	1.85	8.45	15	9.23
P_9	46.22	33.7	11	6.77	27.25	43	26.45
P_{10}	20.3	14.45	4	2.46	11.57	28	17.22
P_{11}	37.27	29.47	7	4.31	21.7	52	31.99
P_{12}	8.12	6.02	2	1.23	4.9	15	9.23
P_{13}	8.12	6.02	2	1.23	4.9	15	9.23

reconfiguration, we obtained up to 47% reduction in schedule length (over the SW-only implementation) with as little as 35% HW fraction. Assuming PDR, we got an extra reduction. The best result is obtained for 20% HW fraction, where the static reconfiguration approach achieves 42% improvement (over the SW-only implementation), while the PDR approach achieves an extra 9% improvement (over the static approach).

5.3 Average Execution Time Minimization

The previous section handled the matter of safety critical applications, considering strict worst-case scenarios, where deadlines have to be satisfied even in case of faults. However, for a large class of applications faults have to be handled, but what is important is the average execution time. Faults appear only seldom, and their handling does not affect the average. What is important is to minimize the impact of the error detection on the average performance. In order to achieve this, a more refined model is needed (compared to Section 5.2), that captures data and control dependencies. The previous section addressed the performance optimization problem at a task level. Using such a task-level coarse granularity is not appropriate for many applications, e.g. those that consist of a single sequential task. In what follows, instead of considering the tasks as black boxes, we will analyze their internal structure and properties.

Note that in this section, similar to Section 3.3, we consider correlations.

However, there are some differences: first of all, the optimization presented in this section is done off-line, as opposed to the technique presented in Section 3.3, which is dynamic and performed on-line. Another difference is that in this section we consider the correlations in an implicit manner, i.e. we will take into account the correlation between a certain program path being followed and the corresponding checking expression being reached. Therefore, the output of our optimization will be a conditional reconfiguration schedule table.

The rest of this section presents an approach to minimize the average execution time of a program by optimizing the hardware/software implementation of error detection. We leverage the advantages of partial dynamic reconfiguration of FPGAs in order to speculatively place in hardware those error detection components that will provide the highest reduction of execution time. Our optimization algorithm uses frequency information from a counter-based execution profile of the program. Starting from a control flow graph representation, we build the interval structure and the control dependence graph, which we then use to guide our error detection optimization algorithm.

5.3.1 Basic Concepts

Before presenting our optimization approach, let us introduce some definitions that will help the reader better understand the rest of this section. Note that the model used in the following is similar to the one used in Section 3.2. Thus, some definitions will be repeated, for the sake of the presentation flow.

Definition 5.1 A *control flow graph* (CFG) of a program is a directed graph $\mathcal{G}_{cf}(\mathcal{N}_{cf}, \mathcal{E}_{cf})$, where each node in $\mathcal{N}_{cf}$ corresponds to a straight-line sequence of operations and the set of edges $\mathcal{E}_{cf}$ corresponds to the possible flow of control within the program. $\mathcal{G}_{cf}$ captures potential execution paths and contains two distinguished nodes, *root* and *sink*, corresponding to the entry and the exit of the program.

Definition 5.2 A node $n \in \mathcal{N}_{cf}$ is *post-dominated* by a node $m \in \mathcal{N}_{cf}$ in the control flow graph $\mathcal{G}_{cf}$ if every directed path from n to *sink* (excluding n) contains m .

Definition 5.3 Given a control flow graph $\mathcal{G}_{cf}$, a node $m \in \mathcal{N}_{cf}$ is *control dependent* upon a node $n \in \mathcal{N}_{cf}$ via a control flow edge $e \in \mathcal{E}_{cf}$ if the next conditions hold:

- There exists a directed path P from n to m in $\mathcal{G}_{cf}$, starting with e , with all nodes in P (except m and n) post-dominated by m ;
- m does not post-dominate n in $\mathcal{G}_{cf}$.

In other words, there is some control edge from n that definitely causes m to execute, and there is some path from n to *sink* that avoids executing m .

Definition 5.4 A *control dependence graph* (CDG) $\mathcal{G}_{cd}(\mathcal{N}_{cd}, \mathcal{E}_{cd})$ corresponding to a control flow graph $\mathcal{G}_{cf}(\mathcal{N}_{cf}, \mathcal{E}_{cf})$ is defined as: $\mathcal{N}_{cd} = \mathcal{N}_{cf}$ and $\mathcal{E}_{cd} = \{((n, m), e) \mid m \text{ is control dependent upon } n \text{ via edge } e\}$. A *forward control dependence graph* (FCDG) is obtained by ignoring all back edges in the CDG [ALSU06].

Definition 5.5 An interval $I(h)$ in a control flow graph $\mathcal{G}_{cf}(\mathcal{N}_{cf}, \mathcal{E}_{cf})$, with header node $h \in \mathcal{N}_{cf}$, is a strongly connected region of $\mathcal{G}_{cf}$ that contains h and has the following properties:

- $I(h)$ can be entered only through its header node h ;
- all nodes in $I(h)$ can be reached from h along a path contained in $I(h)$;
- h can be reached from any node in $I(h)$ along a path contained in $I(h)$.

The interval structure represents the looping constructs of a program.

Definition 5.6 An *extended control flow graph* (ECFG) is obtained by augmenting the initial control flow graph with a pseudo control flow edge (*root*, *sink*) and with explicit interval entry and exit nodes as follows:

- for each interval $I(h)$, a preheader node, ph , is added and all interval entries are redirected to ph (i.e. every edge (n, h) , with n not in $I(h)$ is replaced with the edge (n, ph) and an unconditional edge (ph, h) is added);
- a postexit node, pe , is added and every edge (n, m) with n in $I(h)$ and m not in $I(h)$ is replaced by edges (n, pe) and (pe, m) ;
- a pseudo control flow edge (ph, pe) is added.

The pseudo control flow edges provide a convenient structure to the control dependence graph (causing all nodes in an interval to be directly or indirectly control dependent on the preheader node).

Definition 5.7 Given an edge $(n, m) \in \mathcal{E}_{fcd}$ in the forward control dependence graph (FCDG) $\mathcal{G}_{fcd}(\mathcal{N}_{fcd}, \mathcal{E}_{fcd})$, the execution frequency of the edge [Sar89] is defined as follows:

- if n is a preheader and m is a header, $freq(n, m) \geq 0$ is the average loop frequency for interval $I(m)$;
- otherwise, $0 \leq freq(n, m) \leq 1$ is the branch probability.

Let us exemplify these concepts on the control flow graph from Figure 5.21. Node a is post-dominated by nodes e and h , but not by node b , for example (that is because there exists the path from node a to the sink s going through node d , that excludes node b). Nodes b and d are control dependent upon node a via edges $a - b$ and $a - d$, respectively. Note that the addition of the pseudo control flow edge $r - s$ transforms the initial CFG into an extended control flow graph (ECFG), and causes all nodes to be control dependent on r .

5.3.2 System Model

We consider a *structured* [ALSU06] program, modeled as a control flow graph $\mathcal{G}_{cf}(\mathcal{N}_{cf}, \mathcal{E}_{cf})$. The program runs on an architecture composed of a central processing unit, a memory subsystem and a reconfigurable device (FPGA). As in the previous section, we assume that suitable mitigation techniques are employed (e.g. [LCR03], [NOS14]) in order to provide sufficient reliability of the hardware used for error detection.

We model our FPGA, supporting partial dynamic reconfiguration, as a rectangular matrix of configurable logic blocks (CLBs). Each checker occupies a contiguous rectangular area of this matrix. The model allows modules of any rectangular shape and size. The execution of a checker can proceed in parallel with the reconfiguration of another checker, but only one reconfiguration may be done at a certain time.

5.3.3 Problem Formulation

Input

- A structured program modeled as a CFG $\mathcal{G}_{cf}(\mathcal{N}_{cf}, \mathcal{E}_{cf})$.
- Each node $n \in \mathcal{N}_{cf}$ has a corresponding execution time, given by $time : \mathcal{N}_{cf} \rightarrow \mathbb{Z}^+$.
- Each edge has associated the probability to be taken, given by $prob : \mathcal{E}_{cf} \rightarrow [0; 1]$.

We assume that this information is obtained by profiling the program and it is given.

- The set of checkers (checking expressions), CE , corresponding to the program is also given. Each checker $c \in CE$ has associated the following:
 1. The node where it will be executed, given by $host : CE \rightarrow \mathcal{N}_{cf}$.
 2. The acyclic path for which the checker has been derived (denoted with $path(c) = \text{sequence of edges that lead to } host(c) \text{ in the CFG}$).
 3. The execution time ($sw : CE \rightarrow \mathbb{Z}^+$) if implemented in software.

Figure 5.21: Motivational example for speculative reconfiguration of error detection components

4. The execution time ($hw : CE \rightarrow \mathbb{Z}^+$), necessary FPGA area ($area : CE \rightarrow \mathbb{Z}^+ \times \mathbb{Z}^+$) and reconfiguration overhead ($rec : CE \rightarrow \mathbb{Z}^+$) if implemented in hardware.
- The total available FPGA area which can be used to implement the error detection is also given⁶.

Goal

- To minimize the average execution time of the program, while meeting the imposed hardware area constraints.

Output

- A conditional reconfiguration schedule table (see Section 5.3.5.1).

