Technical University of Crete

Architectural Trade-offs of Partial Reconfiguration in FPGA Systems

A DISSERTATION

Submmitted to the Graduate Program

In Partial Fulfillment of the Requirements

for the Degree

DOCTOR OF PHILOSOPHY

in

Electronic and Computer Engineering

by

Kyprianos D. Papadimitriou

Chania, Crete

October 2011

Abstract

Reconfigurable computing (RC) is increasingly gaining the attention of many researchers and users by the academia and industry alike. The most popular representatives of reconfigurable computing are Field Programmable Gate Arrays (FPGAs). FPGAs are integrated circuits consisting of a large array of uncommitted programmable logic and interconnect, plus large blocks such as memories and Digital Dignal Processing (DSP) units that can be configured to implement digital circuits. Their capability to be programmed and reprogrammed in the field, i.e. forming on demand the digital circuit that will execute the application at hand, offers an unprecedented advantage over other technologies such as the Application Specific Integrated Circuits (ASICs) that cannot be reprogrammed, and the traditional software microprocessors in which flexibility comes at the expense of limited performance due to the fixed instruction set and the lack of parallelism. Moreover, GPUs that have started to be used for accelerating computationally intensive applications, although stand as strong opponents to the FPGAs, they have fixed hardware resources that cannot be customized to the application at hand.

An important portion of the FPGA market concerns static RAM (SRAM) based FPGAs, meaning that the SRAM bits are connected to the configuration points in the chip, and programming the SRAM bits configures the chip. A promising feature of specific families of SRAM-based FPGAs is the ability to reuse the same hardware for different tasks at different phases of an application execution. Moreover, the tasks can be swapped on the fly while part of the hardware continues to operate. This feature is known as run-time or dynamic reconfiguration.

Building upon the idea of dynamically reconfiguring a circuit in SRAM-based FPGAs, this dissertation explores the architectural tradeoffs of implementing applications in partially reconfigurable (PR) FPGA-based systems and proposes new avenues for its use. The dissertation begins with an in-depth study of the literature on reconfigurable devices and concentrates on those that can be configured in part. Next, it proposes a novel way to schedule tasks in PR FPGAs which is evaluated within the context of a simulation framework. Then, a real-world experimental framework allowing to study the functional details

of reconfiguration is presented. Using this framework a theoretical model is shaped which can be used for the early assessment of the overhead that the reconfiguration process incurs to the application execution. Finally, the dissertation proposes a novel way to exploit the PR technology in a specific application domain. In particular, a new method based on the PR capability of specific FPGAs is described, which allows for the self-repairing of FPGA core while operating in a harsh environment. All aspects of the present research have been verified with experiments from different setups using partially reconfigurable FPGA platforms.

Acknowledgements

I would like to thank my Professor Apostolos Dollas for his advice, encouragement and support on this dissertation effort. It was his vision that introduced me in the fascinating area of reconfigurable computing which has started earning its own place as an individual research field. I was fortunate enough to have him as a supervisor to stimulate and struggle me in order to devote a large amount of time seeking the research topic and shaping the subject of this Ph.D on my own. We spent a lot of time discussing and arguing on this subject and it was his directions that eventually led me to complete this dissertation. Also, he was the one who insisted and exhorted me to make the last steps toward completing my work and eventually close this Chapter of my life. I thank him for this deeply.

I would like to thank my Professor Konstantinos Kalaitzakis for his support and advices during my Ph.D candidacy, and my Professor Dionisios Pnevmatikatos for the discussions we had prior delving into this work and for his important hints throughout my research. I deeply thank Professor Scott Hauck for accepting me in his group at the University of Washington and for the long and fruitful discussions we had. Although my visit at the University of Washington was short his advices had a great impact on me. I hope that we will meet and cooperate again in the future.

I want to thank Professors Manolis Katevenis, Georgios Stamoulis, Georgios Stavrakakis, and Ioannis Papaefstathiou for participating in my committee and revising the manuscript.

I deeply thank Robert Pozner for all his advices and the way he has influenced me until now. I thank Anne McKay, Stella Psaroudaki and Magda Markantonaki for all their help. I want to thank the people I have worked with at the Microprocessor and Hardware Laboratory (MHL). It is a pleasure to be in the same environment with Markos - who assisted me in everything I 've needed -, Euripides and Grigoris and interact with all the members of MHL.

I want to thank Stefanos Karasavvidis for his help, and Christoforos Kachris for his detailed comments prior submitting a paper related with my Ph.D work. I would like to thank Amir for the great cooperation we had. Our joint work is not included in the present dissertation but its has been published in the proceedings of a conference. Also, I would

like to thank all the members of the group I worked with during my visit at the University of Washington: Aaron, Robin, Shakil, Jimmy, Michael, Brian, Benjamin, Steve and Ken were all great.

It would have been a miss not to mention Antonis Anyfantis and Argyris Ilias who during their undergraduate studies helped me reinforce the contributions of this research. Antonis made the automatic framework described in Chapter 4 that enabled the measurements during reconfiguration. The results formed the basis for extracting the model of Chapter 5. Argyris during his final-year project implemented a fully functional prototype of the design described in Chapter 6. I instructed and cooperated tightly with both Antonis and Argyris and although they didn't involve with the design, their contribution in building the fully functional systems is acknowledged. Also it should be noted that the results gathered with the experimental setups of the final-year projects of Michalis Vavouras and Giorgos Nikoloudakis were used to verify part of the work described in Chapters 3 and 5 respectively.

I would like to thank all the friends and the people that stood by my side and especially I thank Efi and Fotis.

Also, I would like to acknowledge the support of Xilinx Incorporation on the development platforms and tools. I acknowledge the Greek Ministry of National Education and Religious Affairs for the Ph.D fellowship under the program Heraklitus of EPEAEK II. However, I want to report the delay of the funding, which arrived 20 months after the acceptance of the research proposal. I wish it would have arrived earlier. Also, I want to point out that the subject of the proposal wasn't strictly related with the subject of my Ph.D, but it gave me the motivation to delve into the details of reconfigurable computing.

I thank my parents Dimitris and Ioanna for standing by my side throughout all these years. Without their patience and support I wouldn't be able to complete this task. I thank them with all my heart and I dedicate the dissertation to my father who left us so soon.

To my father

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are suitable for implementing applications that benefit from custom parallelization and pipelining. They combine a large pool of heterogeneous physical resources that when effectively treated can form circuits operating with high performance. Different application domains have benefited from their implementation in FPGAs ranging from bioinformatics on high-end systems [Afratis et al., 2008] to motion detection on low-cost systems [Papademetriou et al., 2006]. Nowadays, the leading companies manufacture FPGA chips in a technology process as low as 28nm. This broadens the capabilities offered by FPGAs as latest technology is capable to incorporate a vast amount of programmable resources made by billions of transistors within a single small die. The resources that co-exist into an FPGA chip range from single gates and flip-flops to hardcore Digital Signal Processing (DSP) units and embedded hardcore processors.

The proper activation of selected resources and the programming of interconnection thereof so as to map effectively an application into an FPGA emerge as important research subjects. Even more, performing these operations at run-time, i.e. dynamically altering part of the FPGA while the rest remains intact continuing its operation [Compton and Hauck, 2002], becomes more challenging although it is still in its infancy. Besides large FPGAs, this technology could benefit applications implemented in small FPGAs in which the gain would have different margins as compared to the former ones. For example, multiple design modules can time-share the physical resources, and the hardware can adapt to the application at hand or even to a segment of an application. In this way, smaller devices

can be employed enabling reduction in cost, size and power, and more efficient use of the board space. In general, the use of PR technology for an application can be justified if benefits are gained with regard to factors such as resource savings, power reduction and higher performance over the static implementation.

1.1 Motivation

Many efforts within the academia and a few among the industrial community exist, trying to establish dynamic reconfiguration as a feasible way to design commercial applications. It is considered to be the "holy grail" of reconfigurable computing and its effective exploitation could result in circuits that run applications more effectively over their static counterparts. In the network domain, a reconfigurable processor that was altered dynamically in order to meet the requirements of the network workload was proposed [Kachris and Vassiliadis, 2006]. Modules like encryption, compression and intrusion detection found in contemporary edge routers, are dynamically loaded according to the traffic distribution to serve the different network flows. In the field of Software Defined Radio (SDR), a prototyping kit was released to the market, which uses partial reconfiguration (PR) to support different communication waveforms and protocols within a single chip [Xilinx Inc., 2006b]; this allows for flexible and efficient communication between equipment that differs in vendor, Radio Frequency (RF) or interface protocol. Dynamic reconfiguration has also been used to support high energy physics research at CERN's Large Hadron Collider [Programmable Logic Design Line, 2008]. The latest achievement by the leading vendor in PR technology targets the networking domain. An integrated Optical Transport Network system has been demonstrated [Xilinx Inc., 2010b], in which considerable resource and power savings were achieved due to the use of PR technology.

The above are some of the systems that can establish dynamic reconfiguration as a feasible way to design commercial applications. At the same time issues such as when to use PR, why to use it, in what kind of applications and in which way remain open, though a considerable effort is made by the academia and industry towards addressing them. Present work aims at bridging the gap between theoretical research and real experimentation. The

latter constitutes an effortful procedure due to the intrinsic difficulties of PR technology. The dissertation delves into the details of reconfiguration with real experiments and by using the feedback of the results it studies ways to use it effectively.

1.2 Contribution

The dissertation is concerned with a task allocation and scheduling mechanism in PR FPGA-based systems, a formula to calculate the reconfiguration time, and a novel way to incorporate PR technology in an application domain. Among the experience gathered and the conclusions drawn, an experimental framework was developed on a PR FPGA-based platform. The contributions of the dissertation consist in:

- an in-depth literature review covering a wide area of issues related with PR technology,
- a resource-aware task scheduling algorithm to exploit effectively the area of a dynamically reconfigurable FPGA,
- a cost model extracted from real experiments and theoretical analysis to quantify the reconfiguration overhead for various setups which can be used in an early-assessment stage of the development procedure,
- evaluation of the impact of reconfiguration overhead in recovering a PR system that is subject to error upsets, and a scheme exhibiting resource savings over the dominant solution in the domain of error diagnosis and recovery.

It should be noted that due to the rapid changes in the specific subject, the direction of the dissertation had to be devised some times. At the same time although PR technology is around for almost 20 years, no killer application exists yet. Present dissertation delves into different research subjects, i.e. task scheduling and allocation, reconfiguration overhead and a novel scheme for fault-prone systems, and ends up with conclusions that can be considered when building systems using PR technology. All experiments were carried out on mature FPGAs, i.e. Xilinx Virtex-II Pro and Virtex-5 FPGAs. Virtex-6 and Virtex-7 FPGAs were released from Xilinx when the dissertation was close to its completion.

1.3 Structure

The dissertation is structured as follows:

Chapter 2 is devoted to the background of reconfigurable computing targeting mainly the dynamic nature of programmable chips. A wide spectrum of subjects related with the PR technology is analyzed to reveal the research status and open problems. It appears that although the PR topic has been studied intensively, several subjects need to be revisited.

Chapter 3 presents a new task scheduling mechanism that increases the utilization of the physical resources. Furthermore, a reusable framework that models dynamically reconfigurable systems and accepts attributes entered by the user has been developed for evaluating the task scheduling mechanism. The framework is generic so as to be used for researching and evaluating similar mechanisms.

Chapter 4 presents an experimental framework deployed on an FPGA-based platform which allows for extensive experimentation and evaluation of PR technology. The values gathered with this platform are used in Chapter 5.

Chapter 5 examines the different setups and factors affecting the reconfiguration time. Also, a cost model that applies in a range of platforms has been developed which was verified using a real-world system and evaluated upon works published by other researchers. This Chapter constitutes an integral work as it surveys the domain related with the reconfiguration overhead at system-level. Also, it analyzes a variety of setups and the factors contributing to the reconfiguration process.

Chapter 6 studies the use of PR technology on a real-world application that has proven (based on the literature) to benefit from partial reconfiguration. It falls into the domain of error diagnosis and recovery for non critical systems. A novel scheme combining software and hardware is described, which supplies the core of an FPGA chip with self-repairing capability. Protection concerns the configuration memory of FPGA as well as instantaneous errors affecting the implemented circuit.

Chapter 7 summarizes the contributions of the dissertation and discusses the future work.

All Chapters begin with an introductory section along with a reference to an up-to-date relevant work. Although each one of the Chapters constitutes a coherent work, they all complement each other. In particular, the experimental framework of Chapter 4 was used to extract the cost model in Chapter 5. In turn, the information and the cost model of Chapter 5 can be used as feedback for inserting realistic attributes to the simulation framework of Chapter 3. Also, the cost model can be used for assessing the system presented in Chapter 6 for different setups.

Chapter 2

State of the Art

Reconfigurable computing has become a subject of a great deal of research during the last decade. The present Chapter begins with a historical evolution of reconfigurable devices. Then, it concentrates on systems supporting dynamic reconfiguration. First, it discusses the different theoretical perspectives that have been proposed to study its performance. Then, several dynamic architectures and the software tools to support them are described. Furthermore, a variety of applications implemented with dynamic reconfiguration is presented. Finally, the overhead incurred by dynamic reconfiguration and the research efforts to reduce its effects on performance are described. Throughout the present work the benefits and the weaknesses of dynamic reconfiguration as well as the effort to deal with it are discussed. This allows to examine the potential improvements of its use in modern systems and applications.

2.1 Introduction

As of 2004, the computer industry has hit a roadblock in getting further performance gains from instruction level parallelism [Olukotun and Hammond, 2005]. Following Moore's law regarding performance, computer architects and designers have started to open new directions in developing computing systems. Hence, the industry is heading towards exploiting higher levels of available parallelism through techniques like multiprocessing and multithreading. On the other hand, reconfigurable computing since its early appearance has

been intended to fill the gap between hardware, e.g. application specific integrated circuits (ASIC), and software, e.g. microprocessors (μP), achieving potentially higher performance than software while maintaining a higher level of flexibility than hardware [Compton and Hauck, 2002]. The stronger representatives of Reconfigurable computing are the SRAM-based Field Programmable Gate Arrays (FPGAs). Systems incorporating a fixed processing unit (FPU) such as software microprocessors and a reconfigurable processing unit (RPU) such as field programmable gate arrays (FPGAs) onto a single chip have become state-of-the-art devices known as reconfigurable processors. Also, super-computing systems combining fixed and reconfigurable resources were released aiming at speeding up custom applications such as systems released by the Convey Computer [Convey, 2011], the Maxeler Technologies [Maxeler, 2011] and Pico Computing [Pico Computing, 2011], which followed the steps of Cray XD1 Supercomputer [Cray, 2008], one of the earliest large-scale products combining microprocessors and FPGAs. Computationally-intensive tasks can be entirely executed in the reconfigurable fabric that can efficiently exploit the application's inherent parallelism. Furthermore, reconfigurable fabrics that adapt to the needs of the application task at hand at run-time have been proposed. Factors such as speed, area, power and energy consumption can be optimized due to the ability of the hardware to adapt to the characteristics of an application. This could boost the necessity of integrating reconfigurable processing units into contemporary systems whether they target the field of general purpose or embedded computing. Currently, almost all applications running in FPGAs as end-products do not make full use of their reprogrammable nature. When FPGAs are used either to rapidly prototype circuits or in designs where a small production run is expected they are usually programmed only once at power-up and after this the circuit remains unchanged.

Before proceeding with the main subject of this Chapter, some terms need to be clarified. Reconfiguration is defined as the capability of the hardware to be modified in the field such as the hardware resources to be used and their interconnection after it is initially configured. It is distinguished in two main categories, static and dynamic reconfiguration. Static reconfiguration is performed at shut-down mode, i.e. the hardware is modified when the device is inactive, while dynamic reconfiguration is performed during execution, i.e. the device is active. Two subcategories exist, full and partial reconfiguration. Full reconfigura-

tion, in which the entire configuration is modified, is performed only under static mode as the reprogramming of the entire device is required. Partial reconfiguration, in which part of the hardware is modified while the rest remains unchanged, can be performed either under static or dynamic mode. Static partial reconfiguration is done when the device is inactive and the unchanged part retains its configuration information. Dynamic partial reconfiguration allows for swapping tasks in and out from specific areas of the hardware while the remaining logic continues undisturbed its execution. The concept of partially reconfiguring the hardware during execution is also known as active partial reconfiguration (APR) [Xilinx Inc., 2004a], run-time reconfiguration (RTR), on-the-fly reconfiguration or simply dynamic reconfiguration. In the present dissertation these terms are used interchangeably. One of the earliest works clarifying terminology on reconfiguration was published in [Lysaght and Dunlop, 1993].

Full reconfiguration incurs a significant amount of data to be swapped in and out of the hardware, which combined with the stall of execution restricts its applicability to applications in which reconfiguration delay degrades the overall performance. On the other hand, dynamic partial reconfiguration provides a more flexible way to deal with versatility and area utilization of reconfigurable resources as part of the hardware logic can be reconfigured while the rest hardware is in operation. This allows to extend the use of reconfigurable systems beyond emulation and rapid system prototyping. Many applications require a large amount of area in the fabric for their processes to be carried out. However, die cost is proportional to the fifth or higher power of the die area [Hennessy and Patterson, 2003] (pp. 21-22). Thus designers can reduce the cost by building smaller chips. Toward the same direction, developers can minimize the resources usage and consequently the chip size when different stages of the application are executed by the same hardware in different time slots. Having flexible means of fabric that can be modified at run-time to execute different tasks of the same application or different tasks serving different applications is appealing. Furthermore, circuits can be specialized according to parameters being changed at run-time, potentially resulting in a superior system over the static alternative [McKay et al., 1998, McKay and Singh, 1998, Gonzalez et al., 2003]. Going one step further, leaving hardware areas unoccupied and configure them according to the state of the application al-

lows for energy savings as it is unnecessary to keep idle tasks in the hardware; unless these tasks are executing, leaving them out of the hardware can result in less energy consumption. The benefits offered by dynamic reconfiguration are recapitulated as follows:

- Hardware sharing by time multiplexing hardware tasks
- On-the-fly adaptation of the hardware according to the changing needs of the application
- Reduced device count
- Reduced power consumption (for infrequent reconfiguration)
- Reduced cost

At its earliest stage dynamic reconfiguration was introduced as virtual hardware which is similar to the concept of virtual memory [Brebner, 1996]. A few years ago, self-reconfiguring systems became feasible by offering the ability to modify the functionality of their hardware at run-time according to the application needs [Blodget et al., 2003]. In such a system an integrated processor undertakes the tasks of controlling and reconfiguring the reconfigurable hardware, while the latter executes the computationally-intensive tasks.

Numerous comprehensive surveys on reconfigurable computing have been published [Villasenor and Hutchings, 1998, Hauck, 1998b, Miyazaki, 1998, Tessier and Burleson, 2001, Hartenstein, 2001, Bondalapati and Prasanna, 2002, Compton and Hauck, 2002, Todman et al., 2005]. Some surveys attempt to classify reconfigurable systems according to specific aspects [Radunovic and Milutinovic, 1998, Enzler, 1999, Barat and Lauwereins, 2000, Schaumont et al., 2001, Barat et al., 2002, Sima et al., 2002, Donthi and Haggard, 2003, Amano, 2006]. In [Gokhale and Graham, 2005], after an introduction to the reconfigurable computing, several reconfigurable architectures, systems with reconfigurable devices, programming languages and implementations of different application domains are presented. An interesting catalog is available in [DeHon et al., 2004], which studies over one hundred works related to reconfigurable computing in order to classify them according to design patterns. Design patterns are defined as solutions to common and recurring design challenges in reconfigurable systems and applications. In [Bobda, 2007], various reconfigurable architectures and

issues like design flow, high-level synthesis, temporal placement, on-line communication, partial reconfiguration and different domains of applications implemented in reconfigurable devices are discussed. Recently, a comprehensive collection was published covering a broad range of topics on reconfigurable computing [Hauck and DeHon, 2008]. This collection covers subjects like reconfigurable architectures, programming languages, operating systems, compilation, placement, mapping, routing, hardware/software partitioning and a variety of applications. Finally, the authors in [Garcia et al., 2006] target the embedded systems domain. They discuss topics such as benefits gained in applications implemented with reconfigurable hardware, basic architectural aspects, critical design issues for embedded systems, and design tools to develop such systems.

Although the above works do not target dynamic reconfiguration only, a few refer or devote a section to it. In [Compton and Hauck, 2002], a dedicated section addresses issues on run-time reconfiguration regarding architecture structures, run-time partial evaluation, compilation and configuration scheduling, supporting software, techniques to reduce reconfiguration overhead as well as existing problems of this technology. A more recent survey [Todman et al., 2005] discusses dynamic reconfiguration and its research directions in an abstract level. A few papers survey some commercial dynamically reconfigurable FPGA devices [Donthi and Haggard, 2003], and commercial coarse-grained dynamically reconfigurable processors [Amano, 2006]. The former evaluates FPGA architectures based on their granularity and their reconfiguration time, whereas the latter examines issues such as structure of the basic processing elements, dynamic reconfigurability, processor-coupling, interconnection and programming software. In [Bobda, 2007] a Chapter is devoted on partial reconfiguration targeting Xilinx devices. In [Hauck and DeHon, 2008] a dedicated section on reconfiguration management and a case study of partial reconfiguration are included. In [Garcia et al., 2006], issues such as real-time operating systems and scheduling for run-time reconfigurable systems, relocation and defragmentation of configurations, and research for hiding configuration overhead are discussed.

Present Chapter intends to complement the above works by presenting details on various aspects of dynamic reconfiguration and providing background information from the earliest up to the latest achievements. This is done by:

- presenting a historical evolution of reconfigurable devices with a critical analysis on academic and industrial research results on dynamic reconfiguration to date.
- presenting various theoretical perspectives that examine the performance of dynamic reconfiguration.
- providing up-to-date information on recent advances in architectures and software tools.
- discussing the application domains that can benefit from their implementation in dynamically reconfigurable hardware.
- presenting the research on techniques for scheduling and partitioning the dynamically reconfigurable tasks as well as on allocation of reconfigurable resources.

This Chapter merges the conducted research, the open directions and the way in which dynamic reconfiguration can be incorporated in computing systems. Although it does not cover every research project, it serves as an in-depth introduction to the rapidly evolving field of dynamically reconfigurable computing. It is organized as follows. Section 2.2 has the historical evolution of reconfigurable computing along with a classification of reconfigurable systems with respect to their structure. In Section 2.3 the main research topics on dynamic reconfiguration are discussed. Section 2.4 presents theoretical approaches studying its performance. Sections 2.5 and **??** examine the various architectures and the software developed to support them respectively. In Section **??** several applications deployed on dynamically reconfigurable systems are presented. Section **??** discusses reconfiguration overhead and research efforts to reduce its effects on performance. Finally, Section **??** discusses the present status and future directions on dynamic reconfiguration research.

2.2 Historical Evolution

Numerous structures have been proposed since the appearance of reconfigurable computing. At the beginning the single-context FPGA was developed. As the research evolved multi-context and partially reconfigurable structures were invented. Present Section overviews this

evolution of reconfigurable computing systems and distinguishes devices from the academic and commercial areas with respect to their structure and the way they are reconfigured. This high-level classification is provided in order to avoid reporting further details at this point such as types and structures of reconfigurable resources, granularity, and supporting software; these are discussed in later Sections.

The following classification concerns SRAM-based reprogrammable devices only that can be potentially programmed unlimited times and in a short time compared with other alternatives. Antifuse and flash devices [Actel, 2007, Quicklogic, 2007] are not included as the former type is one-time programmable only, and the latter type needs long time to be programmed. Table 2.1 consolidates representative SRAM-based reprogrammable devices. Devices combining characteristics of more than one structure type may appear in more than one fields of Table 2.1. The devices of each structure type are put in chronological order according to their first announcement. Especially for the commercial devices, the year in which they entered the market is denoted on the right side of the reference. This can be found in web resources, e.g. a link on Altera mature devices with the year of their first release is available in [Altera, 2007]. In many cases, this information does not match with the dates of the corresponding references - data sheets in most of the cases - because their first releases have been withdrawn for some reason, e.g. maturity of a device or release of newer versions of data sheets that hampered the task of finding and including them in the literature part of the present dissertation. Alternatively, later versions of data sheets which include old devices that are either available in the library of MHL laboratory or still accessible through the web are cited. The remaining Section overviews single-context, multi-context and partially reconfigurable structures. A comprehensive analysis on these structures can be also found in [Compton and Hauck, 2002].

2.2.1 Single Context

Single-context devices were proposed at the evolutionary beginning of reconfigurable computing. In these devices, any change in the configuration requires a complete device reprogramming. Hence, in order to reconfigure a single-context device the whole execution stalls and the new bitstream is loaded off-line. Then, execution starts according to the new

Table 2.1: Reconfigurable devices classified in single-context (SC), multi-context (MC) and partially reconfigurable (PR) structures.

	Academic efforts	Commercial devices
SC	PRISC [Razdan and Smith, 1994] Datapath-FPGA [Cherepacha and Lewis, 1994] HybridFPGA [Kaviani and Brown, 1996]	XC2000/3000/4000/5000 [Xilinx, 1994] '85/'87/'91/? Flex 8000/10K [Altera, 1995] '93/'95 Flex 6000 [Altera, 1998] '98 Spartan [Xilinx, 2002b] '98 Apex 20K [Altera, 1998] '98 Acex 1K [Altera, 2003] '00 Spartan-II [Xilinx, 2004] '00 Mercury [Altera, 2003] '00 Excalibur ARM [Altera, 2001] '00 E5 CSoC [Triscend, 2003] '00 Excalibur MIPS [Altera, 2001] '01 A7 CSoC [Triscend, 2000] '01 Apex II [Altera, 2002] '01 Stratix [Altera, 2006b] '02 Cyclone [Altera, 2005] '02 ispXPGA [Lattice, 2005] '02 Stratix-II [Altera, 2006c] '04 Cyclone-II [Altera, 2006a] '04
MC	WASMII [Ling and Amano, 1993] DPGA [DeHon, 1994] OneChip95 [Wittig, 1995] TMFPGA [Trimberger et al., 1997] CSRC [Scalera and Vazquez, 1998] OneChip98 [Jacob, 1998] MorphoSys [Singh, 1998] DRLE [Fuji et al., 1999] OneChip00 [Esparza, 2000] SRGA [Sidhu et al., 2000] ZIPPY [Enzler and Platzner, 2001] XiRisc [Lodi et al., 2003] PRMC [Smith and Xia, 2004]	MAPL [Hawley, 1991] '91 SIDSA FIPSOC [Faura et al., 1997] '97 CS2112 RCP [Chameleon, 2000] '00 picoArray [PicoChip, 2008] '00 NEC DRP [Motomura, 2002] '02 DAPDNA-2 [IP Flex, 2008] '03
PR	DPGA [DeHon, 1994] KressArray [Hartenstein et al., 1994] COLT [Bittner et al., 1996] MATRIX [Mirsky and Dehon, 1996] RaPiD [Ebeling et al., 1996] Garp [Hauser and Wawrzynek, 1997] Chimaera [Hauck et al., 1997] PipeRench [Schmit, 1997] RAW [Waingold et al., 1997] REMARC [Miyamori and Olukotun, 1998] CoMPARE [Sawitzki et al., 1998] PROTEUS [Dales, 1999] CHESS [Marshall et al., 1999] modified XC6200 [Compton, 1999] unnamed [Pozzi, 2000] DReAM [Alsolaim et al., 2000] Molen [Vassiliadis et al., 2001] HySAM [Bondalapati, 2001] ZIPPY [Enzler and Platzner, 2001] AMDREL [Soudris et al., 2002] ADRES [Mei et al., 2003] WARP [Stitt et al., 2003] XiRisc [Lodi et al., 2003] POEtic [Thoma et al., 2003] DyNoC [Bobda et al., 2004] ARCADE eFPGA [Nowak, 2004] PRMC [Smith and Xia, 2004]	CAL [Algotronix, 1988] '88 ERA60100 [Plessey, 1990] '90 CLAy [National, 1993] '93 AT6000 [Atmel, 1993] '93 CLi6000 [Jenkins, 1994] XC6200 [Xilinx, 1995] '95 NAPA1000 [National, 1996] '96 OR FPGA [Vasilko and Ait-boudaoud, 1996] SIDSA FIPSOC [Faura et al., 1997] '97 PCA [Nagami et al., 1998] DL6000 [DynaChip, 1998] Orca Series 2/3/4 [Lucent, 1998] <'98 Quicksilver ACM [Quicksilver, 2007] '98 AT40K [Atmel, 2006] '98 Virtex [Xilinx, 1999] '98 D-Fabrix [Elixent, 2000] '00 picoArray [PicoChip, 2008] '00 CS2112 RCP [Chameleon, 2000] '00 AT94K FPSLIC [Atmel, 2005] '00 XPP [Baumgarte et al., 2001] '01 Virtex-II [Xilinx, 2007d] '01 Virtex-II Pro [Xilinx, 2007e] '02 Cypress PSoC [Cypress, 2008] '02 DAPDNA-2 [IP Flex, 2008] '03 MeP+D-Fabrix [Toshiba, 2003] '03 Spartan-3 [**?**] '03 Virtex-4 [Xilinx, 2007c] '04 S5-engine [Arnold, 2005] '05 Virtex-5 [Xilinx, 2008b] '06 XPP-III [PACT XPP, 2008] '07 ECA-64 [ElementCXI, 2008] '07

configuration data. Xilinx XC2000 series [Xilinx, 1994] and Altera Flex 8000 series [Altera, 1995] were of this type. An innovative architecture was proposed in [Kaviani and Brown, 1996] combining a simple FPGA structure with CPLD in a single die. A few years later, several vendors coupled single-context structures with a hardware processor in the same die like Altera [Altera, 2001], and Tensilica [Triscend, 2000, Triscend, 2003]. The latter's technology was acquired by Xilinx in 2004. Towards this direction, softcore processors have been developed for reconfigurable logic like Nios [Altera, 2004] and Microblaze [Xilinx, 2007a] for Altera and Xilinx devices respectively. From the academia part, a few efforts proposing innovative single-context structures [Cherepacha and Lewis, 1994], and coupled devices [Razdan and Smith, 1994] were made in the early stages of reconfigurable computing.

2.2.2 Multi Context

Multi-context devices include multiple memory bits for each programming bit location. They can be thought of as multiple planes of configuration information. One plane can be active at a given time, while new configuration data are loaded to another plane. This structure can be also viewed as a multiplexed set of single-context devices, which requires that a context be fully reprogrammed - in most cases - to perform any modification. Switching between planes (or contexts) can be performed in a single cycle. National Semiconductor MAPL was the first device to be considered as a multi-context device [Hawley, 1991]. From the academia part the first published multi-context device was WASMII [Ling and Amano, 1993]. Some of the earliest works were presented in [DeHon, 1994, Wittig, 1995, Trimberger et al., 1997]. Since then, various researchers have been involved with multi-context structures [Scalera and Vazquez, 1998, Jacob, 1998, Singh, 1998, Fuji et al., 1999, Esparza, 2000, Sidhu et al., 2000, Enzler and Platzner, 2001]. A few such devices were commercialized such as the Field Programmable System-on-Chip (FIPSOC) by SIDSA [Faura et al., 1997], the Reconfigurable Communication Processor (RCP) by Chameleon Systems [Chameleon, 2000], the IPFLEX DAPDNA-2 [IP Flex, 2008] and the Dynamic Reconfigurable Processor (DRP) by NEC [Motomura, 2002]. The latter was the successor of the academic Dynamically Reconfigurable Logic Engine (DRLE) [Fuji et al., 1999] and NEC terminated its production in 2007. The FPGA presented in [Scalera and Vazquez, 1998], known as Con-

text Switching Reconfigurable Computer (CSRC), was implemented by British Aerospace Systems [BAE, 2007] and was used for experimental purposes. Most of the above devices were augmented with a processor which controls reconfiguration and context-switching.

2.2.3 Partially Reconfigurable

Partial reconfiguration allows to selectively modify part of the hardware while the remaining hardware retains its configuration. In static devices, execution stalls waiting for the selected part to be reconfigured. This operation resembles single-context devices operation except that a partially reconfigurable device allows partial modifications of the fabric. Such devices are Garp [Hauser and Wawrzynek, 1997], CHESS [Marshall et al., 1999] and its commercial successor D-Fabrix [Elixent, 2000] - later bought by Matsushita Electronics Corp. -, and Molen [Vassiliadis et al., 2001]. The more sophisticated dynamically reconfigurable structure allows for the rest of the device to continue uninterrupted its execution. The first commercial device supporting dynamic reconfiguration was Algotronix CAL [Algotronix, 1988], which was introduced in 1988. A few years later National Semiconductor CLAy [National, 1993] and ATMEL AT6000 [Atmel, 1993] announced the support of this feature. Since then, various vendors like Xilinx [Xilinx, 1995, Xilinx, 1999, Xilinx, 2007d, Xilinx, 2007e, Xilinx, 2007c, Xilinx, 2008b, Xilinx Inc., 2008], Lucent [Lucent, 1998] and PACT XPP [Baumgarte et al., 2001, PACT XPP, 2008] have released such products. For sake of history, Xilinx XC6200 [Xilinx, 1995] was designed based on Algotronix CAL after Xilinx acquired Algotronix technology. The high-end Virtex series have also retained some of its features. Lattice supplied the ORCA series 2/3/4 devices [Lucent, 1998] that support dynamic reconfiguration. The company has entered the market of FPGAs after the purchase of the Agere Systems - formerly Lucent Technologies and before that AT&T Microelectronics. Although Lattice ORCA series support partial reconfiguration while in operation, there is no published record of applications implemented with this feature and as a consequence they have been reported as single-context devices in the literature [Compton and Hauck, 2002]. From the academia part, due to the rich area of research that dynamic reconfiguration technology offers, many groups have been involved with developing such structures [Hartenstein et al., 1994, Bittner et al., 1996, Ebeling et al., 1996, Hauck

et al., 1997, Schmit, 1997, Miyamori and Olukotun, 1998, Alsolaim et al., 2000, Enzler and Platzner, 2001, Mei et al., 2003, Stitt et al., 2003, Bobda et al., 2004]. Designing proprietary architectures rather than using commercial ones allowed researchers to have full control of their characteristics and apply modifications according to the changing needs. There exist several efforts on innovative partially reconfigurable structures and ways to reprogram them [Plessey, 1990, Vasilko and Ait-boudaoud, 1996, Nagami et al., 1998, DynaChip, 1998]. PRMC [Smith and Xia, 2004] suggests a multi-context, partially reconfigurable homogeneous fine-grain array. Moreover, following the concept of the first two structures both academic and commercial partially reconfigurable structures, i.e. single-context and multi-context, tend to incorporate a processor which acts as a reconfiguration controller amongst other tasks. Finally, multi-context structures have been leveraged to allow partial loading of inactive contexts [DeHon, 1994, Mirsky and Dehon, 1996, Faura et al., 1997, Chameleon, 2000, IP Flex, 2008, Lodi et al., 2003, Smith and Xia, 2004]. To the best of the author's knowledge there does not exist an academic or a commercial multi-context device supporting partial reconfiguration of the active context.

2.2.4 Discussion Summary

In this Section a classification of reconfigurable devices according to their structure from the reconfiguration point of view, the year they were announced, and whether they come from the academia or the commercial area has been given. The above list includes fine-, coarse-, medium-, and mixed-grained, coupled and uncoupled with a microprocessor, homogeneous and heterogeneous reconfigurable devices. Some of them fall into the category of FPGAs whereas others are SoCs. In addition, there exist devices and derivatives of the reported commercial devices enhanced with a variety of hardware cores, e.g. PCI and phase-locked loops(PLLs). Similar lists can also be found in active web sites [Berkeley, 2007, Ottawa, 2007, Erasmushogeschool, 2007]. In spite of the large amount of existing reconfigurable architectures the market is still dominated by FPGA devices and in particular those from Xilinx and Altera. The latter vendor has recently started putting effort in developing dynamically reconfigurable FPGAs, e.g. transceivers in Stratix-II GX device can be reprogrammed on the fly to support multiple protocols, data rates and physical medium

attachment settings. Moreover, the coarse-grained reconfigurable devices market has not taken off yet despite the amount of prototypes developed in this direction.

The reconfiguration capabilities of these devices have also been described. Single-context structure requires full reconfiguration whereas in the other two structures only a part can be reconfigured. Although in most of the multi-context devices a full-context is reconfigured while another is active, there are few multi-context devices of which only a part of an inactive context can be reconfigured while another context executes. Partially reconfigurable structures incur small reconfiguration overhead in applications that do not require the device to be completely reconfigured; however, depending on the application needs this might not be negligible with respect to the total execution time. Alternatively, a system consisting of multiple single-context reconfigurable devices each of which is separately reconfigured can be employed, but off-chip communication and extra space on the board increase the cost of speed, area and power consumption. Similarly, a multi-context device carries significant reconfigurable area incurring high interconnection delay, low computational density and suffers from various delays and high-costs due to large storage silicon area, i.e. multi-context configuration memory; thus it is more applicable in coarse-grain devices. In a single partially reconfigurable device the trade-offs between the above factors are more efficiently balanced.

Table 2.1 illustrates that substantial efforts have been made in developing multi-context and partially reconfigurable architectures (statically partially reconfigurable architectures aren't distinguished from the dynamically ones). Although the list is far from being complete it demonstrates that there is an increasing interest from the academia and industry for such devices. Although some commercial efforts have been abandoned the lessons learned - which are basically given by the market applications - are changing the directions; but research for dynamic reconfiguration has never been abandoned. The remaining Chapter focuses on partially reconfigurable devices and particularly on systems combining a fixed processor with a reconfigurable hardware onto the same die. In this scenario the processor usually undertakes the reconfiguration task; however, there exist devices in which a dedicated reconfiguration controller undertakes this task [Baumgarte et al., 2003].

2.3 Research Topics on Dynamic Reconfiguration

Although dynamic reconfiguration is a fascinating feature many scientists have been cautious regarding its feasibility and value. Its use has been restricted within the borders of research, and none to very few commercial applications have been implemented in a dynamic manner. The lack of an operational framework, and its tedious design flow have kept this technology from being widely accepted by the community. As of 2006 a new method has been announced to support dynamic reconfiguration [Lysaght et al., 2006]. It offers a graphical environment and its main scope it to abstract the details of the rigorous design flow of dynamic reconfiguration from the developer.

Commercial applications implemented in a dynamic manner have also been released. A prototyping-to-production kit that accelerates implementation of software defined radio (SDR) modems has been announced by Xilinx and ISR technologies [Xilinx and ISR, 2006]. SDR is a radio communication system supporting different transmission protocols and it allows for the modulation and demodulation of signals to be implemented in software. This provides flexibility to the customer, and Xilinx has taken advantage of the flexibility supported by its devices. Thus it transferred the implementation in hardware to reach high performance and change the algorithms at run-time. It is the industry's first SDR kit to support partial reconfiguration, which exploits this feature to reduce power and cost in developing SDR systems. In the context of networking, Xilinx released an industrial crossbar switch in which the routing resources are modified during operation [Xilinx, 2002a]. These products demonstrate that dynamic reconfiguration technology is now feasible, and it is time to start exhibiting the advantages of reconfigurable computing in more areas. On the other hand, there are several issues that remain to be solved to enable the access of this technology to the mainstream. This Section introduces the main research aspects on dynamic reconfiguration that are discussed later.

- Several works study dynamic reconfiguration from a *theoretical perspective*. This way issues like efficient handling of dynamically reconfigurable tasks and the impact of dynamic reconfiguration on speed and area are theoretically elaborated. As the technology evolves it would be useful to explore the applicability of these studies on recent

systems.

- The demanding adaptation of the hardware to the application at hand has led the exploration of numerous *architectures*. Coarse-grained, fine-grained and architectures integrating a fixed processor onto a single chip that support dynamic reconfiguration have been proposed. Moreover, issues such as which component undertakes the reconfiguration process, i.e. a processor or a dedicated configuration manager, the size of the smallest unit to be reconfigured, and the way the unit to be reconfigured is located should be addressed when designing an architecture.

- A variety of *software tools* have been developed to support dynamically reconfigurable architectures. The design flow of dynamic reconfiguration is rigorous compared with the traditional design flow. Thus abstracting the low-level details from the designer is a major issue and efficient mapping of applications to the hardware without knowing its details is a challenging problem. Nowadays the trend is moving towards designing applications in high-level programming languages with interfaces that can be efficiently handled by the average software programmer. The simplest way is generating data-flow graph from a C-like language and map them into the array.

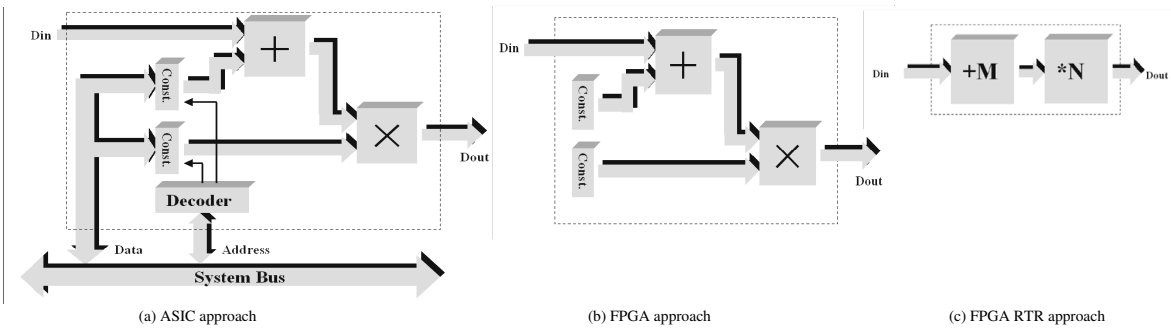
- Significant efforts by the academia and the industry are made to establish dynamic reconfiguration as a feasible way in designing *applications*. Its performance has been investigated with respect to factors such as speed, area and power. Restrictions on these factors have to be considered when employing dynamic reconfiguration to execute an application. The question is under which conditions dynamic reconfiguration prevails over alternative approaches. The type of applications that can benefit from dynamic reconfiguration vs. alternative solutions like ASIC and general-purpose processors are still under research.

- Benefits of dynamic reconfiguration do not come without cost. The inherent high latency and low throughput of the reconfiguration process compared with other processes incurs degradation in overall system performance known as *reconfiguration overhead*. In the 1st NASA/DOD Workshop on Evolvable Hardware, it was reported that

while reconfigurability was a major theme in DARPA's Adaptive Computer Systems program, the big stumbling block at the time was the large reconfiguration time of modern platforms [Munoz, 1999]. In order to harness this overhead which can take microseconds or longer, fast configuration was a critical issue. Since then, the problem has been dealt with techniques such as configuration caching, configuration compression and configuration prefetching [Li, 2002]. Moreover the problem has been addressed in the context of placing the dynamically reconfigurable tasks on the hardware in an efficient manner. Research on task scheduling, resource allocation, HW/SW partitioning mechanisms and efficient placement has been conducted to deal with this problem [Brebner and Diessel, 2001, Noguera and Badia, 2002, Steiger et al., 2004, Banerjee et al., 2005b, Singhal and Bozorgzadeh, 2006].

Several subjects in dynamic reconfiguration research belong to one or more of the above topics. An interesting issue is whether it is preferred to use memory to store parts of the design rather than deploying the whole design on the FPGA fabric. Previous studies suggest that in terms of silicon area it is preferred to store inactive designs in cheaper non-volatile memory [MacBeth and Lysaght, 2001]. On the other hand, no results exist regarding the comparison between the memory needed for the RTR approach and the circuit area of its static counterpart. Another design problem is the division of an application into segments that do not execute concurrently, which is referred to as temporal partitioning [Wirthlin and Hutchings, 1995b]. Another problem involves the inter-configuration communication, i.e. the transmission of results from one configuration to the next one [Wirthlin and Hutchings, 1995b].

Latest efforts and achievements demonstrate that dynamic reconfiguration is an emerging area in the design of reconfigurable systems. If it is accessed in an effective manner it could be the vehicle for applications that otherwise would be difficult or less efficient implemented. There are many known conferences and workshops targeting dynamic reconfiguration [DRS-ARCS, 2007], or raising it as a main topic [FPL, 2011, RAW, 2008, FPL, 2008, FCCM, 2008, ERSA, 2008, FPT, 2007, FPL, 2006]. In present era where multiprocessor systems have started to be incorporated within modern systems, reconfigurable computing



(a) ASIC approach (b) FPGA approach (c) FPGA RTR approach

Figure 2.1: Three different RTR approaches to design an equivalent system [Guccione and Levi 1999].

should deploy its most advanced features like dynamic reconfiguration to be more widely accepted.

2.4 Theoretical Study of Dynamic Reconfiguration

Several researchers have attempted to model the performance of dynamic reconfiguration rather than to conduct any sort of experiments due to the intrinsic difficulties of technology and the lack of supporting tools. As reported previously, most applications of FPGAs do not make full use of the reprogrammable nature of the devices. Typically, they are programmed once and then execute without modifying the configuration data. It is important to investigate theoretically whether the cost of dynamically implementing an application is amortized over the cost of an alternative approach before entering the tedious design flow of dynamic reconfiguration. This Section revisits some representative theoretical approaches.

2.4.1 RTR approaches

In [Guccione and Levi, 1999], a study listing the design issues of run-time reconfiguration and its benefits when applied properly is presented. It claims that its efficient utilization can dramatically reduce the amount of logic, software complexity and IO comparing to other FPGA-based systems. A general definition of RTR is given stating that a system can be said to be run-time reconfigurable if it uses run-time data to alter the function of hardware. Although this is a broad definition, it helps determine what is unique about FPGA reconfigurability and how it can be beneficially employed. According to this concept

and the way RTR is used in different systems three types are distinguished.

First, an ASIC approach of RTR is reported which is based on writable registers that are written by software at run-time through a data bus. The approach is referred with this term as there is nothing that requires FPGA technology to produce this type of RTR circuit. Figure 2.1(a) illustrates this with a block-level diagram of a datapath circuit used to add a constant value M to some input data D_{in} and multiply the result by some other constant value N supplying the function $D_{out} = (D_{in} + M) * N$. The two constant values are held in two registers which may be written by software at run-time. The interface circuitry adds complexity to the system by means of an address decoder for the system address bus, a data bus to and from the registers, and Input Output Buffers (IOBs) to send to and receive from the circuit. This also entails analysis of timing specification of the bus interface. Furthermore, device drivers for the software to communicate with the hardware are needed along with some sort of library interface to the driver which provides a higher level communication to the underlying hardware. Finally, if the hardware platform of the underlying software changes, drivers and libraries need to be rewritten.