5.3.4 Motivational Example

Let us consider the ECFG in Figure 5.21. The graph shows two conditional branches. The execution times of the nodes are: $a = 1, b = 10, d = 20, e =$

⁶Note that in this section we focus our attention on the hardware/software implementation of the checking expression. We assume that the path tracking part of the error detection mechanism (as described in Section 5.1) is implemented fully in hardware, since its overhead is significant in software, while its hardware implementation is very efficient.

Table 5.4: Error detection overheads for the motivational example

c_i	SW	HW		
	time overhead	time overhead	area units	reconfig. time
c_1	22	4	25	25
c_2	18	3	10	10
c_3	15	2	20	20
c_4	29	3	40	40

1, $f = 20, g = 10$, while the $root(r)$ and $sink(s)$ have execution time zero. The conditional edges have their associated probabilities presented on the graph.

Let us assume that inside node h we have a critical variable that needs to be checked and, by applying the technique described in Section 5.1 we derived four checkers for it: c_1 to c_4 . Each of them corresponds to a different acyclic control path in the ECFG, as shown in Figure 5.21. In Table 5.4 we present the time overheads incurred by the software implementation of each checker, as well as the overheads, area and reconfiguration time incurred by the hardware implementation. Since node h is composed of the four checkers, its execution time depends on the actual path followed at run-time, which will determine which of the four, mutually exclusive, checkers will get executed.

For the given example, if we assume that all the checkers are implemented in software, the resulting average execution time of the ECFG is 54.01 time units. On the other hand, if we assume that we have enough FPGA area so that we can place all the checkers in hardware from the beginning (i.e. 95 area units), the average execution time will be 36.44 time units.

Let us now consider an FPGA supporting partial dynamic reconfiguration and having an area of only 40 units. For the reconfiguration decision on each of the outgoing edges of node a , one intuitive solution would be the following: if edge (a, b) is followed, we would start reconfiguring the checker that has the highest probability to be reached on this branch (i.e. c_1). At run-time, when we reach the second conditional node, if the edge (e, f) is followed, then our previous speculation was right, and the reconfiguration of c_1 will have been finished by the time we reach it; otherwise, if the edge (e, g) is taken, then we can drop the reconfiguration of c_1 (because we will not execute it for sure) and we can start the reconfiguration of c_2 . Let us now consider the situation when edge (a, d) is followed. Applying the same strategy as above, we would start reconfiguring c_3 (because its probability is 60% compared to 40% for c_4). In this case, when we reach node e , if the prediction was right, we will have c_3 on FPGA by the time we reach it. But if the prediction was wrong, it is not profitable to reconfigure c_4 (on edge (e, g)) because reconfiguring it and then executing it in hardware would take $40 + 3 = 43$ time units (note that node g is executed in parallel with the re-

Table 5.5: Conditional reconfiguration schedule tables for the motivational example

Condition	S_1 action	Condition	S_2 action
(a, b)	rec. c_1	(a, b)	rec. c_1
$(a, b) \wedge (e, g)$	rec. c_2	$(a, b) \wedge (e, g)$	rec. c_2
(a, d)	rec. c_3	(a, d)	rec. c_4
$(a, d) \wedge (e, g)$	–	$(a, d) \wedge (e, f)$	rec. c_3

configuration), while leaving c_4 in software will result in a shorter execution time ($10 + 29 = 39$ time units). The result is that only c_1 , c_2 and c_3 will possibly be placed in hardware, while c_4 is kept in software. In Table 5.5 we capture the discussion above in the form of a conditional reconfiguration schedule (the alternative denoted with S_1). This schedule table contains a set of conditions and the corresponding reconfigurations that should be started when the condition is true. These conditions are given as a conjunction of edges, meaning that in order for the condition to evaluate to true, the edges have to be followed in the specified order at run-time. The schedule alternative S_1 will lead to an average execution time of 42.16.

However, investigating carefully the example, one can observe that a better scheduling alternative than S_1 can be generated. We can see that, in case edge (a, d) is taken, it is better to start the reconfiguration of c_4 (although it might seem counterintuitive since it has a smaller probability to be reached than c_3). Doing this way, if the prediction is wrong, it is still profitable to later place c_3 in hardware (schedule S_2 from Table 5.5). For c_3 it is possible to postpone its reconfiguration until a later point, since its execution is further away in time and its reconfiguration overhead is smaller than that of c_4 . Even if we start the reconfiguration of c_4 first, we still introduce a waiting time because the reconfiguration is not ready by the time we reach node h (we start reconfiguring c_4 at the end of node a , and the path to h is only $20 + 1 + 10 = 31$ time units, while the reconfiguration takes 40). Nevertheless, waiting 9 time units and then executing the checker in hardware for 3 units is better than executing it in software for 29 time units. So, by applying schedule S_2 , the resulting average execution time is 38.42, with an FPGA of just 40 area units.

When generating the more efficient scheduling alternative S_2 we did not base our decisions exclusively on execution probabilities (as is the case with S_1). Instead, we took into account the time gain resulted from each checker, as well as its reconfiguration overhead in conjunction with the length of the path from the current decision point, up to the location where the checker will be executed.

5.3.5 Speculative Reconfiguration

We propose a constructive algorithm to solve the problem defined in Section 5.3.3. The idea is to use the knowledge regarding already taken branches, in order to take the best possible decision for the future and speculatively reconfigure on FPGA the checkers with the highest potential to provide a performance improvement.

Since for each checker $c \in CE$ we have the corresponding acyclic path specified as a sequence of instrumented branches (edges from the CFG), we define the reach probability, given by $reach_prob : \mathcal{N}_{cf} \times CE \rightarrow [0; 1]$. The value $reach_prob(n, c)$ represents the probability that the checker's path is followed at run-time, given that node $n \in path(c)$ is reached and all the nodes on $path(c)$ up to n were visited. This is computed as the product of the probabilities of the edges in the checker's path, from n up to $host(c)$. If n is not on $path(c)$, then the reach probability will be zero. Considering the example from Figure 5.21, e.g. $reach_prob(a, c_1) = 0.45 \times 0.60 = 0.27$, $reach_prob(b, c_1) = 0.60$ and $reach_prob(d, c_1) = 0$.

For each checker $c \in CE$ we also define its time gain as the difference between its time overhead when implemented in SW versus in HW ($gain : CE \rightarrow \mathbb{Z}^+$, $gain(c) = sw(c) - hw(c)$). Finally, we denote with $iterations(c)$ the average frequency of the loop inside which c is executed. If c is not inside a loop, $iterations(c) = 1$. We will use this value to give higher priority to checkers that get executed inside loops.

We also assign a weight to each checker $c \in CE$:

$$weight(n, c) = \frac{reach_prob(n, c) \cdot iterations(c) \cdot gain(c)}{area(c)} \quad (5.3)$$

This weight represents the average time gain per area unit corresponding to c . As we traverse the CFG of the program, the weights of some checkers might become zero (when certain paths are followed, i.e. $reach_prob$ becomes zero), while the weight of other checkers will increase (as we approach their $host$ on a certain path).

5.3.5.1 Reconfiguration Schedule Generation

The pseudocode of our reconfiguration schedule generation algorithm is presented in Algorithm 11. Considering the CFG received as an input, we first determine its interval structure (line 2), then we build the ECFG (line 3) according to Definition 5.6, and the FCDG (line 4) according to Definition 5.4.

The next step is to traverse the ECFG and build the reconfiguration schedule (line 5). Once the schedule is generated all the checkers have been assigned an implementation (in hardware or in software), on any path that might get executed at run-time. Thus we can estimate the average execution time of the given CFG (line 6).

Algorithm 11 Speculative optimization algorithm

```

1: procedure BUILD_RECONFIGURATION_SCHEDULE( $CFG$ )
2:   determine interval structure of  $CFG$ 
3:   build ECFG
4:   build FCDG
5:    $\triangleright$  traverse ECFG and build reconfiguration schedules
6:   BUILD_SCHEDULE( $root, TRUE$ )
7:   compute average execution time of  $CFG$ 
8: end procedure
9: procedure BUILD_SCHEDULE( $n, cond$ )
10:  for all edge  $e = (n, m)$  do
11:    if ( $e$  not back edge or loop exit edge) or ( $e$  not visited) then
12:      mark  $e$  as visited
13:      if  $e$  is a back edge or a loop exit edge then
14:         $cond \leftarrow TRUE$   $\triangleright$  reset context
15:      end if
16:       $new\_cond \leftarrow cond \wedge e$ 
17:      for all checker  $cinCE$  do
18:        compute  $reach\_prob(m, c)$ 
19:        compute  $weight(m, c)$ 
20:        compute  $path\_length(n, c)$ 
21:      end for
22:      build  $ACE(e)$ 
23:      if reconfiguration controller available and  $ACE(e) \neq \emptyset$  then
24:        SCHEDULE_NEW_RECONFIGURATIONS( $e, new\_cond$ )
25:      end if
26:      BUILD_SCHEDULE( $m, new\_cond$ )
27:    end if
28:  end for
29: procedure SCHEDULE_NEW_RECONFIGURATIONS( $e = (n, m), cond$ )
30:  sort  $ACE(e)$  in descending order by  $weight$ 
31:  repeat
32:    pick and remove  $c \in ACE(e)$  with highest  $weight$ 
33:    if  $c$  already configured on FPGA but inactive then
34:      activate  $c$  at its current location
35:    else
36:       $loc \leftarrow CHOOSE\_FPGA\_LOCATION(c)$ 
37:      mark start of reconfiguration for  $c$ , at  $loc$ , on  $new\_cond$ 
38:      for all checkers  $c' \in ACE(e)$  do
39:        recompute  $path\_length(n, c')$ 
40:      end for
41:    end if

```

(continues on next page)

```

    (continued from previous page)
42:   until end of reconfiguration  $\geq$  end of  $m$ 's execution
43: end procedure
44: function CHOOSE_FPGA_LOCATION( $c$ )
45:   if there is enough free contiguous space on FPGA then
46:     return empty rectangle with lowest fragmentation metric
47:   else
48:     choose the least recently used inactive modules to be replaced
49:     return this location
50:   end if
51: end function

```

The actual HW/SW optimization is performed in the recursive procedure BUILD_SCHEDULE. The ECFG is traversed such that each back edge and each loop exit edge is processed exactly once (line 10). In order to be able to handle the complexity, whenever a back or loop exit edge is taken, we reset the current context (lines 12-13). This means that in each loop iteration we take the same reconfiguration decisions, regardless of what happened in previous loop iterations. The same applies to loop exit: decisions after a loop are independent of what happened inside or before the loop. This guarantees that the conditional reconfiguration schedule table that we generate is complete (i.e. we tried to take the best possible decision for any path that might be followed at run-time).

The generated schedule table contains a set of conditions and the corresponding reconfigurations to be performed when a condition is activated at run-time. Considering the example in Figure 5.21 a condition and its corresponding action is:

$$(a, b) \wedge (e, g): \text{reconfiguration of } c_2$$

At run-time, a condition is activated only if the particular path specified by its constituent edges is followed. At each control node $n \in \mathcal{N}_{ecf}$ in the ECFG we take a different reconfiguration decision on every outgoing edge (line 9). We will next describe the decision process for one such edge, $e = (n, m) \in \mathcal{E}_{ecf}$. First, the new condition is constructed (line 15). This condition corresponds to a conjunction of all the taken edges so far, and the current edge, e .

As we process a control edge $e = (n, m) \in \mathcal{E}_{ecf}$ in the ECFG, we compute the $reach_prob(m, c)$ and the $weight(m, c)$ of all checkers in CE (lines 17-18). These values will obviously be different from the previously computed ones, $reach_prob(n, c)$ and $weight(n, c)$. To be more exact they will increase as we approach $host(c)$.

We also compute the path length corresponding to each checker (denoted with $path_length(n, c)$). This represents the length (in time) of the path that leads from the current node n to $host(c)$ in the CFG. Please note that

all the checkers active along that path will have assigned an implementation (either SW or HW) for the current traversal of the ECFG, so we can compute (line 19) the length of the path (from the current node up to the location of the checker):

$$\begin{aligned} path_length(n, c) &= \\ &= \sum_{\substack{m \in path(c), \\ m \text{ successor of } n}} \left(time(m) + \sum_{\substack{k \in CE, k \neq c, \\ host(k)=m, \\ k \text{ active}}} overhead(k) \right) \end{aligned} \quad (5.4)$$

where

$$overhead(k) = \begin{cases} sw(k), & \text{if } k \text{ is implemented in software} \\ wait(k) + hw(k), & \text{otherwise} \end{cases}$$

and

$wait(k)$ = eventual waiting time introduced due to the fact that reconfiguration is not finished when the checker is reached.

We denote with $ACE(e)$ the set of currently active checking expressions (checkers) if edge e is taken in the ECFG, i.e.:

$$\begin{aligned} ACE(e) = & \{c \in CE \mid e \in path(c) \wedge \\ & area(c) \text{ fits in currently available FPGA area} \wedge \\ & rec(c) + hw(c) < path_length(n, c) + sw(c) \wedge \\ & \text{path from } m \text{ to } host(c) \text{ does not contain any back edge}\} \end{aligned} \quad (5.5)$$

We discard all the checkers for which we do not currently have enough contiguous FPGA area available for a feasible placement, as well as the ones for which it is actually more profitable (at this point) to leave them in software (because they are too close to the current location and reconfiguring and executing the checker in hardware would actually take longer time than executing it in software). We also do not start reconfigurations for checkers over a back edge, because we reset the context on back edges; thus, if a reconfiguration would be running, it would be stopped, and this might possibly lead to a wasteful use of the reconfiguration controller.

The next step after building $ACE(e)$ (line 21) is to check if the reconfiguration controller is available, in order to schedule new reconfigurations (line 22). We distinguish three cases:

1. The reconfiguration controller is free. In this case we can simply proceed and use it.
2. The reconfiguration controller is busy, but it is currently reconfiguring a checker that is not reachable anymore from the current point (we

previously started a speculative reconfiguration and we were wrong). In this case we can drop the current reconfiguration, mark the FPGA space corresponding to the unreachable checker as free and use the reconfiguration controller.

3. The reconfiguration controller is busy configuring a checker that is still reachable from the current point. In this case we leave the reconfiguration running and schedule new reconfigurations only in case the current reconfiguration will finish before the next basic block will finish. Otherwise, we can take a better decision later, on the outgoing edges of the next basic block.

In case $ACE(e) \neq \emptyset$ we start new reconfigurations (line 23) by calling the procedure `SCHEDULE_NEW_RECONFIGURATIONS` (described in Algorithm 11). Otherwise, we do not take any reconfiguration action. Then we continue traversing the ECFG (line 25).

5.3.5.2 FPGA Area Management

When we start a new reconfiguration, we choose a subset of checkers from $ACE(e)$ so that they fit on the currently available FPGA area, trying to maximize their total weight (lines 32-39 in Algorithm 11).

Let us next describe how the FPGA space is managed. After a checker is placed on the FPGA, the corresponding module is marked as active and that space is kept occupied until the checker gets executed. After this point, the module is marked as inactive but it is not physically removed from the FPGA. Instead, if the same checker is needed later (e.g. in a loop), and it has not been overwritten yet, we simply mark it as active and no reconfiguration is needed, since the module is already on the FPGA (lines 33-34). In case the module is not currently loaded on the FPGA we need to choose an FPGA location and reconfigure it. We distinguish two cases:

1. We cannot find enough contiguous space on the FPGA, so we need to pick one inactive module to be replaced. This is done using the least recently used policy (line 48);
2. There is enough contiguous free FPGA space, so we need to pick a location (line 46).

The free space is managed by using the anti-fragmentation policy described in Section 5.2.6.4.

After deciding on a free location, we mark the reconfigurations in the schedule table as active on the previously built condition (line 37). We stop as soon as the sum of reconfiguration times for the selected subset of checkers exceeds the execution time of the next basic block (line 42). For all the other checkers we can take a better decision later, on the outgoing edges of the next basic block.

5.3.6 Experimental Evaluation

In order to evaluate our algorithm we first performed experiments on synthetic examples. We randomly generated control flow graphs with 100 and 300 nodes (15 CFGs for each application size). The execution time of each node was randomly generated in the range of 10 to 250 time units. We then instrumented each CFG with checkers (35 and 75 checkers for CFGs with 100 and 300 nodes, respectively). For each checker, we generated an execution time corresponding to its software implementation, as well as execution time, area and reconfiguration overhead corresponding to its hardware implementation. The ranges used were based on the overheads reported in [LCP⁺09] for this error detection technique.

The size of the FPGA available for placement of error detection modules was varied as follows: we sum up all the hardware areas for all checkers of a certain application:

$$\text{MAX_HW} = \sum_{i=1}^{\text{card}(CE)} \text{area}(c_i) \quad (5.6)$$

Then we generated problem instances by considering the size of the FPGA corresponding to different fractions of MAX_HW: 3%, 5%, 10%, 15%, 20%, 25%, 40%, 60%, 80% and 100%. As a result, we obtained a total of $3 \times 15 \times 10 = 450$ experimental settings. All experiments were run on a PC with CPU frequency 2.83 GHz, 8 GB of RAM, and running Windows Vista.

We compared the results generated by our optimization algorithm (OPT) with a straight-forward implementation (SF), which statically places in hardware those expressions that are used most frequently and give the best gain over a software implementation, until the whole FPGA is occupied. Module placement is done as discussed in Section 5.3.5.2, according to the fragmentation metric (FCC) described in Section 5.2.6.4.

In order to compute our baseline, we considered that all checkers are implemented in hardware (for a particular CFG) and then calculated the execution time of the application ($EX_{\text{HW-only}}$). We also calculated $EX_{\text{SW-only}}$ considering that all checkers are implemented in software. For the same CFGs, we then considered the various hardware fractions assigned and for each resulting FPGA size we computed the execution time after applying our optimization, EX_{OPT} , and after applying the straight-forward approach, EX_{SF} . In order to estimate the average execution time corresponding to a CFG (line 6 in Algorithm 11), we used the methodology described in [Sar89], adapted to take into account reconfiguration of checkers.

For a particular result, EX , we define the normalized distance to the HW-only solution as:

$$D(EX) = \left(\frac{EX - EX_{\text{HW-only}}}{EX_{\text{SW-only}} - EX_{\text{HW-only}}} \times 100 \right) \% \quad (5.7)$$

(a) 100 nodes

(b) 300 nodes

Figure 5.22: Performance improvement

This distance gives a measure of how close to the HW-only solution we manage to stay, although we use less HW area than the maximum needed.