Second, the FPGA approach of RTR eliminates the interface circuitry of Figure 2.1(a). This is illustrated in Figure 2.1(b) where decode logic and IOBs are not included. Moreover, the design does no longer require extensive analysis to perform bus interfacing and the software for register interface, i.e. device drivers and libraries, is eliminated. Supporting software to configure the FPGA is still necessary but it will not be application specific like in the previous approach. In this case, software tools which set the appropriate constants in the circuit configuration registers are needed. The latter are updated with constant values using RTR and no actual logic or routing in the FPGA is modified at run-time.

Third, the FPGA RTR approach, instead of using general purpose units with constants stored in registers, configures specialized units at run-time. Constants are folded into the hardware implemented units. As illustrated in Figure 2.1(c) constant coefficient arithmetic units are used instead of a general purpose multiplier, an adder, and register inputs. The coefficients can also be generated at run-time as the constant values may not be known at compile time and generating all possible configurations will quickly become large. With this approach, size and complexity of the circuit are reduced more dramatically, and the inter-

Table 2.2: Design issues, logic and software requirements for the three RTR approaches of the example system.

Design issues	ASIC approach	FPGA approach	FPGA RTR approach
Adder	general-purpose	general-purpose	specialized
Multiplier	general-purpose	general-purpose	specialized
Registers	hold constants	hold constants	no
Intra-communication	large # of buses	medium # of buses	small # of buses
Address decoder	decodes the data on bus	no	no
System bus	access to registers	no	no
IOBs	send/receive system bus	no	no
Device drivers	app-specific	not app-specific	not app-specific
Libraries	app-specific	not app-specific	not app-specific
Reconfiguration time	low	low	high

connection amongst the components is greatly simplified. One potential drawback of this approach is the amount of configuration data which must be written to the FPGA device. In the previous register-based approaches, only small amounts of data were involved in customizing the behavior of the circuit. The FPGA RTR approach results in fully customized circuits, which entails a substantial configuration cost. Whether the added overhead is excessive depends not only on the FPGA device to be used, but on the application needs as well.

The design issues and requirements of the three approaches are viewed in Table 2.2. ASIC approach requires the most of routing, logic and software. The latter two approaches lead to decreased system costs by using an FPGA device. A hybrid of those two approaches could be considered that would balance trade-offs with regard to resource usage, software overhead, intra-communication and reconfiguration depending on the available hardware and the application needs.

2.4.2 Studying RTR Performance with Mathematical Models

The authors in [Wirthlin, 1997, Wirthlin and Hutchings, 1997] examine the improvement of the run-time implementation of a circuit over its static counterpart using mathematical equations. A performance metric is initially defined, which is inversely proportional to the

time required to complete a computation:

$$P = \frac{1}{T} \tag{2.1}$$

where, P the performance in latency, and T the operating time of a computation. For n number of operations, performance can be measured in terms of throughput:

$$P = \frac{n}{T_n} \tag{2.2}$$

where, P the performance in throughput, n the number of operations, and T_n the operating time of n computations. However, performance gain do not come without cost and a metric is used to measure the cost-performance with respect to the hardware unit resources required for a computation:

$$Cost - Performance = \frac{1}{Circuit\ Area \times Execution\ Time} \tag{2.3}$$

This composite area-time identifies the computational throughput of hardware unit resources, i.e. the density of computation among FPGA resources. It is termed functional density and it is defined as:

$$D = \frac{1}{AT} \tag{2.4}$$

or,

$$D = \frac{n}{AT_n} \tag{2.5}$$

where, A the circuit area, and T and T_n the operating time of a computation and of n computations respectively. Functional density is used to compare a run-time reconfigured circuit with the corresponding static circuit. If the former results in higher functional density it will improve the cost-effectiveness of the computation. The improvement in functional density is evaluated by computing the normalized difference in functional density between the RTR system (D_{rtr}) and the static alternative (D_s):

$$I = \frac{\Delta D}{D_s} = \frac{D_{rtr} - D_s}{D_s} = \frac{D_{rtr}}{D_s} - 1 \tag{2.6}$$

The improvement I, can be measured as a percentage by multiplying with 100. When the functional density of the RTR system is greater than that of the static system, I is positive and RTR is justified. When it is smaller, I is negative and RTR is difficult to be justified.

In a static circuit the operating time is equal to the execution time, i.e. $T = T_e$ whereas a run-time reconfigured circuit has the added cost of configuration, i.e. $T = T_e + T_c$. Hence, the RTR functional density becomes:

$$D_{rtr} = \frac{1}{A(T_e + T_c)} \tag{2.7}$$

where, A the circuit area, T_e the execution time and T_c the configuration time. The cost of reconfiguration can be reduced by increasing the number of operations n completed before reconfiguring the hardware. Stating differently, the more operations n are completed, the more time is spent in execution, and the less time is spent in configuration. For the completion of n operations between configuration, the functional density of Equation 2.7 can be modified as follows:

$$D_{rtr_n} = \frac{1}{A(T_e + \frac{T_c}{n})} \tag{2.8}$$

This represents that the more operations n are completed between configurations, the higher the RTR functional density will be.

Another metric used to examine RTR systems is the configuration ratio which is expressed as the ratio of configuration time over execution time:

$$f = \frac{T_c}{T_e} \tag{2.9}$$

By substituting the configuration time in Equation 2.7, functional density can be expressed in terms of f:

$$D_{rtr} = \frac{1}{AT_e(1 + f)} \tag{2.10}$$

In the limit where $T_e \gg T_c$, i.e. $f \to 0$ the configuration time is negligible. This

corresponds to an RTR system with the maximum functional density:

$$D_{max} = \lim_{f \to 0} D_{rtr} = \frac{1}{AT_e} \tag{2.11}$$

In Equation 2.10, by substituting the term $1/AT_e$ with D_{max}, functional density can be represented in terms of D_{max}:

$$D_{rtr} = \frac{D_{max}}{1+f} \tag{2.12}$$

It is clear that the functional density of an RTR system degrades as the configuration ratio increases. On the other hand, when more operations are completed between reconfigurations, the cost of configuration on a per-operation basis is reduced. Thus when the execution time is much longer than reconfiguration time, i.e. $T_e \gg \frac{T_c}{n}$, the functional density for n operations of Equation 2.8 is reduced to:

$$D_{rtr_n} = \frac{1}{AT_e} \tag{2.13}$$

In another research work a simple metric is used to compute the ratio of the processing time of an application executed on a run-time specialized circuit over the processing time of a general purpose circuit [McKay et al., 1998,McKay and Singh, 1998]. Partial evaluation is employed to study systematic specialization of circuits by taking a general circuit and some data known at run-time, and then the general circuit is transformed into a specialized one. This work proposes dynamic synthesis of circuits according to slowly changing inputs and reprogramming of the hardware on the fly. An example of a decryption circuit is considered in which the key is modified infrequently at run-time. In this case, a data stream consists of a specialization parameter, i.e. the key, followed by n data items, i.e. the decrypted data. Two expressions are given, one for the time needed to process the data by the general purpose circuit and one for the specialized circuit. The expression for the conventional circuit is:

$$T_{conv} = T_k + nT_g \tag{2.14}$$

where, T_k the time to load the specialization parameter, n the number of data items and

T_g the cycle time for the general circuit. The expression for the dynamic circuit is:

$$T_{dyn} = (T_s + T_p) + nT_c \tag{2.15}$$

where, T_s the time to synthesize hardware, T_p the FPGA programming time, n the number of data items and T_c the cycle time for the specialized circuit. In order for the partial evaluation to be proved useful for a given circuit the ratio T_{dyn}/T_{conv} must be less than 1. In this approach the important question is when the cost of calculating new configurations, i.e. specialized circuits, is amortized over sufficient execution time to make the approach worthwhile. Identifying applications in which n is sufficiently large and T_c/T_g sufficiently small will make the approach worthwhile. Although this is an interesting study it does not take into consideration the area factor; it examines the performance of a RTR system over a static one only with respect to speedup.

2.4.3 Other Approaches

The authors in [MacBeth and Lysaght, 2001] study the problem from a different perspective. They use set theory to model a set of circuits that are dynamically placed on an FPGA, while they attempt to identify generic classes of circuits that benefit from being dynamically reconfigured. A recent work proposes a novel analytical model based on queuing theory for studying systems coupling a processor with partially reconfigurable hardware [Lotfifar and Shahhoseini, 2008]. First, it models the operating system, the hardware and the tasks entering the system. The system is formulated as a queuing system, and equations are used to evaluate its performance. Finally, the results of the model are compared with simulation results. This work presents interesting ideas, but at the same time it does not conform to the existing technology and it relies on assumptions that do not correspond to reality. Thus it does not consider contemporary reconfigurable devices that embed RAMs and heterogeneous resources, and that reconfiguration is performed in a column-basis, while it assumes that routing resources are unlimited.

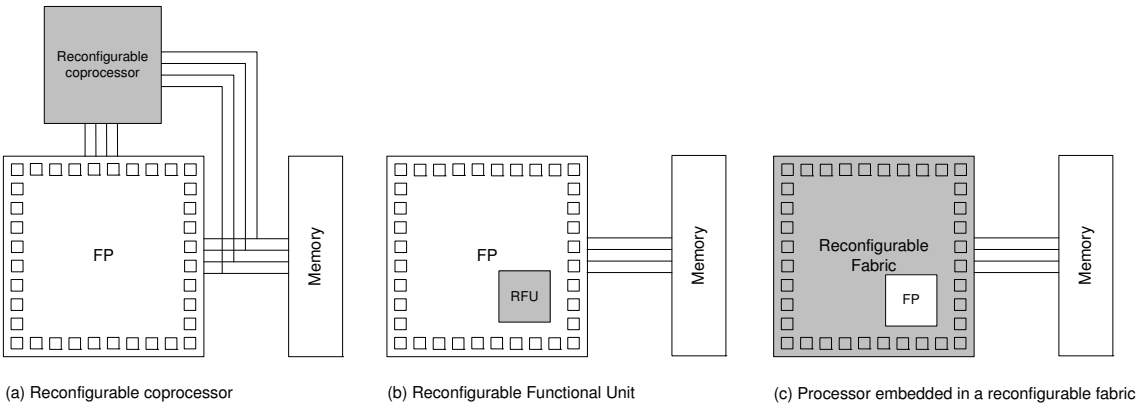


Figure 2.2: Levels of coupling between microprocessor and reconfigurable hardware.

2.4.4 Discussion Summary

The aforementioned works are indicative of the effort put to model RTR. Even though some of them are relatively old they still apply to contemporary systems. The work in [Guccione and Levi, 1999] demonstrates that FPGA RTR approach can outperform alternative approaches. Other works employ mathematical equations to model the performance of dynamic reconfiguration regardless of the device architecture. Toward this direction, the work in [Wirthlin, 1997, Wirthlin and Hutchings, 1997] evaluates the extent to which an application implemented with RTR outperforms the static counterpart. In addition, work has been done on evaluation methodologies relying on queuing theory which target specific device families [Lotfifar and Shahhoseini, 2008].

Toward the concept of modeling RTR with regard to speed and area parameters, it is also worthwhile to assess the power consumed in such systems. Run-time reconfiguration offers the capability to load hardware modules only when they are needed, which allows for power savings of the FPGA chip during the time the modules reside off-chip. On the other hand, run-time reconfiguration process consumes power itself. Thus, a holistic study will reveal the extent to which a partial reconfigurable implementation can result in less power consumption over its static alternative.

2.5 Architectures

Section 2.2 introduced numerous multi-context and partially reconfigurable architectures. Present Section details some of these architectures. Firstly, it concentrates on systems combining a fixed processing unit (FPU) and a reconfigurable processing unit (RPU) onto a single chip. Such systems couple closely the two resources and allow for low reconfiguration delay and fast intercommunication. Figure 2.2 depicts three different architecture types which are classified with respect to the level of coupling:

- Reconfigurable coprocessor: The RPU lies close to the FPU onto the same chip and their communication is performed through a bus. The two units can execute simultaneously and the RPU can operate for a large number of cycles without any intervention from the FPU. The RPU has direct access to the memory hierarchy of the FPU.

- Reconfigurable Functional Unit (RFU): The RPU is tightly coupled with the FPU as part of the latter's datapath. The RPU extends the instruction set of the FPU with custom instructions that can change over time and by taking part in the FPU pipeline execution. The two units communicate frequently.

- FPU embedded in reconfigurable fabric: The FPU is incorporated in the RPU. It can be either hardcore or softcore (implemented with reconfigurable resources). The RPU is larger than in the previous architecture type.

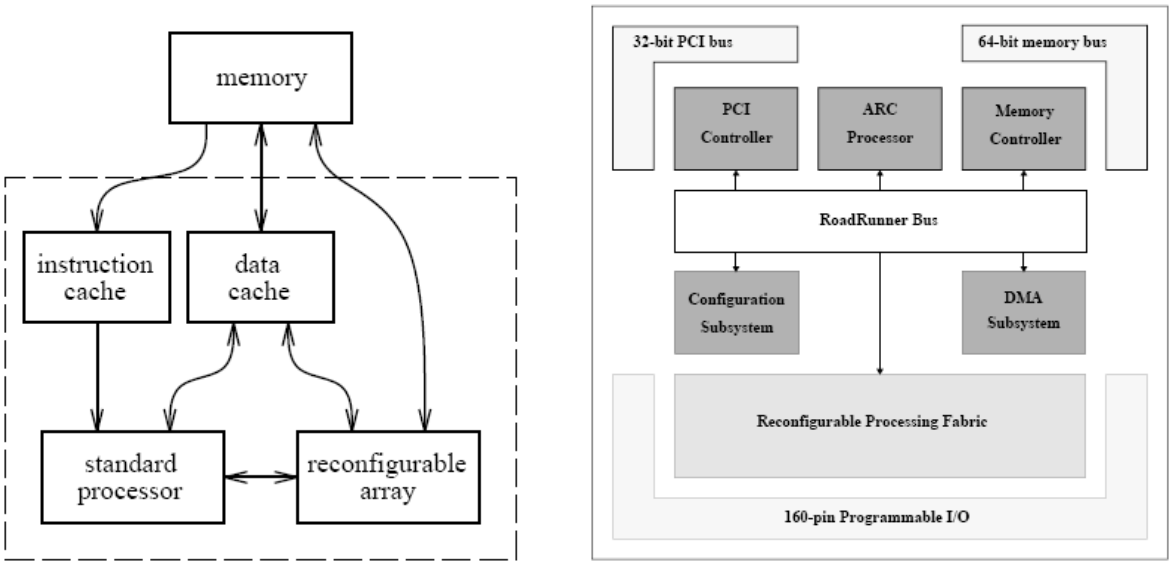
Similar classification can also be found in [Compton and Hauck, 2002, Enzler, 2004, Todman et al., 2005]. In these works more topologies are studied such as attached processing units [Schmit, 1997] which constitute a loosely coupling approach of the reconfigurable coprocessor architecture, and external stand-alone processing units in which the two resources communicate infrequently [Quickturn, 1999]. These are loosely-coupled architectures, and in general reconfiguration is triggered less frequently than in the closely-coupled architectures. Also, the former systems suit applications with high demands in computational power and not in flexibility. The present Section does not study such systems. A later Subsection is devoted on architectures with novel structure that do not incorporate a microprocessor

on the same die. A comprehensive list of closely-coupled architectures, both dynamically and not dynamically reconfigurable, is available in [Berkeley, 2007].

The remaining Section classifies reconfigurable processors belonging to the closely-coupled architectures. The main criterion of the classification is the level of coupling. In addition, some of the following information is given for each device:

- The institution or company that developed it
- The structure (partial or multi-context) and the type of reconfiguration (static or dynamic)
- A block diagram of the architecture (in some cases)
- A brief description of the system-level architecture and operation
- Details of the reconfigurable array (RA)
- Details of the reconfiguration controller, i.e. a microprocessor or a dedicated controller, and whether reconfiguration is performed through DMA
- The basics of system operation and reconfiguration operation, e.g. register scheme, memory hierarchy scheme, shared memory or dedicated memory port can be used for the transfer of data between the microprocessor and the reconfigurable hardware
- Any special features such as mechanisms to relieve the system from reconfiguration overhead
- Target application domain
- Status of the device, i.e. whether it was commercialized, or, reached the simulation, emulation or fabrication stage

At the end of present Section reconfigurable devices are tabulated. They are mainly distinguished by the type of system-level coupling. Details such as the FPU type and the granularity size of the RPU are included. Also, it is reported whether an architecture is multi-context or partially reconfigurable. These structures can also be combined in one device, thus an additional information concerns whether in a multi-context device the inactive



(a) Garp (b) Chameleon RCP

Figure 2.3: Block diagram of reconfigurable coprocessor architectures.

context can be reconfigured partially or fully only. Finally, it is reported if reconfiguration is dynamic or static.

2.5.1 Reconfigurable coprocessor

A representative device of this architecture is the Garp processor illustrated in Figure 2.3 (a), which was developed at the University of California, Berkeley [Hauser and Wawrzynek, 1997]. It supports static partial reconfiguration. It comprises a single-issue MIPS-II FPU and an RPU. The two units cannot execute concurrently meaning that whenever a reconfigurable function is called, the FPU becomes inactive and the RPU carries out the computation of this function. Communication amongst the two resources is accomplished through registers. A memory hierarchy model is used to avoid memory consistency problems. The FPU shares its data cache with the RPU. The latter can access the data cache or memory independently of the FPU. The RPU is a homogeneous, fine-grain 2-D reconfigurable array composed of 2-bit logic blocks. Loading and execution of configurations is controlled by the FPU with dedicated instructions. The smallest configuration unit of the array is one row, and every configuration fills exactly a specific number of contiguous rows.

Off-line reconfiguration is supported only and thus when part of the array is configured the unused rows automatically become inactive. Two independent configurations cannot be active at the same time; there is no way to guarantee that they will not interfere with each others' use of vertical wires. If there is a need to program the array with two different operations at a time, the corresponding configuration data can be packed together into one large configuration file. In order for the reconfiguration to be quickly completed, the RPU has a high bandwidth connection with the main memory and a private configuration cache. The architecture includes a distributed configuration cache to hold recently used configuration for fast reloading. Configuration prefetching is also supported to reduce reconfiguration overhead. Garp targets acceleration of specific loops or subroutines. Thus it targets sequential algorithms spending most of their execution time in relatively small pieces of looping code. The RPU is partially reconfigured to serve efficiently a small amount of code. Garp was an academic project with innovative concepts, and its development stopped at the simulation stage.

The CS2112 RCP developed by Chameleon Systems [Chameleon, 2000] is a multi-context device supporting partial reconfiguration of an inactive context while the other executes. Its architecture is depicted in Figure 2.3 (b). It comprises a 32-bit FPU, a 32-bit RPU, a high-speed system bus and a programmable I/O system. The FPU interfaces with the memory and other processing systems through a PCI-controller, memory and DMA controllers. The different components are linked with an on-chip 128-bit bus offering 2 GBytes/sec bandwidth. The RPU is a heterogeneous, coarse-grain 2-D array of programming elements (PE) that contain ALUs, multipliers and distributed local memory. The FPU schedules the computational-intensive tasks to be executed by the RPU. A configuration bitstream is stored in the main memory and then loaded into the RPU at runtime through DMA. The RPU is divided into slices each of which consists of three tiles and can be independently reconfigured. Each slice has two configuration planes for bitstreams. An active plane executes the working bitstream and a back plane contains the next configuration bitstream. Switching from the back plane to the active plane takes one cycle. Thus, the back plane can be effectively used as cache for loading configuration. Furthermore, prefetching is supported to hide the configuration latency [Tang et al., 2000]. CS2000 family targets the domain of

telecommunications, and especially emerging technologies in which standards, protocols and algorithms need to change continuously. It was not released to the market due to weak macroeconomic reasons in the telecommunication sector at that time.

The DAPDNA-2 developed by IPFlex [IP Flex, 2008, Sato et al., 2005] is a multi-context device supporting partial reconfiguration of an inactive context while the other executes. It comprises a 32-bit RISC FPU, a 32-bit RPU, a high-speed system bus and several peripherals such as a PCI interface, a DDR-SDRAM interface, a UART and a DMA controller. The FPU and the RPU can execute concurrently. The components are linked with an on-chip bus offering 32 Gbps bandwidth. The RPU is a heterogeneous, coarse-grain 2-D array of PEs that contain ALUs, memory, synchronizers, and counters. The FPU controls the dynamic reconfiguration of the RPU. The latter has four banks of configuration memory and can switch among them in one clock cycle. Additional configurations can be loaded from external memory into one of the background banks while the foreground is in operation. DAPDNA-2 was designed mainly for high-end image processing applications. It can be rapidly reconfigured to accelerate loops and repetitive processes. It was released to the market in 2003.

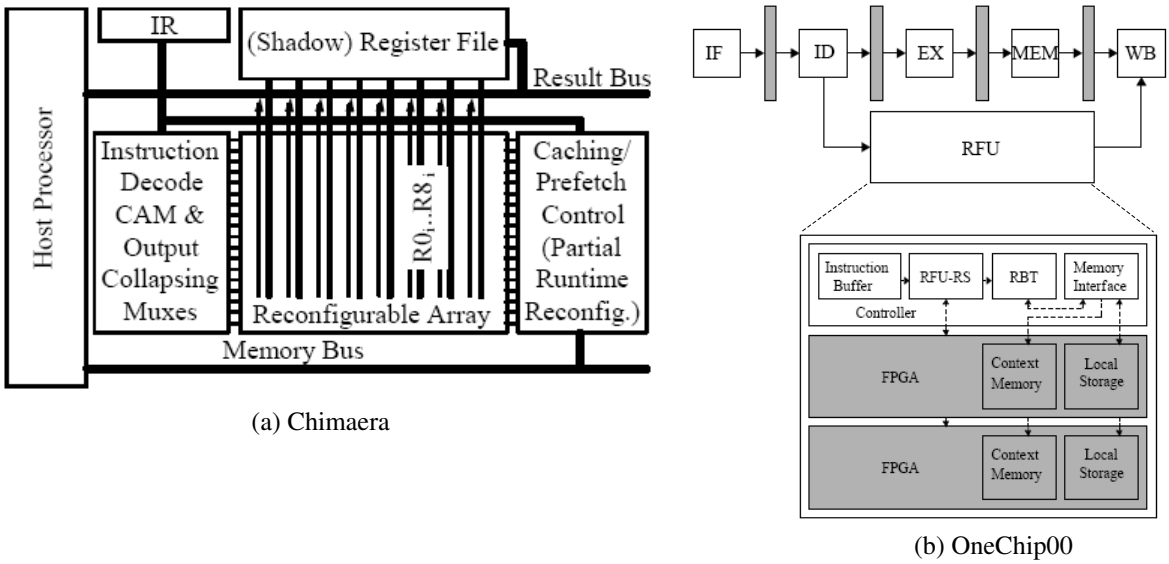
The PSoC developed by Cypress Semiconductor [Cypress, 2008], is a low cost, low speed reconfigurable microcontroller combined with a mixed-signal reconfigurable array that supports dynamic partial reconfiguration. It comprises an 8-bit microcontroller, configurable digital and analog blocks, a few fixed peripherals and programmable interconnect. Complex peripherals can be implemented by combining blocks. PSoC can be viewed as a system falling into the reconfigurable coprocessor architectures due to that the FPU communicates with the peripherals through a programmable system bus. However, unlike FPGAs, the PSoC cannot have its digital functions reprogrammed by HDL; it is configured only with register settings. The RPU is a mixed-signal heterogeneous, coarse-grain array of PEs that can be configured as comparators, A/D and D/A converters, timers, counters, pseudo-random generators, UARTs, filters, amplifiers and so on. The FPU controls the dynamic reconfiguration of the RPU with dedicated instructions. A certain amount of registers can be configured at run-time for each digital and analog block. Configurations can be loaded from flash memory while the system operates. PSoC was designed for low cost and low

power applications and was released to the market in 2002.

Several other devices fall into this architecture type. The DPGA coupled with a microprocessor, was one of the first attempts combining an FPU and an RPU in a single die. It is a homogeneous fine-grain RPU in which the FPU controls the reconfiguration process. The DPGA supports background partial loading of the inactive context but not of the active context. It was developed in MIT [DeHon, 1994] and reached the stage of fabrication. The NAPA 1000 by National Semiconductors [National, 1996], contains a homogeneous fine-grain RPU and the reconfiguration process is undertaken by a reconfiguration controller instead of the FPU. The FIPSOC [Faura et al., 1997] by SIDSA is a mixed-signal architecture with two contexts, which are mapped directly into the FPU's address space who controls the reconfiguration process. The MorphoSys by the University of California at Irvine [Singh, 1998] contains a multi-context homogeneous coarse-grain RPU. Context loads are specified through the FPU and executed by DMA controller. It reached the stage of fabrication. The REMARC by Stanford University [Miyamori and Olukotun, 1998] contains a homogenous coarse-grain RPU, and the FPU controls the reconfiguration process. It reached the stage of simulation. The ZIPPY by the Swiss Federal Institute of Technology [Enzler and Platzner, 2001] integrates a multi-context homogenous coarse-grain RPU with an FPU for controlling the reconfiguration process. It reached the stage of simulation.

2.5.2 Reconfigurable Functional Unit

Chimaera by the University of Washington and Northwestern University [Hauck et al., 1997] is a reconfigurable processor supporting partial reconfiguration. The architecture is depicted in Figure 2.4 (a). It comprises a MIPS FPU and an RPU consisting of combinatorial logic only, tightly integrated with the FPU pipeline. A shadow register file that duplicates a subset of the values in a host register file provides the inputs to the RPU. A set of Content Addressable Memory (CAM) locations, one per row in the reconfigurable array, determines which of the loaded instructions are completed. RFU calls are made with special instructions informing the processor to execute an RFU instruction. As part of this RFU call, an instruction ID determines the instruction to be executed. Every time a new instruction is needed to be loaded into the RFU a reconfiguration takes place. A dedicated controller



(a) Chimaera

(b) OneChip00

Figure 2.4: Block diagram of reconfigurable functional unit architectures.

undertakes the reconfiguration process. Partial reconfiguration is supported on a per-row basis with one or more rows making up a given RFU instruction; contiguous set of rows are configured to hold the new instruction(s). While a configuration is being loaded, execution is stalled [Ye et al., 2000]. When an RFU operation (RFUOP) is loaded into the array, one or more rows are designated as output rows by placing the RFUOP ID into the CAM line corresponding to that row. If the value in the CAM matches the RFUOP ID, the value from that row in the reconfigurable array is written onto the result bus and sent back to the register file. If the instruction corresponding to the RFUOP ID is not present, the caching/prefetch control logic stalls the processor and loads through partial reconfiguration the proper RFU instruction from memory into the reconfigurable array. All loaded RFU instructions speculatively execute during every processor cycle, though their results are written back to the register file only when their corresponding RFU call is actually made. The caching/prefetch logic minimizes the cost of RFU misses; instructions that have recently been executed or predicted that might be needed soon are kept in the RPU. In case another instruction is required, it is brought into the RPU by overwriting one or more of the currently loaded instructions. Chimaera was an academic project that reached the fabrication stage [Hauck et al., 2004].

OneChip by the University of Toronto [Wittig, 1995, Jacob, 1998, Esparza, 2000], is a multi-context reconfigurable processor that supports reconfiguration of an inactive context in its entirety while the other executes. Figure 2.4 (b) has a block diagram of its architecture. It comprises a DLX FPU and an RPU tightly integrated into the former's pipeline. The RPU is integrated in parallel with the EX and MEM stages of the pipeline, performs computations as a functional unit on the main datapath, and shares the same register file and memory with the FPU. Instructions targeting the RPU are forwarded to the FPGA Controller, which contains the reservation stations (RS), a reconfiguration bits table (RBT), a buffer for the instructions and the memory interface. The FPGA Controller is responsible for programming the FPGAs, the context switching and selecting the configurations to be replaced. The RS are responsible for handling data dependencies between RPU and FPU instructions. The RBT keeps track of the location of FPGA configurations. The memory interface consists of a DMA controller for transferring configurations from memory into the context memory according to the values in the RBT. It also transfers the data that an FPGA will operate on into the local storage. The local storage may be considered as the FPGA data cache memory. Multiple FPGAs in the RPU - indicatively two as shown in Figure 2.4 - share the same FPGA Controller and each FPGA has its own context memory and local storage. Each context of the FPGAs is configured independently from the others and acts as a cache for configurations. As the project evolved several changes were proposed. Thus after the system was introduced [Wittig, 1995] several features were added like a memory-consistent interface to support concurrent operation of the processor and the RPU [Jacob, 1998], as well as concurrent operation of two or more FPGAs of the RPU since each FPGA may access data that another FPGA is also accessing [Esparza, 2000]. In the latter work a multiple-issue out of order microprocessor was integrated. In addition, techniques such as configuration compression, configuration preloading and a configuration management scheme were proposed. The system was conceptualized to support dynamic reconfiguration, but the platform on which the system was emulated did not support dynamic reconfiguration. OneChip was designed to speedup mainly memory streaming applications. It was an academic project and its development reached the emulation phase.

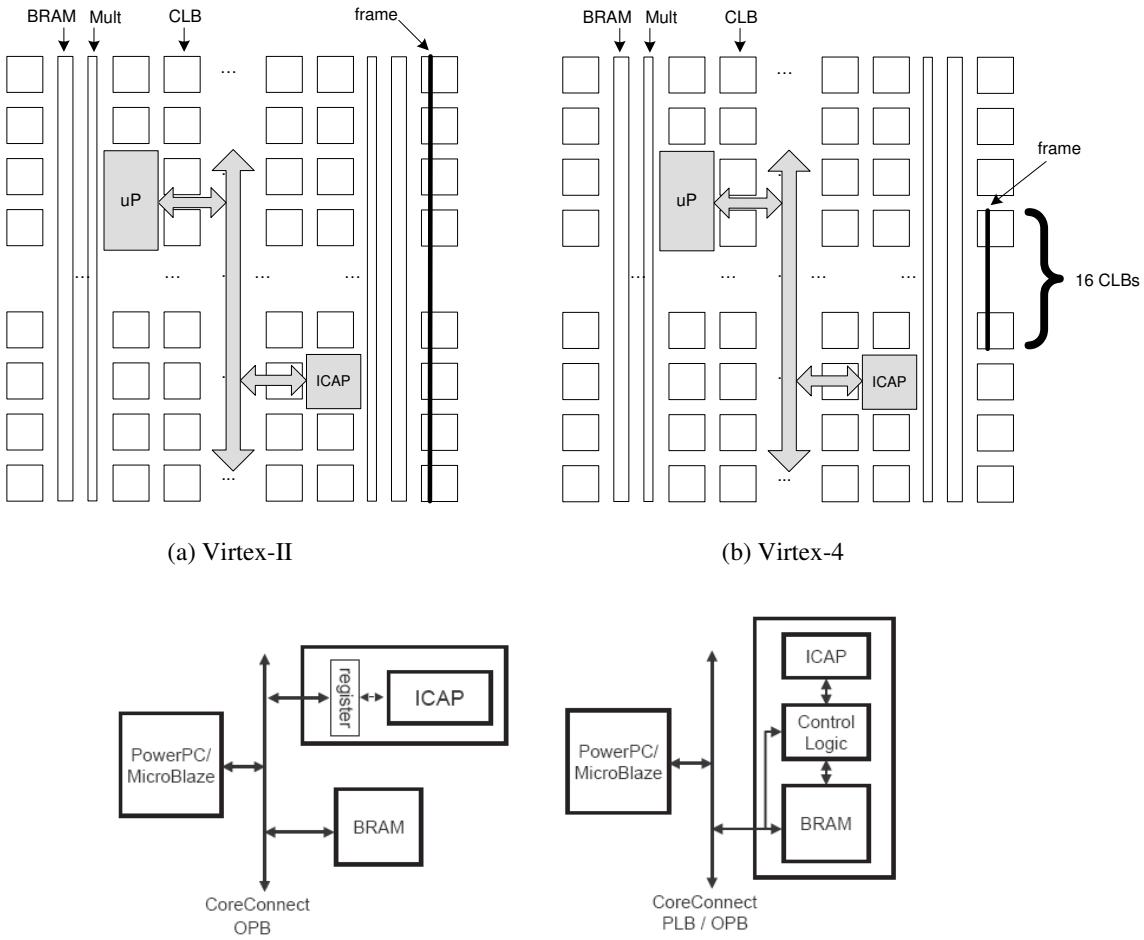
The XiRisc by the University of Bologna and the STMicroelectronics [Lodi et al., 2003]

is a multi-context device supporting partial reconfiguration of an inactive context while another executes. It comprises a VLIW FPU and an RPU called PiCoGa. The two resources share the same register file. The RPU is a homogeneous, medium-grain 2-D array composed of 2-bit logic blocks. Dynamic reconfiguration is controlled by the FPU through a simple interface, which provides information about the context and the RPU region to be reconfigured. A special assembly instruction can configure the RPU from an on-chip configuration cache. To avoid stalls due to reconfiguration, several configurations may be stored inside the RPU. Up to four different functions can be mapped in the same context. Different configurations each corresponding to a different RPU function can be simultaneously loaded into different regions of the same context. The RPU can preserve its state across executions of functions. A new RPU function may use the results of previous ones stored on the RPU, thus reducing the pressure on the register file. A wide configuration bus is incorporated allowing for low reconfiguration time. The XiRisc targets reducing the power consumption in embedded applications and reached the fabrication stage.

Molen is a reconfigurable processor developed at the Delft University of Technology. First, it was announced to support static partial reconfiguration [Vassiliadis et al., 2001], while in a later work it was claimed to support dynamic partial reconfiguration [Wong et al., 2002]. It comprises a superscalar FPU and an RPU. The two units cannot run concurrently [Vassiliadis et al., 2004]. An arbiter decodes the instructions fetched from the main memory in order to issue them to the proper processing unit. A data fetch unit handles data transfers from/to the main memory. Another unit is responsible to distribute data between the RPU or the FPU. Data transfer between FPU and RPU is performed via special exchange registers. The RPU, denoted as Custom Configured Unit (CCU), consists of a fine-grain 2-D reconfigurable array and memory. Part of an application code is loaded into the RPU in order to speed up program execution. Operations in the RPU are divided mainly into two distinct phases: Set and Execute. In the Set phase, CCU is configured in order to support the operations. Next, in the Execute phase, the operations are executed. This decoupling allows the Set phase to be scheduled ahead of the Execute phase, thereby hiding the reconfiguration latency. This policy reflects a configuration prefetching mechanism. Also, caching of configurations is supported. It should be noted that no

specific instructions are associated with operations to configure and execute on the CCU as this would need a considerable amount of opcode space. Instead, pointers to memory locations where the reconfiguration or execution microcode is stored are utilized. In this way, different operations are performed by loading different reconfiguration and execution microcodes. Molen architecture deals with the problem of opcode space explosion, as it requires only a one time extension of the instruction set to incorporate an almost unlimited number of reconfiguration functions per single programming space. It was implemented on an FPGA-based platform.

Several devices exist belonging to the RFU architecture type. The PRISC developed at the Harvard University was one of the fist attempts that envisioned a tight architecture [Razdan and Smith, 1994]. The concept is extended in that the reconfigurable logic can be programmed while the processor executes. The DISC developed at the Birgham Young University [Wirthlin, 1997] also suggested a tight architecture. Whenever a new custom instruction is needed the RPU stalls and it is configured on a row-basis by the FPU. The DISC was emulated on a real platform on which a dedicated configuration controller undertook the reconfiguration process. An off-chip processor was used instead of an integrated processor as a proof-of-concept. A work at the Politecnico di Milano [Pozzi, 2000] proposed a methodology for tightly integrating a VLIW FPU with a multi-context, fine-grain RPU. The FPU is responsible for the reconfiguration process. The architecture reached the phase of simulation. Elixent and Toshiba cooperated in the production of D-Fabrix, which coupled a homogeneous medium-grain reconfigurable array [Elixent, 2000] and the Toshiba MeP configurable processor onto the same die [Toshiba, 2003]. In this product the processor controls reconfiguration through DMA. D-Fabrix was the evolution of the CHESS array [Marshall et al., 1999] which supported static partial reconfiguration. CoMPARE developed at Dresden University of Technology [Sawitzki et al., 1998] assumes one custom instruction per RFU reconfiguration. It can execute instructions in both RFU and ALU individually, in parallel or sequentially, as a super-scalar extension. It reached the emulation stage. The Proteus processor developed at the University of Glascow [Dales, 1999] combines an FPU with a fine-grain RPU in which the FPU controls the reconfiguration process. The ADRES is an architectural template developed at IMEC [Mei et al.,



(a) Virtex-II (b) Virtex-4

(c) 1st generation SRP hardware architecture (d) 2nd generation SRP hardware architecture

Figure 2.5: Block diagram of architectures with FPU embedded in reconfigurable fabric.

2003] that contains a multi-context heterogenous coarse-grain RPU. Only one of the FPU and the RPU can be active at any time. The FPU is responsible for programming the configuration contexts [Mei et al., 2004]. A work conducted in DTU [Wu, 2007] extended the ADRES architecture with multithreading, which allowed for concurrent FPU execution and RPU execution or reconfiguration.

2.5.3 FPU embedded in reconfigurable fabric

Some of the widely used FPGA devices with dynamic partial reconfiguration capability that belong to this category are Virtex-II [Xilinx, 2007d] and Virtex-II Pro [Xilinx, 2007e] families by Xilinx. Their structure is shown in Figure 2.5 (a). Both families can be configured

by a softcore FPU, such as the MicroBlaze and PicoBlaze, whereas in the Virtex-II Pro the hardcore PowerPC can be used [Blodget et al., 2003]. The FPU and the RPU can execute concurrently. They communicate through the CoreConnect bus which provides the high bandwidth Processor Local Bus (PLB) and the slower On-Chip Peripheral Bus (OPB) [IBM, 2007]. FPU can access RPU using a memory mapped interface. Alternatively, MicroBlaze can communicate with the RPU through a low-latency interface called Fast Simplex Link (FSL) which can be attached to the former's pipeline. FSL can be used to extend the FPU's instruction set with custom instructions implemented in RPU, but does not make the speed of the FPU pipeline dependent on the custom function [Xilinx, 2007a]. The RPU in this case has a low latency access to the FPU register file. The RPU is a heterogeneous, fine-grain 2-D array mainly composed of configurable logic blocks (CLBs), memory blocks (BRAMs), digital signal processing blocks (DSPs), and clock management circuits. Loading of configurations can be controlled by the FPU through the Internal Configuration Access Port (ICAP) with dedicated instructions. The ICAP is 8-bit and can run at 100 MHz. Figure 2.5 (c) shows the block diagram of the implementation of the 1^{st} subsystem supporting self-reconfiguration. A 32 bit register is used to interface with the ICAP port. The control logic for reading and writing data to the ICAP is implemented in a low-level software driver. One BlockRAM (BRAM) is used to cache configuration data. In the 2^{nd} generation subsystem of Figure 2.5 (d) several modifications were made to achieve better performance. The ICAP is completely implemented in hardware and the BRAM is moved inside the ICAP peripheral allowing exploitation of its dual port nature. In this way simultaneous transfer from the system bus to the BRAM and from the BRAM to the configuration memory is achieved. Moreover, the ICAP control peripheral is able to fetch configuration data directly from external memory without requiring the FPU involvement. Configuration memory of the Virtex-II/Pro is arranged in vertical frames that are one bit wide and stretch from the top edge to the bottom of the device. Frames are the smallest addressable segments of the device's configuration memory space. In Virtex-II/Pro families the number of frames is proportional to the CLB width of the device whereas the number of bits per frame is proportional to the CLB height of the device. Frames do not directly map to any single piece of hardware; rather, they configure a narrow vertical slice of many

physical resources. A pad frame is required to be added at the end of the configuration data which flushes out the reconfiguration pipeline in order for the last valid frame to be loaded [Xilinx Inc., 2005c]. Therefore, to write even one frame to the device it is necessary to clock in two frames; the data frame plus a pad frame. As previously mentioned Virtex-II/Pro devices have heterogeneous physical resources. Configuration frames are grouped into six column types that correspond roughly to physical device resources (for sake of clarity they are not depicted in Figures 2.5 (a) and (b)). The number of frames per column type is constant for all devices whereas the distribution of frames between CLB columns and other resource columns might be different, e.g. 22 frames configure an entire CLB column while 64 frames are needed for a BRAM column. Moreover, memory elements that define the content of a lookup table (LUT) are all located in one frame. However, this does not hold for all resources. Some resources have their configuration bits scattered across multiple frames [Blodget et al., 2003].

Figure 2.5 (b) has the structure of Xilinx Virtex-4 [Xilinx, 2007c] and Virtex-5 [Xilinx, 2008b] FPGAs. In these devices many features of their predecessors were retained, while their resource heterogeneity was increased. They incorporate the subsystem of Figure 2.5 (d). ICAP has a 32-bit width and it can run up to 100 MHz. Configuration memory is still organized in frames but run-time reconfiguration can be carried out in two dimensions. Virtex-4 and Virtex-5 FPGAs have a fixed frame length of 1312 bits [Xilinx, 2007b, Xilinx, 2008a]. This differs from the Virtex-II family in which configuration frames span the entire height of the device. Thus the fixed frame length in Virtex-4 and Virtex-5 corresponds to rows with a height of 16 CLBs. Rows can be configured without disturbing any logic above or below, which accounts for more flexible floorplanning. Multiple reconfigurable modules and static logic can coexist in the same column as opposed to the Virtex-II FPGAs.

2.5.4 Other Architectures

The ECA-64 developed by ElementCXI supports concurrent dynamic reconfiguration of its processing elements [ElementCXI, 2008, Master, 2006]. Its structure depicted in Figure **??** (a) consists of adaptable functional elements hierarchically communicating via a network on chip. The RPU consists of a homogeneous 2-D array of clusters each of which contains

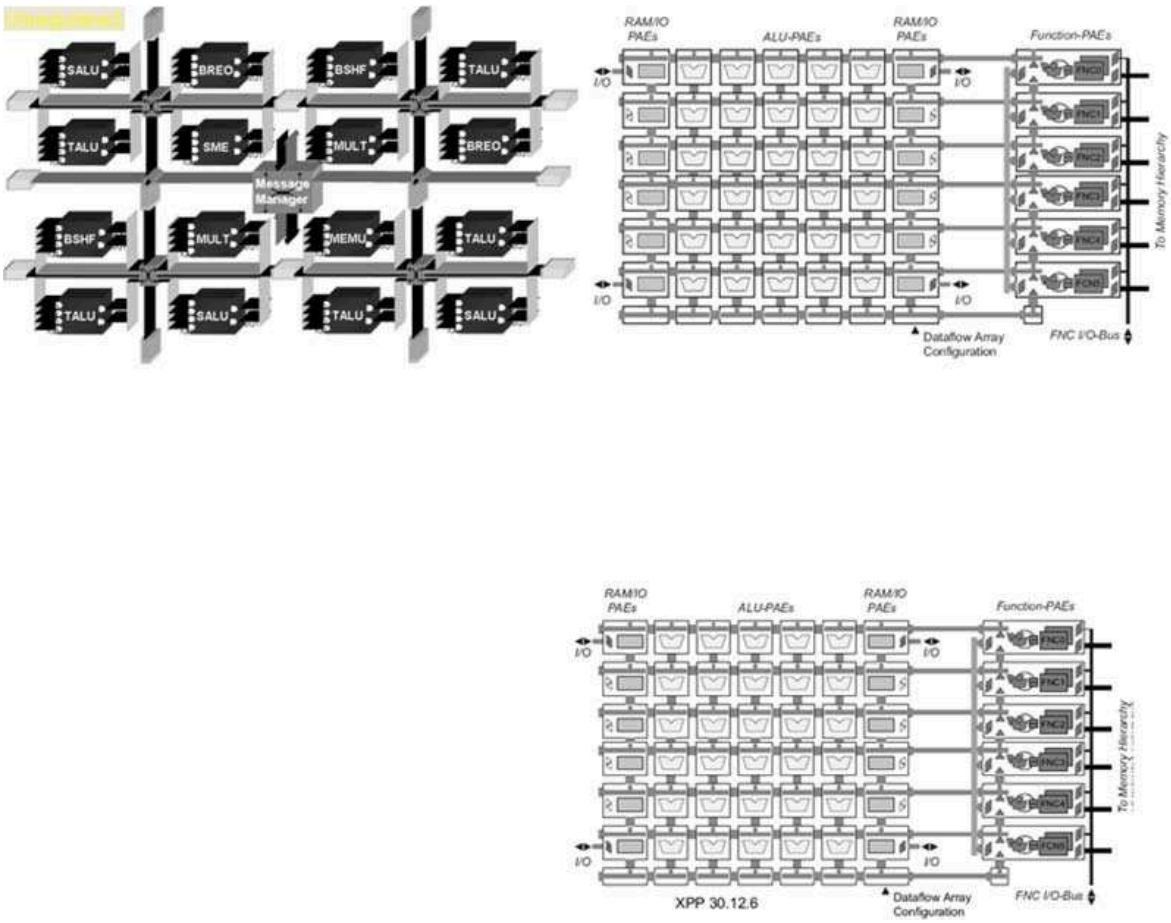


Figure 2.6: Block diagram of innovative architectures.

a heterogeneous, coarse-grain array of different PEs such as ALUs, memory elements, state machine elements, multipliers and shifters. A hardware kernel is responsible for controlling operations of the RPU such as run-time binding of tasks to resources, relocating stalled tasks, and flagging failed hardware resources against future use. The PEs can dynamically form specific computing structures on demand, simultaneously and in parallel in each clock cycle. Each PE has multiple contexts, which offers an additional level of flexibility. One of the technology's strong points is its resiliency which is provided with the use of dynamic reconfiguration, i.e. code is dynamically placed and routed at run-time to work around defects. This allows for graceful system degradation instead of catastrophic failure. At the same time redundant mechanisms, excessive software complexity, and application performance penalties are avoided. One of the keys to resiliency and high-speed execution is the fact that data are kept close to the execution units. This allows for simultaneous fetching

of multiple data, instead of doing this sequentially from a central memory. Distributed memories have the added implication of being fault tolerant. ECA-64 was released to the market in 2007.