5.3.6.1 Experimental Results

In figures 5.22a and 5.22b we compare the average $D(EX_{OPT})$ with $D(EX_{SF})$ over all testcases with 100 and 300 nodes. The setting with 0% HW fraction corresponds to the SW-only solution. It can be seen, for both problem sizes, that our optimization gets results within 18% of the HW-only solution with only 20% of the maximum hardware needed. For big fractions the straight-forward solution (SF) also performs quite well (since there is enough hardware area to place most of the checkers with high frequencies), but for the small hardware fractions our algorithm significantly outperforms the SF solution. Figures 5.23a and 5.23b present the size of the reconfiguration tables (number of entries), for testcases with 100 and 300 nodes, for each hardware fraction considered.

As far as the running time of our optimization is concerned, for the CFGs with 100 nodes the results were generated in less than 3 seconds on average, while for the CFGs with 300 nodes the results were generated in 93 seconds on average.

(a) 100 nodes

(b) 300 nodes

Figure 5.23: Reconfiguration schedule table size

5.3.6.2 Case Study – GSM Encoder

We also tested our approach on a real-life example, a GSM encoder, which implements the European GSM 06.10 provisional standard for full-rate speech transcoding. This application can be decomposed into 10 tasks executed in a sequential order: Init, GetAudioInput, Preprocess, LPC_Analysis, ShortTermAnalysisFilter, LongTermPredictor, RPE_Encoding, Add, Encode, and Output. We instrumented the whole application with 56 checkers, corresponding to the 19 most critical variables, according to the technique described in Section 5.1. The execution times were derived using the MPARM cycle accurate simulator, considering an ARM processor with an operational frequency of 60 MHz. The checking modules were synthesized for an XC6VLX240T Virtex6 device, using the Xilinx ISE design suite. The reconfiguration times were estimated considering a 100 MHz configuration clock frequency and the ICAP 32-bit width configuration interface (see our reconfiguration controller described in Section 3.1.1.2). We used a methodology similar to the one presented in [SBB⁺06] in order to reduce the reconfiguration granularity.

The gain, area and reconfiguration overheads for each checker are given in Table 5.6. The CFGs for each task, as well as the profiling information was generated using the LLVM suite [LA04] as follows: *llvm-gcc* was first used to generate LLVM bytecode from the C files. The *opt* tool was then used to instrument the bytecode with edge and basic block profiling instructions. The bytecode was next run using *lli*, and then the execution profile was

Table 5.6: Time and area overheads of checkers for the GSM encoder case study

c_i	gain (μs)	area (slices)	reconfig. time (μs)	c_i	gain (μs)	area (slices)	reconfig. time (μs)
c_1	0.8	6	3.69	c_{29}	0.7	16	9.84
c_2	0.71	8	4.92	c_{30}	0.66	13	7.995
c_3	0.725	7	4.305	c_{31}	0.515	9	5.535
c_4	0.59	2	1.23	c_{32}	1	18	11.07
c_5	0.88	3	1.845	c_{33}	1.02	22	13.53
c_6	0.45	2	1.23	c_{34}	0.795	17	10.455
c_7	0.675	6	3.69	c_{35}	0.82	18	11.07
c_8	0.665	6	3.69	c_{36}	0.78	16	9.84
c_9	0.82	4	2.46	c_{37}	0.6	19	11.685
c_{10}	1.005	9	5.535	c_{38}	0.625	21	12.915
c_{11}	0.46	6	3.69	c_{39}	0.9	32	19.68
c_{12}	0.67	7	4.305	c_{40}	1.35	45	27.675
c_{13}	0.625	11	6.765	c_{41}	1.34	40	24.6
c_{14}	0.7	13	7.995	c_{42}	1.01	30	18.45
c_{15}	0.73	15	9.225	c_{43}	1.02	40	24.6
c_{16}	0.67	10	6.15	c_{44}	1	35	21.525
c_{17}	1.1	15	9.225	c_{45}	1.01	26	15.99
c_{18}	0.905	15	9.225	c_{46}	1.05	32	19.68
c_{19}	0.91	12	7.38	c_{47}	1.4	34	20.91
c_{20}	0.915	13	7.995	c_{48}	1.45	33	20.295
c_{21}	0.7	14	8.61	c_{49}	1.605	37	22.755
c_{22}	0.62	12	7.38	c_{50}	1.56	37	22.755
c_{23}	0.58	10	6.15	c_{51}	0.5	10	6.15
c_{24}	1.01	16	9.84	c_{52}	0.79	16	9.84
c_{25}	0.705	12	7.38	c_{53}	0.78	14	8.61
c_{26}	0.735	12	7.38	c_{54}	0.7	16	9.84
c_{27}	0.5	14	8.61	c_{55}	0.715	18	11.07
c_{28}	0.96	9	5.535	c_{56}	0.8	22	13.53

generated using *llvm-prof*. Finally, *opt-analyze* was used to print the CFGs to .dot files. We ran the profiling considering several audio files (.au) as input. The results of this step revealed that many checkers (35 out of the 56) were placed in loops and executed on average as much as 375354 times, which suggests that it is important to place them in HW.

Using the information generated we ran our optimization algorithm. The results obtained are shown in Figure 5.24a. It can be seen that the results follow trends similar to those for the synthetic experiments. Our optimization generated results within 27% of the HW-only solution with just 15% hardware fraction. Also note that for hardware fractions between 15% and 40%, the solution generated by our algorithm (OPT) was roughly 2.5 times

(a) Performance improvement

(b) Reconfiguration schedule table size

Figure 5.24: Results for the GSM encoder case study

closer to the HW-only solution than that generated by the straight-forward algorithm (SF). Finally, Figure 5.24b shows the sizes for the reconfiguration tables.

5.4 Summary

This chapter proposed techniques to optimize the hardware/software implementation of error detection mechanisms. First we developed system-level optimization algorithms that perform HW/SW codesign in order to minimize the global worst-case schedule length of a safety-critical application, while meeting the imposed hardware cost constraints and tolerating multiple transient faults. Both statically reconfigurable FPGAs and partially dynamically reconfigurable FPGAs have been considered. In the second part of this chapter we presented an algorithm for performance optimization of error detection based on speculative reconfiguration. We minimize the average execution time of a program by using partially reconfigurable FPGAs to place in hardware only those error detection components that provide the highest performance improvement.

CONCLUSIONS AND FUTURE WORK

IN THIS chapter we shall summarize the research contributions of this thesis and discuss possible directions of future work.

6.1 Conclusions

In order to meet the stringent demands of modern applications and keep costs within reasonable limits, today's engineers are faced with difficult design optimization problems. Often, they deal with multi-objective optimizations, having to deliver solutions that provide high performance as well as flexibility, energy-efficiency as well as real-time features, fault tolerance as well as low cost. The use of reconfigurable and heterogeneous architectures is one approach to deal with these demanding requirements. However, in order to leverage the advantages of such platforms and deliver efficient solutions, the engineers and designers of modern embedded systems need to have available tools and methodologies. It is the responsibility of the research community to develop these tools and to explore the advantages and possible uses of new technologies.

In this thesis we have presented several approaches to the hardware/software codesign and optimization of adaptive real-time systems implemented on reconfigurable and heterogeneous platforms. We have addressed topics that are of interest to the industry: performance enhancement using static and dynamic FPGA configuration prefetching (Chapter 3), energy optimization for multi-mode real-time systems implemented on platforms composed of CPUs, GPUs and FPGAs (Chapter 4), and hardware/software codesign of fault-tolerant safety-critical systems (Chapter 5).

6.1.1 FPGA Configuration Prefetching

In Chapter 3 we proposed a complete framework for partial dynamic reconfiguration of FPGAs, together with optimization algorithms for static and

dynamic FPGA configuration prefetching. The framework is modular and permits rapid integration of user applications with minimal effort from the designer. We proposed an IP-based architecture, together with a comprehensive API, that hides the low-level details from the programmer and can be used to accelerate applications using partial dynamic reconfiguration of FPGAs.

Based on this framework, we next proposed two algorithms for FPGA configuration prefetching:

1. The first one schedules prefetches at design-time and simultaneously performs hardware/software partitioning in order to minimize the expected execution time of an application. The algorithm performs speculative FPGA configuration prefetching based on profiling information in order to reduce the reconfiguration penalty by overlapping FPGA reconfigurations with useful computations.
2. The second algorithm targets applications that exhibit a dynamic and non-stationary behavior. In such cases we can not obtain accurate profiling information, so we need approaches that are able to adapt to the run-time conditions. Our technique uses a piecewise linear predictor that captures correlations in order to launch prefetches for those hardware modules that will provide the highest performance improvements.

The framework and our optimization approaches have been tested using extensive simulations and a proof of concept implementation on a platform from Xilinx.

6.1.2 Multi-Mode Systems

In Chapter 4 we addressed the problem of energy optimization for multi-mode real-time systems. We characterize a functional mode by the composition of the active task set (containing the tasks that are currently releasing periodic jobs). In order to meet the real-time requirements of such dynamic systems and keep a low energy consumption, we propose intelligent on-line resource management algorithms. The resource manager implements run-time policies to decide on-the-fly task admission and the mapping of active tasks to the available resources in a heterogeneous platform composed of CPUs, GPUs and FPGAs. The experimental validation, including both real-life measurements and simulations, has showed the practicality and efficiency of our approach.

6.1.3 Fault-Tolerant Systems

In Chapter 5 we addressed the problem of optimizing the error detection implementation for fault-tolerant safety-critical systems. In order to provide resiliency against transient faults, error detection has to be employed, but

this incurs high performance and cost overheads. In order to deal with this issue, we proposed two optimization strategies:

1. The first one targets real-time applications running on a distributed embedded architecture and addresses the problem at a task level. We decide on the hardware/software implementation of the error detection mechanisms for each task in order to minimize the global worst-case schedule length of the whole application, while meeting the imposed hardware cost constraints and tolerating multiple faults.
2. The second algorithm presents a method to minimize the average execution time of an application composed of a single sequential task. By analyzing the internal structure and properties of the application, and by using the advantages of partial dynamic reconfiguration of FPGAs, we speculatively place in hardware those error detection components that will result in a minimal performance overhead.

Both optimization approaches have been tested using extensive experiments, including real-life case studies.

6.2 Future Work

The entire work presented in this thesis constitutes a solid basis that has a big potential for future extensions, improvements and applications. In this section we will mention some specific points that can be addressed:

- For a large class of applications, optimizing the worst-case behavior is not important, while optimizing for the average is not enough. Instead, statistical guarantees are desirable, like, e.g., optimizing the α -percentile. An interesting direction of future work would be to extend our configuration prefetching strategies presented in Chapter 3 for such a case.
- In Chapter 3 we presented configuration prefetching algorithms to minimize the FPGA reconfiguration penalties. Other techniques that address the same problem are configuration compression and caching. Since they are complementary to the approaches we presented, it would be interesting to investigate holistic solutions that combine the three methods to obtain the best results.
- In the context of multi-mode systems, introducing soft real-time tasks into the model would be very interesting. There exist many application areas in which hard real-time tasks coexist with soft real-time ones, or with tasks that only have certain quality of service demands (but no strict deadlines). Extending the optimizations presented in Chapter 4 to account for such scenarios would be an important research contribution.

- The faults that affect a system might be transient, intermittent or permanent. In Chapter 5 we addressed only the case of transient and intermittent faults. An interesting area of future work would be to consider permanent faults and use the capabilities of partial dynamic reconfiguration of FPGAs to optimize for such scenarios.
- In the context of fault-tolerant systems, we have performed optimization of the error detection implementation. One possible direction of future work is to combine this technique with the prefetching approaches presented in Chapter 3, in order to optimize both the error detection mechanisms and the original application.

BIBLIOGRAPHY

- [ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.
- [ALT08] Joshua A. Anderson, Chris D. Lorenz, and A. Travasset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008.
- [Alt09] Altera. Taking advantage of advances in FPGA floating-point IP cores - white paper 01116. 2009.
- [Alt12] Altera. Using the Stratix V reconfiguration controller to perform dynamic reconfiguration - application note an-664-1.0. 2012.
- [AMHN05] M. Akerholm, A. Moller, H. Hansson, and M. Nolin. Towards a dependable component technology for embedded system applications. *10th IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems*, pages 320–328, 2005.
- [ASG13] A. Ashrafi, Antonio G. M. Strollo, and O. Gustafsson. Hardware implementation of digital signal processing algorithms. *J. Electrical and Computer Engineering*, 2013:782575:1–782575:2, 2013.
- [BBD05] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration. *42nd Design Automation Conference*, pages 335–340, 2005.
- [BBG13] Mehmet E. Belviranli, Laxmi N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiproces-

- processor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013.
- [BBZT14] A. Becher, F. Bauer, D. Ziener, and J. Teich. Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration. *24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.
- [BGFM06] Jason A. Blome, S. Gupta, S. Feng, and S. Mahlke. Cost-efficient soft error protection for embedded microprocessors. *Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, pages 421–431, 2006.
- [BGG13] P.P. Boopal, M. Garrido, and O. Gustafsson. A reconfigurable FFT architecture for variable-length and multi-streaming OFDM standards. *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2066–2070, 2013.
- [BHS⁺13] Rana M. Bilal, R. Hafiz, M. Shafique, S. Shoaib, A. Munawar, and J. Henkel. Isomer: integrated selection, partitioning, and placement methodology for reconfigurable architectures. *International Conference on Computer-Aided Design (ICCAD)*, pages 755–762, 2013.
- [BJRK⁺03] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan. A self-reconfiguring platform. *Proceedings of Field Programmable Logic and Applications*, pages 565–574, 2003.
- [BLBY11] M. Bogdanski, P.R. Lewis, T. Becker, and Xin Yao. Improving scheduling techniques in heterogeneous systems with dynamic, on-line optimisations. *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 496–501, 2011.
- [BMA⁺05] C. Bobda, M. Majer, A. Ahmadiania, T. Haller, A. Linarth, and J. Teich. The Erlangen slot machine: a highly flexible FPGA-based reconfigurable platform. *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 319–320, 2005.
- [BMR⁺08] C. Bolchini, A. Miele, M. Rebaudengo, F. Salice, D. Sciuto, L. Sterpone, and M. Violante. Software and hardware techniques for SEU detection in IP processors. *J. Electron. Test.*, 24(1-3):35–44, 2008.
- [BP05] Z.K. Baker and V.K. Prasanna. Efficient hardware data mining with the apriori algorithm on FPGAs. *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 2005.

- [BPCF13] J. Bispo, N. Paulino, J.M.P. Cardoso, and J.C. Ferreira. Transparent trace-based binary acceleration for reconfigurable HW/SW systems. *IEEE Transactions on Industrial Informatics*, 9(3):1625–1634, 2013.
- [BPPC12] R. Bonamy, H.-M. Pham, S. Pillement, and D. Chillet. UPaRC – ultra-fast power-aware reconfiguration controller. *Design, Automation & Test in Europe (DATE)*, pages 1373–1378, 2012.
- [BSY⁺10] K. Bertels, V.M. Sima, Y. Yankova, G. Kuzmanov, W. Luk, José Gabriel F. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, and A. Michelotti. HArtes: Hardware-software codesign for heterogeneous multicore platforms. *IEEE Micro*, 30(5):88–97, 2010.
- [But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.
- [CBP04] The 1st championship branch prediction competition. *Journal of Instruction-Level Parallelism*, 2004. <http://www.jilp.org/cbp>.
- [CBR⁺14] Juan A. Clemente, I. Beretta, V. Rana, D. Atienza Alonso, and D. Sciuto. A mapping-scheduling algorithm for hardware acceleration on reconfigurable platforms. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(1):1–27, 2014.
- [CLS⁺08] S. Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. *Symposium on Application Specific Processors*, pages 101–107, 2008.
- [CMHM10] E.S. Chung, P.A. Milder, J.C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 225–236, 2010.
- [Con03] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.
- [CRM14] Juan A. Clemente, J. Resano, and D. Mozos. An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):90:1–90:24, 2014.

- [CRR⁺09] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto. Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(5):662–675, 2009.
- [DML11] F. Duhem, F. Muller, and P. Lorenzini. FaRM: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA. Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek A. El-Ghazawi, editors, *ARC*, volume 6578 of *Lecture Notes in Computer Science*, pages 253–260. Springer, 2011.
- [dRGLH06] B. de Ruijsscher, G.N. Gaydadjiev, J. Lichtenauer, and E. Hendriks. FPGA accelerator for real-time skin segmentation. *IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 93–97, 2006.
- [DSB⁺13] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Throughput-constrained DVFS for scenario-aware dataflow graphs. *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 175–184, 2013.
- [EA11] G.A. Elliott and J.H. Anderson. Real-world constraints of GPUs in real-time systems. *IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, volume 2, pages 48–54, 2011.
- [EA12] G.A. Elliott and J.H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Syst.*, 48(1):34–74, 2012.
- [EDPP00] P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):472–491, 2000.
- [EWA13] G.A. Elliott, B.C. Ward, and J.H. Anderson. GPUSync: A framework for real-time GPU management. *IEEE 34th Real-Time Systems Symposium (RTSS)*, pages 33–44, 2013.
- [Fou15] HSA Foundation. Harmonizing the industry around heterogeneous computing. 2015. <http://www.hsafoundation.com/>.
- [GGPM11] Junhe Gan, F. Gruian, P. Pop, and J. Madsen. Energy/reliability trade-offs in fault-tolerant event-triggered distributed embedded systems. *16th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 731–736, 2011.

- [GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, New York, NY, USA, 1990.
- [Hau98] Scott Hauck. Configuration prefetch for single context reconfigurable coprocessors. *Intl. Symp. on Field Programmable Gate Arrays*, 1998.
- [HHSC10] P.-A. Hsiung, C.-H. Huang, J.-S. Shen, and C.-C. Chiang. Scheduling and placement of hardware/software real-time relocatable tasks in dynamically partially reconfigurable systems. *ACM Trans. Reconfigurable Technol. Syst.*, 4(1):9:1–9:32, 2010.
- [HJL⁺07] B. Harris, A.C. Jacob, J.M. Lancaster, J. Buhler, and R.D. Chamberlain. A banded Smith-Waterman FPGA accelerator for Mercury BLASTP. *International Conference on Field Programmable Logic and Applications*, pages 765–769, 2007.
- [hKYS⁺12] S.-H. Kang, H. Yang, L. Schor, I. Bacivarov, Soonhoi Ha, and L. Thiele. Multi-objective mapping optimization via problem decomposition for many-core systems. *IEEE 10th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 28–37, 2012.
- [HLD⁺09] Jie Hu, Feihui Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and Mary J. Irwin. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Trans. Embed. Comput. Syst.*, 2009., 8(4):1–30, 2009.
- [HLS11] Y. He, Jie Liu, and Hongyang Sun. Scheduling functionally heterogeneous systems with utilization balancing. *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 1187–1198, 2011.
- [HMA⁺01] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and Changhong Dai. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. *Symposium on VLSI Technology. Digest of Technical Papers*, pages 73–74, 2001.
- [HMZB12] Ruining He, Yuchun Ma, K. Zhao, and J. Bian. ISBA: An independent set-based algorithm for automated partial reconfiguration module generation. *International Conference on Computer Aided Design*, 2012.
- [HV04] M Handa and R Vemuri. Area fragmentation in reconfigurable operating systems. *Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms*, 2004.