The XPP-III developed by PACT XPP technologies [PACT XPP, 2008, PACT XPP, b, PACT XPP, a], represents innovative IP core families supporting concurrent dynamic reconfiguration of the processing elements. An overview of the architecture is depicted in Figure 2.6(b). It consists of a hierarchical array of adaptive processing elements and a packet oriented communication network. The RPU contains a heterogeneous coarse-grain 2-D array of two basic classes of PEs: one for control flow type and sequential programs, and the other for high bandwidth data streams. The two resources can execute concurrently. A special PE, called system controller (or configuration manager) containing a complete VLIW-like sequential processor kernel, is responsible for reconfiguring the RPU through a DMA channel. For history sake, in the first generation of XPP [Baumgarte et al., 2001, Baumgarte et al., 2003] the reconfiguration was controlled by distributed managers embedded in the array. Special event signals originating within the RPU can also trigger a reconfiguration which is also driven through the system controller. The RPU is reconfigured in a pipelined fashion, i.e. the pipeline distributes the configuration data to all PEs. Several completely independent partial configurations can be loaded to the RPU and execute concurrently. These configurations represent independent tasks operating on their own memories or data streams. Intermediate results of computations are stored in distributed memories or FIFOs for use by subsequent configurations. This programming style is called configuration flow, as opposed to the instruction flow of the classical Von-Neumann architecture. Furthermore, the RPU is equipped with configuration cache means for fast reloading of recently used configurations, and configuration prefetching to reduce reconfiguration overhead. The PEs can be configured rapidly in parallel while neighboring elements are processing data. This way entire applications can be configured and run independently on different parts of the array, which allows to support different types of parallelism like pipelining, instruction level, data flow and task level parallelism.

The first work on self-reconfiguration was published by York University where a small amount of static logic was added in a reconfigurable device to control the reconfiguration

process [French and Taylor, 1993]. The Colt/Stallion by Virginia Tech is a homogeneous coarse grain 2-D array which relies highly on runtime reconfiguration using wormhole routing [Bittner et al., 1996]. Distributed control configuration scheme brought by wormhole RTR allows multiple data ports to configure the device simultaneously. The RAW developed at the Massachusetts Institute of Technology is a heterogeneous coarse-grain 2-D array allowed for the communication pattern of the network to be reconfigured on a cycle-by-cycle basis [Waingold et al., 1997]. The SRGA by the University of Southern California is a heterogeneous fine-grain 2-D array that provides self-reconfiguration capability [Sidhu et al., 2000]. Its strong point is that it allows single context switching and single cycle random access to the on-chip configuration/data memory. The ACM by Quicksilver consists of a homogeneous 2-D array of clusters, each of which contains a heterogeneous, coarse-grain array with dynamic partial reconfiguration capability supported with a DMA engine [Quicksilver, 2007]. A system controller is responsible for reconfiguring the PEs in a cycle-per-cycle manner. The DRP by NEC Electronics contains a homogeneous coarse-grain RPU [Motomura, 2002]. An internal controller can be programmed as a finite state machine to control the reconfiguration process.

An interesting model is the stream-based compute model that has been realized with two different concepts. The first concept comes from the PipeRench developed at the Carnegie Mellon University [Schmit, 1997,Cadambi et al., 1998]. It supports pipeline reconfiguration and simultaneous configuration and execution to realize hardware virtualization [Plessl and Platzner, 2004]. It consists of stripes each of which corresponds to a stage of a pipeline. By dynamically reconfiguring the stripes, a pipeline of arbitrary number of stages can be virtually implemented on the physical stripes. A small configuration controller is responsible for the reconfiguration process. PipeRench has been commercialized with the KiloCore KC256 by Rapport Inc [Rapport Inc., 2008]. The second concept was the SCORE developed by the University of California at Berkeley and the California Institute of Technology [Caspi et al., 2000]. This model virtualize reconfigurable computing resources by dividing a computation into fixed-size pages and time-multiplexing the virtual pages on available physical hardware. This concept was realized on a system combining a microprocessor and an RPU in which a scheduler on the microprocessor is responsible for the reconfiguration process.

2.5.5 Discussion Summary

Present Section has made clear that extensive work has been done on architectures coupling a microprocessor with partially reconfigurable or multi-context hardware on a single die. Table 2.3 classifies such devices and includes some of their characteristics. These characteristics were retrieved from more than one information sources per device such as published papers, surveys, data sheets and web links. In many cases the different information sources do not match each other, thus depending on the information source a device can be categorized as reconfigurable coprocessor or RFU, the RPU of a device can be identified as fine-grained or coarse-grained and so on. All architectures of Table 2.3 support partial reconfiguration, either static or dynamic. The main advantage of dynamic reconfiguration is that hardware execution does not have to be halted in order for the logic to be reconfigured in part. Multi-context devices in which a context is being fully reconfigured while another context is active fall into the category of dynamically reconfigurable devices. Also, there exist multi-context devices allowing only a part of an inactive context to be modified while another context executes. All multi-context architectures of Table 2.3 fall into the category of dynamically reconfigurable devices. On the contrary, there exist partially reconfigurable devices in which part of the logic is altered only statically, e.g. Garp and Spartan-3.

Several issues should by considered when it comes to design an architecture, or, select a device for developing an application. Besides the target application domain, parameters such as the degree of flexibility, cost, complexity, area, speed and energy should be taken into account. Trade-offs should be balanced accordingly, e.g. a rule of thumb says that the higher the flexibility of an architecture is, the higher the complexity and cost. The question is which architecture suits better the needs for each case. For example, there are applications that require devices supporting concurrent operation of FPU and RPU. Below, specific parameters are suggested to be considered prior to designing an architecture, or selecting a device for an application.

Coupling level. Multiple-chip approaches suffer from high communication delay which necessitates the transfer of large unit of code to the reconfigurable hardware to maintain computational efficiency. Such systems suit highly complicated operations, while

Table 2.3: Classification of reconfigurable processors.

Device	Coupling	FPU	Granularity	Structure	Reconf.	Status
Garp	coproc.	MIPS-II	fine	PR	static	S
CS2112	coproc.	ARC	coarse	MC+PR	dynamic	F/W
DAPDNA-2	coproc.	RISC	coarse	MC+PR	dynamic	F
PSoC	coproc.	8-bit M8C	coarse	PR	dynamic	F
uP+DPGA	coproc.	RISC	fine	MC+PR	dynamic	F/R
NAPA1000	coproc.	RISC	fine	PR	dynamic	F/R
SIDSA FIPSOC	coproc.	8051	mixed	MC	dynamic	F/W
MorphoSys	coproc.	TinyRISC	coarse	MC	dynamic	F/R
REMARC	coproc.	MIPS-II	coarse	PR	static	S
ZIPPY	coproc.	MIPS	coarse	MC+PR	dynamic	S
KiloCore KC1025	coproc.	PPC	coarse	PR	dynamic	F
Chimaera	RFU	MIPS	fine	PR	static	F/R
OneChip95	RFU	DLX	fine	MC	dynamic	E
OneChip98	RFU	S-DLX	fine	MC	dynamic	E
OneChip00	RFU	MIPS	fine	MC	dynamic	E
XiRisc	RFU	VLIW	medium	MC+PR	dynamic	F/R
Molen	RFU	superscalar	fine	PR	static	E
PRISC	RFU	R2000	fine	PR	dynamic	S
DISC	RFU	Custom	fine	PR	static	E
Pozzi	RFU	VLIW	fine	MC+PR	dynamic	S
MeP+D-Fabrix	RFU	Toshiba MeP	medium	PR	static	F/W
CoMPARE	RFU	RISC	fine	PR	static	E
Proteus	RFU	ARM	fine	PR	dynamic	S
ADRES	RFU	VLIW	coarse	MC	dynamic	F
Virtex-II	embed.	PPC/MB	fine	PR	dynamic	F
Spartan-3	embed.	MB	fine	PR	static	F
Virtex-4	embed.	PPC/MB	fine	PR	dynamic	F
Virtex-5	embed.	PPC/MB	fine	PR	dynamic	F
ORCA	embed.	LatticeMico	fine	PR	dynamic	F
FPSLIC	embed.	AVR	fine	PR	dynamic	F
ECA	-	-	coarse	MC+PR	dynamic	F
XPP	-	-	coarse	PR	dynamic	F
XPP-III	-	-	coarse	PR	dynamic	F
SRP	-	-	fine	n/a	n/a	S
Colt/Stallion	-	-	coarse	PR	dynamic	S
RAW	-	-	coarse	PR	dynamic	F/R
SRGA	-	-	fine	MC	dynamic	S
ACM	-	-	coarse	PR	dynamic	F/W
NEC DRP	-	-	coarse	MC	n/a	F/W
PipeRench	-	-	coarse	PR	dynamic	F
SCORE	n/a	n/a	n/a	PR	dynamic	S

[a] "MC+PR" refers to multi context devices in which the inactive context can be partially reconfigured; "MC" means that an inactive context can be reconfigured only in its entirety. "n/a" refers to information that was not found. Mark "-" declares that the corresponding column does not apply to the specific device, e.g. if the cell under column "Coupling" is empty a microprocessor is not included; consequently column "FPU" will be empty. Under column "Status", "F" stands for fabrication and commercialization, "F/R" denotes that a fabricated device stopped at the research stage, "W" stands for withdrawn, "S" for simulation, and "E" for emulation.

dynamic reconfiguration might be useless. On the contrary, devices coupling both resources on the same die are characterized by small communication overhead. In general, the loose coupling approach of the coprocessor type requires bigger amount of computations assigned to the RPU than the RFU approach to justify the communication overhead. On the other hand, the RFU approach suits better applications in which communication between the two resources occurs frequently. Placing the RPU on the FPU's datapath simplifies greatly the interface between the two resources. This architecture reduces also the issue latency, but at the cost of a less flexible interface and lower bandwidth. In general in RFU architectures the array is smaller and more frequently reconfigured than in the coprocessor approach. Finally, the tight coupling to FPU's pipeline might not allow the latter to reconfigure the RFU during the time an instruction is in the pipeline.

FPU type. The type of FPU that is coupled with the RPU can affect the performance. Parallelism allows for the FPU to operate concurrently with the RPU or even reconfigure it while the FPU executes. The uniprocessor model does not allow for concurrent operation as opposed to the multiple-issue processors, e.g. superscalar or VLIW.

Granularity. Fine-grained devices like FPGAs have higher utilization rate as compared to coarse-grained devices due to their smaller granularity. Any algorithm can be mapped onto fine-grained logic devices. Especially, they serve well bit-level operation-intensive applications. On the other hand, a fine-grained array uses many configuration points to perform even small computations, thus many elements should be addressed and programmed. This results to large configuration bitstreams and consequently to large configuration times. On the other hand, coarse-grained devices require fewer bits to be configured. Commonly, they consist of tens to hundreds of processing elements, which are capable to execute word- or subword-level operations instead of bit-level operations. This reduces the configuration overhead in terms of memory cost and configuration time. However, this limits the flexibility of the architecture as it cannot be adjusted effectively to applications requiring bit-level operations. Between the two extremes stand the medium-grained architectures. It should be mentioned

that modern FPGAs tend to become hybrid devices in the sense that along with the vast amount of fine-grained resources they immerse coarse-grained elements such as multipliers and digital clock managers (DCM).

Structure. Multi context devices can switch execution from one context to another in one clock cycle. For this to happen an inactive context should be configured prior to when it is needed. The same concept applies to partially reconfigurable devices, although they are not structured in contexts. In multi context devices assuming that each context stores the configuration of one task, different tasks can share the RPU in time rather than in space. However, this technology suffers from several drawbacks. First, MC devices have limited computational resources due to the higher memory cost as compared to the PR technology, thus the tasks that can be accommodated into their RPU are smaller than in PR devices. Tighter area constraints might be needed during synthesis, while large tasks should be partitioned properly so as to be distributed in more than one contexts. Second, tasks lying in different contexts cannot execute concurrently; they can only execute in succession. Finally, inter-task communication might be performed through a special memory device, since communicating tasks cannot be active at the same time slot [Wu, 2007]. The above justify that coarse granularity is preferred in MC structures; more functioning is squeezed into a context with coarse-grained resources.

Static or dynamic reconfiguration. Due to the high flexibility of dynamically reconfigurable devices, they have more complex architecture compared to the static ones. Phenomena such as glitch effects are less likely to disrupt static reconfiguration. Further, task scheduling is more easily handled in statically reconfigurable devices. However, this comes at the expense of performance as in static reconfigurable devices the execution freezes during reconfiguration. Table 2.3 illustrates that most of the partially reconfigurable devices belonging to the RFU category are static. The tight coupling of RPU onto the FPU pipeline might imply that an ongoing reconfiguration controlled by the FPU should complete before the FPU can proceed with another instruction in the pipeline. This depends also on whether the FPU incorporates advanced features

such as a lockup-free cache.

Reconfiguration controller. It exists in two main forms: as software in a μP or as on-chip dedicated hardware. In the latter case it can also be implemented with FPGA logic which allows for greater flexibility. In general, software allows to implement complex control algorithms found in operating systems [Brebner, 1996], but usually it cannot reach the performance of a dedicated hardware controller [Robertson and Irvine, 2004]. Selection of the controller depends on the requirements of the application at hand. Factors affecting this decision can be the frequency of reconfiguration, the size of configuration data and the complexity of scheduling operation [McGregor and Lysaght, 1999, Barat et al., 2002]. For example, if a μP is used to execute part of the application and reconfiguration is occasionally needed, the same μP can undertake the reconfiguration process. Its use as reconfiguration controller can also be justified if the execution time amongst subsequent reconfigurations is considerably longer than the reconfiguration time. On the other hand, a dedicated hardware controller with simple scheduling would fit better in applications demanding frequent and fast reconfiguration. Moreover, there exist systems in which the μP triggers the reconfiguration, but it is not involved in the rest reconfiguration process as another hardware module fetches the configuration data [Blodget et al., 2003].

FPU blocking operation. It is necessary to consider this characteristic in case the FPU will control reconfiguration. Microprocessors that are dynamically scheduled like superscalar, or statically scheduled like VLIW, have more advanced features as compared to the simple scalar. If any of the two former types is employed, its execution does not need to stop during reconfiguration. On the other hand, a simple scalar processor should stop its execution in order to reconfigure the FPU.

Configuration bandwidth. It denotes the maximum configuration rate. Fine-grained devices have longer configuration times than coarse-grained devices due to that more elements need to be addressed and programmed. One of the coarse-grained devices offering high configuration bandwidth is the Colt/Stallion by Virginia Tech [Bittner et al., 1996]. Today, the technology incorporated into modern fine-grained devices

such as Xilinx Virtex-4 and Virtex-5 FPGAs is moving toward increasing configuration bandwidth.

Safety multiprocess execution. The incorporation of reconfigurable logic introduces safety challenges in multiprocess execution. In [Chien and Byun, 1999] a protection architecture is proposed which enables the power of reconfigurability in the processor core while requiring only a small amount of fixed logic to ensure safe and protected multiprocess execution. OneChip by the University of Toronto ensured concurrent execution of RPU and FPU [Jacob, 1998], while its successor ensured concurrent execution of two or more RPUs [Esparza, 2000].

Combinatorial and sequential logic. The RPU might consist of combinatorial logic, sequential logic, or a combination thereof. The inclusion of sequential logic increases complexity but it results in more powerful RPUs which can serve a broader range of applications. Example devices with RPU consisting of combinatorial logic only are PRISC [Razdan and Smith, 1994] and WARP [Stitt et al., 2003].

There exist works that propose to avoid incorporation of partial reconfiguration technology. Researches developed the ConCISe device by Philips Research Laboratories [Kastrup et al., 1999] came up with this conclusion. Targeting the embedded systems domain, the ConCISe suggests a compiler-driven approach to accelerate applications in which multiple custom instructions are encoded into a single RPU configuration. This work claims that PR incurs high overhead, cost and complexity which make it inappropriate for embedded systems. Certainly, triggering a reconfiguration for serving a single custom instruction only might be costly. On the other hand, there are PR architectures suggesting to load multiple instructions in the RPU that are packed in one configuration file only in order to avoid frequent reconfigurations [Wirthlin and Hutchings, 1995a, Ye et al., 2000].

This Section overviewed device architectures that can be reconfigured in part, commencing from the early presence of reconfigurable computing up to date. Many companies try to exhibit the capability of dynamic reconfiguration in their devices as a key solution to improve the performance of specific applications. Several coarse-grained devices have been released but they haven't taken-off yet, thus the companies have shut down their produc-

tion. The area is still dominated by the fine-grained FPGAs, which also tend to incorporate coarse-grained elements in order to serve the changing needs of the market. It seems that a vast amount of work has been done in developing novel structures capable to be reconfigured in part. Research should be strove to other fields as well, such as the efficient support of RTR and effective methods to fit an application into the PR hardware. At the same time, certain issues related with the complexity and cost of PR have to be considered.

Chapter 3

Task Scheduling and Allocation

Partial reconfiguration allows for the reuse of the same physical resources by different tasks of an application at different stages of its execution. However, reconfiguring the hardware during application execution, i.e. run-time or dynamically, incurs an overhead that can cause performance degradation to the total execution time. Present Chapter proposes a method using task graphs to reduce reconfiguration overhead by taking into account the constraints imposed by the physical resources. The method integrates a mechanism to hide the reconfiguration time by fetching tasks ahead of their start time of execution.

3.1 Problem Description

The academic community and industry alike show increasing interest in combining fixed and reconfigurable resources. Towards this direction, reconfigurable processors (RP) couple a fixed processing unit (FPU) with a reconfigurable processing unit (RPU) into a single chip. Commonly, the FPU loads computationally-intensive tasks to the RPU by configuring its memory. Computationally-intensive tasks can be entirely executed in hardware, which due to its parallel nature can potentially exploit fully the application's inherent parallelism and perform well when configured according to the needs of the task at hand. Furthermore, reconfigurable hardware can adapt to the task at hand at run-time based on parameters such as speed and area. This could boost the necessity of integrating reconfigurable processing units into contemporary computing systems whether they target the area of embedded

systems or general purpose computing.

The leading companies in the area of reconfigurable computing have recently released dynamic reconfiguration as a feasible technology to design applications, while the academia works intensively on various research aspects of this technology. However, its advantages do not come without cost yet as in general reconfiguration is performed on demand, and thus its inherent high latency and low throughput compared to other processes can incur degradation to overall system performance. This problem has been addressed within the academic community with techniques such as configuration caching, configuration compression and configuration prefetching [Li, 2002].

3.1.1 Related Work

The work presented in [Brebner, 1996] was among the first ones investigating the implications of incorporating partial reconfiguration capability in an operating system for FPGAs. The idea was to design applications into relocatable cores. Since then, several applications have been designed in partially reconfigurable hardware. Cryptographic applications were the early vehicle in researching the performance of partial reconfiguration. In [**?**] even though the complexity of the subkey generation and the reconfiguration is low, some overhead incurred to the execution time due to the reconfiguration process. A variety of techniques have been proposed attempting to reduce reconfiguration overhead. Configuration prefetching [Li, 2002] allows for overlapping execution with reconfiguration. The idea is based on constructing the control flow graph (CFG) of the application, which is then used to prefetch the configuration of the next operation to be executed on the reconfigurable hardware. In [Jeong et al., 1999] incorporation of a technique for early partial reconfiguration of an FPGA, into hardware-software partitioning stage is proposed. In [Iliopoulos and Antonakopoulos, 2001] instruction forecasting prefetches instructions that are most likely to be used shortly in a cache. The above works exploit flow graphs extracted from code analysis and they use different methods to perform prefetching. However, they do not take into account resource constraints.

Another problem arising that is related with partial reconfiguration is the scheduling of the modules into the available hardware. A work targeting real-time tasks [Steiger et al.,

2003] addresses online task and resource management for partially reconfigurable devices. It suggests integration of heuristics into an operating system to handle real-time tasks, rather than using traditional scheduling. This work does not integrate a prefetching technique. The work in [Shirazi et al., 1998] discusses the trade offs between reconfiguration time and circuit quality depending on the reconfiguration method used and information about the configuration sequence that is available at compile time.

The authors in [Banerjee et al., 2005a] consider reconfiguration overhead and configuration prefetching, while selecting a suitable task granularity. The initial task graph is statically transformed according to the trade-offs between data parallelization exhibited by each task and reconfiguration overhead. Then, simultaneous scheduling and columnar placement are performed, where the scheduling integrates prefetch to reduce reconfiguration overhead.

3.2 Contribution

The present work, which was initially published in [Papademetriou and Dollas, 2006a, Papademetriou and Dollas, 2006b], selects the tasks to be split according to the available hardware, so as to insert prefetch instructions into the processor code. This is done by performing an analysis to the initial task graph of the application. Based on the static prefetching algorithm described in [Li, 2002], an augmented prefething mechanism is proposed. It is also related to [Banerjee et al., 2005a] that schedules tasks according to the physical resource constraints. The difference from the latter is that present work examines specific places in the code, i.e. branches, to select the task to be transformed according to the resource constraints.

A model has been proposed which by fetching configurations in advance hides the configuration latency. But the main problem is whether it loads the correct tasks that will be required for execution in the near future (especially if this depends on input values that cannot be predicted). Present work investigates the advantages as well as the drawbacks of incorporating a prefetching model into a reconfigurable processor.

This Chapter contributes with an experimental framework that models a reconfigurable

processor. It is reusable and the structure of the architecture can be easily tuned to the details of the FPGA. The prefetching mechanism that takes into account the constraints imposed by the physical resources is incorporated within the framework to examine its impact to the overall execution length of an application described by its task graph. A set of experiments along with the results are presented. The problems raised with the proposed model are discussed. Finally, a discussion is made on the benefits gained with the novel experimental framework and the prefetching model, as well as on the way it can be used for different application domains.

3.3 Modeling with Task Graphs

The problem is modeled using task graphs, which constitutes an intuitive and common way for studying problems. A task represents a part of the computation that a system is required to carry out. Although past work in the literature assumes that tasks are mainly coarse-grained, a task can be of any size in terms of the number of instructions that constitute it. Pattern recognition, convolution, encoding, optimization algorithms and even a simple addition are examples of distinct tasks. A task of the same type can be used more than once during the execution of an application.

A task graph is a directed acyclic graph (DAG) consisting of a collection of task nodes, each of which is associated with a task type, and a collection of directed edges, each of which is associated with a scalar denoting the amount of data that must be transferred between the tasks that it connects. Edges represent communication events. In the example of Figure 3.1, each node denoting a task is labeled with its task type. Directed edges can be also labeled with the amount of data flowing along the edges. However, data flow is not examined separately in the present study and thus, the edge labels are omitted in all examples. Each edge points away from its parent task and toward its child task. A task's parents are the tasks to which it is connected by incoming edges, while a task's children are the tasks to which it is connected by outgoing edges. A directed edge may begin executing only after its parent task has completed executing. A task may begin executing only after all its incoming edges have completed executing. Consequently, a task may begin executing

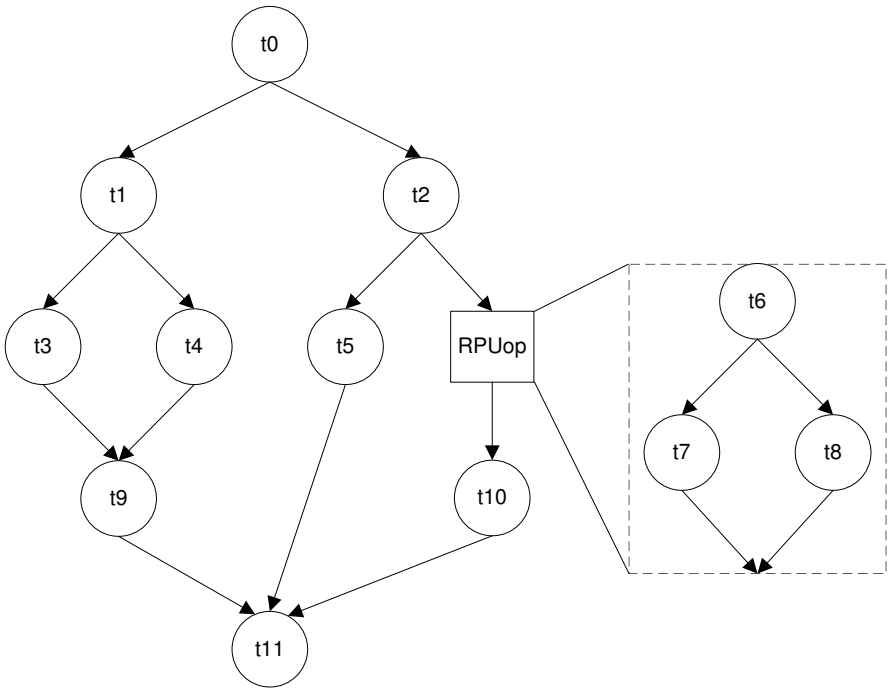


Figure 3.1: Task graph of an example application. The nodes outside the box are tasks executed by the FPU. The square task corresponds to an RPUop call comprising of a set of tasks.

only after the execution of the previous task has completed.

An example is given in the following section in order to better illustrate the problem. Also, the software that was used as part of the system modeling is briefly described.

3.3.1 An Illustrative Example

Figure 3.1 corresponds to an example application represented with a task graph. Nodes (also called vertices) correspond to tasks executed in the FPU and squares correspond to tasks executed in the RPU. Once the RPUop is called, the tasks in the RPU start executing. In task $t6$, along with other instructions, a conditional branch decides which task amongst $t7$ and $t8$ will be executed. Given that the available RPU hardware allows for all three tasks to be simultaneously seated onto the RPU, no time delay is incurred for the transition of execution from $t6$ to the next task, except for the time delay of the decision it self. On the other hand, if a resource-constrained RPU is employed which can hold $t6$ and only one of the $t7$ and $t8$ at most, a delay might be incurred. Assuming that tasks $t6$ and $t7$ have been fetched in advance onto the RPU according to a prefetching algorithm [Li, 2002], in

case the outcome of the decision will eventually need $t8$ the execution is stalled. Then, the FPU reconfigures partially the RPU with the configuration data needed to execute $t8$. This incurs a time overhead which cannot be avoided when a resource-constrained hardware is employed.

3.3.2 Task Graphs For Free

Task Graphs For Free (TGFF) [Dick et al., 1998] is a flexible and standard way for generating pseudo-random task graphs for use in research areas such as scheduling and allocation. It is a free-source software targeting the domain of embedded systems, hardware/software co-design, operating systems, parallel or distributed hardware, and in general any area which requires problem instances consisting of partially-ordered or directed acyclic graphs (DAG) of tasks. Users have parametric control over an arbitrary number of attributes for tasks, processors, and communication resources. Moreover, correlations between attributes can be defined and controlled parametrically. TGFF is capable to generate problem instances tuned to particular domain in scheduling and allocation research that can be used to evaluate and compare different algorithms.

3.4 Enhancing a Prefetching Algorithm

The idea of fetching configuration data in advance is not new. Present work enhances a previous prefetching algorithm [Li, 2002] to suit the problem better. According to the original algorithm (also called original model) the factors that determine prefetches are distance, probability, and configuration latency. Distance (d) is the number of instruction cycles between two task nodes, probability (p) is the branch probability to execute a path, and configuration latency (r) is the time needed to configure a task. The priority of prefetch instructions such as which task should be prefetched first prior to the branch is dictated by a function of the three factors. In this way a selection criterion is formed according to which each path is scored by the function.

Consider the example graph of Figure 3.2(a), which corresponds to an RPUop similar to that of Figure 3.1. Given the values of the three factors for each path that determine the

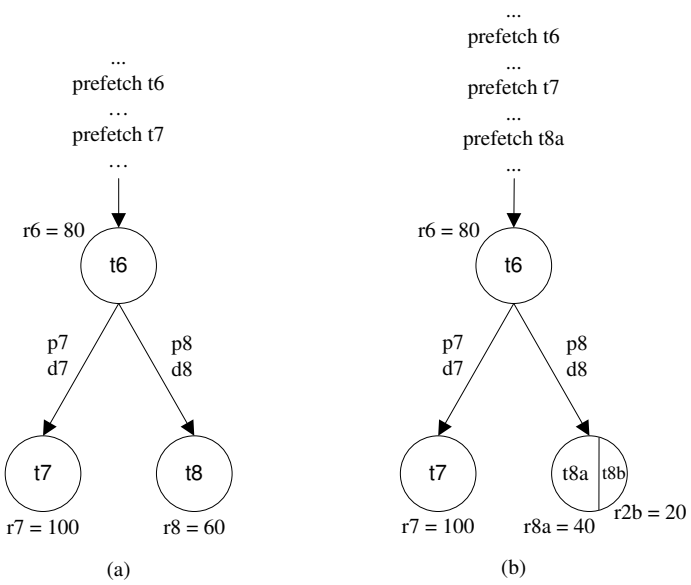


Figure 3.2: Original static prefetching model and augmented model that proposes transformation of the graph and insertion of additional prefetch instructions.

prefetch instructions, reconfiguration overhead is examined when using prefetching. In the specific example, distance and probability are shown as parameters; in a real application these are scalars provided through code analysis. The length of the configuration latency for each task is proportional to its hardware requirements. A resource-constrained RPU is assumed that can hold more than the aggregate size of the two larger tasks, but not all three tasks at a time. This is expressed as $t_{size}(t6, t7) < rpu_{size} < t_{size}(t6, t7, t8)$, where t_{size} is the size of the task(s) in terms of hardware requirements and rpu_{size} is the size of the RPU.

The original static prefetching algorithm [Li, 2002] considers that since the aggregate size of the reachable tasks for a certain node could exceed the capacity of the chip, only prefetches under the size restriction of the chip are generated. The rest of the reachable tasks are ignored. According to this, the prefetches that are inserted into the processor code correspond to the two tasks with the highest selection score, which depends on the distance, probability, and configuration latency. In the example of Figure 3.2(a), it is assumed that $t6$ and $t7$ are fetched due to their higher selection score over $t7$.

The problem objective of the present work is to statically determine the best prefetching that utilizes all the physical resources. The concept behind the proposed approach is to

transform the task graph by breaking down selected RPU tasks in smaller pieces, so as to fully exploit the physical resources by configuring them ahead of their start time of execution. Even tasks with low score that are less likely to be selected are loaded to the configuration memory of the RPU, given the availability of physical resources. To accomplish this the original prefetching algorithm is augmented which is illustrated in Figure 3.2(b). Assume that $rpu_{size} = t_{size}(t6, t7, \frac{2}{3}t8)$. By transforming the task graph, t8 is split into two subtasks. The size of $t8_a$ subtask is equal to $\frac{2}{3}t8$, while the remaining $\frac{1}{3}$ is the size of $t8_b$. Subtasks $t8_a$ and $t8_b$ can now be processed as two separate subtasks. The only factor that is changed for each subtask is configuration latency which is directly proportional to its size in terms of hardware area; distance and probability are retained. Task $t8_a$ is selected for prefething in order to fill up the empty space of RPU, while $t8_b$ will be fetched on demand once the branch outcome needs $t8$.

To obtain the performance gain of the augmented over the original model in the above example, the scalars in Figures 3.2(a) and 3.2(b) are used. Configuration latency is expressed in quantities of time, rather than in real time units. If the outcome of the branch is $t8$, due to that the original algorithm wouldn't have prefetched it a configuration delay equal to 60 is incurred. On the other hand, the augmented algorithm prefetches the part of $t8$ that fits in the empty space, i.e. $t8_a$. Hence, once t8 is needed only $t8_b$ is loaded which incurs a latency of 20.

The factors that contribute to the criterion for selecting the tasks to be prefetched are the probability, the distance and the configuration latency. This criterion can be formed as a mathematical function in order to evaluate each possible path with a selection score, which will be used for inserting the proper prefetch instructions. The terms *most-likely-to-be-selected (mlbs)* and *less-likely-to-be-selected (llbs)* tasks are coined here to characterize the tasks on the paths of a branch. Whether a task is characterized as *mlbs* or *llbs* depends on the outcome of the cost function. The latter can be formed by the designer depending on the needs of the application at hand by taking into account the physical resource constraints. For its development, techniques used in the field of branch prediction can be considered [Hennessy and Patterson, 2003].

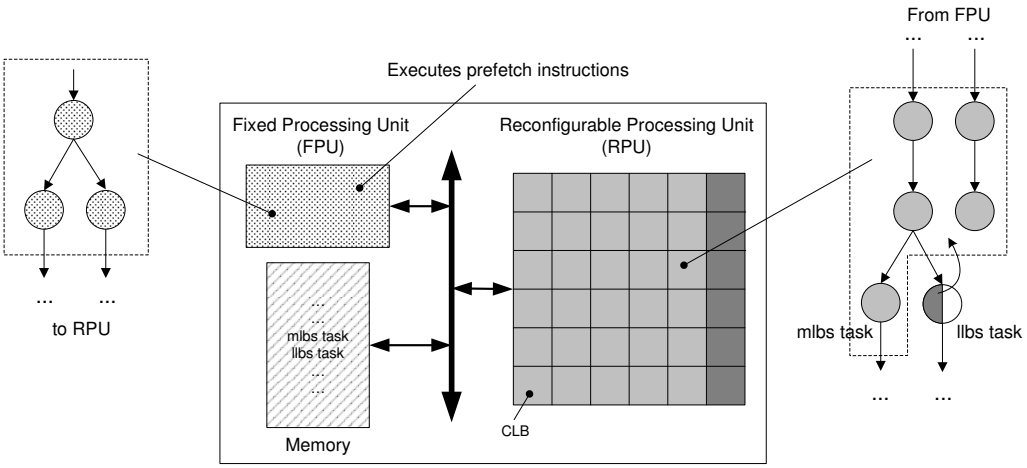


Figure 3.3: Some tasks are executed by the FPU and other tasks are loaded to the RPU for acceleration. FPU triggers reconfiguration of the RPU without entering a stall phase. Prefetch instructions are initiated by the FPU. The partial bitstreams that implement the RPU tasks are stored in a memory.

3.5 System Model

Typically, reconfigurable processors contain an FPU, an RPU and an interface between them. The system model that is used for the simulation framework is shown in Figure 3.3. The FPU can be a hardcore processor, e.g. a PowerPC, or a softcore processor implemented with reconfigurable resources, e.g. a Microblaze. Today's typical FPGAs can incorporate hardcore processors within the same die, or instantiate softcore processors due to their high capacity. Communication can be either bus-based such as OPB and PLB, or point-to-point such as FSL. A memory that stores the full and the partial bitstreams lies outside the chip. A partial bitstream implements a circuit that corresponds either to an RPU task, or an RPU subtask, or even to a set of RPU tasks.

3.5.1 Configuration in Modern FPGAs

In a realistic scenario FPU initiates RPU reconfiguration and continues undisturbed its execution, i.e. FPU execution is not stalled waiting for the configuration data to be loaded. The instruction inserted into FPU's code resembles any other instruction, consuming a single slot in the pipeline.

The configuration memory of Xilinx Virtex-II FPGAs is arranged in one-bit wide vertical

frames spanning the entire height of the device. These frames are the smallest addressable segments of the Virtex-II configuration memory space. Therefore, all operations must act on whole configuration frames. Configuration memory frames do not directly map to any single piece of hardware; rather, they configure a narrow vertical slice of many physical resources. In addition, a pad frame is required to be added at the end of the configuration data which flushes out the reconfiguration pipeline [**?**]. Therefore, to write even one frame to the device it is necessary to clock in two frames, the data frame plus a pad frame.

Virtex-II FPGAs have heterogeneous physical resources. Configuration frames are grouped into six column types that correspond roughly to physical device resources. The number of frames per column type remains constant for all devices whereas the distribution of frames between Configurable Logic Blocks (CLB) and other resource column might be different, e.g. 22 frames are needed to configure an entire CLB column and 64 frames for a BlockRAM column. Moreover, although the memory elements that define the content of a LUT are all located in one frame, this doesn't hold true for all resources. In fact, some resources have their configuration bits scattered across multiple frames [**?**].

3.5.2 Simulation Framework

For the deployment of the framework the task graph representing the application along with attributes related to the execution time and the area requirements of the tasks running on the two resources are needed. These are used for programming the TGFF software. The graph can be extracted from a functional specification in a high-level language like Verilog, VHDL, SystemC etc. There are well known methods and tools for parsing a high-level design language and representing it with a graph [Y. Lin, 1997]. In addition, data gathered from information found in the data sheets of the chosen reconfigurable processor (RP) are used as input to the simulation framework. The chosen attributes for the RP correspond to a Xilinx Virtex-II FPGA and they are very realistic. The FPGAs are mainly divided into CLBs, each of which is capable of being reconfigured to compute a number of logic functions. The FPGA is described by the number of CLBs it contains and the reconfiguration speed; this information is independent of the application. For each pair of tasks and FPGA, there is an execution time, a CLB requirement and a correlation between them. The execution

time refers to the time needed to carry out a task and the CLB requirement is the number of CLBs configured for this task.

Present work assumes a homogeneous device model wherein application tasks are placed in CLB columns only. In particular, it is considered that the circuit of any task is placed in multiples of one CLB column and not in multiples of one frame. As a consequence, reconfiguration is performed in CLB column level only. This was chosen for sake of simplicity, but even in relatively new devices such as Spartan-3 the minimal reconfiguration unit is one CLB column.

The TGFF was programmed using the above data, and a software application was developed that reads the TGFF output data and transforms them in a format capable to be imported in spreadsheets in order to draw graph charts. The above procedure is semi-automated in that the data in the spreadsheets are filtered manually using functions so as to exclude the cases in which mlbs and llbs tasks can be simultaneously seated in the RPU.

The proposed setup lends itself well to the present study. Although the framework is demonstrated using a Virtex-II FPGA, the newer Virtex FPGAs can also be employed by altering the attributes. One basic difference of the new FPGA technology over the previous one is that the unit of reconfiguration granularity is a smaller, bit-wide column corresponding to a standard amount of CLBs (or integer multiples thereof) and is independent of the device size or family. Thus, in Virtex-4 and Virtex-5 the frame is fixed stretching the height of 16 CLBs and 20 CLBs respectively, in contrast to the frame in Virtex-II that stretches a column from the top to the bottom of the device [Lysaght et al., 2006, Xilinx Inc., 2010a, **?**]. Therefore, in the new FPGAs it is feasible to reconfigure separately regions arranged vertically one below the other without disrupting the operation of neighboring frames located either above or beneath the region being reconfigured (2-D reconfiguration), as opposed to the Virtex-II column wise reconfiguration (1-D reconfiguration).

3.6 A Resource-driven Approach

The aim of this research work is to execute an application by fully exploiting the limited resources of a reconfigurable processor. In order to do this, the application task graph is

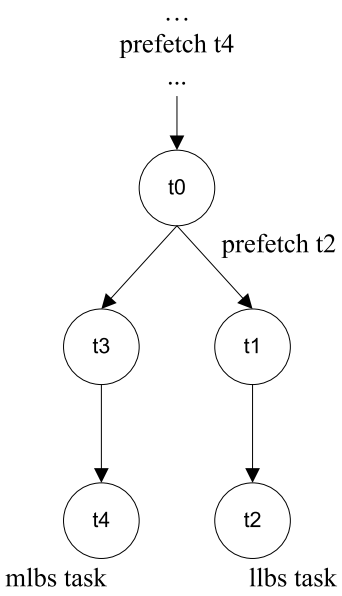


Figure 3.4: Insertion of prefetch instruction of mlbs and llbs tasks according to the original model.

transformed by breaking down selected RPU tasks into smaller pieces, so as to place the split subtasks in the RPU ahead of their start time of execution. In particular, part of the llbs task of a branch is loaded to the leftover physical resources after the placement of the mlbs task. This concept is illustrated in Figure 3.3. Toward this direction, the concept described in Section 3.4 is elaborated in the present Section in terms of the way the tasks are split and fetched to allocate properly RPU resources. The goal is to improve the performance of the total execution of the application, by overlapping RPU reconfiguration with the FPU execution.

The example in Figure 3.4 shows a part of an application comprising of five tasks running on a reconfigurable processor, along with the prefetch instructions inserted by the original algorithm. Figure 3.5 shows the reconfigurable processor with the FPU (implemented as a softcore processor using reconfigurable resources) and the RPU, along with the partitioning of tasks. Tasks $t0$, $t1$ and $t3$ run on the FPU and $t2$, $t4$ run on the RPU. In task $t0$, beside other instructions, a decision is made regarding which task among $t1$ and $t3$ should be followed. Then, the corresponding RPU task is called. Due to resource constraints, the two RPU tasks cannot be simultaneously seated onto the RPU. Thus, after the branch in $t0$ is resolved, reconfiguration of the RPU might be needed.

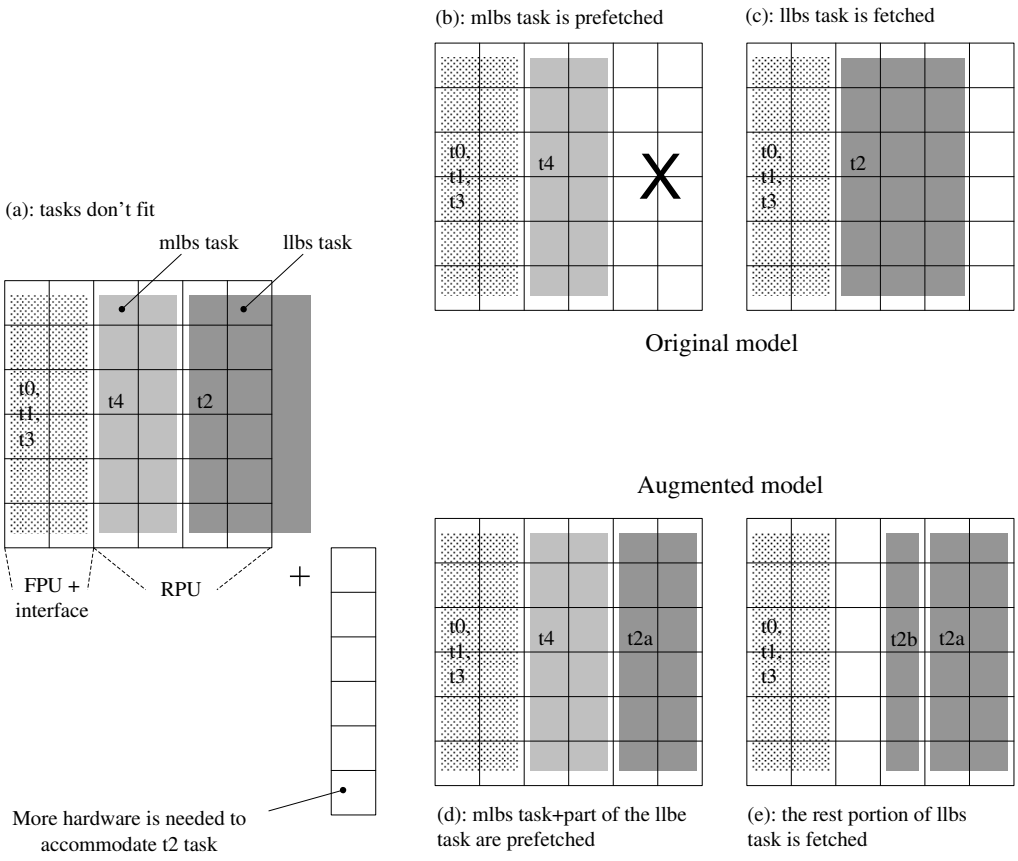


Figure 3.5: Task placement in original vs. augmented model. In (a) tasks do not fit into the RPU. (b) and (c) correspond to the original model. (d) and (e) correspond to the augmented model.

In Figure 3.5(a) it is assumed that $t4$ corresponds to the most likely to be selected (mlbs) RPU task and $t2$ corresponds to the least likely to be selected (llbs) RPU task. One more CLB column would be needed to accommodate both RPU tasks. In Figure 3.5(b), as $t4$ has been prefetched onto the RPU according to the original prefetching algorithm [Li, 2002], in case the outcome of branch in $t0$ requires $t2$ after the intermediate task $t1$, execution might be stalled. The second prefetch instruction of Figure 3.4 reconfigures the RPU with $t2$, which is illustrated in Figure 3.5(c). Given that the system supports concurrent FPU execution and RPU reconfiguration, $t1$ execution will hide part or potentially the entire reconfiguration time. The amount of time that can be hidden depends on the execution length of $t1$ and the configuration latency of $t2$.

The augmented model suggests a more aggressive prefetching that utilizes all the physical resources. This is illustrated in Figures 3.5(d) and 3.5(e). As in the original model the mlbs

RPUop, i.e. $t4$, is selected for prefetching first. RPUOP $t2$ is then broken down into two subtasks in that $t2_a$ fits on the remaining portion of the hardware. Task $t2_a$ is prefetched before $t0$. Therefore, in Figure 3.4 if the outcome of the branch requires $t2$, only subtask $t2_b$ will be loaded after $t0$.

In this work it is assumed that the RPU tasks selected for split are divisible and recombineable. The idea is along with the placement of the mlbs RPU task to automatically break down the llbs RPU task into non-functional subtasks according to the physical constraints. Then one part of the llbs task is placed on the RPU and in case the llbs RPU task is called, the rest part is loaded on demand by displacing the mlbs RPU task that was not finally executed. The cost of disconnecting the displaced RPU task when loading the remaining part of the split RPU task is not examined. In addition, as illustrated in Figure 3.5(d) the proposed model fully utilizes the available area. An issue arisen at this point is the limitation to the placement options of the RPU tasks compared to the original model. To effectively exploit the augmented model, the first subtask should be placed on an appropriate location where the second subtask would be adjacently placed by replacing the mlbs RPU tasks as shown in Figure 3.5(e). This is due to the effort of the augmented model to fit as much as many tasks in the leftover area. The original model doesn't deal with such restrictions as the llbs RPU task is loaded on demand only when the mlbs RPU task is not executed. Thus, despite its greater configuration latency, if offers more options for placing the RPU tasks. The trade-offs between the two models regarding this issue is an interesting study but it isn't studied within the context of the present dissertation.

3.7 Using the Simulation Framework

Experiments are carried out using the framework to evaluate the aforementioned algorithms in a dynamically reconfigurable processor. The setup consists of an application scenario and a set of attributes describing the physical resources of the chosen reconfigurable processor hardware as well as the area and time required to carry out the tasks of the application.