- [HV08] C. Huang and F. Vahid. Dynamic coprocessor management for FPGA-enhanced compute platforms. *Proceedings of the international conference on compilers, architectures and synthesis for embedded systems*, pages 71–78, New York, NY, USA, 2008.
- [HV09] C. Huang and F. Vahid. Transmuting coprocessors: Dynamic loading of FPGA coprocessors. *Design Automation Conference*, pages 848–851, 2009.
- [HWC14] M. Hollander, Douglas A. Wolfe, and E. Chicke, editors. *Non-parametric Statistical Methods, 3rd Edition*. Wiley, 2014.
- [IBM09] IBM. IBM ILOG CPLEX v12.1 - user’s manual for CPLEX. 2009.
- [IPEP05] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems. *Design, Automation & Test in Europe*, pages 864–869, 2005.
- [IPEP06] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems. *Design, Automation & Test in Europe*, pages 1–6, 2006.
- [IPEP08] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Scheduling of fault-tolerant embedded systems with soft and hard timing constraints. *Design, Automation & Test in Europe*, pages 915–920, 2008.
- [IPP⁺09] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng. Analysis and optimization of fault-tolerant embedded systems with hardened processors. *Design, Automation & Test in Europe*, pages 682–687, 2009.
- [JEP13] Ke Jiang, P. Eles, and Z. Peng. Optimization of secure embedded systems with dynamic task sets. *Design, Automation & Test in Europe*, pages 1765–1770, 2013.
- [Jim05] D.A. Jimenez. Piecewise linear branch prediction. *32nd International Symposium on Computer Architecture*, pages 382–393, 2005.
- [JLE⁺13] Ke Jiang, A. Lifa, P. Eles, Z. Peng, and Wei Jiang. Energy-aware design of secure multi-mode real-time embedded systems with FPGA co-processors. *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 109–118, New York, NY, USA, 2013.

- [JLK⁺14] H. Jung, C. Lee, S.-H. Kang, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. Dynamic behavior specification and dynamic mapping for real-time embedded systems: HOPES approach. *ACM Trans. Embed. Comput. Syst.*, 13(4s):135:1–135:26, 2014.
- [JPT⁺10] Zai Jian Jia, A.D. Pimentel, M. Thompson, T. Bautista, and A. Nunez. NASA: A generic infrastructure for system-level MP-SoC design space exploration. *8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 41–50, 2010.
- [JTY⁺99] J. S. N. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook. Dynamic reconfiguration to support concurrent applications. *IEEE Transactions on Computers*, 48(6), 1999.
- [KB03] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
- [KFP⁺15] B. Khemka, R. Friese, S. Pasricha, Anthony A. Maciejewski, Howard J. Siegel, Gregory A. Koenig, S. Powers, M. Hilton, R. Rambharos, and S. Poole. Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system. *Sustainable Computing: Informatics and Systems*, 5(0):14 – 30, 2015.
- [KHM03] N. Kandasamy, J. P. Hayes, and B. T. Murray. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Transactions on Computers*, 52(2):113–125, 2003.
- [KLRI11] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2011.
- [Koc13] Dirk Koch. *Partial Reconfiguration on FPGAs - Architectures, Tools and Applications*, volume 153 of *Lecture Notes in Electrical Engineering*. Springer, 2013.
- [Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2011.
- [LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Intl. Symp. on Code Generation and Optimization*, 2004.

- [LCD⁺00] Yanbing Li, T. Callahan, E. Darnell, Randolph E. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. *37th Design Automation Conference*, pages 507–512, 2000.
- [LCP⁺09] G. Lyle, S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. An end-to-end approach for the automatic derivation of application-aware error detectors. *IEEE/IFIP Intl. Conf. on Dependable Systems & Networks*, pages 584–589, 2009.
- [LCR03] F. Lima, L. Carro, and R. Reis. Designing fault tolerant systems into SRAM-based FPGAs. *40th Design Automation Conference*, pages 650–655, 2003.
- [LEP11] A. Lifa, P. Eles, and Z. Peng. Performance optimization of error detection based on speculative reconfiguration. *Design Automation Conference (DAC)*, pages 369–374, 2011.
- [LEP12a] A. Lifa, P. Eles, and Z. Peng. Context-aware speculative prefetch for soft real-time applications. *IEEE 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 174–183, 2012.
- [LEP12b] A. Lifa, P. Eles, and Z. Peng. Minimization of average execution time based on speculative FPGA configuration prefetch. *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2012.
- [LEP12c] A. Lifa, P. Eles, and Z. Peng. Execution time minimization based on hardware/software partitioning and speculative prefetch. Technical Report 2012:11, Linköping University, Software and Systems, 2012.
- [LEP13] A. Lifa, P. Eles, and Z. Peng. Dynamic configuration prefetching based on piecewise linear prediction. *Design, Automation & Test in Europe (DATE)*, pages 815–820, 2013.
- [LEP15a] A. Lifa, P. Eles, and Z. Peng. A reconfigurable framework for performance enhancement with dynamic FPGA configuration prefetching. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015.
- [LEP15b] A. Lifa, P. Eles, and Z. Peng. On-the-fly energy minimization for multi-mode real-time systems on heterogeneous platforms. *IEEE Symp. on Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2015.
- [LEPI10] A. Lifa, P. Eles, Z. Peng, and V. Izosimov. Hardware/software optimization of error detection implementation for real-time

- embedded systems. *Intl. Conf. on Hardware/Software Codesign and System Synthesis*, pages 40–50, 2010.
- [LH02] Zhiyuan Li and Scott Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. *Intl. Symp. on Field Programmable Gate Arrays*, 2002.
- [LHK09] Chi-Keung Luk, S. Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009.
- [Li02] Zhiyuan Li. Configuration management techniques for reconfigurable computing, 2002. PhD thesis, Northwestern Univ., Evanston, IL.
- [LKLJ09] Ming Liu, W. Kuehn, Zhonghai Lu, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. *International Conference on Field Programmable Logic and Applications*, pages 498–502, 2009.
- [Mar10] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2010.
- [MBH⁺14] A. Maghazeh, U.D. Bordoloi, A. Horga, P. Eles, and Z. Peng. Saving energy without defying deadlines on mobile GPU-based heterogeneous systems. *International Conference on Hardware/Software Codesign and System Synthesis*, pages 8:1–8:10, New York, NY, USA, 2014.
- [MBT04] A. Maheshwari, W. Burleson, and R. Tessier. Trading off transient fault tolerance and power consumption in deep submicron (DSM) VLSI circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):299–311, 2004.
- [MEP05] S. Manolache, P. Eles, and Z. Peng. Fault and energy-aware communication mapping with guaranteed latency for applications implemented on NoC. *Design Automation Conference*, pages 266–269, 2005.
- [ML99] G. McGregor and P. Lysaght. Self controlling dynamic reconfiguration: A case study. *Field Programmable Logic and Applications*, pages 144–154. Springer, 1999.
- [MLH⁺12] R. Membarth, J.-H. Lupp, F. Hannig, J. Teich, M. Körner, and W. Eckert. Dynamic task-scheduling and resource management

- for GPU accelerators in medical imaging. *25th International Conference on Architecture of Computing Systems*, pages 147–159, 2012.
- [MSP⁺12] G. Mariani, V. Sima, G. Palermo, V. Zaccaria, C. Silvano, and K. Bertels. Using multi-objective design space exploration to enable run-time resource management for reconfigurable architectures. *Design, Automation & Test in Europe*, pages 1379–1384, 2012.
- [MT90] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
- [NOS13] B. Navas, J. Oberg, and I. Sander. Towards the generic reconfigurable accelerator: Algorithm development, core design, and performance analysis. *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2013.
- [NOS14] B. Navas, J. Oberg, and I. Sander. On providing scalable self-healing adaptive fault-tolerance to RTR SoCs. *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2014.
- [NSO⁺12] R. Nane, Vlad M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. *International Conference on Field-programmable Logic and Applications (FPL)*, pages 619–622, 2012.
- [NSO13] B. Navas, I. Sander, and J. Öberg. The RecoBlock SoC platform: A flexible array of reusable run-time-reconfigurable IP-blocks. *Design, Automation & Test in Europe*, pages 833–838, 2013.
- [PBV05] Elena M. Panainte, K. Bertels, and S. Vassiliadis. Instruction scheduling for dynamic hardware configurations. *Design, Automation & Test in Europe*, 2005.
- [PBV06] Elena M. Panainte, K. Bertels, and S. Vassiliadis. Interprocedural compiler optimization for partial run-time reconfiguration. *The Journal of VLSI Signal Processing*, 43(2), 2006.
- [PKI05] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Application-based metrics for strategic placement of detectors. *11th Pacific Rim Intl. Symp. on Dependable Computing*, 2005.
- [PKI11] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. Automated derivation of application-aware error detectors using static analysis: The trusted Illiac approach. *IEEE Transactions on Dependable and Secure Computing*, 8(1):44–57, 2011.