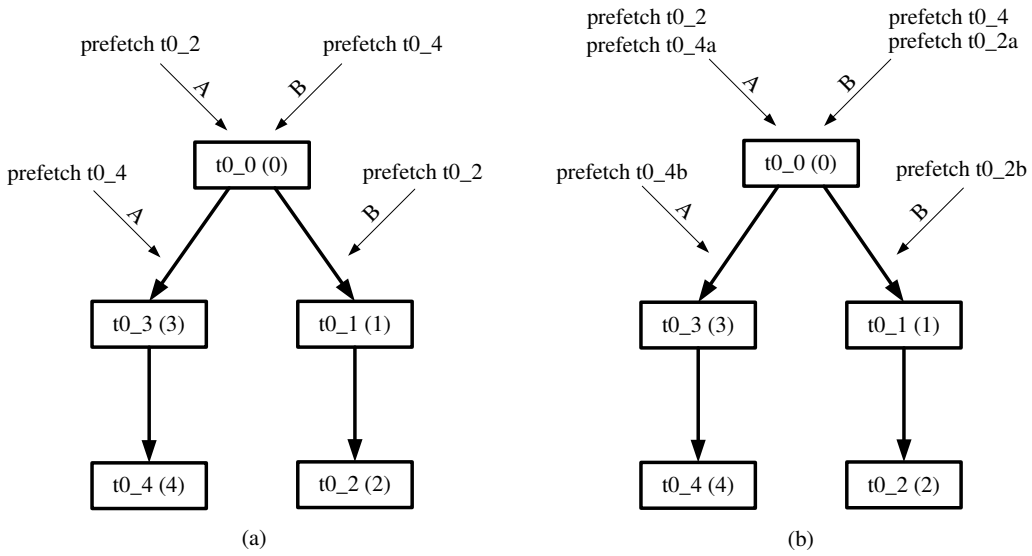


Figure 3.6: TGFF output and insertion of prefetch instructions. (a) and (b) have the prefetches according to the original and the augmented algorithm respectively. Two different scenarios regarding the insertion of prefetch instructions are indicated with the labels A and B at the thin arrows, that are dictated by the mlbs/llbs assignment.

3.7.1 An Experimental Setup

A setup with the same tasks and resource allocation shown in Figure 3.5 is considered. A softcore processor such as the Microblaze is assumed which communicates with the RPU through the OPB/PLB interface. An internal port similar to the ICAP port is used to partially reconfigure the configuration memory of the RPU. Figure 3.6 has the examined task graph as generated by the TGFF. Regarding the task names the number on the left side of the low dash indicates the graph's ID. It is used for identification if more that one graphs are generated. In present case one graph is examined only thus the ID is eliminated, e.g. $t0\_0$ will be referred to as $t0$, $t0\_1$ as $t1$ and so on.

In Figure 3.6(a) the graph is carried out with the original model and in Figure 3.6(b) the same graph is carried out with the augmented model. Two scenarios are possible, A and B, according to which the insertion and the sequence of the prefetch instructions is different. Scenario A is applied when $t1$ is the mlbs task, while scenario B is applied when $t3$ is the mlbs task. Depending on the branch outcome the appropriate path is selected. A description of the operation of the two models follows, assuming that scenario B is applied. In Figure 3.6(a), if the branch outcome is $t3$ no new prefetch is executed (configuration data for task

Table 3.1: Characteristics of the XC2V500 FPGA.

Data from Xilinx's data-sheet:	
chip	XC2V500
# CLBs	768 (24×32)
# CLB columns	24
# frames	928
# frames/CLB column	22
# frames/all CLB columns	528
# bytes/frame	344 (device dependent)
# config. port	ICAP, 8-bit@66MHz
Simple computations give:	
config. time/byte	$15.15ns$
config. time/frame	$15.15ns \times 344 = 5.21\mu s$
config. time/CLB column	$5.21\mu s \times 22 = 114.62\mu s$
config. time/CLB column including pad	$114.62\mu s + 5.2\mu s = 119.83\mu s$
config. time/chip	$4.83ms$

$t4$ have already been loaded or are being loaded to the RPU due to the prefetch instruction ahead of $t0$). On the other hand, if the branch outcome is $t1$ a new prefetch is executed incurring a higher reconfiguration delay (configuration data for task $t2$ aren't contained onto the RPU). In Figure 3.6(b) in scenario B, if the branch outcome is $t3$ no new prefetch is executed (same as in the original model). However, if the branch outcome is $t1$ a new prefetch should be executed in order to load subtask $t2_b$. Thus the incurred reconfiguration delay is smaller than in the original model. A set of experiments was conducted to compare the two models when the llbs task needs to be executed (reconfiguration is performed).

The testbench consists of 500 systems executing the same task graph which was randomly generated by TGFF. Each task node is unique. This is denoted by the values in the parentheses of Figure 3.6. The device contains 24 columns of 32 CLBs each, which resembles the Xilinx Virtex-II XC2V500 FPGA. The FPU with the interface occupies 6 CLB columns which is roughly realistic compared to the amount of area required for the Microblaze softcore processor (800-900 slices out of the 3072 slices are needed which corresponds to the 26%-30% of the device resources) and a bus-based interface with the RPU such as the OPB and PLB. It is assumed that each task carried out by the FPU has an execution time of $300 \pm 250\mu s$ (thus ranging from $50\mu s$ to $550\mu s$). The RPU is implemented with the

Table 3.2: Input to the TGFF.

# task types	5
# CLB columns for FPU+interface	6
# CLB columns/RPU task	10 ± 8
execution time/FPU task	$300 \pm 250\mu s$
execution time/RPU task	$200 \pm 180\mu s$
config. time/CLB col. including pad	$119.83\mu s$

remaining 18 CLB columns. The number of CLB columns required by a task is chosen to be 10 ± 8. The tasks carried out by the RPU are assumed to be executed for $200 \pm 180\mu s$. The only correlated attributes used in the experiments are the CLBs needed for each RPU task and their execution time. Configuration time is also needed for the experiments. A similar to ICAP configuration port is considered with an 8-bit interface running at 66 MHz, capable to sustain its full bandwidth during the transfer of the partial bitstream. This is used for the computations of Table 3.1. Configuration time is proportional to the number of frames to be loaded; this is used to compute the configuration time of the CLB columns to be programmed. The setup of the system is consolidated in Table 3.1, while Table 3.2 has the input attributes for the TGFF.

The input data that supplied the framework were chosen with the consideration to produce systems requiring reconfiguration and systems that do not require reconfigurations. In the specific setup, if the aggregate size of the RPU tasks is bigger than 18 CLB columns, the tasks cannot be simultaneously placed onto the RPU and reconfiguration is needed. Moreover, the aim was to explore the impact of reconfiguration delay and therefore the amount of CLB columns per task was varied substantially, i.e from 2 to 18 CLB columns per task. Aside from the reconfiguration overhead, the scope was to study the impact of the utilization of leftover CLB columns, after prefetching the mlbs task, to the execution length of the application. The above data can be reused to reproduce the same experimental setup and to gather more results.

3.7.2 Results and Evaluation

Figure 3.7 compares the two models with respect to the total execution length of the application over different number of leftover CLB columns after prefetching the mlbs RPU

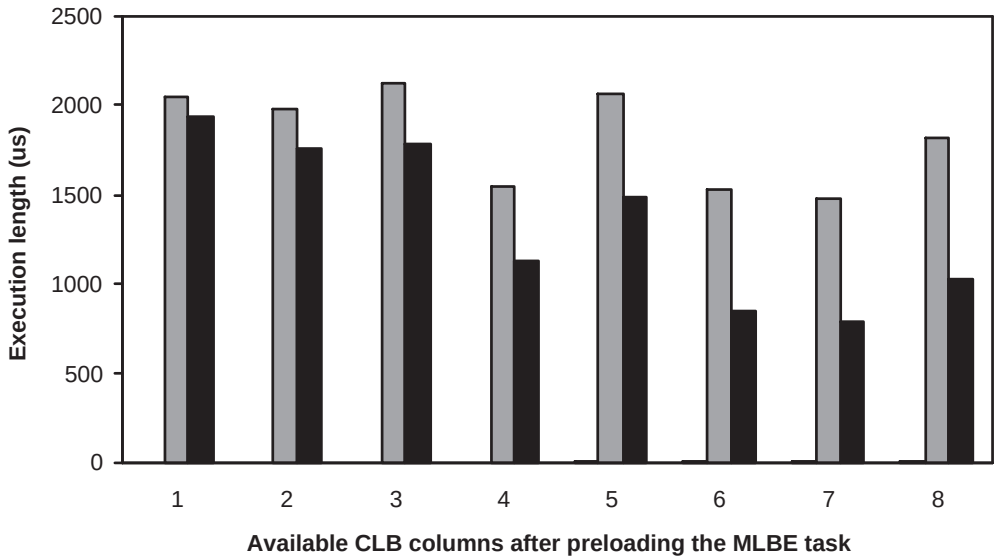


Figure 3.7: Execution lengths for the original and the augmented model for different values of leftover CLB columns after prefetching the mlbs RPU task. The llbs RPU task is chosen for execution. Average values are shown that were obtained from 500 experiments with the simulation framework.

task. All experiments concern the same task graph and the average values were used to construct the chart graph (for each value of leftover CLB columns the average execution length is shown). In some cases the RPU tasks were completely prefetched, i.e. their total size was smaller than the available hardware; these cases are not included. The results concern execution of the llbs task only. It is observed that the more the CLB columns that are utilized for prefetching the llbs RPU task, the greater the benefit of the augmented model over the original model. The improvement in the execution length is given by the following equation:

$$100 \times \frac{(EL_{orig} - EL_{augm})}{EL_{orig}}$$

where, EL_{orig} and EL_{augm} are the execution lengths for the original and the augmented model respectively. For one leftover CLB column the improvement was 5.2%, whereas for seven leftover CLB columns the biggest improvement was achieved, equal to 46.5%.

The preceding results were obtained assuming that the execution speed of the RPU tasks are equal in both models, i.e. in the augmented model the operational frequency

Table 3.3: Performance in original vs. augmented model. Worst cases regarding CLAM time are shown. Times are measured in μs.

FPU task	CLOM	CLAM	# free CLB columns	ROOM	ROAM
167	1890	1775	1	-1722	-1607
335	1752	1522	2	-1417	-1187
335	1616	1271	3	-1281	-936
125	1195	735	4	-1070	-610
525	1285	710	5	-759	-184
227	815	125	6	-698	-8.4
236	923	118	7	-686	118
333	1116	196	8	-783	136

of the llbs task once it is assembled with the placement of the second part, would be the same to that of the original model. However, this may not be realistic. As reported in Section 3.6 limitations of the task placement imposed by the augmented model can affect the performance improvement.

Table 3.3 has the reconfiguration overhead of the augmented model in contrast to the original model. It consolidates the worst cases of the experiments with regard to the length of configuration latency of the augmented model, i.e. the cases in which for a specific number of leftover CLB columns after prefetching the mlbs task, the second part of the llbs task to be loaded is the largest. Hence, the comparison is in favor of the original model. *FPU task* column has the execution time of the task prior to which the prefetched instruction is inserted, i.e $t1$ or $t3$ of Figure 3.6. *CLOM* (Configuration Latency of the Original Model) refers to the original model and is the time needed to load the entire llbs RPU task. *CLAM* (Configuration Latency of the Augmented Model) refers to the augmented model and is the time needed to load the second part of the llbs RPU task. The *# CLB columns* is the amount of leftover CLB columns after prefetching the mlbs RPU task. The meaning of Reconfiguration Overhead is introduced here as the amount of time that can/cannot be hidden by overlapping reconfiguration with processor execution. If its value is negative, it is not hidden by the processor execution and it is called reconfiguration penalty; if it is positive it is hidden by the processor execution and no reconfiguration penalty occurs. *ROOM* (Reconfiguration Overhead of the Original Model) is the overhead caused by loading the

llbs RPU task before the FPU task (after the branch). *ROAM* (Reconfiguration Overhead of the Augmented Model) is the overhead caused by loading the second part of llbs RPU task before the FPU task (after the branch). All times are measured in μs.

An interesting remark is obtained when scrutinizing the 7th and 8th rows of the table. As CLOM increases (from 923 to 1116 μs), ROOM decreases (from -686 to -783 μs). On the contrary, as CLAM increases (from 118 to 196 μs), ROAM continues to increase (from 118 to 136 μs). This is due to the increase of the available CLB columns that the augmented model can utilize (from 7 to 8), and as the FPU task time increases (from 236 to 333 μs) it hides more overhead (even if it is not necessary as any positive number indicates completely hidden overhead).

The above results illustrate the relation between configuration latency and reconfiguration overhead and whether reconfiguration can be hidden by the processor's execution. In a system where the FPU execution can overlap with RPU reconfiguration, depending on the RPU task execution time and the amount of leftover CLB columns after prefetching the mlbs RPU task, the designer can decide whether it is worthwhile trying to hide the llbs RPU task configuration latency by applying a proper split operation. The original model in all cases exhibited negative reconfiguration overhead which entails that the reconfiguration was not hidden. Conversely, the augmented model in some cases managed to completely hide reconfiguration and in all cases performed better than the original algorithm resulting in lower overhead.

The preceding results were obtained assuming that the execution speed of the RPU tasks are equal in both models, i.e. in the augmented model the operational frequency of the llbs task once it is assembled with the placement of the second part, would be the same to that of the original model. However, this may not be realistic. As reported in Section 3.6 limitations of the task placement imposed by the augmented model can affect the performance improvement.

3.8 Discussion Summary

The proposed model takes into account the area constraints and the reconfiguration overhead to place tasks onto a partially reconfigurable processor. Its main advantage is the increase in the utilization of the available hardware resources by splitting the least likely to be selected (llbs) task. When part of the task is prefetched, the augmented model reduces the total execution time of an application by overlapping reconfiguration with processor execution. A study is needed to assess the extent to which the processor involves in the reconfiguration process. In a system where the prefetch instruction blocks the processor execution until the RPU is reconfigured the performance might be degraded. This will affect both models. A solution to this problem can be given if the process of reconfiguration is thought of as accessing the processor memory system. In many processors, whenever a data cache read miss occurs, the processor stalls until the outstanding miss is serviced. To remedy this situation, non-blocking (lockup-free) cache is employed, which allows the processor to continue performing useful work even in the presence of cache misses. In particular, non-blocking loads reduce the time stalled due to cache misses by allowing the processor to overlap the servicing of a miss with the execution of other instructions. The amount of overlap depends on the number of instructions that are available for execution that do not use the register being targeted by the load instruction [Farkas et al., 1994, Belayneh and Kaeli, 1996]. Under the same concept, given that RPU reconfiguration has no dependency with the active FPU instructions, FPU execution overlaps completely with RPU reconfiguration. In order for this capability to add value, the FPU needs to allow instructions to execute out of order [Hennessy and Patterson, 2003]. Although the low-end Microblaze and PowerPC 405 that are available for the Virtex-II and Virtex-4 FPGA families do not support out-of-order execution (they are equipped only with some advanced features such as branch prediction and victim cache), the high-end PowerPC 440 and ARM Cortex A9 processors that are immersed into the newer Xilinx Virtex FPGAs allow out-of-order execution. Toward dealing with this problem but from a different aspect, Xilinx suggests developing systems able to fetch configuration data directly from external memory without requiring the processor to be involved during reconfiguration [Blodget et al., 2003]. Therefore, in the

proposed approach no extra time will be consumed in the FPU for configuring the RPU except for the reconfiguration triggering event through the prefetch instruction.

A problem arisen is the limitation of the placement options due to the restriction of the area where the task can be placed. This might cause degradation in task's execution speed. Moreover, unless the llbs task is called, an overhead is paid on the processor as more prefetch instructions are inserted into its code, and an overhead for transferring the configuration data of the first part of the llbs task. In case the latter will not be eventually executed the performance of the system could be degraded. The trade-offs between these limitations and keeping the system at an acceptable performance level need to be considered by the designer and depend on the application at hand.

The simulation framework devised here for dynamically reconfigurable processors can be tuned to the problem at hand for algorithm evaluation. In its present form it constitutes an integrated framework allowing to adjust different attributes for the FPU and the RPU such as size and reconfiguration time, and for the tasks carried out in both resources such as execution time and hardware requirements. Also, it can be extended by explicitly defining the time needed for data flowing among the tasks. This will allow to separate the time spent in data transferring from the time for task execution so as to quantify the time spent in the two processes. To do this, scalars need to be associated with the edges to denote the amount of data transferred between the connected tasks. Also, it is necessary to study the throughput of the connection means between the modules running the tasks, either if this is realized with a bus such as the OPB and PLB, or with a point to point link such as the FSL. Presently, data flowing is considered as part of the execution length of the tasks.

In the experiments of Section 3.7.2 the input values used to feed the framework weren't derived from a real case and for this reason their range was big enough. For real-world applications real data should be used. For example, consider that a genetic algorithm [Vavouras et al., 2009a, Effraimidis et al., 2009] is part of an RPUop that is swapped in and out of the RPU. The generation of one population consisting of 32 members takes 1780 clock cycles. When implemented in a Virtex-II Pro running at 127 MHz this translates to $14\mu s$. For the objective functions studied in [Effraimidis et al., 2009], 7 generations are needed to get an optimized result which translates to a total execution time of $98\mu s$. This

value evidences that the range for the execution time per RPU task in Table 3.2 that was selected to compare the two algorithms, i.e. $20\mu s$ to $380\mu s$, is quite realistic.

The framework can be used to evaluate research algorithms such as the two models presented in this Chapter on different application domains. For example, in a real-time system if the deadline of an llbs task needs to be met, by breaking it down into smaller subtasks and prefetching one of them into the free hardware, the time to load the rest part will be smaller as compared to the original model; thus the total time to complete the task - which is composed of reconfiguration and execution - is reduced; it is more likely that the deadline is met in this case. Indicatively, consider the genetic algorithm as the llbs task of a large application. The genetic algorithm comprises five main distinct modules, i.e. population sequencer, selection, random generator, crossover and mutation, and fitness evaluation [Vavouras et al., 2009a]. Given a resource-constraint hardware, some modules can be prefetched into the RPU, but they will be activated once they are assembled with the rest modules. The latter will be fetched on demand when the genetic algorithm needs to be executed.

The framework concerns systems utilizing the full bandwidth of the ICAP configuration port. However, this is not usually true and in a realistic FPGA-based system a more holistic approach should be considered in which data are fetched from external memories, then transferred through a bus or a point-to-point interface, and eventually loaded to the FPGA configuration memory. This will affect the computations for both the original and the augmented model, and thus their comparison might produce different results. Clearly, reconfiguration time affects the system performance as it takes a considerable amount of time (see Table 3.1), and the capability to feed the framework with a realistic value would increase its value. In fact, the capability to estimate the reconfiguration time for different system setups can assist the designer during experimentation in making proper decisions prior to entering the implementation phase.

Part of the work in this Chapter has been published in [Papademetriou and Dollas, 2006a] and [Papademetriou and Dollas, 2006b]. The setup of the latter work considers column-based reconfiguration and compared to the former which examines reconfiguration per CLB unit, it is more realistic due to its conformity with the way the configuration

memory of Virtex-II FPGA is addressed, i.e. frames are spanning the entire height of the FPGA thus affecting part of the entire CLB column. As mentioned above, the granularity of reconfiguration differs amongst the FPGA families, ranging from one frame stretching the height of 16 CLBs in Virtex-4 to an entire CLB column in Spartan-3. This issue needs to be taken into account with respect to the FPGA model used for the experiments as it affects the size of the smallest configurable element, in multiples of which a task can be arranged.

Chapter 4

Experimental Framework

Many efforts within the academia and a few among the industrial community exist, trying to establish dynamic reconfiguration as a feasible way to design commercial applications. However, it can degrade the execution time due to the time required to download the configuration data before the system is ready to execute. This is known as reconfiguration overhead, and quantitative analysis is needed to examine whether dynamic reconfiguration is justified for an application. In addition, this analysis can be used to evaluate mechanisms proposed to reduce reconfiguration overhead such as configuration caching, compression, and prefetching [Li, 2002, Papademetriou and Dollas, 2006b].

Performance evaluation constitutes an important procedure in a wide range of research domains as it allows to analyze thoroughly a system. It can take considerable amount of time to be completed, but usually it pays-off by revealing the details of system functionality. Its use can benefit contemporary FPGA-based platforms that have started to be used as end products for deploying industrial applications. Toward this direction evaluation of high-end dynamically reconfigurable systems presents interesting research challenges.

This Chapter introduces a methodology to evaluate dynamic reconfiguration from a system perspective and defines the time components that add up to the total reconfiguration overhead. Initial results using this methodology were published in [Papadimitriou et al., 2007]. The shortcomings of that work showed that the process was tedious due to the time needed to prepare each experiment and the amount of data that had to be sampled and processed. In particular the subsequent steps needed to be carried out by the user are

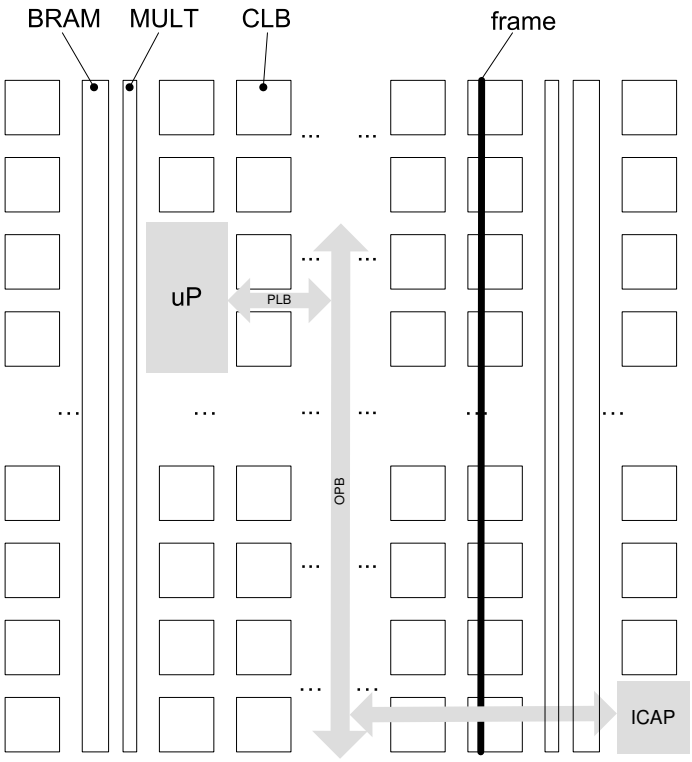


Figure 4.1: The Virtex II-Pro FPGA resources and configuration frames.

the setup of the different system parameters, sampling and processing of the experimental data. The different values of these parameters can affect the performance of applications running in a dynamically reconfigurable system, and therefore a new method was necessary to examine thoroughly the system. To this end, a semi-automated framework was developed within the context of a final-year project by a student of the MHL laboratory [Anyfantis, 2007]. That framework allowed to quickly set up the system parameters, sample and depict the experimental data in a user-friendly way. Present work elaborated further on that framework and concluded with results showing the variation of the reconfiguration time over different values of the system parameters. Also a study of the level of importance of the parameters with respect to reconfiguration performance was conducted which formed the basis of the next Chapter.

4.1 Background

This section discusses the Xilinx Virtex-II Pro FPGAs [Xilinx Inc., 2007b] and the reconfiguration mechanism. Although the experiments were performed on the specific device, the framework is rather general and can be used for other FPGA platforms.

4.1.1 The Virtex-II Pro FPGA and Reconfiguration

The generic structure of the Virtex-II Pro FPGA is shown in Figure 4.1. It consists of the hardcore processor PowerPC, and the high-bandwidth Processor Local Bus (PLB) and the slower On-Chip Peripheral Bus (OPB) [IBM, 2007] for communication with the array. The modules implemented in the array logic act as peripherals of the processor. The array is 2-D fine-grain heterogeneous, mainly consisting of configurable logic blocks (CLBs), hardcore memory blocks (BlockRAMs or BRAMs), and hardcore multipliers. Each CLB contains look up tables (LUTs), flip-flops, multiplexers and gates that are configured to implement the design logic. The array can be configured by the processor with dedicated instructions through the Internal Configuration Access Port (ICAP), an 8-bit built-in interface that configures the FPGA at a maximum rate of 100 MHz. A BRAM attached to the ICAP caches configuration bits prior loading to the FPGA configuration memory (CM).

The CM of the Virtex-II Pro is arranged in vertical frames that are 1 bit wide and stretch from the top edge to the bottom of the device. Frames are the smallest addressable segments of the device's CM space, so all operations must act on whole configuration frames. They do not directly map to any single piece of hardware; rather, they configure a narrow vertical slice of many physical resources [Xilinx Inc., 2007b]. A pad frame is required to be added at the end of the configuration data, which flushes out the reconfiguration pipeline for the last valid frame to be loaded. Therefore, to write even one frame to the device, it is necessary to clock in two frames, i.e. a data frame and a pad frame.

The configuration data produced for programming the FPGA is called bitstream. When only a portion of the FPGA is to be configured which corresponds to an amount of frames, a partial bitstream is produced. To create the partially reconfigurable modules, the difference-based flow [Xilinx Inc., 2004b, Xilinx Inc., 2007a] was followed by making small changes to

a design and then generating the bitstreams based on the differences between the designs. Transition to the module-based flow [Lysaght et al., 2006] does not require any modification in the proposed framework. The same design flow also applies to the latest Xilinx high-end FPGAs, i.e. Virtex-4 and Virtex-5, except that the configuration granularity is smaller.

4.1.2 Evaluation of Reconfiguration

McGregor and Lysaght evaluated a self-controlling dynamically reconfigurable system using a logic analyzer and reported that the reconfiguration process was significantly slower than the execution speed of the FPGA logic [McGregor and Lysaght, 1999]. In [McKay and Singh, 1999], the authors developed tools and techniques for debugging a dynamically reconfigurable system. A logic analyzer was used to evaluate the improvement of specialized circuits, such as constant coefficient multipliers over the corresponding general circuits. In [Tan et al., 2006] a performance comparison between two interfaces used for partial reconfiguration of FPGAs is made to evaluate the tradeoffs between design complexity, area overhead, reconfiguration flexibility and reconfiguration latency. For this purpose the Xilinx Chipscope Pro tool was used that operates as an internal logic analyzer [Xilinx Inc., 2009a]. In [Hymel et al., 2007], the authors studied the performance impact on timing and resource utilization of the Xilinx's new partial reconfiguration design flow when targeting Virtex-4 FPGAs through remote updating.

The foregoing works demonstrate that performance evaluation of dynamic reconfiguration is an interesting area. In addition, as the existing tools do not support simulation of dynamic reconfiguration due to the lack of behavioral and hardware models, the foregoing works employed a logic analyzer, either for measuring the reconfiguration time for a few bits, for evaluating specialized circuits over the general counterparts, or for evaluating partial reconfiguration interfaces. However, neither of them had reported a method for automatically gathering experimental results, nor examined the overhead incurred by the components that participate in the reconfiguration process. Thus, the development of a general infrastructure for elaborating dynamic reconfiguration seems promising.

The present framework evaluates the overhead added by the physical components that participate in the reconfiguration process. In [Papadimitriou et al., 2007], some piecewise

delays during reconfiguration were defined and measurements with a logic analyzer were introduced, but results taken with software methods only were included. The present Chapter discusses logic analyzer measurements along with the automatic method. According to a publication by Xilinx researchers, once data are available in the configuration cache, the time to reconfigure a single frame in a Virtex-II Pro FPGA through the ICAP is in the order of decades of microseconds [Blodget et al., 2003]. A more recent work reveals the reconfiguration time of many frames for Virtex-II and Virtex-4 FPGAs, again after the configuration data are available in the configuration cache [Lysaght et al., 2006]. This time was measured and verified with software methods in [Papadimitriou et al., 2007]. In a later Section this time is measured using a logic analyzer also, showing that it conforms with the Xilinx's published value and the software measurements. However, this time is not the only aspect in the reconfiguration process, and other physical components of the system add significant delays, causing the reconfiguration time to increase more than three orders of magnitude as compared with the foregoing time. Moreover, although the reconfiguration time over the number of frames is linear [Lysaght et al., 2006], it is proven that this does not hold always at platform-level.

4.2 Experimental Setup

The setup for the experiments shown in Figure 4.2 consists of a XUPV2P platform with a Virtex-II Pro FPGA [Digilent Inc., 2008], a board with LEDs, an Agilent 1680A logic analyzer, and a PC. The platform is connected through the serial port to the PC for evaluation and through the onboard expansion headers to the LED board and to the logic analyzer for monitoring internal FPGA signals. This setup allows for the measurement of the time components that add up to the total reconfiguration latency. Just as the total latency of a dynamic memory is substantially higher than its access time, the total reconfiguration latency is substantially higher than the execution time. Hence, proper definition and measurement of the constituent latencies will allow to evaluate reconfigurability in the development of an application.

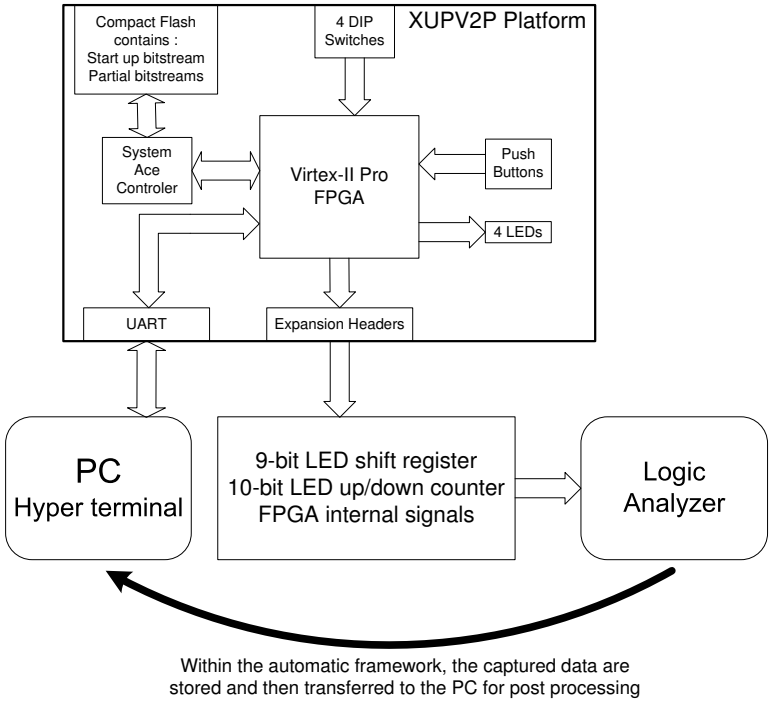


Figure 4.2: Block diagram of the experimental setup.

4.2.1 XUPV2P Platform

Figure 4.2 shows the platform parts of the system that was studied. A nonvolatile compact flash (CF) memory holds the configuration bitstreams, i.e. the initial and the partial bitstreams. The System ACE Controller supervises the transfer of data from the CF to the FPGA. In the FPGA, the PowerPC and several peripherals have been configured. A push button allows the user to trigger a reconfiguration at any time during operation. Four dual in-line package (DIP) switches control the functionality of an FPGA peripheral. The universal asynchronous receiver/transmitter (UART) sends status messages and debugging information to the PC. The LED board is connected to the expansion headers for displaying the peripherals' operation and monitoring FPGA signals with the logic analyzer.

4.2.2 FPGA System

The FPGA internal system is shown in Figure 4.3. The PLB and OPB buses communicate through a bridge. The PowerPC controls the reconfiguration process. A cyclic shift register peripheral has been implemented as static logic; a logic function controlling an up/down

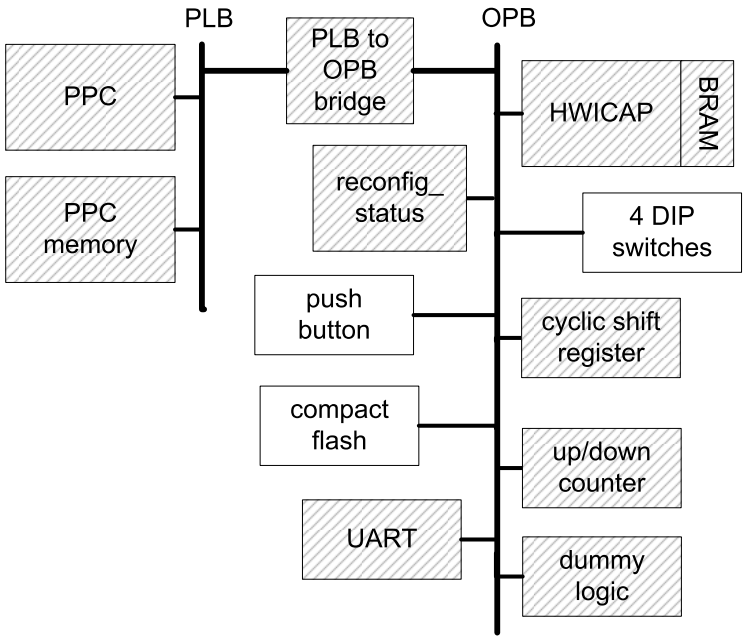


Figure 4.3: The shadowed boxes represent the internal components of the FPGA. The white boxes are parts of the platform connected externally with the FPGA.

counter and ten dummy logic peripherals have been designed as partially reconfigurable modules. This way, dynamic reconfiguration is demonstrated as the cyclic shift register continues to operate undisturbed while a part of the array is being reconfigured.

The push button forces the PowerPC to request a partial bitstream from the CF, then write it in the PowerPC memory, and subsequently transmit it to the HWICAP module[1]. The bitstream is not written to the PowerPC memory with one transaction only. A specific amount of bits, the so called data chunk, is first written from the CF to the PowerPC memory. The size of the data chunk depends on a system parameter called buffer cache (bc), which affects the amount of bits transferred with one transaction. More specifically, the bc size defines the amount of memory for buffering *read* and *write* calls to the System ACE Controller. The size of the processor memory allocated for the data chunks, which is called processor array (pa), affects the amount of configuration data stored before writing to the HWICAP takes place. Then, the configuration data are written to the BRAM of the HWICAP, which is called configuration cache, and when it gets full, the bits are written

[1]The HWICAP module is provided by the vendor as a ready-to-use IP core and allows for the embedded processor to control the ICAP for reading and writing the FPGA CM at run time.

into the FPGA CM via the ICAP; the writing through the ICAP is controlled with software instructions. The foregoing process is repeated until the entire bitstream is loaded. Hence, iterations of the following operations occur until the entire bitstream is written to the FPGA CM: (i) CF to PowerPC memory, (ii) PowerPC memory to HWICAP configuration cache, and (iii) HWICAP configuration cache to FPGA CM.

Two more peripherals have been implemented for evaluation purposes; the reconfig_status peripheral, which outputs a signal indicating the duration of reconfiguration process, and the UART peripheral, which transmits status messages to the PC.

4.2.3 Creating the Partial Bitstreams

In order to create the partial bitstreams the difference-based design flow [Xilinx Inc., 2004b, Xilinx Inc., 2007a] was followd. A logic function that controls a simple up/down counter was implemented as a partially reconfigurable module by affecting the LUTs on the same column. Also, 10 dummy peripherals of different sizes were implemented as partially reconfigurable modules by changing the values in the LUTs of contiguous columns, starting from the left and moving to the right side of the chip. Thus, 11 peripherals were initially created, with one having the logic function that controls the up/down counter only and the other ones having the logic function plus one dummy peripheral. This resulted in 11 modules each having a different size. Then, for each module, a second configuration with equivalent size but different functionality was created. This resulted in an overall of 22 partially reconfigurable modules that were stored as partial bitstreams in the CF. The two smallest partial bitstreams reconfigure the logic function only, the next two larger partial bitstreams reconfigure the logic function plus the smallest dummy peripheral, and so on. Table 4.3 has the sizes of the experimental partial bitstreams as produced with the Xilinx tools. The succeeding sections present the measurements of the time needed to reconfigure the FPGA with partial bitstreams ranging from 20,352 to 119,872 bits.

It is clear that the advantages of dynamic reconfiguration do not come without cost, as the bitstreams must be stored elsewhere in the system. The most obvious tradeoff is between external nonvolatile memory and FPGA size. In terms of silicon area and hence cost, it is preferred to store the inactive designs in cheaper non-volatile memory [MacBeth

and Lysaght, 2001]. Alternatively, internal BRAMs can be used to store partial bitstreams [Blodget et al., 2003], but this poses limitations to the size of the bitstreams that can be stored.

4.2.4 Demonstration of Dynamic Reconfiguration

The cyclic shift register is static, and its output was monitored with the LED board. The up/down counter counts upward or downward, and its output was monitored with the LED board as well. Its operation depends on the Boolean logic function. The latter's output was monitored with four small LEDs located on the XUPV2P platform. Two different Boolean logic functions were created such as to be partially reconfigured, the bitwise "OR" and "AND" between four 1-bit operands. The operands' values are directly given from the four DIP switches. The output of the logic function determines the counter's behavior.

During operation, when the reconfiguration button is pressed, a partial bitstream is loaded. Depending on the experiment, the logic function that is equal to 20,352 bits up to the logic function plus the largest dummy peripheral that are equal to 119,872 bits are configured. The control of the counter - the logic function - changes on the fly, whereas the shift register continues its operation. This scenario is simplistic, but it serves the scope of reconfiguring the chip with bitstreams of different sizes. Real experiments for the different bitstreams of Table 4.3 demonstrate the usefulness of the framework.

4.2.5 System Parameters

Some parameters were configured as fixed values, and others were varied during experimentation. The PowerPC main memory was 48 KBytes and the stack was 6,000 Bytes. The bc of the processor was varied between 512 and 4,096 Bytes. Its size defines the amount of memory for buffering read and write calls to the System ACE controller. In addition, the size of the pa was varied, i.e. the array allocated in the processor memory for storing the configuration data chunks[2] that were read from the CF. The HWICAP configuration cache

[2]Transactions were conducted in multiples of one sector per processor request using a software routine. A sector is the smallest unit the CF is organized in and is equal to 512 Bytes. Thus, the pa size was varied in multiples of a sector size.

is implemented with one BRAM and is equal to 2,048 Bytes. Interrupts of the processor were not enabled in order to configure a system with low resources.

4.2.6 System Operation Flow

Figure 4.4 has the operation flow of the system. In step 1, at power-up the FPGA is configured, execution of user application starts, and the PowerPC starts operation. In steps 2-3, the PowerPC does polling on the push buttons waiting for a reconfiguration to occur. In steps 4-5, the PowerPC writes configuration data from the CF to its memory. In step 6, the data are written word by word to the configuration cache of the HWICAP. In step 7, the HWICAP BRAM is checked to determine if it has fully been loaded. If the HWICAP BRAM is full, then reconfiguration is performed in step 10 and all data contained in the HWICAP BRAM are written to the FPGA. Step 11 checks if the reconfiguration of the FPGA has been completed. Execution of the new configuration starts in step 12, and the pipeline is flushed in step 13 if the reconfiguration was verified as complete in step 11; if it is not complete, then new configuration data will be loaded in step 9. In step 14, the PowerPC detects reconfiguration completion. Back to step 7, if the HWICAP BRAM is not full, then, in step 8, it is checked if all configuration data have been sent. If this is false, new configuration data are loaded from the CF or the PowerPc memory in step 9. If it is true, FPGA reconfiguration is performed in step 10.

4.3 Reconfiguration Time Breakdown and Measurement

To gain complete understanding of the reconfiguration time, a definition of the delays that add up to it is given:

1. CF-PPC is the time to copy configuration data from the CF to the processor memory with one transaction.

2. PPC-HWICAP is the time to write configuration data from the PPC memory to the HWICAP BRAM[3].

[3]Due to the HWICAP BRAM size, the maximum data size per transmission is equal to the size of one BRAM, i.e. 2,048 Bytes.

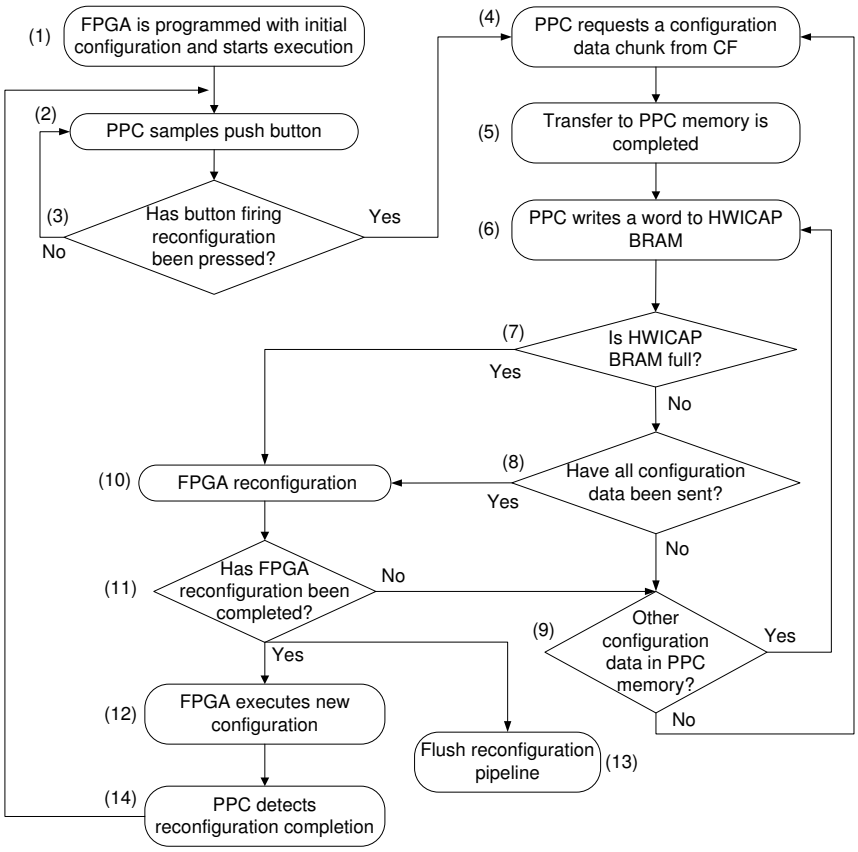


Figure 4.4: System operation flow.

3. HWICAP BRAM-CM is the time to load the configuration data from one HWICAP BRAM to the FPGA CM.

4. Rec-HWICAP is the time elapsed between the PPC detection that a reconfiguration has been fired and the first launch of the configuration data from the HWICAP BRAM to the FPGA CM.

5. HWICAP-CM is the time for loading all configuration data from the HWICAP BRAM to the FPGA CM, including the pad frame[4].

6. RT is the time elapsed between PPC detection that a reconfiguration has been fired and switching to the new execution; this is the total reconfiguration time.

[4]Note that this delay differs from the HWICAP BRAM-CM delay. The latter corresponds to the time needed to release the data that have filled the HWICAP BRAM only *once*. Contrarily, the HWICAP-CM is the time elapsed between the *first* configuration data start being written to the CM and the *last* configuration data that have been written to the CM. Figure 4.5 helps to clarify this.

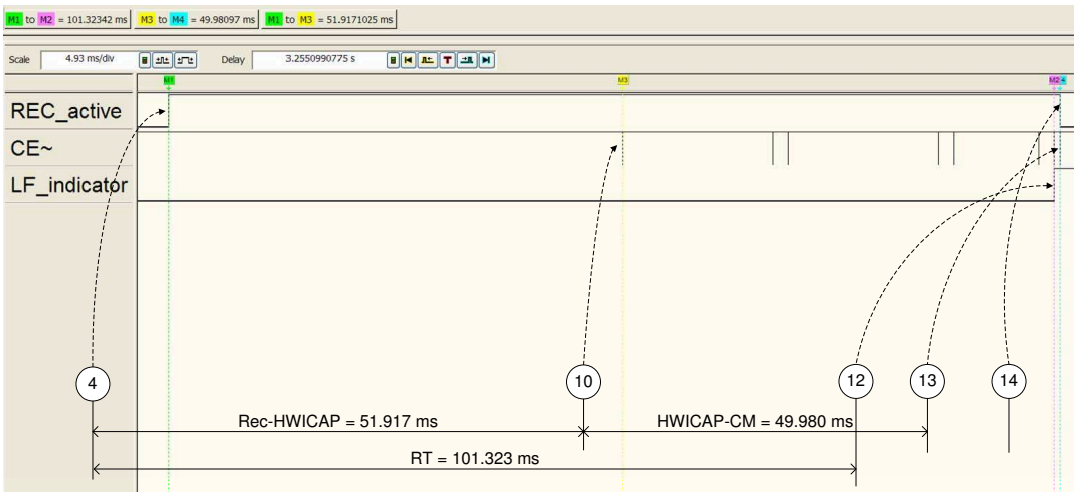


Figure 4.5: Timing mode trace of the logic analyzer for one reconfiguration of bitstream 11 of Table 4.3. The parameters used are bc=4,096 Bytes and pa=4,096 Bytes.

The first three delays were measured with software timers to evaluate the corresponding processor instructions [Papadimitriou et al., 2007]. It was shown that, i) when the partial bitstream is smaller than the pa, it is written with one processor request only and stored in the processor memory in its entirety prior to transmission to the HWICAP, ii) from the HWICAP side, configuration cache is 2,048 Bytes and can not be changed, thus inhibiting writing and accommodation of the entire bitstream at once (even bitstream 1 in Table 4.3 requires 20,352 bits = 2,544 Bytes to be configured, which cannot be accommodated at once by the HWICAP BRAM), and iii) HWICAP is not the bottleneck. Moreover, the time to write the data in the CM after they filled the configuration cache was measured. This was made by measuring the HWICAP BRAM-CM delay, which reflects the time to configure one frame, and it was found to be equal to 25.26μs, which matches the Xilinx's published values [Blodget et al., 2003, Lysaght et al., 2006].

The remaining three delays of the foregoing list were measured with the following signals using the logic analyzer:

- ICAP signals (the bar line denotes active-low signal).
  - $\overline{CE}$ (input): The ICAP chip is enabled.
  - $\overline{WRITE}$ (input): Indicates writing to the FPGA CM. It is deactivated during read.

– $\overline{BUSY}$ (output): Indicates that ICAP is busy, either during write or read.

- REC_active: It indicates that reconfiguration is in progress. It is set high when the PowerPC is notified that reconfiguration has been fired and low when it is notified that reconfiguration has finished. This signal is exported from the reconfig_status peripheral.

- LF_indicator: It marks the moment the FPGA starts execution of a new configuration. This signal is exported from the up/down counter peripheral to indicate that the logic function has been changed.

The delays were measured by the intervals between the edges of these signals, as shown in the example of Figure 4.5. This Figure shows a logic analyzer trace during the reconfiguration of the largest bitstream of Table 4.3. The numbers in the circles correspond to the numbers of steps of Figure 4.4. First, the REC_active signal is asserted, indicating that a reconfiguration has been requested. Activation of the $\overline{CE}$ signal indicates loading of configuration data from the HWICAP to the FPGA. Transition of the LF_indicator signal, either from "0" to "1" or from "1" to "0", marks the moment the FPGA switches to the new execution. Completion of the HWICAP's BRAM write to the CM is shown with the last rising edge of the $\overline{CE}$ signal. Finally, deactivation of the REC_active signal indicates that the PowerPC has detected reconfiguration completion.

In addition, the time the $\overline{CE}$ signal is active was measured, which is set low during one write transaction from the HWICAP configuration cache to the CM. It was found to be equal to $24.5\mu s$, which matches the time to write one frame as published by Xilinx [Lysaght et al., 2006, Blodget et al., 2003], and the software measurements [Papadimitriou et al., 2007].

4.4 Experimentation Phase

The experimentation phase consists of stages from the setting of parameters to the data plotting. In the first part the parameters that were changed and the stages of the experimentation are specified. Then the manual and the automatic experimentation method are

discussed.

4.4.1 Parameters and Stages

The effect of the parameters introduced in Section 4.2 is examined:

- Partial bitstream size. Experiments were conducted with 11 different sizes varying from 20,352 to 119,872 bits.
- bc size. It is varied from 512 to 4,096 Bytes with a step size of 512, resulting in eight different experiments.
- pa size. It is varied from 1 to 8 sectors with a step size of 1, i.e. 512-4,096 Bytes, resulting in eight different experiments.

The total number of experiments is given by all parameter combinations, which is found by multiplying the number (#) of different values of the parameters:

$$\begin{aligned} \#experiments &= (\#bitstreams) \times (\#pa\ sizes) \times (\#bc\ sizes) \\ &= 11 \times 8 \times 8 = 704 \end{aligned} \tag{4.1}$$

Initially, the values of the parameters are set, the code is compiled and downloaded, and a self-test routine runs. Next, the logic analyzer is prepared to be triggered, the user pushes the button to fire the reconfiguration, and the logic analyzer captures the data. Then, the delays are measured and sorted in a proper format in order to be plotted. The definition of these stages is rather general and the remaining section discusses the manual method and the transition to the automatic method.

4.4.2 The Manual Method

It consists of the following stages:

1. Change of parameters: The filename of the partial bitstream that is fetched from the CF, the pa size, and the bc size are separately changed for each experiment.
2. Compilation and downloading: For the changes to take effect on the partial bitstream filename and the pa size, only the user code (C program) should be recompiled as

Table 4.1: Time duration per user, system, or combined user/system action, for one experiment with the manual method.

action	bitstream	pa size	bc size
parameter change	15 s	15 s	25 s
compilation,downloading	12 s	12 s	45 s
self-test	16 s	16 s	16 s
sampling	30 s	30 s	30 s
measurement,sorting	60 s	60 s	60 s
total time (for 1 experiment)	133 s	133 s	176 s

these two parameters are set in the user code. The bc is part of the PowerPC settings, and in order to be altered, the entire project should be rebuilt, which incurs a long compilation. After downloading, a self-test routine is executed.

3. Sampling: The user prepares the logic analyzer for triggering, fires reconfiguration, and stops the logic analyzer data capturing.

4. Measurement: The logic analyzer markers measure the intervals between the signal transitions. Each marker is programmed to be automatically positioned on the edge of the signal that is used to measure a delay.

5. Sorting: The user inserts the values into spreadsheet cells and sorts them according to the analysis (s)he wants to conduct.

6. Plotting of the spreadsheet values.

The user has to carry out 704 iterations of stages 1-5 before data plotting. Table 4.1 shows the lower bounds of the time duration of each action[5]. The overall experimentation time is equal to the total number of experiments multiplied with their respective duration, as shown in the following equation:

$$\begin{aligned} overall\ time &= [(\#experiments - \#bc\ sizes) \times (fast\ total\ time)] \\ &\quad +(\#bc\ sizes \times slow\ total\ time) \\ &= [(704 - 8) \times (133s)] + (8 \times 176s) \\ &= 93,976s = 26.1h \end{aligned} \tag{4.2}$$

[5]Actions are distinguished in those carried out only by the user, e.g. parameter change; carried out only by the system, e.g. self-test; or those employing both, e.g. sampling. The actions are the same for other FPGA platforms, and only the time duration of the system actions would be different.

In this equation, the amount of times needed to modify the bc size, which is the most time-consuming process according to Table 4.1, is reduced to provide the optimal sequence of experiments with respect to time. The *fast total time* corresponds to the time to complete one experiment when any parameter except for the bc size is modified, and the *slow total time* corresponds to the time to complete one experiment when the bc size is modified. In addition, to reduce the overall time, some actions were overlapped, i.e. distinct actions that do not require the same resources of the setup, so they can be conducted simultaneously. Two of the authors participated in the experimentation phase and observed which actions were overlapped. For example, preparation of the parameters for the next experiment (15 or 25 sec) can be carried out simultaneously with the sampling of the current experiment (30 sec). Specifically, while one user prepares the logic analyzer, fires reconfiguration, and stops the capturing, the other user changes the parameter(s) for the next experiment. In addition, compilation, downloading (12 sec) and self-test (16 sec) of the next experiment can be carried out at the same time the user measures and sorts the current captured data (60 sec). Hence, in the previous equation the time duration of the actions that overlap with the longer durations in Table 4.1 can be eliminated:

$$\begin{aligned} \textit{overall time w/ overlap} &= (\#\textit{experiments}) \times [\textit{sampling time} \\ &\quad +(\textit{measurement}, \textit{sorting time})] \\ &= (704) \times (30s + 60s) \\ &= 63,360s = 17.6h \end{aligned} \tag{4.3}$$

Although the foregoing time durations are optimistic, repetitions of the user actions were boring and thus the process was automated.

4.4.3 The Framework

Firstly, to reduce the compilations and downloads per experiment, the change of the parameters which *correspond to the first and second stages of the manual method* was automated. Specifically, all combinations of bitstream sizes and pa sizes are included in the processor code, and their values are changed at run-time within loop structures, resulting in $11 \times 8 = 88$ experiments per one compilation and download. Regarding the bc size change, recompilation is inevitable as its value is entered during the setup of the PowerPC and not

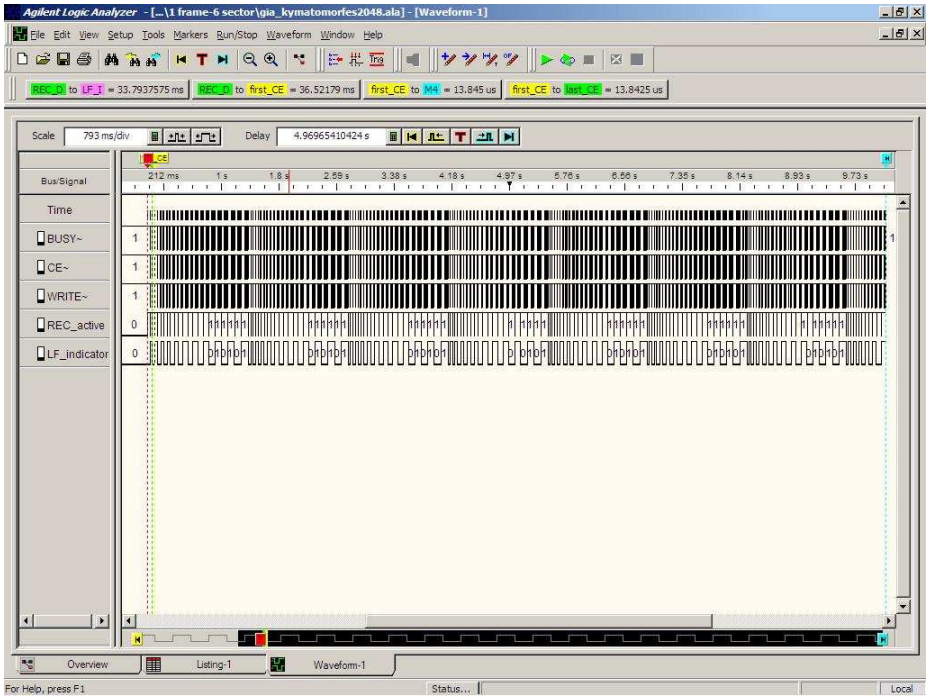


Figure 4.6: Logic analyzer trace for successive reconfigurations of bitstreams 1-11 of Table 4.3 and pa sizes ranging from 512 to 4,096 Bytes.

within the C code; thus, eight compilations and downloads are required. After bc change, the 88 experiments can be conducted with a single run.

Then the data sampling that *corresponds to the third stage of the manual method* was automated. This allows exploiting the full logic analyzer memory and relieves the user from triggering and stopping manually the logic analyzer. However, due to the latter's memory limitations, the automatic run of 88 parameter combinations does not fit (eight combinations do not fit), and hence, another sampling should be triggered.

The measurement and sorting that *correspond to the fourth and fifth stages of the manual method* were also automated. The captured data of each run are written in a .csv file and then transferred to the PC. A C program operates on windows of continuous data for identifying the edges of the signals where the measurements are to be taken by searching for "01" and "10" patterns. Within a window, subtractions between the time values of the appropriate signal transitions are made, and the results are written in a new .csv file.

Recapitulating, the framework consists of the following stages:

1. Change of the bc size only.

2. Compilation and downloading: Combinations of all bitstreams and pa sizes are compiled and downloaded at once, and then, a self-test routine is executed.

3. Sampling: The logic analyzer is prepared for triggering, and the reconfiguration button is pushed. Successive reconfigurations for all combinations of bitstreams and pa sizes are performed. Once the logic analyzer memory is filled, capturing stops.

4. Export to .csv: Captured data are written in a .csv file.

5. Measurement and sorting: The .csv file is loaded to the C program for measurements and calculations. The results are sorted and written in a new .csv file, which is then imported into a spreadsheet.

6. Plotting.

Stages 1-5 are repeated for all the bc sizes, resulting in eight recompilations/downloads. In addition, eight iterations should be executed due to the inadequacy of the specific logic analyzer memory. Table 4.2 has the time duration of each action. The overall time for the experiments is the number of user interventions multiplied with the time to complete one run, and it is shown in equation 4.4. A user intervention is either a recompilation due to the change of the bc size or a new sampling due to the inadequate logic analyzer memory.

$$\begin{aligned} overall\ time &= (\#experiments \times buffer\ cache) \\ &\quad +(\#experiments \times inadequate\ mem) \\ &= (8 \times 221s) + (8 \times 123s) = 2,752s = 0.77h \end{aligned} \tag{4.4}$$

The framework offers an improved productivity of $17.6h \div 0.77h = 22.8$ times as compared with the manual method. Figure 4.6 shows a logic analyzer trace, which was captured with one triggering within the framework. Successive reconfigurations of all bitstreams for different pa sizes result in 80 combinations that are then processed in an automatic way. With the manual method shown in Figure 4.5, only one reconfiguration for a parameter combination can be sampled.

Table 4.2: Time duration per user, system, or combined user/system action for one run using the framework. 88 different experiments, with each one corresponding to a different parameter combination, are executed with a single run.

action	bc size	inadequate memory
parameter change	25 s	15 s
compilation,downloading	45 s	12 s
self-test	16 s	16 s
sampling	60 s	15 s
export to .csv	35 s	25 s
measurement,sorting	40 s	40 s
total time (for 88 experiments)	221 s	123 s

4.5 Results

Thorough experiments for all parameter combinations were conducted. In each graph of Figure 4.7, the bitstream size varies, whereas the bc size and the pa size are kept constant. The three delays defined in Section 4.3 were measured with the logic analyzer within the framework.

For most parameter combinations, the behavior of the delays resembles Figure 4.7 (a). It reflects the frequent case according to which the delay increases linearly with respect to the bitstream size. Figure 4.7 (b) shows the less frequent case, wherein the reconfiguration time for bitstreams 3 and 4 decreases as opposed to the bitstream size. Thus the bc of 4,096 Bytes affects the time to write the configuration data. The Rec-HWICAP time is reduced, which causes reduction to the reconfiguration time (RT). This is due to the specific bc size only and does not depend on the pa size. It is clear that the increase in memory means, i.e. the bc size, is utilized more efficiently when reconfiguring bitstreams 3 and 4. Table 4.3 has the total reconfiguration times for both cases of Figure 4.7.

An interesting result derives from the comparison between the Rec-HWICAP delays of the two graphs for the same bitstream size. When bc=4,096, for bitstreams 1,3 and 4, the delay is significantly lower as compared with the case bc=3,072. As a consequence, the total reconfiguration time decreases; this is also illustrated in Table 4.3. However, this is not true for larger bitstreams. Therefore, depending on the size of the partial bitstreams, the selection of system parameters might improve or degrade the application performance.

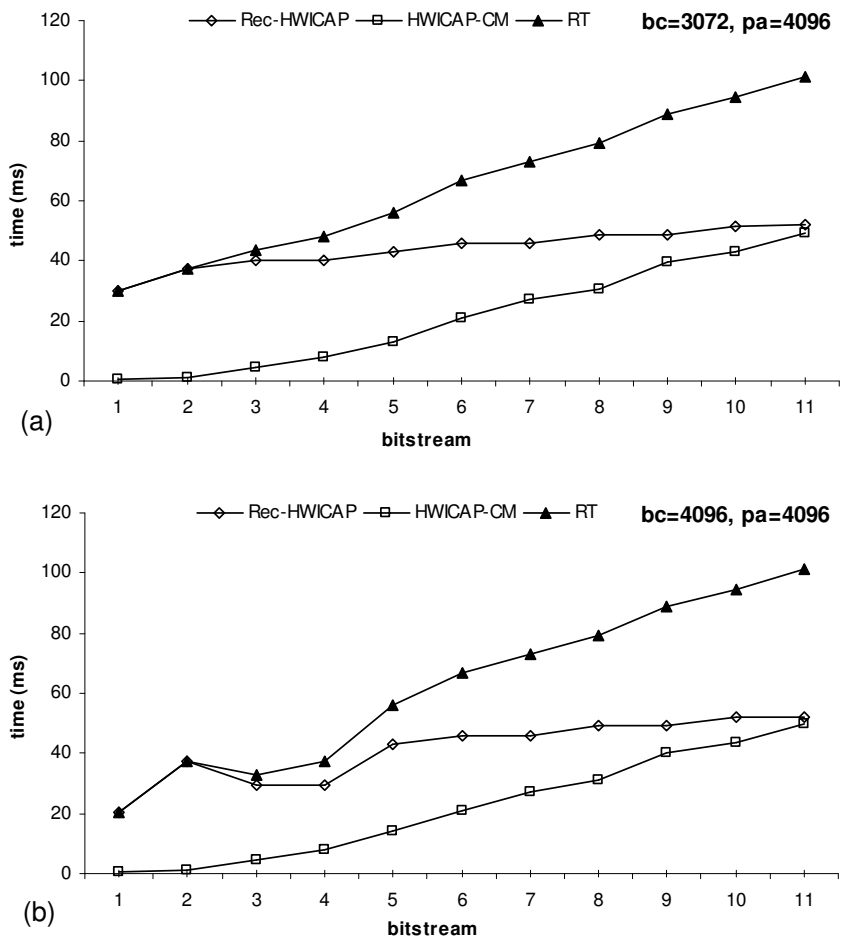


Figure 4.7: Reconfiguration time and piecewise delays for different bitstreams and bc, and fixed pa.

4.6 Conclusion

A methodology for the rapid evaluation of dynamic reconfiguration of FPGA platforms has been presented. It is proven that for some system parameters the reconfiguration time over the configuration size is not always linear at platform level. To gather such results a thorough experimentation is required, but doing this manually is a tedious process.

The framework can be ported to any reconfigurable platform with an embedded processor and an external memory for storing the partial bitstreams, such as platforms with the Xilinx Virtex-4 and Virtex-5 FPGAs that incorporate the hardcore PowerPC and/or the softcore Microblaze and the ICAP port, and contain a Compact Flash and the System ACE

Table 4.3: Size of the experimental partial bitstreams and reconfiguration times for the parameters of Figure 4.7.

bitstream	#words	#bits	$RT_{(a)}(ms)$	$RT_{(b)}(ms)$
1	636	20,352	29.7	20.2
2	842	26,944	37.2	37.2
3	1,258	40,256	43.7	32.8
4	1,464	46,848	48.3	37.4
5	1,880	60,160	56.1	56.2
6	2,086	66,752	66.7	66.8
7	2,502	80,064	73.1	73.2
8	2,708	86,656	79.2	79.3
9	3,124	99,968	88.6	88.8
10	3,330	106,560	94.5	94.7
11	3,746	119,872	101.1	101.3

Controller [Xilinx Inc., 2009b,Xilinx Inc., 2009c,Xilinx Inc., 2009d,Digilent Inc., 2008]. The only steps needed to adjust the framework in these platforms are i) recompilation of the project for the corresponding platform and ii) modifications in the user constraint file (.ucf) for the appropriate FPGA pins to be connected to the DIP switch, the push buttons and the expansion headers.

The reconfiguration time along with the piecewise delays were measured at the platform level in order to provide a holistic approach. Previous works that employed a logic analyzer to measure such delays concern obsolete FPGAs [McGregor and Lysaght, 1999,McKay and Singh, 1999] and do not target measurements from a system perspective [Tan et al., 2006]. Present Chapter provided detailed data for a particular platform concerning reconfiguration of bitstreams of different sizes loaded from a compact flash memory. The output of this work is used in the next Chapter to shape a model for assessing the reconfiguration time in a wide range of FPGA-based systems.

Chapter 5

Performance of Reconfiguration

Fine-grain reconfigurable devices suffer from the time needed to load the configuration bitstream. Even for small bitstreams in partially reconfigurable FPGAs this time cannot be neglected. This Chapter surveys the performance of the factors that contribute to the reconfiguration speed. Then, it elaborates further on the PR system discussed in the previous Chapter and by using the experimental results along with straightforward maths it produces a cost model of partial reconfiguration (PR). This model is introduced to calculate the expected reconfiguration time and throughput. In order to develop a realistic model all the physical components that participate in the reconfiguration process are taken into account. The parameters affecting the generality of the model and the adjustments needed per system for error-free evaluation are analyzed. The cost model is verified with real measurements, and then it is employed to evaluate existing systems published by other researchers. The percentage error of the cost model when comparing its results with the actual values of those publications varies from 36% to 63%, whereas existing works report differences up to two orders of magnitude. The cost model enables a user to evaluate PR and decide whether it is suitable for a certain application prior entering the complex PR design flow.

The present Chapter constitutes an integral work in the sense that it concentrates only on the reconfiguration time and the parameters affecting it. A taxonomy is made in order to explore the factors affecting reconfiguration time. Then a new model for estimating reconfiguration time is formed that can be used for various setups. Besides the cost model

itself, the novelty of this Chapter lies in that by applying straightforward maths a safe model is extracted that can be used for the early assessment of such systems with even a different setup.

5.1 Overview of Reconfiguration Performance Issues

During recent years many applications that exploit the dynamic nature of reconfigurable devices have been developed. Due to their SRAM-based technology, both fine-grain and coarse-grain reconfigurable devices can be reprogrammed potentially unlimited times. This reconfigurability can serve as the vehicle to customize the design of applications even during operation in order to improve their performance in terms of different aspects, such as speed, power, and area. The dissertation and thus present Chapter focus on fine-grain partially reconfigurable (PR) devices. Previous published work has shown that i)specialized circuits can operate at higher speeds vs. their general static counterparts [McKay and Singh, 1998], ii)chip area is saved by programming only the physical resources that are needed in each operation phase [Gholamipour et al., 2009], iii)power can be saved by programming only the circuit that is needed, which allows for static leakage reduction, and by programming optimized circuits, which allows for dynamic power reduction [Noguera and Kennedy, 2007, Paulsson et al., 2008].

The configuration data produced to program a reconfigurable device is called a bitstream. When only a portion of a PR device is to be reconfigured a partial bitstream is produced. Although fine-grain devices like FPGAs offer customization at the bit-level allowing for great flexibility, at the same time they require large bitstreams to be configured as opposed to the coarse-grain devices. This induces a considerable reconfiguration time, which cannot be neglected even for devices supporting partial reconfiguration. Usually, published reconfiguration times refer to the configuration port of the chip only [Lysaght et al., 2006]. There exist works proposing the incorporation of PR in their systems, which use such theoretical values and due to unrealistic assumptions the results can be questionable. When incorrectly used the reconfiguration time might deviate one or two orders of magnitude from the realistic value [Griese et al., 2004,Galindo et al., 2008]. For "real-world" systems a more

holistic approach is needed, taking into account all the system components that contribute to the total reconfiguration overhead, such as the internal and external memory, the internal configuration port, the reconfiguration controller, the FPGA configuration memory, and the connections means. As today FPGA platforms are used as final products and not just for rapid system prototyping, partial reconfiguration can play a significant role, and thus it would be useful to effectively precalculate its overhead.

Performance evaluation frameworks are being developed in order to leverage existing partial reconfiguration design flows with design space exploration at the early stages of development [Hsiung et al., 2008]. Toward this direction present work relies on experiments with a "real-world" platform used as reference, and with straightforward math, a model to evaluate reconfiguration time for different platforms without being necessary to enter the tedious PR design flow is constructed. The model is verified by comparing the results with real measured data and is used to evaluate previously published systems. The contributions are:

- a survey with the most recent works on measuring reconfiguration times of PR FPGAs, and an investigation of the performance margins for different system setups.
- an analysis of the system factors that contribute to the reconfiguration overhead of which a designer should be aware to evaluate the PR system.
- the formulation of a cost model[1]; to evaluate reconfiguration in platforms using an embedded processor to control the reconfiguration the proposed model does not rely on the throughput of the configuration port alone, and it can be used to compute the reconfiguration time without performing experiments.

The paper is structured as follows: Section 5.2 has the basics of partial reconfiguration and discusses recent works that include measurement of reconfiguration time. In Section 5.3 we categorize different system setups of published works, and we analyze the characteristics that affect the performance of reconfiguration. Section 5.4 describes the reference system architecture. Section 5.5 analyzes the components that contribute to the total reconfiguration time and proposes the cost model. In Section 5.6 we first verify the model and then

[1]available on line in [PRCC, 2010].

use it on different platforms and compare its results with real measurements appeared in previous publications of other researchers, in order to extract the percentage error. Also we discuss the merits and limitations of the model. Finally, Section 5.7 summarizes the benefits of our work.

5.2 State of the Art

Several vendors are striving to develop devices supporting partial reconfiguration. Xilinx offers the largest FPGA matrices which incorporate this feature, and most research efforts on this field use devices from this vendor. We focus on these devices and we provide some background on partial reconfiguration technology along with the corresponding terminology. Then we discuss recent works that measure the reconfiguration time and throughput on "real-world" platforms with modern FPGAs. Some of them include theoretical formulas to calculate the expected results.

5.2.1 Background on Partial Reconfiguration

The generic structure of the Xilinx Virtex FPGAs is shown in Figure 5.1. It is a Virtex-II/Pro device [Xilinx Inc., 2007b], one of the most widely used FPGA devices with partial reconfiguration capability. The newer Virtex-4 and Virtex-5 FPGAs have similar structure. It comprises an embedded processor such as a hardcore PowerPC or a softcore Microblaze, a reconfigurable array, the Processor Local Bus (PLB) [IBM Inc., 2000] and the less efficient On-Chip Peripheral Bus (OPB) [IBM Inc., 2001]. The processor is attached on the PLB bus, which communicates with the OPB through a bridge. The modules implemented in the array logic can be attached to the buses and act as peripherals to the processor. The array is 2-D fine-grain heterogeneous, mainly composed of configurable logic blocks (CLBs), hardcore memory blocks (BRAMs), hardcore DSP units (MULTs) and I/O resources. Each CLB contains look up tables (LUTs), flip-flops, multiplexers and gates that are configured to implement the design logic. The array can be configured either externally or internally, by a processor or a dedicated reconfiguration controller. The serial JTAG, the SPI and the parallel SelectMAP allow for external configuration, whereas the parallel

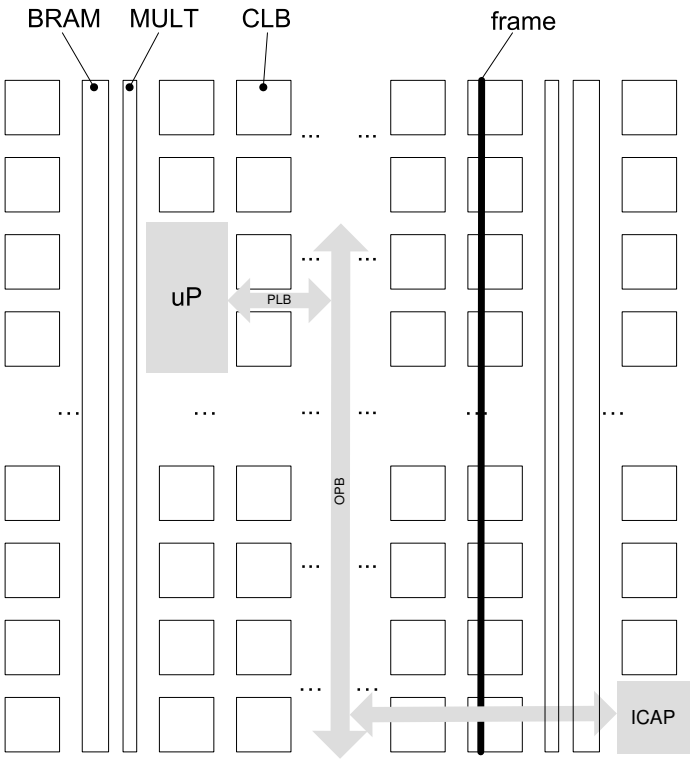


Figure 5.1: The Virtex II-Pro FPGA resources and configuration frames.

Internal Configuration Access Port (ICAP) allows for internal - partial only - configuration. The ICAP in Virtex-II/Pro has a width of 8-bits, whereas Virtex-4 and Virtex-5 families support 16-bit and 32-bit transfers as well. The maximum operational frequency of ICAP suggested by the vendor is 100 MHz. A BRAM attached to the ICAP caches the configuration bits before they are loaded to the FPGA configuration memory. An IP core called OPBHWICAP [Xilinx Inc., 2006a] is provided by the vendor which is connected on the OPB bus as a slave peripheral, and enables a processor to access the configuration memory through the ICAP, by using a library and software routines. For the Virtex-4 and Virtex-5 FPGAs, the XPSHWICAP [Xilinx Inc., 2007c] has been released, which works similarly with the OPBHWICAP, but it is connected on the PLB bus allowing for lower-latency reconfiguration.

The configuration memory of Virtex-II/Pro is arranged in vertical frames that are one bit wide and stretch from the top edge to the bottom edge of the device. They are the smallest addressable segments of the device's configuration memory space, so all operations

must act on entire configuration frames. They do not directly map to any single piece of hardware; rather, they configure a narrow vertical slice of many physical resources [Xilinx Inc., 2007b]. A pad frame is added at the end of the configuration data to flush out the pipeline of the reconfiguration processing machine in order for the last valid frame to be loaded. Therefore, to write even one frame to the device it is necessary to clock in two frames; the configuration data frame plus a pad frame. In the newer Virtex devices, the structure and the PR technology is essentially the same, with a few changes. In the previous Virtex-II/Pro family the frame size is not fixed as it is proportional to the entire height of the device, which differs amongst the parts. By contrast, in Virtex-4 and -5 families the frame spans a fixed height of the device; its width is one bit. Moreover, the OPB bus is not suggested anymore due to its low performance, and although it will not likely be supported in the future it remains optional within the infrastructure. The PLBv4.6 (also called XPS) is now the default, which provides more bandwidth, less latency and it is more deterministic and stable as compared with the OPB. Also, PLB allows for simultaneous read and write transfers and can be instantiated as either a shared bus, or for point-to-point connections; the latter option reduces the latency of transfers between the modules connected on the bus.

The PR design flow suggested for complex designs is the module-based flow [Lysaght et al., 2006]. One or more areas of the FPGA are defined as partially reconfigurable regions (PRR). For each PRR more than one module called partially reconfigurable modules (PRM), can be created and imported. A partial bitstream is generated to create each PRM. During operation, the PRMs can be swapped in and out of the corresponding PRR without interrupting the remaining hardware.

5.2.2 Reconfiguration Time and Throughput

Table 5.1 has some of the most representative works measuring the reconfiguration time, sorted according to the publication year[2]. It contains the characteristics of "real-world" platforms, along with the respective reconfiguration time and throughput measured with

[2]At this point the sorting of Table 5.1 is done according to the publication year (first column), rather than according to the reconfiguration throughput (last column). A categorization according to the throughput is provided in the following Table.

hardware and/or software means, such as a logic analyzer and software time-stamps. The first column has the *Reference*. The *Storage* column has the memory means from which the bitstream is loaded to the configuration memory. The next column consolidates the type, the bit-width and the operational frequency in MHz of the *Configuration port*, e.g. ICAP8@100 refers to the 8-bit ICAP port running at 100 MHz. In case a characteristic is missing it means the information is not provided by the authors. The column *Cntlr* refers to the type of reconfiguration controller. The prefixes "v-" and "c-" define whether the solution is furnished by the "vendor" or by the authors as a "custom" solution respectively. Thus the c-OPB and the c-PLB refer to a custom reconfiguration controller located on the OPB or the PLB bus respectively, whereas the v-OPB and the v-XPS refer to the vendor's OPBHWICAP and XPSHWICAP solutions respectively that are controlled by an embedded processor (alternatively a custom core can be implemented to interface with the vendor's OPBHWICAP and XPSHWICAP). The last three columns have the *Bitstream Size (BS)* in KBytes, the corresponding *Reconfiguration Time (RT)* in msec and the system's *Actual Reconfiguration Throughput (ARTP)* in MBytes/sec. The values are included as published, and in the rare case a value is not explicitly reported in a reference it is calculated, e.g. the fraction of the bitstream size over the reconfiguration time calculates the actual reconfiguration throughput and vice-versa with the formula $ARTP = \frac{BS}{RT} \Leftrightarrow RT = \frac{BS}{ARTP}$. All experiments of Table 5.1 concern Virtex-II and Virtex-4 FPGAs, mainly the XC2VP30, XC4FX20 and XC4FX60 devices. The 8-bit ICAP corresponds to the Virtex-II, and the 32-bit to the Virtex-4 FPGA families.

Along with the Table 5.1, a discussion on partially reconfigurable systems follows by concentrating on i)the type of the external storage from which the partial bitstream is loaded to the array, ii)the type of the reconfiguration controller, iii)the measurement of the reconfiguration process and its phases, and iv)the theoretical formulas to calculate the expected reconfiguration time and throughput. The configuration studied here focuses on ICAP only which provides the fastest way to partially reconfigure the array.

Commonly, the partial bitstreams are stored in an external non-volatile memory. Therefore a host PC can be used which, after the system startup, transfers the bitstream to the FPGA either through PCI [Griese et al., 2004] or serial port [Fong et al., 2003]. For embed-

Table 5.1: Reconfiguration-related characteristics and measured reconfiguration time and throughput. The Bitstream Size (BS) is in KBytes, the Reconfiguration Time (RT) in milliseconds, and the Actual Reconfiguration Throughput (ARTP) in MBytes/sec.

Reference	Storage	Conf. Port	Cntlr	BS	RT	ARTP
[Fong et al., 2003]	PC/RS232	ICAP8	c-HW	34.8	6,200	0.005
[Griese et al., 2004]	PC/PCI	SMAP8@25	c-HW	57.0	14.328	3.88
[Gelado et al., 2006]	$BRAM_{PPC}$	ICAP8	v-OPB	110.0	25.99	4.13
[Delahaye et al., 2007]	SRAM	ICAP8	v-OPB	25.7	0.510	49.20
[Papadimitriou et al., 2007]	$BRAM_{PPC}$	ICAP8@100	v-OPB	2.5	1.67	1.46
[Papadimitriou et al., 2010]	CF	ICAP8@100	v-OPB	14.6	101.1	0.15
[Claus et al., 2007]	DDR	ICAP8@100	c-PLB	350.75	3.75	91.34
[Claus et al., 2007]	DDR	ICAP8@100	v-OPB	90.28	19.39	4.66
[Claus et al., 2008]	DDR	ICAP8@100	c-PLB	70.5	0.803	89.90
[Claus et al., 2008]	DDR	ICAP8@100	v-OPB	70.5	15.0	4.77
[Claus et al., 2008]	DDR2	ICAP32@100	c-PLB	1.125	0.004	295.40
[Claus et al., 2008]	DDR2	ICAP32@100	v-OPB	1.125	0.227	5.07
[French et al., 2008]	DDR2	ICAP32@66	v-OPB	514.0	112.0	4.48
[Manet et al., 2008]	DDR2/ZBT	ICAP32@100	c-OPB	166.0	0.47	353.20
[Liu et al., 2009]	DDR2	ICAP32@100	v-OPB	79.9	135.6	0.61
[Liu et al., 2009]	DDR2	ICAP32@100	v-XPS	80.0	99.7	0.82
[Liu et al., 2009]	DDR2	ICAP32@100	v-OPB	75.9	7.8	10.10
[Liu et al., 2009]	DDR2	ICAP32@100	v-XPS	74.6	4.2	19.10
[Liu et al., 2009]	DDR2	ICAP32@100	c-PLB	81.9	0.991	82.10
[Liu et al., 2009]	DDR2	ICAP32@100	c-PLB	76.0	0.323	234.50
[Liu et al., 2009]	$BRAM_{ICAP}$	ICAP32@100	c-PLB	45.2	0.121	332.10

ded solutions a compact flash located on the FPGA platform is preferred, which is the case for most works of Table 5.1. Other non-volatile memories that can be used are linear flash, SPI flash, and platform flash PROM. After the system boots, the bitstream can be loaded to a high-speed memory, either off-chip or on-chip, in order to perform faster reconfiguration. If only the compact flash is used, a longer reconfiguration time is required [Papadimitriou et al., 2007]. The different memory types are characterized by certain advantages and disadvantages [Möller et al., 2006].

With respect to the type of reconfiguration controller various solutions exist, each of which can be employed depending on the application needs. Some works used the OPBHWICAP [Gelado et al., 2006, Papadimitriou et al., 2007, Delahaye et al., 2007, Claus et al., 2007, Claus et al., 2008, French et al., 2008, Liu et al., 2009] whereas a few experimented with the XPSHWICAP [Liu et al., 2009]. On the other hand, customized reconfiguration controllers aim to speedup reconfiguration and/or release the processor time to other tasks,

such as the reconfiguration management under the supervision of an operating system [Santambrogio et al., 2008]. This can be done either through Direct Memory Access (DMA) with a customized controller on the OPB [Manet et al., 2008] or the PLB bus [Claus et al., 2007, Claus et al., 2008, Liu et al., 2009], or with a PLB master connected directly with the controller of the external memory, allowing for burst transmissions [Liu et al., 2009]. Moreover, to speed up reconfiguration, partial bitstreams can be loaded immediately after the system startup into large on-chip memory implemented with BRAMs attached to the reconfiguration controller [Liu et al., 2009]. In general, the systems with the processor acting as the reconfiguration controller suffer from long delays due to the large time needed to access its program memory to call and execute the software instructions. Also, when several modules are connected to the bus the delay is longer and unstable due to the contention between reconfiguration and data transfers.

The distinct phases during reconfiguration can vary depending on the system setup. The measured times mainly concern the phase to pull the bitstream from the off-chip memory to the local memory of the processor, copy it from the local memory of the processor to the ICAP, and send it from the ICAP to the FPGA configuration memory [Gelado et al., 2006,Papadimitriou et al., 2007,French et al., 2008]. These subtasks iterate until the entire partial bitstream is written to the configuration memory [Papadimitriou et al., 2010]. For specific systems other subtasks have been measured, such as the time for configuration code analysis to guarantee safety, initialization and start time of the reconfiguration controller [Griese et al., 2004], the time to send the adequate instructions to the ICAP [Gelado et al., 2006], and the time to copy the configuration data from user space to kernel space in Linux [French et al., 2008]. In all systems the largest overhead comes from the bitstream transfer from the external memory to the on-chip memory [Griese et al., 2004,Papadimitriou et al., 2007, Papadimitriou et al., 2010], and from the processor to the ICAP through the bus [Gelado et al., 2006,French et al., 2008,Delahaye et al., 2007].

Although some of the above works evaluate the time spent in each reconfiguration phase, they do not provide a theoretical analysis taking into account the system characteristics. Some of them calculate the expected reconfiguration throughput based on the bandwidth of the configuration port only [Claus et al., 2007,Manet et al., 2008]. However, in [Griese

et al., 2004] the authors demonstrated that the actual reconfiguration throughput deviates one order of magnitude from the theoretical bandwidth of the configuration port. In another work this deviation reaches up to two orders of magnitude [Galindo et al., 2008]. Towards the same direction, the reconfiguration times for Virtex-II Pro and Virtex-4 devices are reported in [Lysaght et al., 2006]. For example the partial bitstream for 25% of a XC2VP30, which is equal to 353 KBytes, can be loaded in less than 6 ms when the ICAP operates at 100 MHz and is fully utilized. This has a considerable difference with the end-to-end system times, and depending on the system setup the actual reconfiguration times can take significantly longer, as demonstrated in Table 5.1. The authors in [Claus et al., 2008] provide a more sophisticated model to calculate the expected reconfiguration throughput and latency. However, that model is based on the characteristics of the configuration port itself only; these are the operational frequency, the interface width and the frequency of activation of the ICAP's "BUSY" handshaking signal. That model cannot be safely applied on the Virtex-4 devices due to the different behavior of the "BUSY" signal in the specific family. Although it can be fairly accurate under specific circumstances, if a designer wants to evaluate a system, (s)he would need to built it first, then evaluate it, and then examine whether it meets the application needs. The present work aims to develop a model that will provide better estimation regardless of the maximum throughput of the configuration port early in the design stage.

5.3 Investigation of Reconfiguration Performance

This Section analyzes further the characteristics affecting the reconfiguration performance. Table 5.2 categorizes the existing approaches according to the configuration port and its bandwidth, the reconfiguration controller and the storage means. It also summarizes the expected range of the actual reconfiguration throughput according to the system characteristics. It is observed that the throughput varies heavily depending on the three characteristics.

When comparing the *cases A and B* where the PowerPC (PPC) processor is the reconfiguration controller, loading the partial bitstream from the local memory of the processor

Table 5.2: Comparison between the bandwidth of the configuration port and the actual reconfiguration throughput for different published system setups.

	Configuration Port				
case	Type	Bandwidth	Cntlr	Storage	ARTP
A	ICAP8@100	95.3 MB/s	v-OPB	CF	0.15 MB/s
B			v-OPB	$BRAM_{PPC}$	1.46 MB/s
C			v-OPB	DDR	4.54-4.77 MB/s
D			c-PLB	DDR	89.90-91.34 MB/s
E	ICAP32@100	381.5 MB/s	v-OPB	DDR2	0.61-11.1 MB/s
F			v-XPS	DDR2	0.82-22.9 MB/s
G			c-OPB	DDR2/ZBT	353.2 MB/s
H			c-PLB	DDR2	82.1-295.4 MB/s
I			c-PLB	$BRAM_{ICAP}$	332.1-371.4 MB/s

implemented with BRAMs offers a speedup of 10x over the compact flash solution [Papadimitriou et al., 2007]. In *case I*, where on-chip memory implemented with BRAMs attached to the ICAP is used to fetch the entire partial bitstream prior reconfiguration takes place, much better results are produced. In this case a dedicated reconfiguration controller accounts for the high throughput, as it continuously pulls the prefetched configuration data; this allows for the almost full utilization of the configuration port [Liu et al., 2009].

In *case D* where a customized reconfiguration controller located on the PLB bus is utilized, the reconfiguration throughput reaches almost the bandwidth of the 8-bit ICAP configuration port [Claus et al., 2007, Claus et al., 2008]. However in *case H* when the 32-bit ICAP is used, the same setup cannot achieve the configuration port bandwidth. This might be due to the DDR2 SDRAM controller which is not able to feed the reconfiguration controller fast enough in order to attain the ICAP bandwidth [Claus et al., 2008, Liu et al., 2009]. High throughput approaching the theoretical maximum is also achieved in *case G* with a custom reconfiguration controller located on OPB [Manet et al., 2008]. In this work, the authors reported that the throughput was ranging from 0.46 to 353.2 MB/sec for different system setups depending on the reconfiguration controller and the type of external memory. Their fastest solution was provided with a custom OPB ICAP reconfiguration controller when the partial bitstreams were loaded from a high-speed external memory; however the authors do not report whether the DDR2 SDRAM or the ZBT SRAM was

used. They were not able to reach the maximum throughput due to the DMA overhead. Also, due to a clock skew problem they sent the configuration data on the falling edge of the ICAP clock. It is worth noticing that the throughput of the best case of the custom OPB controller of [Manet et al., 2008] which is 353.2 MB/s is higher than the custom PLB controller of [Claus et al., 2008] which is 295.4 MB/s, despite the fact that the former is attached on the OPB bus and the latter on the more-efficient PLB bus (see Table 5.1 of Section 5.2). Two can be the possible reasons the system with the OPB reconfiguration controller offers higher throughput; either i)the bitstreams are fetched from the ZBT SRAM which provides higher throughput than the DDR2 SDRAM and consequently allows for higher utilization of the ICAP bandwidth, or ii)if the bitstreams are fetched from the DDR2 SDRAM, the corresponding controller is implemented more efficiently.

When the processor acts as the reconfiguration controller its settings can affect significantly the reconfiguration speed. This holds for both the OPBHWICAP and the XPSHWICAP of *cases E and F* respectively. Such settings are the selection of a hardcore processor (e.g. PowerPC), or a softcore processor (e.g. Microblaze), and the incorporation of separate instruction cache and data cache [Liu et al., 2009]. Another study shows that the change of the amount of memory allocated in the processor local memory for buffering reads and write calls to the compact flash caused variations in the reconfiguration time with respect to the number of reconfiguration frames [Papadimitriou et al., 2010]. Another action towards increase of reconfiguration performance is the replacement of the vendor's Application Program Interface (API) that handles the ICAP accesses with a better one [Möller et al., 2006].

Based on the foregoing observations the following conclusions are drawn for recent PR technology on "real-world" systems:

- The reconfiguration throughput of the system cannot be calculated from the bandwidth of the configuration port alone. To evaluate the system holistically, the overhead added by the components participating in the reconfiguration process should be taken into account.

- The characteristics that affect the reconfiguration overhead depend on the system

setup, such as the external memory and the memory controller, the reconfiguration controller and its interface with the configuration port, and the user space to kernel space copy penalty when an operating system running on the processor controls the reconfiguration.

- A bus-based system that connects the processor, the partially reconfigurable module(s), the static module(s), and the configuration port, can be non-deterministic and unstable due to the contention between data and reconfiguration transfers.

- A large on-chip memory implemented with BRAMs attached to the configuration port that prefetches the bitstreams after the system boots can allow for fast reconfiguration. Due to the limited size of BRAMs in FPGAs, the utilization cost should be considered according to the application needs, i.e. using BRAMs to prefetch configuration vs. saving BRAMs to deploy the circuits.

- The settings of the processor can affect the reconfiguration time. Moreover, the size of the local memory of the processor can affect significantly the reconfiguration time. However, similar to the above case the cost of BRAM utilization should be considered.

- The selection of the reconfiguration controller depends on the application needs, the available resources, the speed and the power constraints. For example, in case area is of greater concern than speed, and an operating system executes on an embedded processor, the latter can undertake the reconfiguration process in order to avoid the consumption of logic resources to implement a dedicated reconfiguration controller. Previous works suggested ways to reduce the reconfiguration overhead if this is needed [Hauck, 1998a, Papademetriou and Dollas, 2006b].

- Nearly the full bandwidth of the configuration port can be used with a dedicated reconfiguration controller that is either equipped with DMA capabilities, or acts as a master directly connected to the configuration port allowing for burst transmissions. Also, effective design choices at the system level should be made such as the usage of a high-speed memory to store the partial bitstreams after power-up and an efficient memory controller.

Obviously only the development of an efficient reconfiguration controller can allow for the actual reconfiguration throughput to approach the ICAP bandwidth. However, this is not the common case either due to the system components that bound the full utilization of the ICAP or the designer's choices and application needs. A cost model to precalculate the expected reconfiguration time can assist the performance evaluation phase. Moreover, it would be beneficial if this model could be used early in the design flow in order to either i)prevent the designer from entering the tedious PR design flow, or, ii)timely change the settings of the system setup. The evaluation can be done either right after the bitstream estimation of the partially reconfigurable modules at the initial budgeting stage of the floorplanning stage, or even after the synthesis results. Although the latter method is less accurate, the comparison with experimental values has shown that the cost model still produces satisfactory results.

Several works exploring applications that benefit from partial reconfiguration can be augmented with a cost model for partial reconfiguration in order to provide realistic data. For the dynamically reconfigurable network processor proposed in [Kachris and Vassiliadis, 2006] the cost model can calculate the time needed for reconfiguration after which the system will meet the network workload. Bitstreams of different encoding, encryption and transcoding modules are fairly large and should be located in an external memory, so, the added overhead before the system is ready to execute is useful to know. In the field of multi-mode communication systems the cost model can be used to calculate the time to load the optimized configurations of the filters [Gholamipour et al., 2009].

Furthermore, assuming that unlimited amount of on-chip memory is available to fetch the entire bitstream and then load it at the ICAP's speed is not realistic. An analysis on the memory resources required by the circuit itself when deployed in the field should be performed firstly by the designer. Also, an analysis of the local memory of the processor and the bitstream size is needed, prior deriving the amount of BRAMs that can be allocated for bitstream prefetching. Table 5.3 shows that moderate-size FPGAs commonly used in research have limited BRAM resources. In many cases the amount of on-chip memory imposes limitations to the size of the bitstreams that can be prefetched; this is obtained from Table 5.1 that contains bitstream sizes for several "real-world" applications.

Table 5.3: Size of BRAM resources for moderate-sized FPGAs of Virtex-II,-4,and -5 families. The Table illustrates the maximum size of the bitstreams that can be prefetched in the extreme case none of the actual circuits utilizes BRAMs.

FPGA	BRAM size (KBytes)
XC2VP30	306
XC4FX60	522
XC5VLX50T	270

5.4 System Architecture

This Section initially presents a general architecture of a partially reconfigurable system, and then it describes a realistic system with a a Xilinx Virtex-II Pro FPGA that is identical to the one studied in the previous Chapter. It is used as the reference system to develop the cost model.

5.4.1 General Architectural Model

Figure 5.2 illustrates the general architectural model of a PR FPGA-based system. The FPGA main components are the array, the configuration memory that programs the array, the reconfiguration controller, the configuration port and the on-chip memory buffer. The connection between them is made either point-to-point or through a bus. Other system components are the volatile high-speed memory from which the partial bitstreams are loaded and its memory controller. The memory controller can reside either off-chip or on-chip implemented as an IP module. Other necessary components are the non-volatile memory used as a repository for the partial bitstreams and its controller. However, these are omitted from Figure 5.2 as after the system boots, the bitstreams can be copied to the high-speed memory (for faster reconfiguration), and thus they do not need to be involved in the process again. If the off-chip high-speed memory is left out and each time the bitstream is loaded directly from the compact flash, throughput becomes flash-dominated and little transfer speed improvement is possible.