- [PRP⁺15] E. Paone, F. Robino, G. Palermo, V. Zaccaria, I. Sander, and C. Silvano. Customization of OpenCL applications for efficient task mapping under heterogeneous platform constraints. *Design, Automation & Test in Europe*, pages 736–741, 2015.
- [PSA10] T. Pionteck, S. Sammann, and C. Albrecht. Optimizing runtime reconfiguration decisions. *IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 39–46, 2010.
- [PTW10] Marco Platzner, Jrgen Teich, and Norbert Wehn, editors. *Dynamically Reconfigurable Systems*. Springer, 2010.
- [QP13a] Wei Quan and A.D. Pimentel. An iterative multi-application mapping algorithm for heterogeneous MPSoCs. *IEEE 11th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 115–124, 2013.
- [QP13b] Wei Quan and A.D. Pimentel. A scenario-based run-time task mapping algorithm for MPSoCs. *Design Automation Conference (DAC)*, pages 1–6, 2013.
- [R⁺05] J. Resano, D. Mozos, and F. Catthoor. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. *Design, Automation & Test in Europe*, 2005.
- [RB93] Colin R. Reeves and John E. Beasley. *Modern Heuristic Techniques For Combinatorial Problems*. John Wiley & Sons, Inc, New York, NY, USA, 1993.
- [RCG⁺08] J. Resano, Juan A. Clemente, C. Gonzalez, D. Mozos, and F. Catthoor. Efficiently scheduling runtime reconfigurations. *ACM Trans. Des. Autom. Electron. Syst.*, 13(4):58:1–58:12, 2008.
- [RCV⁺05] George A. Reis, J. Chang, N. Vachharajani, R. Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2(4):366–396, 2005.
- [S⁺10] Joon E. Sim, W.-F. Wong, G. Walla, T. Ziermann, and J. Teich. Interprocedural placement-aware configuration prefetching for FPGA-based systems. *IEEE Symp. on Field-Programmable Custom Computing Machines*, 2010.
- [SAHE05] M.T. Schmitz, B.M. Al-Hashimi, and P. Eles. Cosynthesis of energy-efficient multimode embedded systems with consideration of mode-execution probabilities. *IEEE Transactions*

- on *Computer-Aided Design of Integrated Circuits and Systems*, 24(2):153–169, 2005.
- [Sar89] V. Sarkar. Determining average program execution times and their variance. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, New York, NY, USA, 1989.
- [SB97] Stephen M. Smith and J. Michael Brady. SUSAN — a new approach to low level image processing. *International Journal of Computer Vision*, 23(1):45–78, 1997.
- [SBB⁺06] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in Virtex FPGAs. *IEEE Proceedings - Computers and Digital Techniques*, 153(3):157 – 164, 2006.
- [SBE⁺12] S. Samii, U.D. Bordoloi, P. Eles, Z. Peng, and A. Cervin. Control-quality optimization for distributed embedded systems with adaptive fault tolerance. *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 68–77, 2012.
- [SBH14] M. Shafique, L. Bauer, and J. Henkel. Adaptive energy management for dynamically reconfigurable processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(1):50–63, 2014.
- [SBHT13] M. Schmid, M. Blocherer, F. Hannig, and J. Teich. Real-time range image preprocessing on FPGAs. *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2013.
- [SBR⁺12] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. *15th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 71–80, 2012.
- [SBSH12] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase behavior in serial and parallel applications. *IEEE International Symposium on Workload Characterization (IISWC)*, pages 47–58, 2012.
- [SEPC09] S. Samii, P. Eles, Z. Peng, and A. Cervin. Quality-driven synthesis of embedded multi-mode control systems. *Design Automation Conference*, pages 864–869, 2009.

- [SGO⁺09] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 175–184, 2009.
- [SHTT13] M. Schmid, F. Hannig, A. Tanase, and J. Teich. High-level synthesis revised - generation of FPGA accelerators from a domain-specific language using the polyhedron model. *International Conference on Parallel Computing*, pages 497–506, 2013.
- [SPM10] P.K. Saraswat, P. Pop, and J. Madsen. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 89–98, 2010.
- [SSC03] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. *Proceedings of the 30th annual international symposium on Computer architecture (ISCA)*, pages 336–349, New York, NY, USA, 2003.
- [TBEP10] B. Tanasa, U.D. Bordoloi, P. Eles, and Z. Peng. Scheduling for fault-tolerant communication on the static segment of Flexray. *Real-Time Systems Symposium (RTSS)*, pages 385–394, 2010.
- [TBEP11] B. Tanasa, U.D. Bordoloi, P. Eles, and Z. Peng. Reliability-aware frame packing for the static segment of Flexray. *International Conference on Embedded Software (EMSOFT)*, pages 175–184, 2011.
- [TK11] Jim Torresen and Dirk Koch. Can run-time reconfigurable hardware be more accessible? *Intl. Conf. on Engineering of Reconfigurable Systems & Algorithms*, 2011.
- [TL10] Kuen H. Tsoi and Wayne Luk. Axel: A heterogeneous cluster with FPGAs and GPUs. *18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 115–124, New York, NY, USA, 2010.
- [VMS14] U. Verner, A. Mendelson, and A. Schuster. Batch method for efficient resource sharing in real-time multi-GPU systems. *15th International Conference on Distributed Computing and Networking*, pages 347–362, 2014.
- [WAST12] S. Wildermann, J. Angermeier, E. Sibirko, and J. Teich. Placing multimode streaming applications on dynamically partially reconfigurable architectures. *Int. J. Reconfig. Comp.*, 2012.
- [WEE⁺08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra,

- F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008., 7(3):1–53, 2008.
- [WJR⁺03] M. Wirthlin, E. Johnson, N. Rollins, M. Caffrey, and P. Graham. The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets. *11th IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 133–142, 2003.
- [WPQ⁺12] Z. Wang, Geguang Pu, Shengchao Qin, Jianwen Li, Kim G. Larsen, J. Madsen, Bin Gu, and Jifeng He. MDM: A mode diagram modeling framework for periodic control systems. *Computing Research Repository (CoRR)*, 2012.
- [WRZT13] S. Wildermann, F. Reimann, D. Ziener, and J. Teich. Symbolic system-level design methodology for multi-mode reconfigurable systems. *Design Autom. for Emb. Sys.*, 17(2):343–375, 2013.
- [Xil12] Xilinx. Partial reconfiguration user guide ug702. 2012.
- [ZSJ10] Jun Zhu, I. Sander, and A. Jantsch. Pareto efficient design for reconfigurable streaming applications on CPU/FPGAs. *Design, Automation & Test in Europe (DATE)*, pages 1035–1040, 2010.

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Science
Linköping Studies in Statistics
Linköpings Studies in Information Science

Linköping Studies in Science and Technology

- No 14 **Anders Haraldsson**: A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.
- No 17 **Bengt Mågnhagen**: Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.
- No 18 **Mats Cedwall**: Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.
- No 22 **Jaak Urmi**: A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.
- No 33 **Tore Risch**: Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.
- No 51 **Erlend Jungert**: Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.
- No 54 **Sture Hägglund**: Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.
- No 55 **Pär Emanuelson**: Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.
- No 58 **Bengt Johnsson, Bertil Andersson**: The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.
- No 69 **H. Jan Komorowski**: A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.
- No 71 **René Reboh**: Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.
- No 77 **Östen Oskarsson**: Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.
- No 94 **Hans Lunell**: Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.
- No 97 **Andrzej Lingas**: Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.
- No 109 **Peter Fritzson**: Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.
- No 111 **Erik Tengvald**: The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.
- No 155 **Christos Levcopoulos**: Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.
- No 165 **James W. Goodwin**: A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.
- No 170 **Zebo Peng**: A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.
- No 174 **Johan Fagerström**: A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.
- No 192 **Dimitar Driankov**: Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

- No 213 **Lin Padgham**: Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.
- No 214 **Tony Larsson**: A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.
- No 221 **Michael Reinfrank**: Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.
- No 239 **Jonas Löwgren**: Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.
- No 244 **Henrik Eriksson**: Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.
- No 252 **Peter Eklund**: An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.
- No 258 **Patrick Doherty**: NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.
- No 260 **Nahid Shahmehri**: Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.
- No 264 **Nils Dahlbäck**: Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.
- No 265 **Ulf Nilsson**: Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.
- No 270 **Ralph Rönquist**: Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.
- No 273 **Björn Fjellborg**: Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.
- No 276 **Staffan Bonnier**: A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.
- No 277 **Kristian Sandahl**: Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.
- No 281 **Christer Bäckström**: Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.
- No 292 **Mats Wirén**: Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.
- No 297 **Mariam Kamkar**: Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.
- No 302 **Tingting Zhang**: A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.
- No 312 **Arne Jönsson**: Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.
- No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.
- No 371 **Bengt Savén**: Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

- No 375 **Ulf Söderman**: Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.
- No 383 **Andreas Kågedal**: Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.
- No 396 **George Fodor**: Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.
- No 413 **Mikael Pettersson**: Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.
- No 414 **Xinli Gu**: RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.
- No 416 **Hua Shu**: Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.
- No 429 **Jaime Villegas**: Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.
- No 431 **Peter Jonsson**: Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.
- No 437 **Johan Boye**: Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.
- No 439 **Cecilia Sjöberg**: Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.
- No 448 **Patrick Lambrix**: Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.
- No 452 **Kjell Orsborn**: On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.
- No 459 **Olof Johansson**: Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.
- No 461 **Lena Strömbäck**: User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.
- No 462 **Lars Degerstedt**: Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.
- No 475 **Fredrik Nilsson**: Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.
- No 480 **Mikael Lindvall**: An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.
- No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.
- No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.
- No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.
- No 498 **Thomas Drakengren**: Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.
- No 502 **Jakob Axelsson**: Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.
- No 503 **Johan Ringström**: Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.
- No 512 **Anna Moberg**: Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.
- No 520 **Mikael Ronström**: Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.
- No 522 **Nidas Ohlsson**: Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.
- No 526 **Joachim Karlsson**: A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.
- No 530 **Henrik Nilsson**: Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.
- No 555 **Jonas Hallberg**: Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.
- No 561 **Ling Lin**: Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.
- No 563 **Eva L Ragnemalm**: Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.
- No 567 **Jörgen Lindström**: Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.
- No 582 **Vanja Josifovski**: Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.
- No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.
- No 592 **Mikael Ericsson**: Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.
- No 593 **Lars Karlsson**: Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.
- No 594 **C. G. Mikael Johansson**: Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.
- No 595 **Jörgen Hansson**: Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.
- No 596 **Niklas Hallberg**: Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.
- No 597 **Vivian Vimarlund**: An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.
- No 598 **Johan Jönvald**: Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.
- No 607 **Magnus Merkel**: Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.
- No 611 **Silvia Coradeschi**: Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.
- No 613 **Man Lin**: Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.
- No 618 **Jimmy Tjäder**: Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.
- No 627 **Vadim Engelson**: Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

- No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.
- No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.
- No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.
- No 688 **Marcus Bjärelund:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.
- No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.
- No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN 91-7373-126-9.
- No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.
- No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.
- No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.
- No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.
- No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.
- No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.
- No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.
- No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.
- No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.
- No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.
- No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.
- No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.
- No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.
- No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.
- No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.
- No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.
- No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.
- No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems' Development, 2003, ISBN 91-7373-604-X.
- No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.
- No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.
- No 823 **Christina Ölvingsson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.
- No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.
- No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.
- No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.
- No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.
- No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.
- No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.
- No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.
- No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.
- No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.
- No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.
- No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5.
- No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.
- No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.
- No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.
- No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.
- No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.
- No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.
- No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.
- No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.
- No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

- No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.
- No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.
- No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.
- No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.
- No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.
- No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.
- No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.
- No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.
- No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.
- No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.
- No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.
- No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.
- No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.
- No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.
- No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.
- No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.
- No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.
- No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.
- No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8.
- No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.
- No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.
- No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.
- No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.
- No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.
- No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.
- No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.
- No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.
- No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.
- No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.
- No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.
- No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.
- No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.
- No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.
- No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.
- No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.
- No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.
- No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.
- No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.
- No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.
- No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.
- No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.
- No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.
- No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.
- No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.
- No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.
- No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.
- No 1155 **Beatrice Ailenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

- No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.
- No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.
- No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.
- No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.
- No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.
- No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.
- No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.
- No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.
- No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.
- No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.
- No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.
- No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.
- No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.
- No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.
- No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.
- No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.
- No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.
- No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.
- No 1294 **Johan Thapper:** Aspects of a Constraint Optimisation Problem, 2010, ISBN 978-91-7393-464-0.
- No 1306 **Susanna Nilsson:** Augmentation in the Wild: User Centered Development and Evaluation of Augmented Reality Applications, 2010, ISBN 978-91-7393-416-9.
- No 1313 **Christer Thörn:** On the Quality of Feature Models, 2010, ISBN 978-91-7393-394-0.
- No 1321 **Zhiyuan He:** Temperature Aware and Defect-Probability Driven Test Scheduling for System-on-Chip, 2010, ISBN 978-91-7393-378-0.
- No 1333 **David Broman:** Meta-Languages and Semantics for Equation-Based Modeling and Simulation, 2010, ISBN 978-91-7393-335-3.
- No 1337 **Alexander Siemers:** Contributions to Modelling and Visualisation of Multibody Systems Simulations with Detailed Contact Analysis, 2010, ISBN 978-91-7393-317-9.
- No 1354 **Mikael Asplund:** Disconnected Discoveries: Availability Studies in Partitioned Networks, 2010, ISBN 978-91-7393-278-3.
- No 1359 **Jana Rambusch:** Mind Games Extended: Understanding Gameplay as Situated Activity, 2010, ISBN 978-91-7393-252-3.
- No 1373 **Sonia Sangari:** Head Movement Correlates to Focus Assignment in Swedish, 2011, ISBN 978-91-7393-154-0.
- No 1374 **Jan-Erik Källhammer:** Using False Alarms when Developing Automotive Active Safety Systems, 2011, ISBN 978-91-7393-153-3.
- No 1375 **Mattias Eriksson:** Integrated Code Generation, 2011, ISBN 978-91-7393-147-2.
- No 1381 **Ola Leifler:** Affordances and Constraints of Intelligent Decision Support for Military Command and Control - Three Case Studies of Support Systems, 2011, ISBN 978-91-7393-133-5.
- No 1386 **Soheil Samii:** Quality-Driven Synthesis and Optimization of Embedded Control Systems, 2011, ISBN 978-91-7393-102-1.
- No 1419 **Erik Kuiper:** Geographic Routing in Intermittently-connected Mobile Ad Hoc Networks: Algorithms and Performance Models, 2012, ISBN 978-91-7519-981-8.
- No 1451 **Sara Stymne:** Text Harmonization Strategies for Phrase-Based Statistical Machine Translation, 2012, ISBN 978-91-7519-887-3.
- No 1455 **Alberto Montebelli:** Modeling the Role of Energy Management in Embodied Cognition, 2012, ISBN 978-91-7519-882-8.
- No 1465 **Mohammad Saifullah:** Biologically-Based Interactive Neural Network Models for Visual Attention and Object Recognition, 2012, ISBN 978-91-7519-838-5.
- No 1490 **Tomas Bengtsson:** Testing and Logic Optimization Techniques for Systems on Chip, 2012, ISBN 978-91-7519-742-5.
- No 1481 **David Byers:** Improving Software Security by Preventing Known Vulnerabilities, 2012, ISBN 978-91-7519-784-5.
- No 1496 **Tommy Färnqvist:** Exploiting Structure in CSP-related Problems, 2013, ISBN 978-91-7519-711-1.
- No 1503 **John Wilander:** Contributions to Specification, Implementation, and Execution of Secure Software, 2013, ISBN 978-91-7519-681-7.
- No 1506 **Magnus Ingmarsson:** Creating and Enabling the Useful Service Discovery Experience, 2013, ISBN 978-91-7519-662-6.
- No 1547 **Wladimir Schamai:** Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool, 2013, ISBN 978-91-7519-505-6.
- No 1551 **Henrik Svensson:** Simulations, 2013, ISBN 978-91-7519-491-2.
- No 1559 **Sergiu Rafiliiu:** Stability of Adaptive Distributed Real-Time Systems with Dynamic Resource Management, 2013, ISBN 978-91-7519-471-4.
- No 1581 **Usman Dastgeer:** Performance-aware Component Composition for GPU-based Systems, 2014, ISBN 978-91-7519-383-0.
- No 1602 **Cai Li:** Reinforcement Learning of Locomotion based on Central Pattern Generators, 2014, ISBN 978-91-7519-313-7.
- No 1652 **Roland Samlaus:** An Integrated Development Environment with Enhanced Domain-Specific

Interactive Model Validation, 2015, ISBN 978-91-7519-090-7.

- No 1663 **Hannes Uppman:** On Some Combinatorial Optimization Problems: Algorithms and Complexity, 2015, ISBN 978-91-7519-072-3.
- No 1664 **Martin Sjölund:** Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models, 2015, ISBN 978-91-7519-071-6.
- No 1666 **Kristian Stavåker:** Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures, 2015, ISBN 978-91-7519-068-6.
- No 1680 **Adrian Lifa:** Hardware/ Software Codesign of Embedded Systems with Reconfigurable and Heterogeneous Platforms, 2015, ISBN 978-91-7519-040-2.
- No 1685 **Bogdan Tanasa:** Timing Analysis of Distributed Embedded Systems with Stochastic Workload and Reliability Constraints, 2015, ISBN 978-91-7519-022-8.

Linköping Studies in Arts and Science

- No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.
- No 586 **Fabian Segelström:** Stakeholder Engagement for Service Design: How service designers identify and communicate insights, 2013, ISBN 978-91-7519-554-4.
- No 618 **Johan Blomkvist:** Representing Future Situations of Service: Prototyping in Service Design, 2014, ISBN 978-91-7519-343-4.
- No 620 **Marcus Mast:** Human-Robot Interaction for Semi-Autonomous Assistive Robots, 2014, ISBN 978-91-7519-319-9.

Linköping Studies in Statistics

- No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.
- No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.
- No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.
- No 13 **Agné Burauskaite-Harju:** Characterizing Temporal Change and Inter-Site Correlations in Daily and Sub-daily Precipitation Extremes, 2011, ISBN 978-91-7393-110-6.

Linköping Studies in Information Science

- No 1 **Karin Axelsson:** Metodisk systemstruktureringsätt skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998, ISBN-9172-19-296-8.
- No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN-9172-19-299-2.
- No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999, ISBN-91-7219-606-8.
- No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.
- No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN 91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.