The configuration takes place in two distinct phases shown with the dashed lines. Once the memory controller is instructed to load the bitstream, it is copied from the off-chip memory to the on-chip memory buffer. Then the reconfiguration controller loads the bitstream

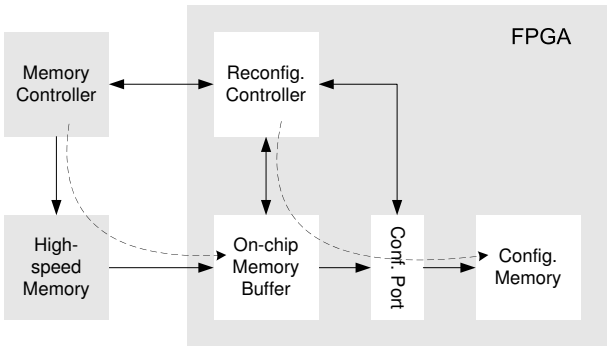


Figure 5.2: General architectural model and flow of partial reconfiguration.

to the FPGA configuration memory through the configuration port. These phases occur alternately in succession until the entire bitstream is copied to the configuration memory, except for the case when the entire bitstream is prefetched in the on-chip memory. If this is desirable (depending on the user application) and feasible (depending on the available on-chip memory) the first phase can be executed before reconfiguration is actually needed, i.e. after the system boots, or, scheduled during system operation before the task is needed for execution. It is important to notice that the on-chip memory can be composed of separate memory blocks that communicate hierarchically; this holds for the following system.

5.4.2 The Reference System

Figure 5.3 shows the PR system implemented on a platform with a Virtex-II Pro FPGA as shown in the previous Chapter and published in [Papadimitriou et al., 2007, Papadimitriou et al., 2010]. The white boxes are part of the FPGA, while the grey boxes lie outside of the FPGA. The Device Control Register (DCR) bus is used for communicating with the status and control registers of the peripherals. The PPC is attached on the PLB bus and contains two interfaces capable of accessing memory, the Processor Local Bus (PLB) interface and the On-Chip Memory (OCM) interface; in the present case the first interface is used by attaching on the bus a PLB_BRAM controller connected with the PPC local memory. The OPBHWICAP incorporates a finite state machine for control, a configuration cache implemented with one BRAM (set 2 KBytes by the vendor) and the ICAP. The PRR and the static part are OPB peripherals. The OPB sysace peripheral is used to interface with the external System ACE controller in order to communicate with the compact flash. A

DDR controller attached on the PLB controls the DDR SDRAM memory; however it is not used in the current setup.

The PR system is representative as it belongs to the largest vendor in the field that supports fine-grain partial reconfiguration. Also, although the PowerPC is currently used as the embedded processor to control the reconfiguration, the softcore Microblaze can be used instead, which also supports the reconfiguration functions. Other custom processors can be used as well but the appropriate functionality should be developed from the scratch to support reconfiguration.

At this point it is important to discuss the way the bus-based system works. The IBM CoreConnect infrastructure is the backbone of the internal system connecting the processor to the peripherals using the PLB, the OPB, and the DCR buses to build a complete system [IBM Inc., 2000, IBM Inc., 2001, IBM Inc., 2006]. The DCR bus is used for accessing status and control registers within the PLB and OPB masters and slaves. It off-loads the PLB and the OPB from the lower performance read/write transfers from/to the status and control registers. It also decreases access latency during periods of high processor bus utilization. DCR is not part of the system memory map, and thus it removes configuration registers from the memory address map, allowing for data transfers to occur independently from, and concurrent with, PLB and OPB transfers.

In the setup of Figure 5.3, the PPC is the controller of the reconfiguration process carried out in three distinct phases that are repeated until the entire bitstream is written in the configuration memory [Papadimitriou et al., 2010]. These phases are shown with the dashed lines: i)First the PPC requests the bitstream from the external storage means (the compact flash in the present case) and writes it in its local memory (first level of the on-chip memory hierarchy) with block-by-block transactions, ii)then the PPC transfers the bitstream word-by-word to the ICAP configuration cache (second level of the on-chip memory hierarchy), and iii)once the configuration cache is full the PPC instructs the OPBHWICAP to load the bitstream to the FPGA configuration memory through the ICAP. These actions are determined with software commands residing in the PPC program memory. For sake of simplicity they will be referred to as SM-PPC[3], PPC-ICAP and ICAP-CM respectively.

[3]SM and CF acronyms are used throughout this Chapter depending on the type of storage is being

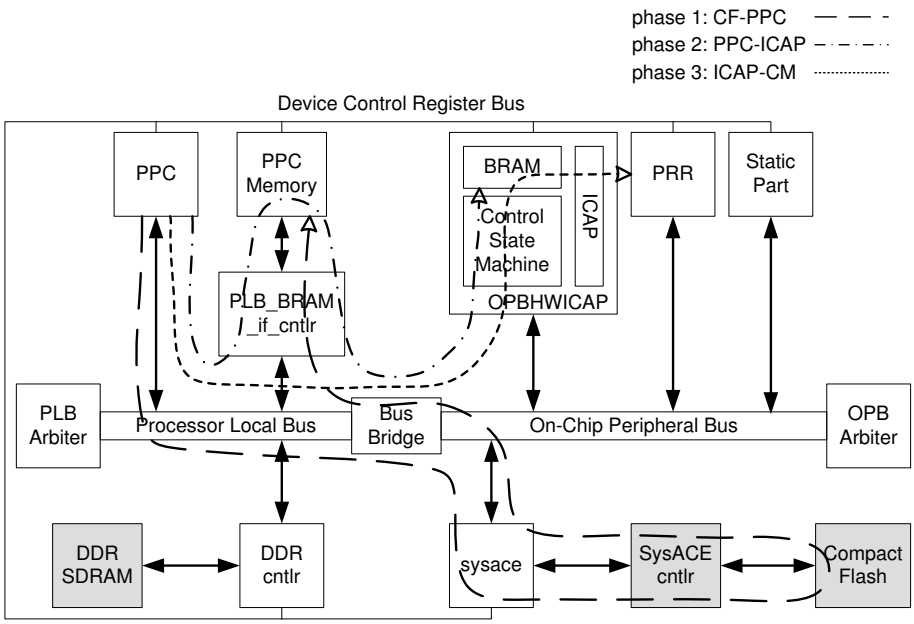


Figure 5.3: Flow of partial reconfiguration in the reference system with a Virtex-II Pro.

Table 5.4: System settings.

Parameters	Size
CF sector	512 Bytes
PLB BRAM block	48 KB
stack	6 KB
buffer cache	0.5-4 KB
processor array	0.5-4 KB
ICAP BRAM cache	2 KB

Table 5.4 has the system parameters that were configured as fixed and those that were varied to examine their affect on performance. The PPC local memory was 48 KBytes and the stack was 6 KBytes. The buffer cache (bc) defines the amount of memory for buffering read and write calls to the System ACE controller. The processor array (pa) is allocated in the PPC local memory to store the configuration data, and determines the amount of data read from the compact flash per transaction[4]. The ICAP cache has fixed size set by the

referred, i.e. SM stands for storage means referring to a general type of memory, either volatile or non-volatile, high-speed or low-speed, and CF stands for the compact flash which is non-volatile low speed memory.

[4]Transactions are conducted in multiples of one sector per processor call. A sector is the smallest unit the compact flash is organized in and is equal to 512 Bytes. Thus during experimentation the processor array size was varied in multiples of one sector.

vendor one BRAM, which is equal to 2 KBytes. In order to configure a system with low resources the instruction/data caches and interrupts of the PPC were not enabled.

Table 5.5 has the time spent in the processor calls (that force the transactions) during reconfiguration for different processor array sizes. They were measured with software time-stamps. It is observed that the processor array dictates the amount of data transferred during the SM-PPC phase. In the same way it dictates the transfer in the PPC-ICAP phase. However, in the case the processor array is larger than the ICAP cache, the amount of data per call is bounded by the latter's size. This is illustrated in the fifth row of Table 5.5 where the PPC-ICAP time is the same with the one shown in the above row. Moreover, if the processor array is larger than the bitstream, the latter is transferred with one processor call and stored in the processor local memory in its entirety prior the transfer to the ICAP. From the ICAP side, the configuration cache size is fixed and thus inhibits accommodation and transfer of the entire bitstream at once. Also, the ICAP cache should be filled up with 2048 Bytes before it is ready to write the configuration memory. This is the reason the ICAP-CM time is constant in all cases. For example, in the first row of Table 5.5 where processor array = 512 Bytes, iterations of the first and the second phases are executed in succession (with the corresponding processor calls) until the ICAP cache is filled up. In particular, four consecutive iterations of the two first phases occur until the ICAP cache will become full with 2048 Bytes. Then, in the third phase, the contents of the ICAP cache are loaded to the FPGA configuration memory with one transaction (caused by the corresponding processor call). The foregoing sequence is repeated until the entire bitstream is copied to the configuration memory.

The above observations revealed the way the reconfiguration operation works. The data of Table 5.5 were processed to obtain a metric that has a meaningful system-level design correspondent. Specifically, the "average time-per-processor-call" was extracted . It is observed that for the SM-PPC and PPC-ICAP phases, the time per 512 bytes (the smallest unit that can be transferred) is almost constant. For the SM-PPC this time is $\sim 1.45ms$, while for PPC-ICAP it is $\sim 0.42ms$. For the ICAP-CM phase the quantity of 512 Bytes is not the smallest quantity, as exactly 2048 Bytes are needed to initiate a transfer to the configuration memory. In turn, the Table is formed 5.6 which will be used in Section 5.5 to

Table 5.5: Execution time-per-processor-call for different sizes of the processor array. The amount of data transferred per call is dictated by the smallest amongst the processor array and the ICAP cache. In the ICAP-CM phase, the processor call needs 2048 Bytes to perform a transaction.

processor array (Bytes)	SM-PPC (ms)	PPC-ICAP (ms)	ICAP-CM (ms)
512	1.5	0.42	
1024	2.94	0.83	
1536	4.35	1.25	0.02526
2048	5.79	1.67	
2560 ($> ICAP\ cache$)	7.24	1.67	

Table 5.6: Average time-per-processor-call. For the analysis to be realistic it is adhered to the way the phases are carried out by the corresponding processor calls; the minimum amount of data for the SM-PPC and PPC-ICAP phases to be carried out is 512 Bytes, while ICAP-CM phase needs exactly 2048 Bytes.

SM-PPC	PPC-ICAP	ICAP-CM
1.45 ms (per-512 Bytes)	0.42 ms (per-512 Bytes)	0.02526 ms (per-2048 Bytes)

construct the cost model.

5.4.3 Options Affecting Reconfiguration Performance

Based upon the foregoing observations the optional processor features that can improve reconfiguration speed are listed:

- The processor array puts the upper limit on the amount of configuration data transferred from the compact flash per processor call. Moreover, when the processor array is smaller than the ICAP cache, the former puts a limit on the amount of configuration data sent to the ICAP cache per processor call.

- Enabling the I-Cache and D-Cache of the processor can improve significantly the reconfiguration throughput. Although their deactivation accounts for resource savings, it causes diminishing results in performance. A study for the Virtex-4 has proven that for a system the overall improvement ranges from 16.6 to 23.3 times [Liu et al., 2009].

- The Application Program Interface (API) provided by Xilinx allows the software control of the IP core (OPBHWICAP or XPSHWICAP) to access ICAP. It requires the software controller on the processor to fetch 512-word blocks of the configuration

data and store them in a BRAM attached as cache to the ICAP. Then the API instructs ICAP to program the configuration memory. This process is slow and an improved API can increase the performance. [Möller et al., 2006].

- The stack of the processor should be set at least 4 KBytes. Otherwise the system hangs while reading from the compact flash.
- Reading from the compact flash is a very slow process. Fetching the configuration data to a volatile high speed memory after the system boots can improve the performance.
- Setting the PPC memory on the shared PLB through the PLB_BRAM controller might not be effective. The dedicated On Chip Memory (OCM) interface provides lower latency and faster access [Lund, 2004].

The above conclusions, in combination with the Table 5.2 of Section 5.3, can support the designer to make initial decisions on the system setup.

5.5 Development of a Cost Model

Initially the cost model on the reference system is formulated and then it is extended in order to suit more generic cases. This cost model was published in [Papadimitriou et al., 2011].

5.5.1 The Cost Model

The total reconfiguration time (RT) can be expressed by the sum of the times spent in each phase of the reconfiguration process:

$$RT = RT_{SM-PPC} + RT_{PPC-ICAP} + RT_{ICAP-CM} \tag{5.1}$$

The phases occur in a successive manner and are repeated until the entire bitstream is loaded to the configuration memory. Thus the aim is to find the aggregate time spent in each phase. All phases are controlled with software instructions residing in the processor's program memory, and there is no overlap between the phases as the instructions are executed

in-order. The processor calls the routines to transfer the configuration data from the storage means (compact flash in present case) to its local memory, then to the ICAP configuration cache, and then to the configuration memory. If for every phase, the number of executed processor calls, the amount of configuration data per call, and the time per call are known, it will be possible to compute the aggregate reconfiguration time per phase, and finally the total reconfiguration time. Thus the Equation (5.1) becomes

$$RT = SM_{calls} \times SM_{time} + IC_{calls} \times IC_{time} + CM_{calls} \times CM_{time} \tag{5.2}$$

To compute the number of processor calls to the compact flash, SM_{calls}, first it is necessary to know the amount of data (measured in bytes) to be transferred. If fs the frame size (measured in bytes) of the corresponding FPGA (e.g. 824 bytes for XC2VP30 and 164 bytes for all Virtex-4 and Virtex-5), then for n frames, the amount of reconfiguration bytes including the pad frame is $fs \times (n + 1)$. At this point it is recalled from Section 5.4 the way transactions from the compact flash to the processor are performed. The amount of configuration data transferred with one transaction depends on the size of the processor array, denoted as pa, allocated in the processor memory. Its size is defined in multiples of one sector, and thus transactions are conducted in multiples of one sector with the corresponding processor call. The number of calls for a given number of reconfiguration frames is

$$SM_{calls} = \frac{fs \times (n + 1)}{pa} \tag{5.3}$$

The time per processor call to the compact flash, SM_{time}, was measured with software time-stamps. It depends directly on the pa size. From Table 5.6 the time per 512 bytes has an average value of 1.45 ms. Thus the time per call is

$$SM_{time} = \frac{pa \times 1.45ms}{512} = \frac{pa}{353}ms \tag{5.4}$$

From Equations (5.3) and (5.4) the aggregate time of the 1st phase of reconfiguration process is

$$SM_{calls} \times SM_{time} = \frac{fs \times (n + 1)}{353}ms \tag{5.5}$$

The same concept is followed to compute the time spent in the processor calls for transferring the bitstream to the ICAP cache. The amount of configuration data transferred from the processor local memory to the ICAP cache with one transaction depends on the *pa* size, unless the latter is larger than the ICAP cache. In this case the size of the ICAP cache, denoted as ic_{size}, puts the upper limit on the amount of bytes transferred per transaction. The reason is that a new write transfer to the ICAP cache is accepted only if the transfer of the previous configuration data to the configuration memory has been completed. A quantity named *block* is introduced which is

$$block = min(pa, ic_{size})$$

The number of the corresponding processor calls to transfer the bitstream is

$$IC_{calls} = \frac{fs \times (n+1)}{block} \tag{5.6}$$

The time per processor call to the ICAP cache, IC_{time}, was measured with software timestamps. It depends directly on the *block* size. As obtained from Table 5.6 the time spent per 512 bytes has an average value of 0.42 ms. Also, it stops increasing after the amount of configuration data exceeds the size of ICAP cache as the latter cannot accept more than 2 KBytes per transaction. Thus the time per call is

$$IC_{time} = \frac{block \times 0.42ms}{512} = \frac{block}{1219}ms \tag{5.7}$$

From Equations (5.6) and (5.7) the aggregate time of the 2nd phase of reconfiguration process is

$$IC_{calls} \times IC_{time} = \frac{fs \times (n+1)}{1219}ms \tag{5.8}$$

Similarly, the time spent in the processor calls for loading the contents of the ICAP cache to the configuration memory is computed. The ic_{size} puts the upper limit on the amount of data written per transaction. The number of the corresponding processor calls

Table 5.7: Percentage of the time spent in each phase of reconfiguration and the corresponding measured throughput and theoretical bandwidth. The values concern the reference system.

	CF-PPC	PPC-ICAP	ICAP-CM
%RT	77.28%	22.38%	0.34 %
ARTP (calculated from Table 5.5)	0.33 MB/s	1.17 MB/s	77.32 MB/s
Theoretical Bandwidth (provided in data-sheets)	64 MB/s	95.3 MB/s	95.3 MB/s

to transfer the bitstream is

$$CM_{calls} = \frac{fs \times (n+1)}{ic_{size}} \tag{5.9}$$

The time per processor call to the configuration memory, CM_{time}, was measured with software time-stamps. As shown in Table 5.6 it is steady due to the fixed volume transfers from the ICAP cache to the configuration memory, which directly depends on the ic_{size}. In Virtex-II Pro the ICAP cache size is limited to one BRAM, thus $ic_{size} = 2048\ Bytes$. For a different ic_{size}

$$CM_{time} = \frac{ic_{size} \times 0.02526ms}{2048} = \frac{ic_{size}}{81077}ms \tag{5.10}$$

From Equations (5.9) and (5.10) the aggregate time of the 3rd phase of reconfiguration process is

$$CM_{calls} \times CM_{time} = \frac{fs \times (n+1)}{81077}ms \tag{5.11}$$

Therefore from the Equations (5.5), (5.8), (5.11), the Equation (5.2) becomes

$$\begin{aligned} RT &= fs \times (n+1) \times (\frac{1}{353} + \frac{1}{1219} + \frac{1}{81077})\ ms \\ &= fs \times (n+1) \times 3.66 \times 10^{-3}\ ms \end{aligned} \tag{5.12}$$

where fs is measured in bytes. Equation (5.12) holds for a PR system where the bitstreams are loaded directly from a compact flash under the control of a processor. It is observed that the variables pa and ic_{size} do not affect the reconfiguration time.

5.5.2 Model Extension

Using the above Equations, the percentage of the reconfiguration time (RT) spent in each phase is derived, which is shown in the first line of Table 5.7. Also, based upon the values of the Table 5.5 and from the fraction of the amount of bytes transferred over the transfer time the maximum achievable throughput (ARTP) of each phase is calculated directly, which is shown in the second line of Table 5.7. The throughput refers to the number of total bytes - including the overhead - successfully delivered per second. The values in the Table illustrate the magnitude of the difference between the throughput and the theoretical bandwidth, shown in the second and the third line of the Table 5.7 respectively. The theoretical bandwidth is determined by the slowest physical component involved in the corresponding reconfiguration phase, and it is obtained from the data-sheets; for the first phase this is the compact flash system, for the second phase it is the access to the ICAP cache, and for the third phase it is the access to the configuration memory through the ICAP.

Presently, the slowest phase of the reconfiguration process is the SM-PPC, which consumes 77.28% (see Table 5.7) of the reconfiguration process. If instead of the compact flash a faster type of external memory is used the system performance would increase. To find this speedup Amdahl's law is applied in a straightforward manner. The law is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S. The total speedup of applying the improvement is

$$Speedup_{total} = \frac{Time_{old}}{Time_{new}} = \frac{1}{(1-P) + \frac{P}{S}} \tag{5.13}$$

The problem is adjusted to the present case concerned with *the reconfiguration speedup achievable from an improvement to the reconfiguration time that affects a proportion P (which corresponds to the SM-PPC phase) of that reconfiguration time where the improve-*

ment has a speedup of S_{SM-PPC}, which results in

$$\frac{RT_{CF}}{RT_{SM}} = \frac{1}{(1-P)+\frac{P}{S_{SM-PPC}}} \Leftrightarrow RT_{SM} = (1-P+\frac{P}{S_{SM-PPC}}) \times RT_{CF} \tag{5.14}$$

where RT_{CF} and RT_{SM} are the reconfiguration times for the reference system with the compact flash and the optimized system with the faster external storage respectively. For $P = 77.28\%$ and by using the Equation (5.12) to calculate the RT_{CF}, the Equation (5.14) becomes

$$RT_{SM} = fs \times (n+1) \times (0.83 + \frac{2.83}{S_{SM-PPC}}) \times 10^{-3}\ ms \tag{5.15}$$

Equation (5.15) determines the performance margins when the improvement is applied on the first phase of the reconfiguration process. It is more generic than Equation (5.12) in that it calculates the reconfiguration time not only for compact flash but for other storage means such as a DDR SDRAM or a ZBT SRAM. If an identical compact flash to the one of the reference system setup is used to load directly the bitstream, the speedup factor would be $S_{SM-PPC} = 1$. In case a different storage is used, a proportional analysis regarding the performance gained over the compact flash is needed, in order to quantify the S_{SM-PPC}. This analysis along with the usage and effectiveness of the formula is described in Section 5.6.

Equation (5.15) can be used to calculate the reconfiguration throughput of useful frames, i.e. without the pad frame. The bitstream size that corresponds to the useful configuration is $BS = fs \times n$. Thus

$$useful\ ARTP_{SM} = \frac{BS}{RT_{SM}} = \frac{n}{n+1} \times \frac{10^6}{0.83 + \frac{2.83}{S_{SM-PPC}}}\ Bytes/s \tag{5.16}$$

For the compact flash where $S_{SM-PPC} = 1$, Equation (5.16) becomes

$$useful\ ARTP_{CF} = \frac{n}{n+1} \times 0.26\ MBytes/s \tag{5.17}$$

Similarly, the reconfiguration throughput of each phase can be computed from the corre-

sponding aggregate reconfiguration time provided in Equations (5.5),(5.8) and (5.11).

5.5.3 Discussion

Table 5.7 shows clearly that for each phase the throughput vs. the theoretical bandwidth is degraded considerably. This is due to the distinct components involved in the reconfiguration process. For example, in the PPC-ICAP phase, the way the PPC sends the configuration data to the ICAP cache has a negative impact on the reconfiguration time. The processor transfers the data from the PPC local memory to the ICAP cache in a word-by-word fashion, which is iterated until the ICAP cache is filled up. During experimentation it was found that to transfer one word 633 PPC cycles are required when the PPC runs at 300 MHz, which translates to 2.11 μs. This time was found to be fairly constant. The reason this large amount of cycles is needed is that the transfer from the PPC local memory to the ICAP is bus-based plus the function calls residing in the processor local memory are dictated by an Application Program Interface (API). Moreover the buses operate at 1/3 of the PPC frequency, i.e. 100 MHz, which accounts for the large number of clock cycles in the processor. Also, the time spent for the processor call in the ICAP-CM phase was measured. For the present case where the ICAP cache was 2048 Bytes, 7541 cycles of the PPC running at 300 MHz were needed, which translates to 25.26 μs (see Table 5.6). This measurement was repeated and was found to be constant as well. The above delays add up to the total reconfiguration overhead every time a processor function is called, as the reconfiguration process is not pipelined, but it is controlled with sequential functions. Moreover, the throughput is bounded by the slowest CF-PPC phase, during which several components shown in Figure 5.3 interact, such as the compact flash, the external System ACE controller, the OPB sysace module, the PPC memory and its memory controller, the bus bridge, and the library maintained in the PPC memory to supervise the transfers.

It is obtained from Equation (5.15) that the parameters pa and ic_{size} are irrelevant. In particular, the user needs to be aware only of the frame size, fs, and the number of frames, n, to calculate the expected reconfiguration time. The former is available in the data-sheets. The quantity $fs \times (n+1)$ can be replaced with the number of reconfiguration bytes, rb, of

the partial bitstream

$$rb = fs \times (n+1) = (fs \times n) + fs = partial\ bitstream\ size + fs \tag{5.18}$$

where the *partial bitstream size* is extracted after running the place & route tool, or can be estimated from the synthesis results by using the percentage of occupied area and the size of configuration memory of the corresponding FPGA. Finally, the S_{SM-PPC} factor is calculated with a proportional analysis described in the next Section.

5.6 Verification and Usage

First the cost model is verified on the base compact flash system, and then it is used to investigate the performance of systems where the bitstreams are loaded from a DDR memory. The verification process was done through non-trivial "real-world" examples with actual measurements.

5.6.1 Verification

A platform for cryptography applications developed on a Virtex-II Pro FPGA with the Xilinx PR design flow was used. Two cryptography modules, the Advanced Encryption Standard (AES) and the Device Encryption Standard (DES/3DES), are switched on-the-fly without disturbing the remaining logic. The setup is the same with the one presented in Figure 5.3 except that the PPC operates at 100 MHz instead of 300 MHz. The PPC is the reconfiguration controller and the bitstreams are loaded directly from the compact flash. Table 5.8 has the details for the two designs as well as for the blanking bitstream which is used for power-saving when none of the cryptography modules is needed. The first two columns show the type of the partially reconfigurable modules and the measured bitstream size. The third and the fourth columns have the reconfiguration time as calculated with Equation (5.15) and as measured with the software time-stamps, respectively. The fifth column shows the absolute difference between the calculated and the measured times. The sixth column has the percentage error of the cost model. This metric was used to express

Table 5.8: Comparison between the calculated and measured reconfiguration times for a partially reconfigurable cryptography system.

Partial bitstream	BS (Bytes)	Calculated RT (ms)	RT (ms)	Diff (ms)	%Error
AES	749,737	2,748.19	3,732.16	983.97	26.36%
DES/3DES	744,037	2,727.30	3,649.75	922.45	25.27%
Blanking	673,895	2,470.19	3,359.19	889.00	26.46%

the magnitude of the deviation between the theoretical value (also called approximate) and the measured value (also called actual), given by the following formula

$$\%error = \frac{|approximate\ value - actual\ value|}{actual\ value} \tag{5.19}$$

It is observed that there is a discrepancy between the cost model results and the measured times. However, the percentage error is fairly constant for all bitstreams. A factor that contributes to this error is the lower frequency of the processor in the cryptography system (100 MHz vs. 300 MHz in the reference system), which accounts for the larger reconfiguration time. As an extension the cost model could include the processor clock as a factor. Nevertheless, in its present form it estimates the expected reconfiguration time with a good accuracy, far better than the one and two orders of magnitude reported in other works [Griese et al., 2004, Galindo et al., 2008].

5.6.2 Reaping the Benefits of the Cost Model

The cost model was employed to evaluate PR systems with different setups. In particular, in the systems that are studied after boot-up the partial bitstreams are placed in a DDR; this allows for faster reconfiguration. A DDR controller, either as a discrete component or implemented as an IP core within the FPGA, is required. In both cases the DDR controller interfaces with the internal system through the on-chip buses, i.e. OPB or PLB. The reconfiguration is controlled by the PPC running at 300 MHz. The OPB is 32-bit and the PLB is 64-bit, and with a typical clock of 100 MHz they sustain a maximum throughput of 400 MB/s and 800 MB/s respectively. The reconfiguration time and throughput are calculated for different setups using the Equations (5.15) and (5.16). The speedup factor

S_{SM-PPC} affects the phase of reconfiguration process during which the bitstream is loaded from the external storage means. It is calculated as a fraction of the available bandwidth (BW) - which is dictated by the slowest component amongst the on-chip bus and the selected DDR - over the bandwidth of the compact flash; this is expressed with Equation (5.20). In the reference system that was used to develop the cost model, the theoretical bandwidth of the compact flash was 64 MB/s.

$$S_{SM-PPC} = \begin{cases} \dfrac{DDR\ BW}{CF\ BW} & \text{if } DDR\ BW \leq on-chip\ Bus\ BW \\ \dfrac{on-chip\ Bus\ BW}{CF\ BW} & \text{if } DDR\ BW > on-chip\ Bus\ BW \end{cases} \tag{5.20}$$

Table 5.9 has the expected reconfiguration times for a partial bitstream of 80 KBytes (the size is indicative; a different size could be chosen) and the corresponding throughput for various DDR memories when the DDR controller is attached on the OPB or the PLB bus. These results are drawn in Figure 5.4. It is observed that for the fastest DDR memories, the DDR controller on the PLB offers higher performance as compared to the OPB. Also, in both cases there is a point beyond which the transfer from the external storage is no longer the bottleneck of the system. Therefore, even though the available bandwidth of the external storage increases considerably, the benefit to the reconfiguration time is negligible.

The effectiveness of the proposed approach is explored by evaluating systems presented in previous publications by other researchers and comparing the results with the measured times contained in those publications. In particular, based on the information included in published works in which the embedded processor acts as the reconfiguration controller, the expected reconfiguration time is calculated using the cost model. Then, the results are compared with the measured times of Table 5.1. These data and their comparison are consolidated in Table 5.10. In [Liu et al., 2009] the PPC running at 300 MHz is the reconfiguration controller. A DDR controller is implemented as an IP core attached on the PLB bus, and has 32-bit interface and 100 MHz clock, which results in a theoretical bandwidth of 800 MB/s. According to Equation (5.20) this offers a speedup of 12.5 over the reference system. Equation (5.15) produces a reconfiguration time of 82.3 ms, denoted

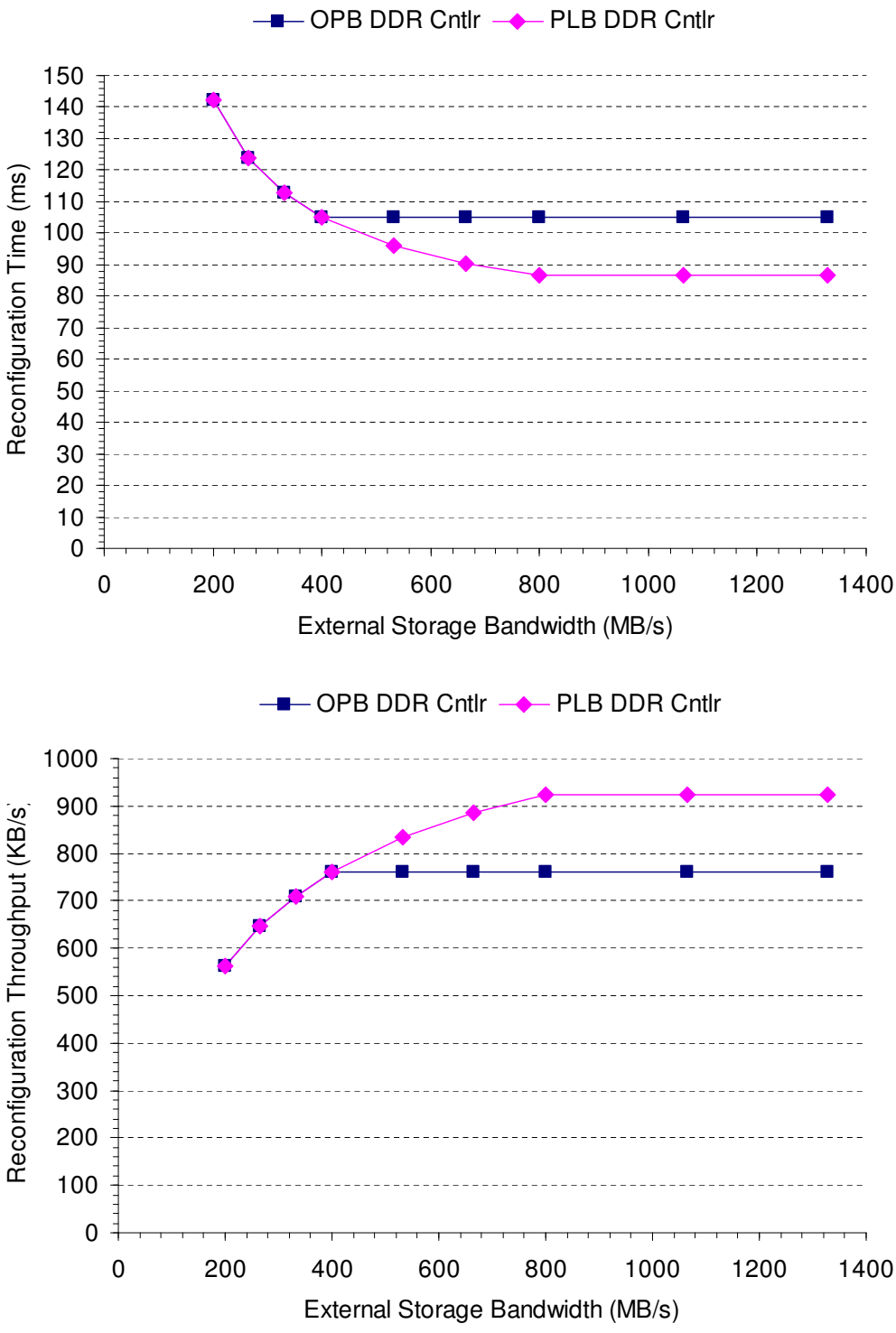


Figure 5.4: Reconfiguration time for a partial bitstream of 80 KBytes and the corresponding throughput for different DDR memory bandwidths. The data are taken from Table 5.9.

Table 5.9: Reconfiguration time (RT) and throughput (ARTP) for setups with various DDR memories. The values are calculated with the cost model for a partial bitstream of 80 KBytes. They concern a system, where the DDR controller is implemented as an IP core attached on the OPB (400 MB/s) or the PLB (800 MB/s) bus.

Storage	Clock(MHz)/ Width(bits)	Bandwidth (MB/s)	S_{SM-PPC}		RT_{SM} (ms)		$ARTP_{SM}$ (KB/s)	
			OPB	PLB	OPB	PLB	OPB	PLB
CF	32/16	64	1.00	1.00	299.83	299.83	266.82	266.82
DDR	100/8	200	3.13	3.13	142.18	142.18	562.67	562.67
	133/8	266	4.16	4.16	123.77	123.77	646.34	646.34
	166/8	332	5.19	5.19	112.68	112.68	709.95	709.95
	100/16	400	6.25	6.25	105.09	105.09	761.27	761.27
	133/16	532	6.25	8.31	105.09	95.88	761.27	834.35
	166/16	664	6.25	10.38	105.09	90.34	761.27	885.55
	100/32	800	6.25	12.50	105.09	86.54	761.27	924.42
	133/32	1064	6.25	12.50	105.09	86.54	761.27	924.42
	166/32	1328	6.25	12.50	105.09	86.54	761.27	924.42

Table 5.10: Comparison between the calculated and published reconfiguration times for different setups of partially reconfigurable systems using the processor to control reconfiguration (Calculated RT was extracted using the cost model, while Published RT is reported in the corresponding reference paper).

Reference paper	Storage	BS(KB)	Calculated RT(ms)	Published RT(ms)	%Error
[Liu et al., 2009]	DDR2@800MB/s	79.9	82.3	135.6	39.3%
[Liu et al., 2009]	DDR2@800MB/s	75.9	5.0	7.8	35.9%
[Claus et al., 2007]	DDR@400MB/s	90.3	7.2	19.39	63.1%
[Claus et al., 2008]	DDR@400MB/s	70.5	5.6	15.13	62.9%
[Papadimitriou et al., 2010]	CF@64MB/s	14.6	57.8	101.1	42.8%

in the first row of Table 5.10, which deviates from the published measured time [Liu et al., 2009] by 39.3%. In that work it was explicitly reported that the ICache and DCache of the PPC were disabled, which holds for the reference system too. In the second system of Table 5.10, the ICache and DCache of the processor were enabled. According to [Liu et al., 2009] the activation of the processor caches offers an overall enhancement of 16.6 times in the reconfiguration time. Using the cost model and by dividing the result with 16.6 the time reported in the second row of the Table is obtained, equal to 5 ms, which deviates from the published measured time by 35.9%. The values for all systems of Table 5.10 are extracted in the same way.

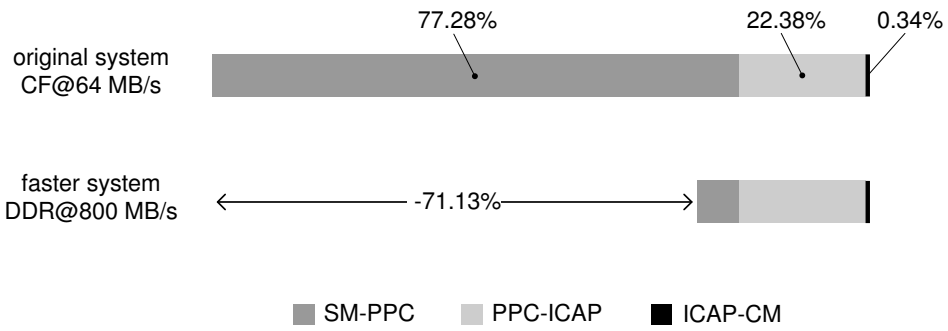


Figure 5.5: The reconfiguration operation has three independent phases. In the original system, the SM-PPC phase (dark-gray part) takes 77.28% of the total reconfiguration operation (see Table 5.7). Making this part 12.5 times faster and leaving intact the rest of the operation (light-gray and black parts) in the faster system reduces the total reconfiguration time by 71.13%.

The present analysis allows for the examination of the enhancement margins when using the DDR memory, which is illustrated in Figure 5.5. The speedup of $S_{SM-PPC} = 12.5$ concerns the SM-PPC phase only, which takes the 77.28% of the reconfiguration time in the original system (see Table 5.7). This offers a total speedup of 3.46 times (the speedup is calculated using Equation (5.13) for $P = 77.28\%$ and $S = 12.5$). It is obvious that the total improvement from the increase of the external storage bandwidth becomes saturated, and the transfer from the PPC to the ICAP bounds now the reconfiguration performance.

5.6.3 Strengths, Weaknesses and Potential Extensions

Present work is novel in that except for the verification of the cost model, it examines its effectiveness on systems presented in previous works. The cost model provides results with a percentage error ranging from 36% to 63% when evaluated upon published measurements. This error is small considering that other works relying on the throughput of the configuration port show a discrepancy of two orders of magnitude between the theoretical and the measured values [Griese et al., 2004, Galindo et al., 2008]. Other theoretical models based on the characteristics of the configuration port, e.g. the ICAP's "BUSY" signal, cannot be applied to other FPGA families except for the Virtex-II/Pro, and they focus on custom reconfiguration controllers targeting ICAP full utilization [Claus et al., 2008]. Present approach does not rely on the throughput of the configuration port, nor on characteristics

that vary with the FPGA technology, e.g. behavior of "BUSY" signal differs in Virtex-II, Virtex-4 and Virtex-5 FPGAs.

The proposed cost model can be augmented with more factors in the same way the simple model for the compact flash on the reference system was extended to support other types of external memories. Specifically, the second part of Section 5.5 was devoted to extending the model with the optimization factor S_{SM-PPC}. Following the same concept, Equation (5.15) can be enriched with optimization factors for the PPC-ICAP and the ICAP-CM phases (namely $S_{PPC-ICAP}$ and $S_{ICAP-CM}$ respectively). To do this further experimentation with a different setup is needed. In particular, Figure 5.4 shows that beyond a specific point the larger bandwidth of the external memory does not necessarily result in a higher throughput. Instead, the PPC-ICAP phase becomes the bottleneck as illustrated in Figure 5.5. Therefore, other alternatives can be explored such as the usage of the On Chip Memory (OCM) instead of the PLB memory, which is likely to provide lower latency communication. Furthermore, the cost model can be extended with a factor that relates to the impact of changing the processor clock. Experimentation will quantify this as well to reduce even more the uncertainty for the expected reconfiguration time.

The present approach considers systems in which the reconfiguration is controlled by the processor. Therefore, it serves users who desire to pre-evaluate their system without putting the effort to implement a custom reconfiguration controller. The values that were used to develop the cost model were obtained with real measurements discussed in the previous Chapter. Even though they are of small scale they were proven adequate to foresee well the reconfiguration performance. After gaining insights from the reference system, a model was developed without delving into details such as the latency of the distinct components involved in the reconfiguration process. The only factors that someone should consider of is whether the processor is the reconfiguration controller and its caches are enabled, and the type of external storage the bitstreams are loaded from. In particular, for a given system setup, simple characteristics gathered from data-sheets such as the available bandwidth, the frame size, and the bitstream size are enough to predict the reconfiguration time.

5.7 Discussion Summary

This Chapter delivers an up-to-date survey and an exploration of parameters affecting reconfiguration performance. The system of Chapter 4 is analyzed to identify the features that increase reconfiguration speed. Using this system as reference a simple cost model for the early prediction of reconfiguration time is developed. The model is verified, then used to evaluate several "real-world" systems, and the results are compared with published measurements. Also, the study quantifies the improvement of reconfiguration performance given the speedup of data transfer offered by the DDR memory over the compact flash. Finally, the strengths and weaknesses of this approach are discussed.

It is concluded that experimental evaluation is necessary for delving into the details of system-level operation. With data gathered from experiments a method and a cost model were devised currently applied in a category of real PR systems, which can be enhanced with more parameters to broaden its range of applicability.

Nowadays FPGA platforms are delivered as end-products and the benefits of partial reconfiguration technology are still being explored in numerous application domains. With the proposed model an opportunity arises for people not involved with partial reconfiguration yet but are interested to study whether their applications benefit from this technology. It can assist researchers from different domains such as telecommunications, cryptography and networking to estimate the reconfiguration overhead without entering the complex and rigid PR design flow, and also to make effective choices of the system setup. Hence, based on specific constraints they can decide if it is worthwhile to proceed with designing an application with PR technology by either assigning this task to experts having the know-how of PR design flow, or putting this effort on their own.

Chapter 6

HMR: A Novel Case Study

The application domains that will eventually benefit from dynamic reconfiguration technology are still under research. In fact, although a large body of research on applications implemented in hardware using PR technology exists none of them has been widely adopted by the industry. An interesting domain which has proven to avail from this technology concerns error recovery. Within this domain the issues raised in FPGAs need to be addressed in a different manner as compared with other types of integrated circuits. In particular, to cope with errors occurring in the FPGA circuit during operation might require reprogramming of the configuration memory.

This Chapter introduces a Hybrid Modular Redundant (HMR) scheme for handling faults in non time-critical FPGA systems. It targets SRAM-based FPGAs operating with real-time input data, in which processing should be sustained without losses and resource savings in silicon is of major concern. This scheme relies on the duplication of the hardware core and a software counterpart of the application running in a hardcore processor. Once a mismatch between the outputs of the two cores is found, the processor temporarily undertakes the execution of the application. A hardcore processor has smaller cross section and it is more immune to upsets as opposed to the configuration memory, while it occupies less area than reconfigurable resources. Experimental elaboration shows that HMR can achieve significant area reduction and faster operation vs. the dominant Triple Modular Redundancy (TMR) solution for realistic large designs at the cost of a reduction in the processing rate of the input.

6.1 Introduction

SRAM-based FPGAs are an appealing solution to accelerate applications and re-customize them with upgrades in the field. However, in radiation-exposed environments their functionality can be disrupted leading to erroneous operation and results. The most common faults induced in such environments are the Single Event Upsets (SEUs). In SRAM-based FPGAs, an SEU occurs when the noise caused by radiation sources exceeds the critical charge of a memory cell forcing it to alter its value a.k.a bit flip. SEUs are soft errors meaning that they don't damage permanently the device. They can cause bit flips in the configuration memory inducing undesirable changes in logic and routing, or affecting directly the contents in memory resources like the user SRAM and flip-flops [Quinn et al., 2008]. In extreme cases over two hundred SEUs during a day have been reported [Xilinx Inc., 2005a]. Except for the SEUs other type of single events can occur. A single event may cause a voltage pulse, i.e. glitch, to propagate through the circuit, which is referred to as a Single Event Transient (SET). Since the transient is not an actual change of the state in the way the SEU impacts a memory cell, SETs are differentiated from SEUs. However, if a SET propagates and results in an incorrect value that is latched in a sequential logic unit, i.e. flip-flops, it is then considered an SEU. In addition to the errors experienced in radiation-induced environments, as the technology process shrinks, the probability to encounter such events in SRAM-based FPGAs increases [Stott et al., 2008]. This has made necessary the development of techniques for the recovery of such systems upon fault occurrences. However, no single method suits perfectly all cases and its selection depends heavily on the needs of the application at hand [Cheatham et al., 2006].

Currently, the field is dominated by the Triple Modular Redundancy (TMR) scheme. Present work aims at forming a scheme having nearly the same performance and robustness with TMR but with smaller area overhead and without sacrificing reliability. It relies on partial reconfiguration (PR), combined with the use of embedded hardcore processors to control the detection, diagnosis and recovery from faults. A hardcore processor doesn't suffer from SEUs at the same rate as the configuration logic, and it is more reliable under harsh conditions [Petrick et al., 2005]. The real-time domain is targeted in which the FPGA

needs to react effectively upon a fault occurrence in order to sustain the processing of continuous input data. Applications of particular concern include gathering meteorological data, aviation, object recognition, and in-orbit missions [Caffrey et al., 2009] where devices are exposed to high levels of radiation. Such phenomena can also appear at the ground level [Bolchini et al., 2007].

Present approach pursues a solution combining continuous check for error detection, immediate action for error recovery, and low area overhead, while maintaining valid output of the system by processing correctly the input data. During normal operation two replicas of the hardware core are processing the input data. Once upon a mismatch between the two replicas is detected, the processing is undertaken by the processor, which typically runs with lower performance than hardware. If the software processing rate is slower than the input rate, a FIFO queue at the input is needed to avoid losing data. The contributions of the present work are:

- formulation of a hybrid scheme combining hardware and software for handling faults in non time-critical systems.
- examination of the overhead added by the distinct operations carried out in two different designs implemented according to the proposed scheme.
- performance study of each of the two designs in terms of the input rate that they can sustain without data loss for a given FIFO size.
- qualitative and quantitative comparison between the HMR and TMR schemes.

As HMR relies on the use of hardcore processors, FPGAs with embedded processors are considered. The radiation hardened family of Xilinx Virtex-II/-4/-5 series (XQR FPGAs) are equipped with PowerPC processors, a fact that accounts for the present approach. A proof-of-concept system was built as part of a final-year project in the MHL laboratory [Ilias, 2009]. A commercial platform equipped with a Xilinx Virtex-II Pro FPGA which immerses two hardcores PowerPCs was used. Two designs of the HMR were implemented and the system along with preliminary results were published in [Ilias et al., 2010]. The present dissertation contributed with the concept and the system architecture. As the project

evolved the limitations were taken into consideration and the FIFO was added as an integral part of the system architecture. This Chapter performs a comparison of HMR with TMR and discusses its benefits and its drawbacks.

The Chapter is structured as follows. Section 6.2 discusses state of the art on fault detection and recovery. Section 6.3 presents the generic architecture of the HMR along with an initial evaluation. Section 6.4 has the implementation of two alternative designs of HMR in a Xilinx FPGA. Section 6.5 has the experimental results. In Section 6.6 HMR is compared against the TMR solution. Finally, Section 6.7 discusses the advantages of the proposed approach.

6.2 Related Work and Motivation

An SEU is a soft error caused by radiation sources such as alpha particles or neutrons. If these radiation noises exceed the critical charge of a memory cell or a flip-flop, the noise results in an SEU. SEUs can occur in different environments. Terrestrial SEUs arise due to cosmic particles colliding with atoms in the atmosphere, creating cascades or showers of neutrons and protons, which in turn may interact with electronics. At deep sub-micron geometries, this can affect semiconductor devices in the atmosphere at the ground level. The space environment is characterized by high energy ionizing particles referred to as cosmic rays. This translates to a radiation environment very rich in electrons, protons and heavy ions that can affect greatly the semiconductor devices. Hence, either near the ground level or in space it is necessary to cope with SEUs to increase the safety of the operation of electronic systems. The selection of the reliability technique depends heavily on the environment in which the system will be placed and the mission requirements. Parameters such as performance, cost and area have an important role in selecting a technique, and they should be balanced in order to select the must suitable one.

Prior deploying an electronic device in an environment it is tested for the SEU response against radiation resources. This response is called bit cross section, which is a measure of the per-bit sensitive area to the particular radiation source and has the units of cm^2/bit [Quinn et al., 2008, Quinn et al., 2009]. This metric is taken into account to decide the

safety level to be applied.

Except for the development of methods to handle faults, special SRAM-based FPGAs are fabricated, which are more immune to radiation-exposed environments than commercial FPGAs. Examples of using this kind of FPGAs along with proper methods for reliable operation in real-world applications include the motor and landing control of the rovers in the mission to Mars [Ratter, 2004] and applications such as software defined radio (SDR), demodulators, decoders, and FFTs in satellites [Caffrey et al., 2009].

Currently, the field is dominated by the Triple Modular Redundancy (TMR) scheme, which falls into the category of logic redundancy. The concept is to use three replicas of the system and a voter to select and identify the correct result amongst the three with respect to the majority vote. Its effectiveness has been demonstrated in numerous works [Bolchini et al., 2007, Caffrey et al., 2009]. When combined with partial reconfiguration technology of FPGAs it is claimed to be able to handle any error while the system operates [CarMichael et al., 2000, CarMichael, 2006]. The main drawback of TMR is the resource overhead and the degradation of speed performance due to its complexity. It is reported to have at least 3.2x resource utilization cost [Xilinx Inc., 2005b], while clock frequencies over 100 MHz in a Virtex-II FPGA are difficult to achieve [Bridgford et al., 2008]. This scenario is very optimistic as even for small circuits such as simple filters, resource utilization from 4.2x to 6.3x has been reported [Bolchini et al., 2007]. Recent works propose modifications to existing SRAM-based FPGA architectures by applying TMR locally to the Look-Up-Tables in order to support fine grain redundancy at a much less area cost [Kyriakoulakos and Pnevmatikatos, 2009]. Another redundancy method with smaller overhead but also smaller fault coverage than the TMR is the Duplication With Compare (DWC), which uses two replicas and a comparator to check their outputs [Johnson et al., 2008].

The most promising methods for handling faults exploit heavily the ability of SRAM-based FPGAs to be reconfigured [Stott et al., 2008]. In particular, correction of soft faults in SRAM-based FPGAs is conducted by reconfiguring their memory. This way, the bits affected from a fault presence are overwritten by reprogramming the FPGA with the initial configuration. This process is called "scrubbing" and when applied periodically it mitigates the device from the accumulation of SEU errors. However, this method incurs a considerable

time overhead to the FPGA operation. With the advent of partially reconfigurable FPGAs, reprogramming of addressable segments of the device, i.e. configuration frames, became feasible. When a fault is detected and the affected area is identified, correction can be applied by reconfiguring the corresponding frames only. This method has been demonstrated effectively in several projects [Bolchini et al., 2007, Gokhale et al.,]. Partial reconfiguration has also been employed to tolerate faults due to defects of the chip. Specifically, upon the occurrence of such faults - also called hard faults - alternative configurations can be loaded that will not use the defected resources. If the FPGA is not fully utilized, unused resources preassigned as spares can accommodate portions of the design affected from hard faults [Emmert and Bhatia, 1997, Huang and McCluskey, 2001].

Present study differs from others in the sense that along with the resource requirements it quantifies the FIFO depth needed to continue processing the input data without losses for various data rates.

As the concept relies on using hardcore processor(s) present study targets FPGAs with embedded processors. The radiation tolerant family of Virtex-II/-4/-5 series (XQR FPGAs) are equipped with PowerPC processors, a fact that accounts for present approach. The proof-of-concept system uses the commercial Xilinx Virtex-II Pro FPGA that incorporates two hardcore PowerPCs.

The proposed approach lies between the DWC and TMR as two hardware cores coexist with a third model of the application running in software. Upon a failure real-time processing is undertaken by the processor, which executes with less performance than the hardware. Thus, it is explored whether traditional sequential processors suffice in assisting application running in SRAM-based reconfigurable FPGAs.

6.3 A Generic HMR Architecture

First, some terms used throughput the Chapter need to be clarified. The operation of fault handling is distinguished in the detection phase, i.e. finds that an error occurred, the diagnosis phase, i.e. identifies the faulty part, and the correction or recovery phase, i.e. corrects the fault. Also, the term "restoration time" is coined here as the time elapsed from

a fault detection until the hardware core begins processing again the input data.

6.3.1 Architecture

Figure 6.1 draws the generic architecture of HMR and highlights its differences with the TMR. The grey components indicate the differences in the resource overhead between the two schemes. The method used for the detection of faults is the Duplication With Compare (DWC) [Johnson et al., 2008]. The outputs of the two hardware (HW) cores are compared using a XOR gate and once a disagreement is found successive steps for error recovery take place. Correction is conducted through partial reconfiguration. The scope of the software subsystem is threefold: checking the XOR gate for fault detection, execution of the software version - also called software model - of the application upon a failure, and reconfiguring the faulty part. The software model due to its low performance as compared to the hardware would degrade the processing rate. In particular, although the reconfigurable hardware can service a high data rate link due to its customized parallelization and pipelining, it is likely that the software processor would not be able to sustain it. Thus a FIFO is needed to hold temporarily the input data during fault handling; its depth is a matter of study in order to avoid losses.

For the implementation of HMR the replicas of the core are designed as partially reconfigurable modules (PRM). They are placed into predefined regions of the FPGA called partially reconfigurable regions (PRR). Upon a fault occurrence the bitstream of the corresponding PRM is loaded to reconfigure a PRR. During reconfiguration the PRR is isolated so as to leave intact the remaining part of the FPGA by controlling special communication primitives called bus macros - the I/Os of the PRR - placed at the edge of the boundaries of PRRs.

Two different designs can take effect according to the HMR. The architecture is the same for both designs. Their main difference lies in whether the two replicas are placed in the same PRR or in two separate PRRs. In the first case, the so called 1-PPR design, the two replicas are implemented as one PRM that is loaded into one PRR. In the second case, the so called 2-PRRs design, each replica is created as one PRM each of which is loaded into a separate PRR. Only the operation of fault detection is the same for both designs;

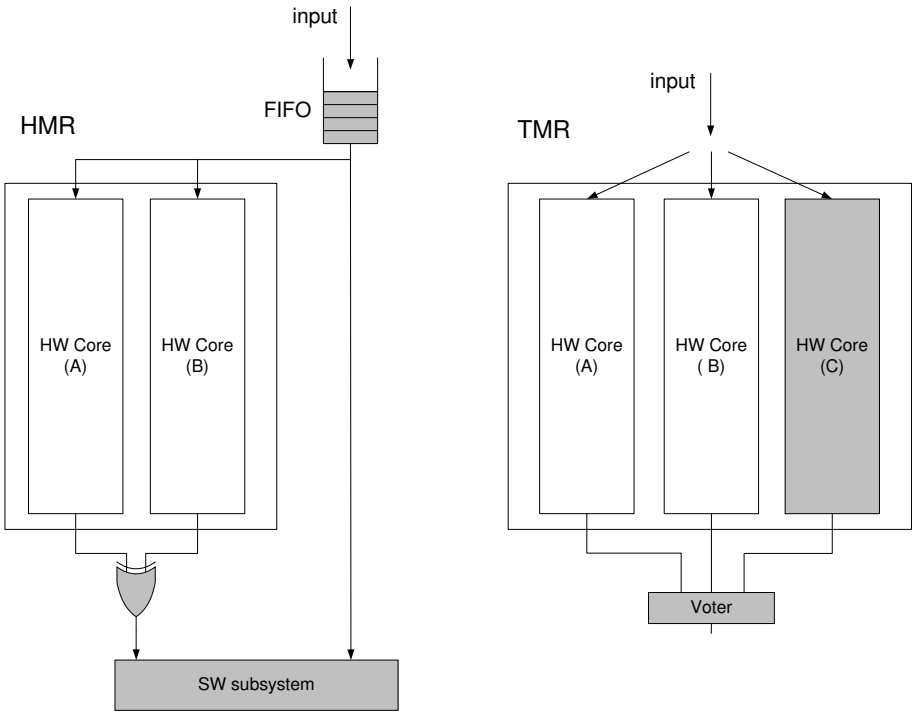


Figure 6.1: HMR vs. TMR resource overhead.

the other operations differ.

6.3.2 1-PRR design vs. 2-PRRs design

In both designs once a fault is detected the processing in the HW cores is halted. In the 1-PPR design no diagnosis is made and reconfiguration is triggered immediately. During that time the software undertakes the processing of the data. On the other hand, in the 2-PRRs design the software processes a specific amount of the next input data simultaneously with the two HW cores in order to identify the core that disagrees with the output of the SW model. Then, the software instructs the uncorrupted HW core to resume its operation and triggers the reconfiguration of the PRR carrying the corrupted HW core. Thus the main difference between the two designs is that the 1-PRR design lacks of the diagnosis phase, while the 2-PRRs design exploits time redundancy to diagnose the faulty part.

In the 1-PRR design the PRR is reconfigured in its entirety as imposed by the partial reconfiguration technology. During reconfiguration the PRR cannot operate and thus both HW cores are non operational. Therefore, identifying the corrupted HW core would be

a meaningless operation. On the other hand, in the 2-PRRs design the diagnosis of the corrupted HW core is meaningful as it identifies the PRR that will be reconfigured. Also, it identifies the uncorrupted HW core that resumes processing without waiting for the reconfiguration to complete.

The effectiveness of each design is determined by its "restoration time". The smaller this time is, the faster the system enables the hardware to undertake again the processing of the input and consequently the smaller the FIFO that is needed. Clearly, the 1-PRR design doesn't allow any HW core to process the input until the reconfiguration completes. Contrarily, the 2-PRRs design releases the uncorrupted HW core once diagnosis is made. A work published in [Ilias et al., 2010] presented that for a given FIFO size the 2-PRRs design can sustain a much higher data rate during fault handling than the 1 PRR.

6.3.3 Benefits and Weaknesses

The question is whether HMR can surpass TMR in terms of area overhead, complexity and performance. HMR can be justified if the overhead of the additional components in Figure 6.1 is smaller than the overhead added by TMR. The resources needed for the software model doesn't scale with the same rate hardware resources would do for large application designs. In particular, the requirements of the software for a large design would increase only with respect to the program memory. This is of low concern compared to the hardware counterpart that besides memory requires more logic and routing resources. Especially for a large design, triplication will result in a dense circuit and in clock degradation [Bridgford et al., 2008]. It is reasonable to anticipate that the benefits of the HMR over TMR will be exhibited for large HW cores. This will be possible if a generic software subsystem with relatively fixed amount of resources that won't increase with the application size and with a small FIFO can be built.

The HMR scheme can handle permanent as well as transient errors affecting the HW cores. In particular, the 2-PRRs design is capable of distinguishing the two error types using time redundancy: once a fault is detected the SW processes the next input data simultaneously with the HW cores; if all results match, the fault has disappeared, i.e. the error was transient, and thus the HW cores resume their operation, otherwise, the fault was

permanent and reconfiguration is triggered to correct the faulty core. Therefore, unless the SW model disagrees with one of the HW cores, reconfiguration is not triggered. It is worth noting that in the presence of both error types the "restoration time" is added as overhead to the total application execution.

One of the weaknesses of HMR is that until the system is restored the processing rate decreases. This disturbs the output, either by halting it, or lowering the rate the processed data occur at the output. Thus the applications that can benefit from the HMR scheme should be explored. The most promising candidates belong to the non time-critical domain that requires safety in data-delivery while resource savings is a major concern [Wang and Bolotin, 2004].

Finally, it should be mentioned that the HMR during fault handling doesn't perform any error checking. Therefore, it won't be able to detect and handle a new fault unless reconfiguration is finished and both HW cores are in-place and operational. The same holds for TMR as during correction of the faulty part the two uncorrupted cores will be able to detect a fault but not to identify the corrupted core. Thus the system will not continue working properly. Additional control logic could prevent such cases.

6.4 Implementation of HMR in a Virtex-II Pro

The guarded application was a 7^{th}-order FIR filter operating on 12-bit data. It was developed both in hardware and software. The hardware filter was designed in a pipelining fashion able to process one sample per cycle. In order to evaluate HMR for an as much as possible susceptible core, the hardware filter was built with configurable elements only, i.e. LUTs. The static design of the filter - also called base design - occupied 901 slices (6% of the FPGA's slices) and operated at 111 MHz, while its partially reconfigurable counterpart occupied 992 slices (7% of the FPGA's slices) and operated at 107 MHz.

Figure 6.2 depicts the implementation of the HMR in a Virtex-II Pro FPGA. Currently, the software subsystem uses both processors communicating through a shared memory implemented with BRAMs[1]. The PPC0 checks the XOR gate for fault detection, informs

[1]With slight modifications one processor only can be used thus eliminating the overhead of communication between the processors.

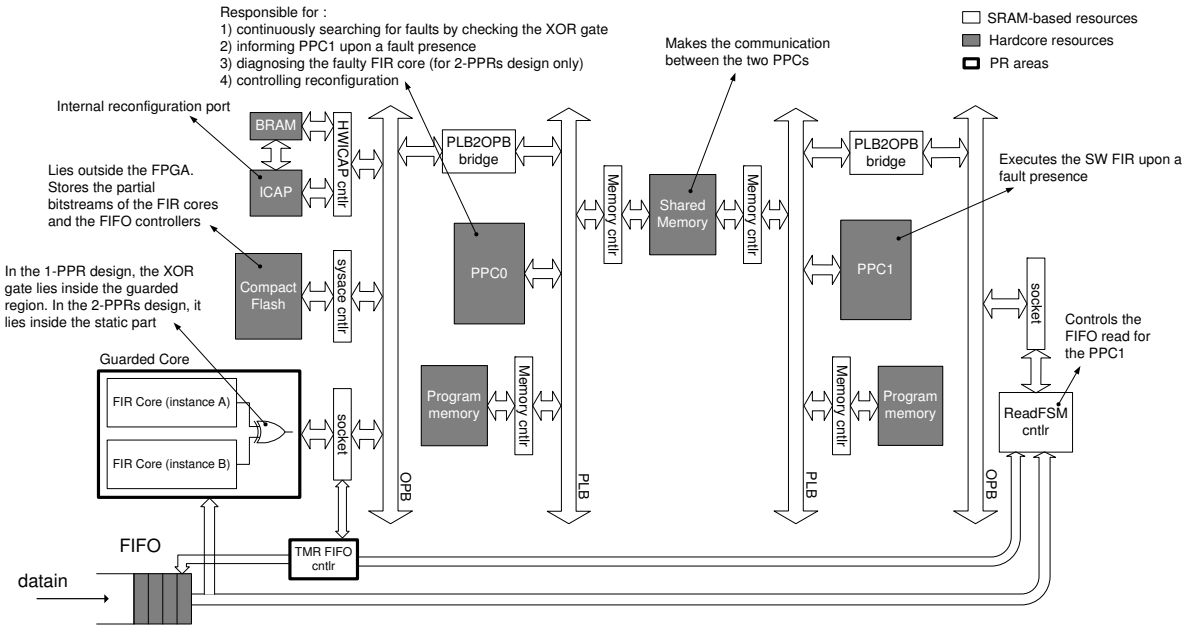


Figure 6.2: Block diagram of the HMR implemented in a Virtex-II Pro FPGA.

the processor PPC1 when a fault occurs, and reconfigures the faulty part. The main task of the PPC1 is the execution of the software filter upon a failure. The partial bitstreams are loaded by the PPC0 from an external compact flash memory. The FIFO is implemented with embedded BRAM blocks and its controller is implemented with the TMR technique. The FIFO is connected with the FIR cores and the PPC1. The ReadFSM controller is used for the synchronization of the FIFO reading from the PPC1. The guarded components implemented in partially reconfigurable regions are the FIR cores and the TMR FIFO controller. The FIFO subsystem constitutes an integral part of HMR scheme and thus it was deemed necessary to guard the FIFO controller. This will allow to perform a fair side-by-side comparison between the HMR and the TMR regarding the area overhead.

Both designs of the HMR scheme we implemented in order to evaluate their performance:

- *1-PRR design*: once a fault is detected, the PPC0 informs the PPC1 to start processing the input data from FIFO. Then, the PPC0 triggers immediately the reconfiguration of the PRR. The PPC1 continues the processing until it is notified from PPC0 that the reconfiguration has ended.

- *2-PRRs design*: once a fault is detected, the PPC0 informs the PPC1 that en error

has occurred and waits for its response. The PPC1 processes the next 8 input data (7^{th} order filter) simultaneously with the two FIR cores and it sends the result to the PPC0. The latter compares the result of the PPC1 with those of the FIR cores in order to identify the corrupted core. Then, the PPC0 triggers the reconfiguration of the PRR containing the corrupted FIR core, while the uncorrupted one resumes processing the input.

Figure 6.3 shows the floorplanning of the partially reconfigurable regions in the 2-PRRs design. It illustrates the resource requirements for the two FIR cores and the TMR FIFO controller. Obviously, the FIFO controller consumes much less resources as compared to the FIR core. The interesting point here is that even if a larger application core will be deployed such as a larger filter with more taps, the area occupied by the TMR FIFO controller would remain about the same. The floorplanning of the 1-PRR design is similar except that both FIR cores are placed in one large PRR.

6.4.1 Resource Utilization

Table 6.1 has the resource requirements of HMR derived from Figure 6.2. The first column has the component type and the second column reports whether it is SRAM-based (S), or Hard IP (H). Amongst the SRAM-based components only the HW cores and the FIFO controllers are guarded, while the others belong to the static part of the system. The fourth column of the Table 6.1 has the components needed for the alternative TMR solution, which is used for comparison purposes in Section 6.6.

Table 6.2 has the resource utilization of the 2-PRRs design of HMR. The BRAMs are used mainly for the software subsystem, i.e. the PPC program memories and the shared memory. This is illustrated in Table 6.3. Clearly, the 31.4% out of the 36% of the BRAMs is used for the software subsystem. Part of the remaining BRAMs, i.e. $36\% - 31.4\% = 4.6\%$, is used for the configuration cache of the ICAP (one BRAM only is used for the ICAP) and the FIFO (in the experimental setup a 1024×12 FIFO was used). The amount of BRAMs allocated for the software subsystem is the minimum that can be set as imposed by the vendor's tools. However, it is much larger than the actual memory needs and thus

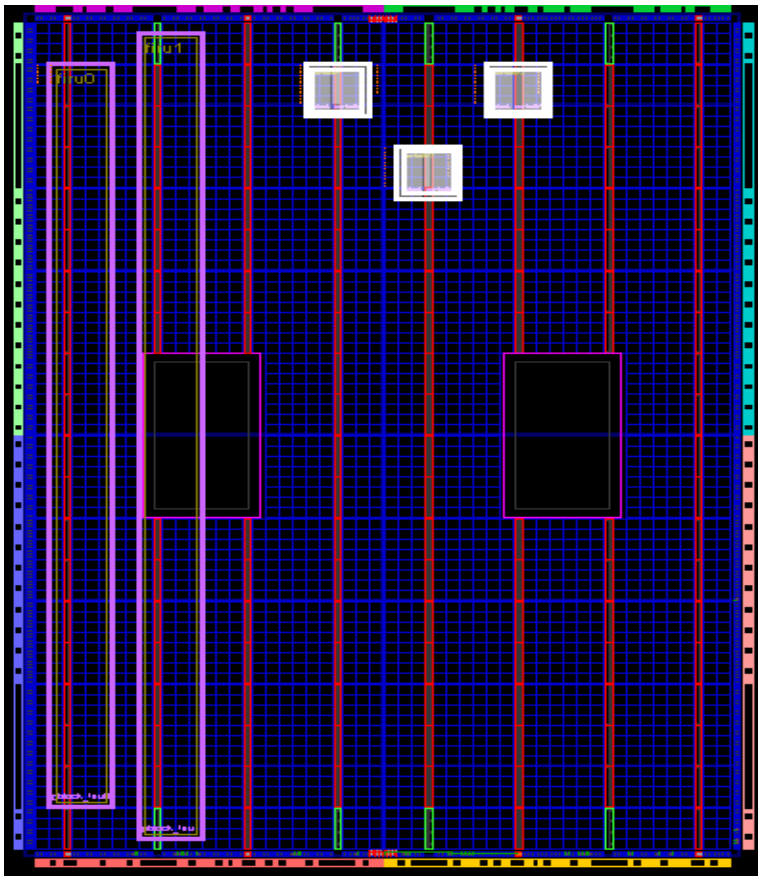


Figure 6.3: Floorplanning of the partially reconfigurable regions in the HMR when implemented with the 2-PRRs design. The two FIR cores stand on the left and the TMR FIFO controller on the top of the device.

no additional BRAMs would be needed to develop in software a larger application. The guarded area that hosts the FIR cores requires 15% of the FPGA slices. It is obtained that the amount of slices needed to duplicate and implement the partially reconfigurable version of the FIR core is 2.3x the base design (the results of the calculation $2144 \div 901$).

6.4.2 Benefits, Drawbacks and Extensions

HMR detects faults and recovers HW cores at real time. For a larger application it is expected that the static area will not increase. This is due to that the static part composes of the processor, the controllers and the program memories; these components won't get larger. But even if the area occupied by these components will increase it won't scale as much as a hardware core would do. Only the FIFO size relates to the "restoration time"

Table 6.1: Resource requirements for HMR and TMR.

Component	S/H	HMR	TMR
HW cores	S	2	3
majority voters	S	-	√
minority voters	S	-	√
feedback logic	S	-	√
bus macros	S	√	√
ICAP port	H	√	√
HWICAP controller	S	√	√
ICAP's BRAM	H	√	√
sysace controller	S	√	√
I/O interfaces	S	√	√
hardcore processor	H	2	1
program memory	H	2	1
shared memory	H	1	-
memory controllers	S	4	1
internal bus (OPB and PLB)	S	√	√
FIFO	H	√	-
TMR FIFO controller	S	√	-
voter for the TMR FIFO controller	S	√	-
ReadFSM controller	S	√	-
XOR gate	S	√	-

and the input data rate. The faster the "restoration time" and the smaller the input rate is, the smaller the FIFO that is needed.

In the present architecture reliability issues outside the guarded core haven't been taken into account. In particular, except for the application core and the FIFO controller components such as the memories, controllers, buses and interfaces of Table 6.1 aren't guarded. The same practice is followed in other projects as well [Bolchini et al., 2007]. To increase the safety at system level such concerns should be addressed. For instance, in order to guarantee the operation of memories Error Correction Codes (ECC) can be employed. Alternatively, the SRAM resources of the static part can be placed in a PRR region and checked periodically so as to "scrub" it when the output of the software subsystem disagrees with both HW cores' outputs.

Currently, the software subsystem of HMR is implemented with two hardcore processors. With slight modifications all tasks can be carried out by one processor only, which will

Table 6.2: Resource utilization for the static and the partially reconfigurable parts of the 2-PRRs design of HMR implemented in a Virtex-II Pro.

Resource type	Available	Static (%)	FIR cores (%)	TMR FIFO Cntlr (%)	Total (%)
PPCs	2	2 (100%)	-	-	2 (100%)
LUTs	27392	3654 (13%)	4288 (15%)	480 (2%)	8422 (30%)
Flip-Flops	27392	3488 (12%)	4288 (15%)	480 (2%)	8256 (29%)
Slices	13696	3181 (23%)	2144 (15%)	240 (2%)	5565 (40%)
BRAMs	136	50 (36%)	-	-	50 (36%)

Table 6.3: BRAM utilization for implementing the program memories and the shared memory of the PPC's subsystem. 136 BRAMs are available in the Virtex-II Pro FPGA, each of which equals 2.25 KBytes resulting in an overall of 306 KBytes available in the device.

Memory component	Allocated	Utilization%
PPC0 program memory	64 KB	21%
PPC1 program memory	16 KB	5.2%
Shared memory	16 KB	5.2%
Total	96 KB out of 306 KB	31.4% (<36%)

eliminate the communication overhead with the second processor. In particular, in the 2-PRRs design, after the PPC0 detects a fault it can itself execute the SW model in order to diagnose the corrupted HW core, and then reconfigure it while at the same time the uncorrupted HW core resumes processing.

6.5 Experimental Results

This part of the work explored the efficacy of the HMR scheme by conducting experiments on both the 1-PRR and the 2-PRRs designs. The purpose was to study the impact of the software operations on the restoration procedure. The duration of each operation and the restoration time in each design was measured. For testing purposes an artificial way of injecting faults was created. A new fault is injected only after the system has been fully recovered from a previous fault. This means that both HW cores are in place and operational and the DWC mechanism is able to detect a fault. It is controlled by an external push button that causes the DWC mechanism to experience faults. To emulate a streaming environment in which data are processed by the filters as they are arrived a simple data generator connected directly with the FIFO was instantiated. It generates

Table 6.4: Operations affecting the restoration time in the HMR when implemented with the two different designs. The time duration of an operation is shown and summed only if it is part of the restoration procedure.

Operation	1-PRR		2-PRRs	
	What/How	Duration	What/How	Duration
Reconfiguration	both filters	$907.64ms$	one filter	0
Processors' communication	2 messages	$3.36\mu s$	1 message	$1.68\mu s$
FIR initialization	7 samples	$19.046\mu s$	7 samples	0
Exec. of SW (for diagnosis)	n/a	0	8 samples proc. in PPC1	$8 \times 2.57\mu s$
Compare SW and HW results	n/a	0	performed in PPC0	$1.09 \times \mu s$
Restoration time (sum)		$907.67ms$		$23.33\mu s$

12-bit samples with adjustable rate. Also, a 1024×12 FIFO was configured.

Table 6.4 has the duration of the operations that take place during fault handling. Clearly, in the 1-PPR design the restoration time is dominated by the reconfiguration time. In the 2-PPRs design, reconfiguration is irrelevant and the time is mainly consumed in the diagnosis phase. Both designs are able to handle faults but the 2-PRRs design is restored much faster than the 1-PRR design. Therefore, it can serve considerably higher input rates without losing data, i.e. the FIFO didn't overflow [Ilias et al., 2010].

6.6 HMR vs. TMR

The primary difference between the HMR and the TMR is that in the former the third copy of the application runs in software. This allows for better scalability in terms of area. To compare the two schemes the Table 6.1 is examined further. Just like in HMR, TMR requires a circuit supporting partial reconfiguration in order to "scrub" the corrupted core. This circuit comprises the PPC and the ICAP port along with the supporting peripherals.

Figure 6.3 illustrates that the TMR FIFO controller consumes much less resources as compared to the FIR core. Even if the application increases, the aggregate area occupied by the three FIFO controllers would remain about the same. In the TMR solution the third replica of the FIR core would scale with the application size. Although the exact area needed for the implementation of the specific filter using TMR is unknown, the Table 6.5 was formed which consolidates the components scaling with the application size. This Table compares quantitatively the two schemes. It is obvious that in TMR the third HW

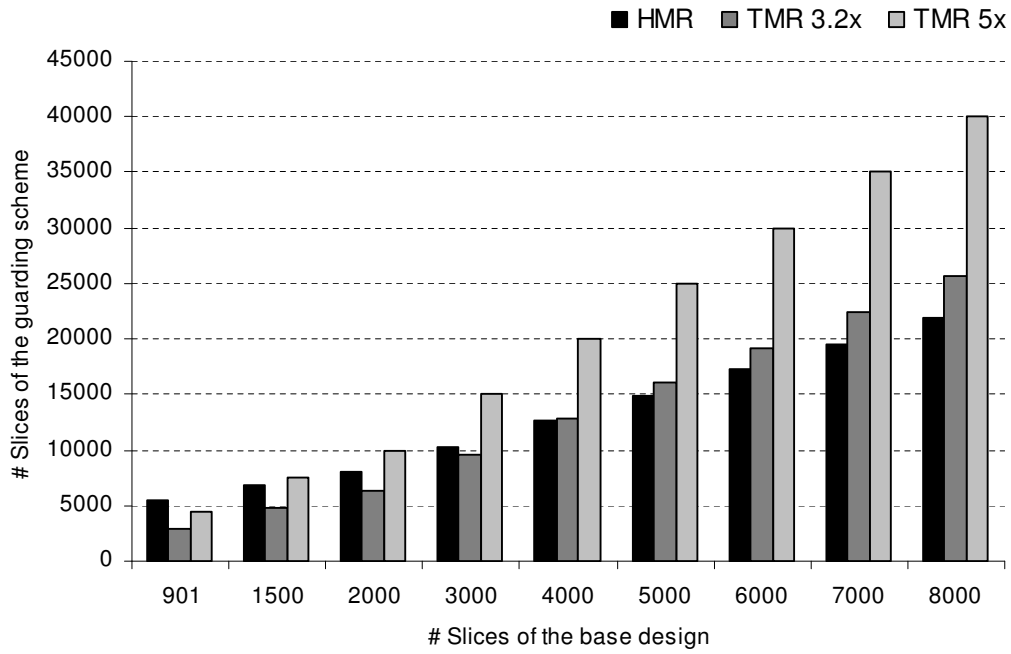


Figure 6.4: Slice utilization of HMR vs. TMR for various base designs in the Virtex-II Pro.

core apart from itself, incurs an additional overhead due to the bus macros and the I/O interfaces which is not paid by HMR. Furthermore, it incurs more complexity in routing, i.e. wires and switch matrixes, which is also not paid by HMR. The same stands for the voters and the feedback logic. In HMR the main concern is to keep as small as possible two components: i) the program memory that stores the SW model of the application and ii) the FIFO; both are implemented with embedded BRAMs. Regarding the former too many BRAMs have been allocated unnecessarily for the SW FIR filter and thus for a larger application no extra BRAMs would be needed. Specifically, the amount of memory in Table **??** is the minimum that can be set as imposed by the vendor's tool. Therefore, the problem translates mostly to whether the FIFO can be kept small in order to justify the usage of HMR instead of TMR.

Figure 6.4 projects the amount or slices needed for a larger design than the above filter, according to the scalability factors derived from Table 6.2 for the HMR, and from the literature for the TMR [Xilinx Inc., 2005b, Bolchini et al., 2007]. In HMR, along with the 2.3x utilization over the base design due to the duplication, the amount of slices for the static

Table 6.5: Components in HMR and TMR schemes that scale according to the application size.

Component	HMR	TMR
HW cores	√ (2)	√ (3)
majority voters	-	√
minority voters	-	√
feedback logic	-	√
bus macros	√	√
I/O interfaces	√	√
program memory	√	-
FIFO	√	-

part and for the TMR FIFO controller is also added, i.e. 3181 and 240 slices respectively. For larger designs than the filter, HMR starts exhibiting resource savings over the best case of TMR (overhead is 3.2x) when the size of the base design is around 4000. When compared to the worst case of TMR (overhead is 5x), the HMR scheme exhibits resource savings before the base design reaches the 1500 slices. It is obvious that the larger the hardware core, the more the percentage savings in resources that the HMR offers over the TMR. It is observed that for the largest base design the HMR achieves an area reduction from 15% to 46% over the TMR 3.2x and TMR 5x respectively. Further, more reduction can be achieved if the PPC1 will be removed; its entire operation can be transferred to the PPC0. Besides the PPC1 itself this improvement will eliminate its program memory along with its controller, the shared memory with the two controllers, and the communication buses between the two processors. Therefore, this configuration will account for less overhead as compared with the TMR solution.

The main drawback of HMR is that upon a fault presence the output is halted until the uncorrupted core is identified. If the performance can be kept at a satisfactory level, and the FIFO won't overflow, then the HMR can suit well for non time-critical systems [Wang and Bolotin, 2004]. It is worth noting that currently the system can execute at 107 MHz in the Virtex-II Pro, which is relatively bigger than the 100 MHz that is difficult to achieve with TMR in the same device [Bridgford et al., 2008].

6.7 Conclusions

The concept of combining hardware redundancy with software is new. The software undertakes the processing during fault handling and it doesn't allow for the hardware execution to resume processing until i) the entire PRR is reconfigured (1-PRR design), or ii) the uncorrupted core is identified based on a similar to majority voting mechanism (2-PPRs design). In order to harness the benefits of FPGAs in radiation-exposed environments it is necessary need to study the criticality-level of the application-at-hand and the availability of hardware resources. In the HMR scheme the cost of reduction in the processing rate is amortized by the smaller resource overhead over the TMR.

Although the concept of HMR is relatively simple, significant effort was put to implement the architecture using the error-prone PR flow. However, the results are interesting enough to justify the development of novel ways allowing the reliable operation of a system under the presence of faults. The HMR has been implemented and tested in a Virtex-II Pro FPGA. Newer FPGAs equipped with more powerful processors can be used such as the Virtex-5FX FPGA embedding the PowerPC440 processor and the Zynq-7000 Extensible Processing Platform with the ARM Cortex A-9 processor. Also it will be worth evaluating the system with larger applications such as filters used in the telecommunication domain [Gholamipour et al., 2009].

Chapter 7

Conclusions and Future Work

This Chapter summarizes the dissertation work and proposes some potential extensions. Also, it provides suggestions for further work in fields that worth to be studied in the context of dynamic reconfiguration. This technology emerges as a new model in developing applications and a lot of work remains to be done in order to become mainstream. Despite that not everyone agrees with this premise, it is a fact that several researchers are interested in using it for exploring its benefits in their applications. Currently only a few FPGA families support it, but FPGA vendors show persistent interest in incorporating it in their devices.

The field of this dissertation has been the exploration of effective use of dynamic reconfiguration and the trade-offs when using it to develop applications. I involved with three main subjects appeared to be worth studying after the completion of literature work in Chapter 2. I conducted experiments using two different platforms equipped with Xilinx Virtex FPGAs. Part of my work was verified with results derived by research conducted either in MHL or by groups of other Universities.

7.1 Contributions

A considerable amount of time was devoted to seeking the unexplored areas related with dynamic reconfiguration. I concluded that significant research has been done but many aspects of dynamic reconfiguration are still uncovered and worth revisiting. The technology undergoes continuous changes due to inherent difficulties related with the hardware and the

software tools supporting it.

Initially, I focused on the way the partially reconfigurable tasks can allocate the reconfigurable hardware by taking into account the constraints imposed by the physical constraints. I built a simulation framework which takes as input an application represented by its task graph, the execution time of each task, the task dependencies and the architecture information such as the amount of available physical resources, the number of frames, and the reconfiguration time. To develop the framework I used C and Python programming languages. The processed data can be imported in a spreadsheet to assess the application performance in a resource-constrained PR FPGA.

A problem raised at this point was that the value of reconfiguration time inserted as attribute to the framework was incorrect. More specifically, this time was calculated based on the throughput of configuration port. The same routine has been followed in other published works, but it is far from being realistic. Reconfiguration overhead is usually large and not negligible and such an assumption can result to misleading conclusions. Moreover, it is heavily dependent on the system setup such as the memory from which the bitstreams are fetched and the type of buses. As proven in this dissertation, in some cases the reconfiguration time computed by relying only on theoretical characteristics of the system could differ as much as three to four orders of magnitude compared to the actual value. To address this issue I developed a parameterized cost model which applies to a range of FPGA-based development platforms widely used within the research community. It relies on real-world experiments, on values of the theoretical characteristics of the system components and on existing well proven mathematical models, while maths were used to shape its final form. The cost model is versatile in the sense it provides results for a broad range of different setups. Its novelty lies in that it can accept values from theoretical characteristics of the system components (information that can be found in data sheets), and calculate directly the reconfiguration time with a good accuracy. I verified it with credible resources from experimental work in our laboratory and published works containing measurements of researchers in other Universities. The resulted cost model provides a holistic approach for evaluating dynamic reconfiguration from a system perspective. Its output can provide the value of reconfiguration time which is an attribute to the simulation framework. The latter

can serve researchers willing to assess the performance of their applications that would run on a PR FPGA, at an early stage of the design cycle.

Finally, I concentrated on a domain that can benefit from this technology. I designed a hybrid scheme combining software and hardware called HMR which allows to diagnose and recover systems that are subject to upsets. A FIFO is included as integral part of the system architecture, so I studied the way its integration affects the system performance and stability. The system was implemented and a prototype was made available for demonstration [Ilias, 2009]. I compared the new scheme against the dominant scheme in the domain, i.e. TMR. Although HMR fits well in resource-consuming applications, it poses some drawbacks as upon fault correction it might require reducing the processing rate of the input. Thus, HMR cannot be used in critical systems with the same performance as TMR. However, it can be adequate for non-critical applications that do not pose strict real-time requirements.

My dissertation considers the difficulties accompanied with dynamic reconfiguration technology. I went down into the details while I was following its changing progress. I suggested solutions in specific subjects within the context of dynamic reconfiguration and I attempted to open new directions in this emerging field.

7.2 Directions for Future Research

Present work can be extended toward different directions related with all three subjects. Moreover, due to the continuous changes of PR technology that appear to be device-specific, it is necessary to revisit these subjects.

- It is worth studying the way the proposed task allocation and scheduling fit in the 2D reconfiguration. Moreover, in the present task graphs, data exchange amongst the tasks swapped in and out of the reconfigurable regions is not considered. To do so, the edges should be annotated with additional information. Also, the simulation framework can be extended so as to support 2D reconfigurable devices. Further, in its present form it is semi-automated; certain processes can be automated such as the task graph insertion and the data importing for post processing. Finally, the ARM

processor that is embedded in the new Virtex-7 FPGA should be studied with regard to the way it executes instructions. The specific ARM processor has a lockup-free cache that allows for non-blocking execution, which corresponds to the architecture considered in the simulation framework.

- The cost model can be enriched with more attributes such as the frequency of the processor. New experiments with the latest Virtex-7 FPGA equipped with the ARM processor and with the new bus, i.e. AIX, will allow to extend the cost model of [PRCC, 2010] so as to cover more cases. It is very likely that the IBM PowerPC and the PLB bus will not be supported in future devices.

- The implementation of a large application using the HMR scheme can empower the results of the latter's performance. The FIFO requirements need to be examined further so as to provide a range of FIFO sizes that are adequate for serving different input rates. Finally, HMR can be built on a Virtex-7 FPGA embedding an ARM processor and by using point-to-point links instead of the PLB bus; this setup will result in fewer execution cycles and smaller communication overheads.

In order to strengthen the reasons for dynamic reconfiguration use, other subjects need to be studied with respect to the way they fit and interact with this technology, as well as the areas that benefit from it. To this direction, it is worthwhile to examine the way PR can be incorporated in Electronic System Level (ESL) methodologies. In general, ESL aims to elevate the design to a higher abstraction level in order to reinforce the adoption of hardware design by software-oriented programmers.

The application domains that can benefit from PR technology should be explored. Many applications seem promising but none has been commercialized yet. Researchers are working in different application domains ranging from modular robotics [Upegui et al., 2005] to wireless sensors networks [Garcia et al., 2009] and biometrics [Fons and Fons, 2010]. In order for the PR to be justified in developing an application, it is necessary to examine if its use satisfies the requirements possibly set by the designer. Reconfiguration overhead appears to be a significant burden when it comes to compare the performance of a PR design over its static alternative.

One important research field concerns the extent to which power consumption can be reduced with the use of PR. PR makes it possible to leave empty regions of the reconfigurable fabric during operation in case there are tasks that do not need to execute for a period of time. The alternative solution is to deactivate the tasks either with control signals or clock gating, but PR goes beyond this capability as it makes possible to create completely non-configured regions during run-time. This can result in both dynamic and static power savings. Representative work on this field has been published in [Noguera and Kennedy, 2007, Paulsson et al., 2008, Becker et al., 2010]. Two important factors that need to be studied are the power consumed by the reconfiguration process itself, and the power consumed during reconfiguration from the system perspective. A framework can be built that will allow to shape a power cost model similar to the one created for the reconfiguration overhead as part of the present dissertation work [PRCC, 2010]. Xilinx already offers such a model to estimate the power consumption for static implementations. Finally, it would be interesting to experiment with the Xilinx low-end Spartan-6 FPGAs which support partial reconfiguration. Recently, battery-operated development platforms with such FPGAs have been released. Research is needed to compare the battery drawn in static designs vs. their PR alternative for specific applications.

Bibliography

[Actel, 2007] Actel (2007). http://www.actel.com.

[Afratis et al., 2008] Afratis, P., Sotiriades, E., Chrysos, G., Fytraki, S., and Pnevmatikatos, D. (2008). A Rate-based Prefiltering Approach to BLAST Acceleration. In *Proceedings of the International Conference on Field Programmable Logic and Applications*.

[Algotronix, 1988] Algotronix (1988). Data Book. Algotronix, Edinburgh, UK.

[Alsolaim et al., 2000] Alsolaim, A., Starzyk, J. A., Becker, J., and Glesner, M. (2000). Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–216, Napa, CA.

[Altera, 1995] Altera (1995). Data Book. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 1998] Altera (1998). Data Book. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 2001] Altera (2001). Excalibur: Embedded Processor Programmable Solutions. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 2002] Altera (2002). APEX II Programmable Logic Device Family. Data Sheet. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 2003] Altera (2003). Mercury Programmable Logic Device Family. Data Sheet. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 2004] Altera (2004). Nios 3.0 CPU. Altera Corp., San Jose, CA. www.altera.com.

[Altera, 2005] Altera (2005). Cyclone Device Family Data Sheet. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 2006a] Altera (2006a). Cyclone II Device Family Data Sheet. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 2006b] Altera (2006b). Stratix Device Family Data Sheet. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 2006c] Altera (2006c). Stratix II Device Family Data Sheet. Altera Corp., San Jose, CA. http://www.altera.com.

[Altera, 2007] Altera (2007). http://www.altera.com/products/devices/mature/mat-index.html.

[Amano, 2006] Amano, H. (2006). A Survey on Dynamically Reconfigurable Processors. *IECIE Trans. Comm.*, (12):3179–3187.

[Anyfantis, 2007] Anyfantis, A. (2007). *Evaluation of Dynamic Reconfiguration in Modern Reconfigurable Logic Systems*. Diploma Thesis, Technical University of Crete.

[Arnold, 2005] Arnold, J. (2005). S5:The Architecture and Development Flow of a Software Configurable Processor.

[Atmel, 1993] Atmel (1993). Configurable Logic: Design and Application Book. Atmel Inc., San Jose, CA. www.atmel.com.

[Atmel, 2005] Atmel (2005). AT94KAL Series Field Programmable System Level Integrated Circuit. Atmel Inc., San Jose, CA. www.atmel.com.

[Atmel, 2006] Atmel (2006). AT40K: 5K-50K Gates Coprocessor FPGA with FreeRAM. Atmel Inc., San Jose, CA. www.atmel.com.

[BAE, 2007] BAE (2007). http://www.baesystems.com.

[Banerjee et al., 2005a] Banerjee, S., Bozorgzadeh, E., and Dutt (2005a). Physically-aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration. In *DAC*, pages 335–340.

[Banerjee et al., 2005b] Banerjee, S., Bozorgzadeh, E., and Dutt, N. (2005b). Physically-aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration. In *DAC*, pages 335–340.

[Barat and Lauwereins, 2000] Barat, F. and Lauwereins, R. (2000). Reconfigurable Instruction Set Processors: A Survey. In *IEEE International Workshop on Rapid System Prototyping*, pages 168–173.

[Barat et al., 2002] Barat, F., Lauwereins, R., and Deconinck, G. (2002). Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. *IEEE Transactions on Software Engineering*, 28(9):847–862.

[Baumgarte et al., 2003] Baumgarte, V., Ehlers, G., May, F., Nuckel, A., Vorbach, M., and Weinhardt, M. (2003). PACT XPP-A Self-Reconfigurable DataProcessing Architectures. *The Journal of Supercomputing*, 26(3):167–184.

[Baumgarte et al., 2001] Baumgarte, V., May, F., Nuckel, A., Vorbach, M., and Weinhardt, M. (2001). PACT XPPA self reconfigurable data processing architecture. In *In Procedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms ERSA*, Las Vegas, NV.

[Becker et al., 2010] Becker, T., Luk, W., and Cheung, P. Y. K. (2010). Energy-Aware Optimization for Run-Time Reconfiguration. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 55–62.

[Belayneh and Kaeli, 1996] Belayneh, S. and Kaeli, D. R. (1996). A Discussion on Non-Blocking/Lockup-Free Caches. Technical report, Northeastern University.

[Berkeley, 2007] Berkeley (2007). `http://brass.cs.berkeley.edu/reproc.html`.

[Bittner et al., 1996] Bittner, R. A., Athanas, P., and Musgrove, M. D. (1996). Colt: An Experiment in Wormhole Run-time Reconfiguration. In *Proceedings of the High-Speed Computing, Digital Signal Processing, and Filtering Using reconfigurable Logic*, pages 180–186.

[Blodget et al., 2003] Blodget, B., James-Roxby, P., Keller, E., McMillan, S., and Sundararajan, P. (2003). A Self-reconfiguring Platform. In *Proceedings of the International Conference on Field Programmable Logic and Applications(FPL)*, pages 565–574.

[Bobda, 2007] Bobda, C. (2007). *Introduction to Reconfigurable Computing Architectures Algorithms and Applications*. Springer.

[Bobda et al., 2004] Bobda, C., Majer, M., Koch, D., Ahmadinia, A., and Teich, J. (2004). A Dynamic NoC Approach for Communication in Reconfigurable Devices. In *Field Programmable Logic and Application*, pages 1032–1036.

[Bolchini et al., 2007] Bolchini, C., Miele, A., and Santambrogio, M. D. (2007). TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs. In *Proceedings of the IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 87–95.

[Bondalapati, 2001] Bondalapati, K. (2001). *Modeling and Mapping for Dynamically Reconfigurable Hybrid Architectures*. PhD thesis, University of Southern California.

[Bondalapati and Prasanna, 2002] Bondalapati, K. and Prasanna, V. K. (2002). Reconfigurable Computing Systems. *Proceedings of the IEEE*, 90(7):1201–1217.

[Brebner, 1996] Brebner, G. (1996). A Virtual Hardware Operating System for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field Programmable Logic and Applications*, pages 327–336.

[Brebner and Diessel, 2001] Brebner, G. and Diessel, O. (2001). Chip-Based Reconfigurable Task Management. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*, pages 182–191.

[Bridgford et al., 2008] Bridgford, B., CarMichael, C., and Tseng, C. W. (2008). Single-Event Upset Mitigation Selection Guide. Application Note XAPP987, Xilinx Inc.

[Cadambi et al., 1998] Cadambi, S., Weener, J., Goldstein, S. C., Schmit, H., and Thomas, D. (1998). Managing Pipeline Reconfigurable FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 55–64.

[Caffrey et al., 2009] Caffrey, M., Morgan, K., Roussel-Dupre, D., Robinson, S., Nelson, A., Salazar, A., Wirthlin, M., Howes, W., and Richins, D. (2009). On-Orbit Flight Results from the Reconfigurable Cibola Flight Experiment Satellite (CFESat). In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 3–10.

[CarMichael, 2006] CarMichael, C. (2006). Triple Module Redundancy Design Techniques for Virtex FPGAs. Application Note XAPP197, Xilinx Inc.

[CarMichael et al., 2000] CarMichael, C., Caffrey, M., and Salazar, A. (2000). Correcting Single-Event Upsets Through Virtex Partial Configuration. Application Note XAPP216, Xilinx Inc. and Los Alamos National Laboratories.

[Caspi et al., 2000] Caspi, E., Chu, M., Huang, R., Yeh, J., Wawrzynek, J., and DeHon, A. (2000). Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proceedings of the Field-Programmable Logic and Applications*, pages 605–614.

[Chameleon, 2000] Chameleon (2000). CS2000 Advance Product Specification. Chameleon Systems Inc., San Jose, Ca.

[Cheatham et al., 2006] Cheatham, J. A., Emmert, J. M., and Baumgart, S. (2006). A Survey of Fault Tolerant Methodologies for FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, 11(2):501–533.

[Cherepacha and Lewis, 1994] Cherepacha, D. and Lewis, D. (1994). A Datapath Oriented Architecture for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Array*.

[Chien and Byun, 1999] Chien, A. A. and Byun, J. H. (1999). Safe and Protected Execution for the Morph/AMRM Reconfigurable Processor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 209–221, Napa, CA.

[Claus et al., 2007] Claus, C., Muller, F. H., Zeppenfeld, J., and Stechele, W. (2007). A New Framework to Accelerate Virtex-II Pro Dynamic Partial Self-Reconfiguration. In *IPDPS*, pages 1–7.

[Claus et al., 2008] Claus, C., Zhang, B., Stechele, W., Braun, L., Hübner, M., and Becker, J. (2008). A Multi-Platform Controller Allowing for Maximum Dynamic Partial Reconfiguration Throughput. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 535–538.

[Compton, 1999] Compton, K. (1999). Programming Architectures for Run-Time Reconfigurable Systems. Master's thesis, Northwestern University, Evanston, USA.

[Compton and Hauck, 2002] Compton, K. and Hauck, S. (2002). Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210.

[Convey, 2011] Convey (2011). Convey Computer. `http://www.conveycomputer.com`.

[Cray, 2008] Cray (2008). Cray XD1 supercomputer. `http://www.cray.com`.

[Cypress, 2008] Cypress (2008). `http://www.cypress.com`.

[Dales, 1999] Dales, M. (1999). The Proteus Processor - A Conventional CPU with Reconfigurable Functionality. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*.

[DeHon, 1994] DeHon, A. (1994). DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proceedings of the IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 31–39, Napa, CA.

[DeHon et al., 2004] DeHon, A., Adams, J., DeLorimier, M., Kapre, N., Matsuda, Y., Naeimi, H., Vanier, M., and Wrighton, M. (2004). Design Patterns for Reconfigurable Computing. In *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 13–23.

[Delahaye et al., 2007] Delahaye, J.-P., Palicot, J., Moy, C., and Leray, P. (2007). Partial Reconfiguration of FPGAs for Dynamical Reconfiguration of a Software Radio Platform. In *Mobile and Wireless Communications Summit, 2007. 16th IST*, pages 1–5.

[Dick et al., 1998] Dick, R. P., Rhodes, D. L., and Wolf, W. (1998). TGFF: Task Graphs For Free. In *Proceedings of the IEEE International Workshop on Hardware/Software Codesign (CODES)*, pages 97–101.

[Digilent Inc., 2008] Digilent Inc. (2008). `http://www.digilentinc.com`.

[Dollas et al., 2004] Dollas, A., Papademetriou, K., Sotiriades, E., Theodoropoulos, D., Koidis, I., and Vernardos, G. (2004). A Case Study on Rapid Prototyping of Hardware Systems: the Effect of CAD Tool Capabilities, Design Flows, and Design Styles. In *International IEEE Workshop on Rapid System Prototyping (RSP)*, pages 180–186.

[Donthi and Haggard, 2003] Donthi, S. and Haggard, R. L. (2003). A Survey on Dynamically Reconfigurable Devices. In *Proceedings of the 35th IEEE Southeastern Symposium on System Theory*, pages 422–426.

[DRS-ARCS, 2007] DRS-ARCS (2007). `http://arcs07.ethz.ch/`.

[DynaChip, 1998] DynaChip (1998). DL6000 - Fast Field Programmable Gate Array. Datasheet. DynaChip Inc.

[Ebeling et al., 1996] Ebeling, C., Cronquist, D., and Franklin, P. (1996). RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications,*, pages 126–135.

[Effraimidis et al., 2009] Effraimidis, C., Papadimitriou, K., Dollas, A., and Papaefstathiou, I. (2009). A Self-Reconfiguring Architecture Supporting Multiple Objective Functions in Genetic Algorithms. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 453–456.

[ElementCXI, 2008] ElementCXI (2008). `http://www.elementcxi.com/`.

[Elixent, 2000] Elixent (2000). DFA 1000 Accelerator Data Sheet. Elixent Corp., Bristol, England.

[Emmert and Bhatia, 1997] Emmert, J. M. and Bhatia, D. (1997). Partial reconfiguration of FPGA mapped designs with applications to fault tolerance and yield enhancement. In *Proceedings of Field Programmable Logic and Applications (FPL)*, pages 141–150.

[Enzler, 1999] Enzler, R. (1999). The Current Status of Reconfigurable Computing. Technical report, Electronics Laboratory, Swiss Federal Institute of Technology.

[Enzler, 2004] Enzler, R. (2004). *Architectural Trade-offs in Dynamically Reconfigurable Processors*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland.

[Enzler and Platzner, 2001] Enzler, R. and Platzner, M. (2001). Dynamically Reconfigurable Processors. In *World Conference on Educational Multimedia, Hypermedia and Telecommunications*.

[Erasmushogeschool, 2007] Erasmushogeschool (2007). `http://elektronica.ehb.be/reco/Components.htm`.

[ERSA, 2008] ERSA (2008). `http://webest.uk.com/ersaco/`.

[Esparza, 2000] Esparza, J. E. C. (2000). Evaluation of the OneChip Reconfigurable Processor. Master's thesis, University of Toronto, Toronto, Canada.

[Farkas et al., 1994] Farkas, K. I., Jouppi, N. P., and Chow, P. (1994). How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors? Technical report, Western Research Laboratory.

[Faura et al., 1997] Faura, J., Moreno, J. N., Aguirre, M. A., van Duong, P., and Insenser, J. M. (1997). Multicontext dynamic reconfiguration and real-time probing on a novel mixed signal programmable device with onchip microprocessor. In *Procedings of th 7th International Workshop on Field Programmable Logic and Applications*, pages 1–10.

[FCCM, 2008] FCCM (2008). `http://www.fccm.org/`.

[Fong et al., 2003] Fong, R. J., Harper, S. J., and Athanas, P. M. (2003). A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguation. In *Proceedings of the IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 117–123.

[Fons and Fons, 2010] Fons, F. and Fons, M. (2010). Making Biometrics the Killer App of FPGA Dynamic Partial Reconfiguration. Xcell journal online.

[FPL, 2006] FPL (2006). `http://arantxa.ii.uam.es/~fpl06/`.

[FPL, 2008] FPL (2008). `http://www.kip.uni-heidelberg.de/fpl08`.

[FPL, 2011] FPL (2011). `http://fpl2011.org`.

[FPT, 2007] FPT (2007). `http://www.icfpt.org/`.

[French et al., 2008] French, M., Anderson, E., and Kang, D.-I. (2008). Autonomous System on a Chip Adaptation through Partial Runtime Reconfiguration. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 77–86.

[French and Taylor, 1993] French, P. C. and Taylor, R. W. (1993). A Self-Reconfigurable Processor. In *Proceedings of the IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 50–59, Napa, CA.

[Fuji et al., 1999] Fuji, T., i. Furuta, K., Motomura, M., Nomura, M., Mizuno, M., i. Anjo, K., Wakabayashi, K., Hirota, Y., e. Nakazawa, Y., Itoh, H., and Yamashina, M. (1999). A Dynamically Reconfigurable Logic Engine with a Multi-context/Multi-mode Unified Cell Architecture. In *Proceedings of International Solid-State Circuits Conference*, pages 364–365.

[Galindo et al., 2008] Galindo, J., Peskin, E., Larson, B., and Roylance, G. (2008). Leveraging Firmware in Multichip Systems to Maximize FPGA Resources: An Application of Self-Partial Reconfiguration. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 139–144.

[Garcia et al., 2006] Garcia, P., Compton, K., Schulte, M., Blem, E., and Fu, W. (2006). An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP Journal on Embedded Systems*, 2006(1):1–19.

[Garcia et al., 2009] Garcia, R., Gordon-Ross, A., and George, A. D. (2009). Exploiting Partially Reconfigurable FPGAs for Situation-Based Reconfiguration in Wireless Sensor Networks. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 243–246.

[Gelado et al., 2006] Gelado, I., Morancho, E., and Navarro, N. (2006). Experimental Support for Reconfigurable Application-Specific Accelerators. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA 2006), in conjuction with ISCA*, pages 50–57.

[Gholamipour et al., 2011] Gholamipour, A., Papadimitriou, K., Kurdahi, F., Dollas, A., and Eltawil, A. (2011). Area, Reconfiguration Delay and Reliability Trade-Offs in Designing Reliable Multi-Mode FIR Filters. In *Proceedings of the IEEE International Design and Test Workshop (IDT)*.

[Gholamipour et al., 2009] Gholamipour, A. H., Eslami, H., Eltawil, A., and Kurdahi, F. (2009). Size-Reconfiguration Delay Tradeoffs for a Class of DSP Block in Multi-mode Communication Systems. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 71–78.

[Gokhale et al.,] Gokhale, M., Graham, P., Wirthlin, M., Johnson, D. E., and Rollins, N. Dynamic Reconfiguration for Management of Radiation-Induced Faults in FPGAs. *International Journal of Embedded Systems 2006*, 2(1).

[Gokhale and Graham, 2005] Gokhale, M. and Graham, P. S. (2005). *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer.

[Gonzalez et al., 2003] Gonzalez, I., Lopez-Buedo, S., Gomez, F., and Martinez, J. (2003). Using Partial Reconfiguration in Cryptographic Applications: An Implementation of the IDEA Algorithm. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*, pages 194–203.

[Griese et al., 2004] Griese, B., Vonnahme, E., Porrmann, M., and Ruckert, U. (2004). Hardware Support for Dynamic Reconfiguration in Reconfigurable SoC Architectures. In *Proceedings of the International Workshop on Field Programmable Logic and Applications (FPL)*, pages 842–846.

[Guccione and Levi, 1999] Guccione, S. A. and Levi, D. (1999). The Advantages of Run-Time Reconfiguration. In *Proceedings of the International Society for Optical Engineering*.

[Hartenstein, 2001] Hartenstein, R. W. (2001). A Decade of Reconfigurable Computing: a Visionary Retrospective. In *DATE*, pages 642–649.

[Hartenstein et al., 1994] Hartenstein, R. W., Kress, R., and Reinig, H. (1994). A new FPGA architecture for word-oriented datapaths. In *Proceedings of the 4th International Workshop on Field Programmable Logic and Applications*.

[Hauck, 1998a] Hauck, S. (1998a). Configuration Prefetch for Single Context Reconfigurable Coprocessors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 65–74.

[Hauck, 1998b] Hauck, S. (1998b). The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–639.

[Hauck and DeHon, 2008] Hauck, S. and DeHon, A. (2008). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Elsevier.

[Hauck et al., 1997] Hauck, S., Fry, T. W., Hosler, M. M., and Kao, J. P. (1997). The Chimaera Reconfigurable Functional Unit. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, Napa, CA.

[Hauck et al., 2004] Hauck, S., Fry, T. W., Hosler, M. M., and Kao, J. P. (2004). The Chimaera Reconfigurable Functional Unit. *IEEE Transactions on VLSI Systems*, 12(2):206–217.

[Hauser and Wawrzynek, 1997] Hauser, J. R. and Wawrzynek, J. (1997). Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 24–33, Napa, CA.

[Hawley, 1991] Hawley, D. (1991). Advanced PLD Architectures. *Abingdon EE and CS Books*, pages 11–23.

[Hennessy and Patterson, 2003] Hennessy, J. and Patterson, D. (2003). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

[Hsiung et al., 2008] Hsiung, P.-A., Lin, C.-S., and Liao, C.-F. (2008). Perfecto: A Systemc-based Design-space Exploration Framework for Dynamically Reconfigurable Architectures. *ACM Transactions on Reconfigurable Technology and Systems*, 1(3):1–30.

[Huang and McCluskey, 2001] Huang, W.-J. and McCluskey, E. J. (2001). Column-Based Precompiled Configuration Techniques for FPGA. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 137–146.

[Hymel et al., 2007] Hymel, R., George, A. F., and Lam, H. (2007). Evaluating Partial Reconfiguration for Embedded FPGA Applications Interfaces. In *Proceedings of the High Performance Embedded Computing Workshop*.

[IBM, 2007] IBM (2007). `http://www.chips.ibm.com/products/coreconnect`.

[IBM Inc., 2000] IBM Inc. (2000). 128-bit Processor Local Bus. Architecture Specifications Version 4.6.

[IBM Inc., 2001] IBM Inc. (2001). On-Chip Peripheral Bus. Architecture Specifications Version 2.1.

[IBM Inc., 2006] IBM Inc. (2006). Device Control Register Bus 3.5. Architecture Specifications.

[Ilias, 2009] Ilias, A. (2009). *Dynamically Reconfigurable Systems for Handling Failures in Real-Time Conditions*. Diploma Thesis, Technical University of Crete.

[Ilias et al., 2010] Ilias, A., Papadimitriou, K., and Dollas, A. (2010). Combining Duplication, Partial Reconfiguration and Software for On-line Error Diagnosis and Recovery in SRAM-based FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–76.

[Iliopoulos and Antonakopoulos, 2001] Iliopoulos, M. and Antonakopoulos, T. (2001). Run-Time Optimized Reconfiguation Using Instruction Forecasting. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 286–295.

[IP Flex, 2008] IP Flex (2008). `http://www.ipflex.com`.

[Jacob, 1998] Jacob, J. (1998). Memory Interfacing for the OneChip Reconfigurable Processor. Master's thesis, University of Toronto, Toronto, Canada.

[Jenkins, 1994] Jenkins, J. H. (1994). Designing with FPGAs and CPLDs. In *Englewood Cliffs, New Jersey: PTR Prentice Hall*.

[Jeong et al., 1999] Jeong, B., Yoo, S., S.Lee, and Choi, K. (1999). Exploiting Early Partial Reconfiguration of Run-Time Reconfigurable FPGAs in Embedded Systems Design. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 247–247.

[Johnson et al., 2008] Johnson, J., Howes, W., Wirthlin, M., McMurtrey, D., Caffrey, M., Graham, P., and Morgan, K. (2008). Using Duplication with Compare for On-line Error Detection in FPGA-based Designs. In *IEEE Aerospace Conference*, pages 1–11.

[Kachris and Vassiliadis, 2006] Kachris, C. and Vassiliadis, S. (2006). Performance Evaluation of an Adaptive FPGA for Network Applications. In *Proceedings of the 17th IEEE International Workshop on Rapid System Prototyping*.

[Kastrup et al., 1999] Kastrup, B., Bink, A., and Hoogerbrugge, J. (1999). ConCISe: A compiler-driven CPLD-based instruction set accelerator. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 92–101, Napa, CA.

[Kaviani and Brown, 1996] Kaviani, A. and Brown, S. (1996). Hybrid FPGA Architecture. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*.

[Kyriakoulakos and Pnevmatikatos, 2009] Kyriakoulakos, K. and Pnevmatikatos, D. (2009). A Novel SRAM-Based FPGA Architecture for Efficient TMR Fault Tolerance Support. In *Proceedings of the International Workshop on Field Programmable Logic and Applications (FPL)*, pages 193–198.

[Lattice, 2005] Lattice (2005). ispXPGA Family. Lattice Semiconductor Corp., Hillsboro, Oregon. `http://www.latticesemi.com`.

[Li, 2002] Li, Z. (2002). *Configuration Management Techniques for Reconfigurable Computing*. Phd thesis, Northwestern University.

[Ling and Amano, 1993] Ling, X. P. and Amano, H. (1993). WASMII: a data driven computer on a virtual hardware. In *Proceedings of the IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 33–42, Napa, CA.

[Liu et al., 2009] Liu, M., Kuehn, W., Lu, Z., and Jantsch, A. (2009). Run-Time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration. In *Proceedings of the IEEE International Workshop on Field Programmable Logic and Applications (FPL)*.

[Lodi et al., 2003] Lodi, A., Toma, M., Campi, F., Cappelli, A., Canegallo, R., and Guerrieri, R. (2003). A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications. *IEEE Journal of Solid-State Circuits*, 38(11):1876–1886.

[Lotfifar and Shahhoseini, 2008] Lotfifar, F. and Shahhoseini, H. S. (2008). Performance Evaluation of Partially Reconfigurable Computing Systems. In *2nd HiPEAC Workshop on Reconfigurable Computing*.

[Lucent, 1998] Lucent (1998). FPGA Data Book. Lucent Technologies Inc. Allentown, PA.

[Lund, 2004] Lund, K. (2004). PLB vs. OCM Comparison Using the Packet Processor Software. Application Note: Virtex-II Pro Family XAPP644 (v1.1), Xilinx.

[Lysaght et al., 2006] Lysaght, P., Blodget, B., Mason, J., Young, J., and Bridgford, B. (2006). Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *Proceedings of the IEEE International Conference Field Programmable Logic and Applications (FPL)*, pages 1–6.

[Lysaght and Dunlop, 1993] Lysaght, P. and Dunlop, J. (1993). Dynamic reconfiguration of Field Programmable Gate Arrays. In *Proceedings of the International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 82–94.

[MacBeth and Lysaght, 2001] MacBeth, J. and Lysaght, P. (2001). Dynamically Reconfigurable Cores. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*.

[Manet et al., 2008] Manet, P., Maufroid, D., Tosi, L., Gailliard, G., Mulertt, O., Ciano, M. D., Legat, J.-D., Aulagnier, D., Gamrat, C., Liberati, R., Barba, V. L., Cuvelier, P., Rousseau, B., and Gelineau, P. (2008). An Evaluation of Dynamic Partial Reconfiguration for Signal and Image Processing in Professional Electronics Applications. *EURASIP Journal on Embedded Systems*, 2008:11 pages.

[Marshall et al., 1999] Marshall, A., Vuillemin, J., Stansfield, T., Kostarnov, I., and Hutchings, B. (1999). A Reconfigurable Arithmetic Array for Multimedia Applications. In *International Symposium on Field Programmable Gate Arrays*, pages 135–144.

[Master, 2006] Master, P. L. (2006). Reconfigurable Hardware and Software Architectural Constructs for the Enablement of Resilient Computing Systems. In *Proceedings of the ASAP*, pages 50–55.

[Maxeler, 2011] Maxeler (2011). Maxeler Technologies. `http://www.maxeler.com`.

[McGregor and Lysaght, 1999] McGregor, G. and Lysaght, P. (1999). Self Controlling Dynamic Reconfiguration: A Case Study. In *Proceedings of the International Workshop Field-Programmable Logic and Applications*, pages 144–154.

[McKay et al., 1998] McKay, N., Melham, T., and Susanto, K. (1998). Dynamic specialisation of XC6200 FPGAs by partial evaluation. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–309.

[McKay and Singh, 1998] McKay, N. and Singh, S. (1998). Dynamic Specialisation of XC6200 FPGAs by Partial Evaluation. In *Proceedings of International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 298–307.

[McKay and Singh, 1999] McKay, N. and Singh, S. (1999). Debugging Techniques for Dynamically Reconfigurable Hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[Mei et al., 2004] Mei, B., Vernalde, S., Verkest, D., and Lauwereins, R. (2004). Design Methodology for a Tightly Coupled VLIW-Reconfigurable Matrix Architecture: A Case Study. In *DATE*, pages 1224–1229.

[Mei et al., 2003] Mei, B., Vernalde, S., Verkest, D., Man, H. D., and Lauwereins, R. (2003). ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proceedings of the 13th International Workshop on Field Programmable Logic and Applications*.

[Mirsky and Dehon, 1996] Mirsky, E. and Dehon, A. (1996). MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA.

[Miyamori and Olukotun, 1998] Miyamori, T. and Olukotun, K. (1998). A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 2–11, Napa, CA.

[Miyazaki, 1998] Miyazaki, T. (1998). Reconfigurable Systems: A Survey (Embedded Toturial). In *ASP-DAC*, pages 447–452.

[Möller et al., 2006] Möller, L., Soares, R., Carvalho, E., Grehs, I., Calazans, N., and Moraes, F. (2006). Infrastructure for Dynamic Reconfigurable Systems: Choices and Trade-offs. In *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design, SBCCI*, pages 44–49.

[Motomura, 2002] Motomura, M. (2002). A Dynamically Reconfigurable Processor Architecture. In *Microprocessor Forum*.

[Munoz, 1999] Munoz, J. (1999). DARPA's Adaptive Computing Systems Program, Invited talk. In *1st NASA/DOD Workshop on Evolvable Hardware*.

[Nagami et al., 1998] Nagami, K., Oguri, K., Shiozawa, T., Ito, H., and Konishi, R. (1998). Plastic Cell Architecture: Towards Reconfigurable Computing for General-Purpose. In

Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, pages 68–77, Napa, CA.

[National, 1993] National (1993). Configurable Logic Array (CLAy) Data Sheet. National Semiconductor.

[National, 1996] National (1996). NAPA1000 Data Sheet. National Semiconductor.

[Noguera and Badia, 2002] Noguera, J. and Badia, R. M. (2002). Dynamic Run-Time H/W Scheduling Techniques for Reconfigurable Architectures. In *CODES*, pages 205–210.

[Noguera and Kennedy, 2007] Noguera, J. and Kennedy, I. O. (2007). Power Reduction in Network Equipment through Adaptive Partial Reconfiguration. In *Proceedings of the IEEE International Workshop on Field Programmable Logic and Applications (FPL)*, pages 240–245.

[Nowak, 2004] Nowak, K. (2004). *Template-Based Embedded Reconfigurable Computing*. PhD thesis, Technische Universiteit Eindhoven.

[Olukotun and Hammond, 2005] Olukotun, K. and Hammond, L. (2005). The Future of Microprocessors. *ACM QUEUE Magazine*, 3(7):26–34.

[Ottawa, 2007] Ottawa (2007). `http://www.site.uottawa.ca/~rabielmo/personal/rc.html`.

[PACT XPP, a] PACT XPP. Reconfiguration on XPP-III Processors.

[PACT XPP, b] PACT XPP. XPP-III Processor Overview.

[PACT XPP, 2008] PACT XPP (2008). `http://www.pactxpp.com`.

[Papademetriou and Dollas, 2006a] Papademetriou, K. and Dollas, A. (2006a). A Task Graph Approach for Efficient Exploitation of Reconfiguration in Dynamically Reconfigurable Systems. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 307–308.

[Papademetriou and Dollas, 2006b] Papademetriou, K. and Dollas, A. (2006b). Performance Evaluation of a Preloading Model in Dynamically Reconfigurable Processors. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 901–904.

[Papademetriou et al., 2006] Papademetriou, K., Dollas, A., and Sotiropoulos, S. (2006). Low Cost Real-Time 2-D Motion Detection based on Reconfigurable Computing. *IEEE Transactions on Instrumentation and Measurement (TIM)*, 55(6):2234–2243.

[Papadimitriou et al., 2007] Papadimitriou, K., Anyfantis, A., and Dollas, A. (2007). Methodology and Experimental Setup for the Determination of System-level Dynamic Reconfiguration Overhead. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 335–336.

[Papadimitriou et al., 2010] Papadimitriou, K., Anyfantis, A., and Dollas, A. (2010). An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems. *IEEE Transactions on Instrumentation and Measurement (TIM)*, 59(6):1642–1651.

[Papadimitriou et al., 2011] Papadimitriou, K., Dollas, A., and Hauck, S. (2011). Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model. *ACM Transactions on Reconfigurable Technoclogy and Systems (TRETS)*, 4(4):1–27.

[Paulsson et al., 2008] Paulsson, K., Hubner, M., and Becker, J. (2008). Cost-and Power Optimized FPGA based System Integration: Methodologies and Integration of a Low-Power Capacity-based Measurement Application on Xilinx FPGAs. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, pages 50–55.

[Petrick et al., 2005] Petrick, D., Powell, W., Howard, J., and LaBel, K. (2005). Virtex-II Pro PowerPC SEE Characterization Test Methods and Results. In *International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, page 7 pages.

[Pico Computing, 2011] Pico Computing (2011). Pico Computing. `http://www.picocomputing.com`.

[PicoChip, 2008] PicoChip (2008). `http://www.picochip.com`.

[Plessey, 1990] Plessey (1990). ERA60100 Datasheet Electrically Reconfigurable Array. Plessey Semiconductors. Cheney Manor, Sindown, Wiltshire SN2 2QW, UK.

[Plessl and Platzner, 2004] Plessl, C. and Platzner, M. (2004). Virtualization of Hardware - Introduction and Survey. In *ERSA*, pages 63–69.

[Pozzi, 2000] Pozzi, L. (2000). *Methodologies for the Design of Application-Specific Reconfigurable VLIW Processors*. PhD thesis, Politecnico di Milano.

[PRCC, 2010] PRCC (2010). Partial Reconfiguration Cost Calculator. `http://users.isc.tuc.gr/~kpapadimitriou/prcc.html`.

[Programmable Logic Design Line, 2008] Programmable Logic Design Line (2008). Xilinx honored for enabling technology in the ALICE experiment at CERN. Xilinx Inc., San Jose, CA. http://www.pldesignline.com/news/207101017.

[Quicklogic, 2007] Quicklogic (2007). `http://www.quicklogic.com` (unavailable).

[Quicksilver, 2007] Quicksilver (2007). `http://www.qstech.com`.

[Quickturn, 1999] Quickturn (1999). A Cadence Company, System Realizer, San Jose, CA. `http://www.quickturn.com/products/systemrealizer.htm`.

[Quinn et al., 2008] Quinn, H., Graham, P., Morgan, K., Krone, J., Caffrey, M., and Wirthlin, M. (2008). An introduction to radiation-induced failure modes and related mitigation methods for Xilinx SRAM FPGAs. In *Proceedings of the ERSA*, page 139.

[Quinn et al., 2009] Quinn, H. M., Graham, P. S., Wirthlin, M. J., Pratt, B. H., Morgan, K., Caffrey, M. P., and Krone, J. (2009). A Test Methodology for Determining Space Readiness of Xilinx SRAM-Based FPGA Devices and Designs. *IEEE Transactions on Instrumentation and Measurement*, 58(10):3380–3395.

[Radunovic and Milutinovic, 1998] Radunovic, B. and Milutinovic, V. (1998). A Survey of Reconfigurable Computing Architectures. In *Proceedings of the 1998 Field-Programmable Logic and Applications*, pages 376–385. Springer.

[Rapport Inc., 2008] Rapport Inc. (2008). `http://www.rapportincorporated.com/`.

[Ratter, 2004] Ratter, D. (2004). FPGAs on Mars. Xcell journal online, NU Horizons Electronics.

[RAW, 2008] RAW (2008). `http://www.ece.lsu.edu/vaidy/raw/`.

[Razdan and Smith, 1994] Razdan, R. and Smith, M. D. (1994). A High-Performance Microarchitecture with hardware-programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180.

[Robertson and Irvine, 2004] Robertson, I. and Irvine, J. (2004). A Design Flow for Partially Reconfigurable Hardware. *ACM Transactions on Embedded Computing Systems*, 3(2):257–283.

[Santambrogio et al., 2008] Santambrogio, M. D., Rana, V., and Sciuto, D. (2008). Operating System Support for Online Partial Dynamic Reconfiguration Management. In *Proceedings of the IEEE International Workshop on Field Programmable Logic and Applications (FPL)*, pages 455–458.

[Sato et al., 2005] Sato, T., Watanabe, H., and Shiba, K. (2005). Implementation of Dynamically Reconfigurable Processor DAPDNA-2. In *IEEE VLSI Design and Test*.

[Sawitzki et al., 1998] Sawitzki, S., Gratz, A., and Spallek, R. G. (1998). CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism. In *In Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems*.

[Scalera and Vazquez, 1998] Scalera, S. M. and Vazquez, J. R. (1998). The design and implementation of a context-switching FPGA. In *Proceedings of the IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 78–85, Napa, CA.

[Schaumont et al., 2001] Schaumont, P., Verbauwhede, I., Keutzer, K., and Sarrafzadeh, M. (2001). A Quick Safari Through the Reconfiguration Jungle. In *Proceedings of the 2001 Design Automation Conference*, pages 172–177.

[Schmit, 1997] Schmit, H. (1997). Incremental Reconfiguration for Pipelined Applications. In *Proceedings of the IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 47–55, Napa, CA.

[Shirazi et al., 1998] Shirazi, N., Luk, W., and Cheung, P. (1998). Run-Time Management of Dynamically Reconfigurable Designs. In *Proceedings of the International Workshop Field-Programmable Logic and Applications*, pages 59–68.

[Sidhu et al., 2000] Sidhu, R., Wadhwa, S., Mei, A., and Prasanna, V. K. (2000). A self-reconfigurable gate array architecture. In *Proceedings of the 10th International Workshop on Field Programmable Logic and Applications*, pages 106–120.

[Sima et al., 2002] Sima, M., Vassiliadis, S., Cotofana, S., van Eijndhoven, J. T. J., and Vissers, K. A. (2002). Field-Programmable Custom Computing Machines - A Taxonomy -. In *Proceedings of the 2002 Field-Programmable Logic and Applications*, pages 79–88.

[Singh, 1998] Singh, H. (1998). MorphoSys: An Integrated Re-configurable Architecture. In *Proceedings of the NATO RTO Symposium on System Concepts and Integration*.

[Singhal and Bozorgzadeh, 2006] Singhal, L. and Bozorgzadeh, E. (2006). Milti-Layer Floorplanning on a Sequence of Reconfigurable Designs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 605–612.

[Smith and Xia, 2004] Smith, J. and Xia, T. (2004). PRMC: A Multicontext FPGA with Partially Reconfigurable Logic Planes. In *Proceedings of the IEEE Northeast Workshop on Circuits and Systems*, pages 393–396.

[Soudris et al., 2002] Soudris, D., Masselos, K., Blionas, S., Siskos, S., Nikolaidis, S., Tatas, K., and Mignolet, J.-Y. (2002). AMDREL: On Designing Reconfigurable Embedded Structures for the Future Reconfigurable SoC for Wireless Communication Applications. In *Proceedings of IEEE Workshop on Heterogeneous Reconfigurable Systems on Chip*.

[Steiger et al., 2003] Steiger, C., Walder, H., and Platzner, M. (2003). Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*, pages 575–584.

[Steiger et al., 2004] Steiger, C., Walder, H., and Platzner, M. (2004). Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE Transactions on Computers*, 53(11):1393–1407.

[Stitt et al., 2003] Stitt, G., Lysecky, R., and Vahid, F. (2003). Dynamic Hardware/Software Partitioning: A First Approach. In *Design Automation Conference*, pages 250–255.

[Stott et al., 2008] Stott, E., Sedcole, P., and Cheung, P. Y. K. (2008). Fault Tolerant Methods for Reliability in FPGAs. In *Proceedings of the International Workshop on Field Programmable Logic and Applications (FPL)*, pages 415–420.

[Tan et al., 2006] Tan, H., DeMara, R. F., Thakkar, A. J., Ejnioui, A., and Sattler, J. D. (2006). Complexity and Performance Evaluation of Two Partial Reconfiguration Interfaces on FPGAs: a Case Study. In *Proceedings of the ERSA*.

[Tang et al., 2000] Tang, X., Aalsma, M., and Jou, R. (2000). A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors. In *Proceedings of the Field-Programmable Logic and Applications*, pages 29–38. Springer.

[Tessier and Burleson, 2001] Tessier, R. and Burleson, W. (2001). Reconfigurable Computing for Digital Signal Processing: A Survey. *Journal of VLSI Signal Processing*, 28:7–27.

[Thoma et al., 2003] Thoma, Y., Sanchez, E., Arostegui, J.-M. M., and Tempesti1, G. (2003). A Dynamic Routing Algorithm for a Bio-inspired Reconfigurable Circuit. In *Proceedings of the Field Programmable Logic and Applications*, pages 681–690.

[Todman et al., 2005] Todman, T., Constantinides, G., Wilton, S., Mencer, O., Luk, W., and Cheung, P. (2005). Reconfigurable Computing Architectures and Design Methods. *IEE Proc.-Comput. Digit. Tech*, 152(2):193–207.

[Toshiba, 2003] Toshiba (2003). Press Release: Toshiba and Elixent Agree to Jointly Develop Reconfigurable Platform SoCs. First Platform Device to Be Available in 2003. Toshiba Inc. `http://www.eetimes.com/press_releases/bizwire/showPressRelease.jhtml?articleID=34403&CompanyId=2`.

[Trimberger et al., 1997] Trimberger, S., Carberry, D., Johnson, A., and Wong, J. (1997). A time-multiplexed FPGA. In *Proceedings of the IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 22–28, Napa, CA.

[Triscend, 2000] Triscend (2000). Triscend A7 Configurable System-on-Chip Family Data Sheet. Triscend Corp., Mountain View, CA.

[Triscend, 2003] Triscend (2003). Triscend E5 Customizable Microcontroller Platform: Product Description. Triscend Corp., Mountain View, CA.

[Upegui et al., 2005] Upegui, A., Moeckel, R., Dittrich, E., Ijspeert, A. J., and Sanchez, E. (2005). An FPGA Dynamically Reconfigurable Framework for Modular Robotics. In *Proceedings of the Workshop on Architecture and Computing Systems Workshop*, pages 83–89.

[Vasilko and Ait-boudaoud, 1996] Vasilko, M. and Ait-boudaoud, D. (1996). Optically Reconfigurable FPGAs: Is this a Future Trend? In *Proceedings of the International Workshop on Field Programmable Logic and Applications*, pages 270–279.

[Vassiliadis et al., 2001] Vassiliadis, S., Wong, S., , and Cotofana, S. (2001). The MOLEN $\rho\mu$-coded Processor. In *Proceedings of the 11th International Workshop on Field Programmable Logic and Applications*, pages 275–285.

[Vassiliadis et al., 2004] Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., and Panainte, E. M. (2004). The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363–1375.

[Vavouras et al., 2009a] Vavouras, M., Papadimitriou, K., and Papaefstathiou, I. (2009a). High-speed FPGA-based Implementations of a Genetic Algorithm. In *IEEE International Conference on Embedded Computer Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 9–16.

[Vavouras et al., 2009b] Vavouras, M., Papadimitriou, K., and Papaefstathiou, I. (2009b). Implementation of a Genetic Algorithm on a Virtex-II Pro FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 287–287.

[Villasenor and Hutchings, 1998] Villasenor, J. and Hutchings, B. (1998). The Flexibility of Configurable Computing. *IEEE Signal Processing Magazine*, pages 67–84.

[Waingold et al., 1997] Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S. P., and Agarwal, A. (1997). Baring It All to Software: RAW Machines. *IEEE Computer*, 30(9):86–93.

[Wang and Bolotin, 2004] Wang, M. and Bolotin, G. (2004). IBM PowerPC 405 SEU Mitigation Using Processor Voting Techniques in Xilinx Virtex-II Pro FPGA. In *International Conference on Military and Aerospace Programmable Logic Devices (MAPLD).*

[Wirthlin, 1997] Wirthlin, M. (1997). *Improving Functional Density Through Run-Time Circuit Reconfiguration.* PhD thesis, Brigham Young University.

[Wirthlin and Hutchings, 1995a] Wirthlin, M. and Hutchings, B. (1995a). A Dynamic Instruction Set Computer. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 99–107, Napa, CA. IEEE Computer Society Press.

[Wirthlin and Hutchings, 1995b] Wirthlin, M. and Hutchings, B. (1995b). Implementation Approaches for Reconfigurable Logic Applications. In *Proceedings of the Field-Programmable Logic and Applications*. Springer.

[Wirthlin and Hutchings, 1997] Wirthlin, M. and Hutchings, B. (1997). Improving Functional Density Through Run-Time Constant Propagation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*.

[Wittig, 1995] Wittig, R. D. (1995). OneChip: An FPGA Processor with Reconfigurable Logic. Master's thesis, University of Toronto, Toronto, Canada.

[Wong et al., 2002] Wong, S., Vassiliadis, S., and Cotofana, S. (2002). Microcoded Reconfigurable Embedded Processors: Current Developments. In *SAMOS*, pages 207–224.

[Wu, 2007] Wu, K. (2007). *Reconfigurable Architectures: from Physical Implementation to Dynamic Behaviour Modelling*. PhD thesis, Technical University of Denmark.

[Xilinx, 1994] Xilinx (1994). The Programmable Logic Data Book. Xilinx Inc., San Jose, CA. `http://www.xilinx.com`.

[Xilinx, 1995] Xilinx (1995). The XC6200 Fine Grain FPGA. Xilinx Inc., San Jose, CA. `www.xilinx.com`.

[Xilinx, 1999] Xilinx (1999). Virtex 2.5 V Field Programmable Gate Arrays: Advance Product Specification. Xilinx Inc., San Jose, CA. `www.xilinx.com`.

[Xilinx, 2002a] Xilinx (2002a). Press Release: Xilinx Announces Industry's First Programmable Crossbar Switch Solution. Xilinx Inc., San Jose, CA. `http://www.xilinx.com/prs_rls/end_markets/02151crossbar.htm`.

[Xilinx, 2002b] Xilinx (2002b). Spartan and Spartan-XL Families Field Programmable Gate Arrays. Xilinx Inc., San Jose, CA. `http://www.xilinx.com`.

[Xilinx, 2004] Xilinx (2004). Spartan-II 2.5V FPGA Family: Complete Data Sheet. Xilinx Inc., San Jose, CA. `http://www.xilinx.com`.

[Xilinx, 2007a] Xilinx (2007a). MicroBlaze Processor Reference Guide. Xilinx Inc., San Jose, CA. `www.xilinx.com`.

[Xilinx, 2007b] Xilinx (2007b). Virtex-4 Configuration Guide v1.9. Xilinx Inc., San Jose, CA. `www.xilinx.com`.

[Xilinx, 2007c] Xilinx (2007c). Virtex-4 User Guide. Xilinx Inc., San Jose, CA. `www.xilinx.com`.

[Xilinx, 2007d] Xilinx (2007d). Virtex-II Platform FPGA User Guide. Xilinx Inc., San Jose, CA. `www.xilinx.com`.

[Xilinx, 2007e] Xilinx (2007e). Virtex-II Pro and Virtex-II Pro X FPGA User Guide. Xilinx Inc., San Jose, CA. `www.xilinx.com`.

[Xilinx, 2008a] Xilinx (2008a). Virtex-5 Configuration Guide Xilinx Inc., San Jose, CA. www.xilinx.com.

[Xilinx, 2008b] Xilinx (2008b). Virtex-5 User Guide. Xilinx Inc., San Jose, CA. www.xilinx.com.

[Xilinx and ISR, 2006] Xilinx and ISR (2006). Press Release: ISR and Xilinx Roll Out Ready-to-Wear SDR. Xilinx Inc., San Jose, CA. http://www.fpgajournal.com/articles_2006/20060228_sdr.htm.

[Xilinx Inc., 2004a] Xilinx Inc. (2004a). Two Flows for Partial Reconfiguration: Module Based or Difference Based. XAPP290 (v1.2) September 9, 2004.

[Xilinx Inc., 2004b] Xilinx Inc. (2004b). Two Flows for Partial Reconfiguration: Module Based or Difference Based. XAPP290 (v1.2) September 9, 2004.

[Xilinx Inc., 2005a] Xilinx Inc. (2005a). Defense and Aerospace. Presentation.

[Xilinx Inc., 2005b] Xilinx Inc. (2005b). Radiation Effects and Mitigation Overview. Presentation.

[Xilinx Inc., 2005c] Xilinx Inc. (2005c). Virtex-II Pro and Virtex-II Pro X FPGA User Guide. Datasheet. UG012 (v4.0) March 23, 2005.

[Xilinx Inc., 2006a] Xilinx Inc. (2006a). OPB HWICAP (v1.00.b) Product Specification DS280 July 26, 2006.

[Xilinx Inc., 2006b] Xilinx Inc. (2006b). Press Release: ISR and Xilinx Roll Out Ready-to-Wear SDR. Xilinx Inc., San Jose, CA. www.fpgajournal.com.

[Xilinx Inc., 2007a] Xilinx Inc. (2007a). Difference Based Partial Reconfiguration. XAPP290 (v2.0) December 3, 2007.

[Xilinx Inc., 2007b] Xilinx Inc. (2007b). Virtex-II Pro and Virtex-II Pro X FPGA User Guide. Datasheet. UG012 (v4.2).

[Xilinx Inc., 2007c] Xilinx Inc. (2007c). XPS HWICAP (v1.00.a) Product Specification DS586 November 05, 2007.

[Xilinx Inc., 2008] Xilinx Inc. (2008). Spartan-3 Generation FPGA User Guide,. www.xilinx.com.

[Xilinx Inc., 2009a] Xilinx Inc. (2009a). http://www.xilinx.com/ise/optional_prod/cspro.htm.

[Xilinx Inc., 2009b] Xilinx Inc. (2009b). http://www.xilinx.com/products/devkits/HW-V5-ML507-UNI-G.htm.

[Xilinx Inc., 2009c] Xilinx Inc. (2009c). http://www.xilinx.com/products/devkits/HW-V5-ML505-UNI-G.htm.

[Xilinx Inc., 2009d] Xilinx Inc. (2009d). http://www.xilinx.com/products/devkits/HW-V4-ML401-UNI-G.htm.

[Xilinx Inc., 2010a] Xilinx Inc. (2010a). Virtex-5 FPGA Configuration Guide, v3.9.1. www.xilinx.com.

[Xilinx Inc., 2010b] Xilinx Inc. (2010b). Xilinx Demonstrates Cost and Power Reductions for 40G OTN Muxponder With Partial Reconfiguration Technology. http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1402650&highlight=.

[Y. Lin, 1997] Y. Lin (1997). Recent Developments in High Level Synthesis. *ACM Transactions on Design Automation*.

[Ye et al., 2000] Ye, Z. A., Moshovos, A., Hauck, S., and Banerjee, P. (2000). CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. *International Symposium on Computer Architecture*, pages 225–235.

List of Publications

Within the context of the Ph.D research:

1. Papademetriou, K. and Dollas, A. (2006a). A Task Graph Approach for Efficient Exploitation of Reconfiguration in Dynamically Reconfigurable Systems. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 307–308

2. Papademetriou, K. and Dollas, A. (2006b). Performance Evaluation of a Preloading Model in Dynamically Reconfigurable Processors. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 901–904

3. Papadimitriou, K., Anyfantis, A., and Dollas, A. (2007). Methodology and Experimental Setup for the Determination of System-level Dynamic Reconfiguration Overhead. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 335–336

4. Papadimitriou, K., Anyfantis, A., and Dollas, A. (2010). An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems. *IEEE Transactions on Instrumentation and Measurement (TIM)*, 59(6):1642–1651

5. Ilias, A., Papadimitriou, K., and Dollas, A. (2010). Combining Duplication, Partial Reconfiguration and Software for On-line Error Diagnosis and Recovery in SRAM-based FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–76

6. Papadimitriou, K., Dollas, A., and Hauck, S. (2011). Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model. *ACM Transactions on Reconfigurable Technoclogy and Systems (TRETS)*, 4(4):1–27

7. Gholamipour, A., Papadimitriou, K., Kurdahi, F., Dollas, A., and Eltawil, A. (2011). Area, Reconfiguration Delay and Reliability Trade-Offs in Designing Reliable Multi-

Mode FIR Filters. In *Proceedings of the IEEE International Design and Test Workshop (IDT)*

Less related to the Ph.D research:

8. Dollas, A., Papademetriou, K., Sotiriades, E., Theodoropoulos, D., Koidis, I., and Vernardos, G. (2004). A Case Study on Rapid Prototyping of Hardware Systems: the Effect of CAD Tool Capabilities, Design Flows, and Design Styles. In *International IEEE Workshop on Rapid System Prototyping (RSP)*, pages 180–186

9. Vavouras, M., Papadimitriou, K., and Papaefstathiou, I. (2009b). Implementation of a Genetic Algorithm on a Virtex-II Pro FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 287–287

10. Vavouras, M., Papadimitriou, K., and Papaefstathiou, I. (2009a). High-speed FPGA-based Implementations of a Genetic Algorithm. In *IEEE International Conference on Embedded Computer Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 9–16

11. Effraimidis, C., Papadimitriou, K., Dollas, A., and Papaefstathiou, I. (2009). A Self-Reconfiguring Architecture Supporting Multiple Objective Functions in Genetic Algorithms. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 453–456