Diss. ETH No. 15423

Architectural Trade-offs in Dynamically Reconfigurable Processors

A dissertation submitted to the

Swiss Federal Institute of Technology
Zurich

for the degree of
Doctor of Technical Sciences

presented by

Rolf Enzler

Dipl. El.-Ing. ETH
born 30 September 1971
citizen of Walchwil ZG

accepted on the recommendation of

Prof. Dr. Gerhard Tröster, examiner
PD Dr. Marco Platzner, co-examiner

2004

To Susana

Acknowledgments

I am grateful to my advisor, Prof. Dr. Gerhard Tröster, for his support and for providing me with extraordinary research facilities.

Special thanks belong to Christian Plessl and Dr. Marco Platzner for the uncountable hours of fruitful discussions and joint work within the research project ZIPPY. Further, I want to thank Marco for co-examining this thesis and for providing his valuable input.

I would like to thank Dr. Hubert Kaeslin and Francisco Camarero of the Microelectronics Design Center at ETH Zurich for their technological advice and the support with the Synopsys synthesis tools.

I am very much obliged to the members of the Electronics Laboratory and the Computer Engineering and Networks Laboratory – way too many to name all of them personally – for the pleasant and inspiring research atmosphere. Special thanks go to Didier Cottet who always had an open ear for my diverse requests.

I also want to thank Adrian M. Whatley for the careful proofreading of the manuscript.

Finally, I would like to express my gratitude to my wife Susana as well as to my parents; without their support and encouragement this work would not have been possible.

Zurich, January 2004 Rolf Enzler

Contents

Abstract

This dissertation deals with the design and evaluation of dynamically reconfigurable processor architectures targeting the embedded computing domain. The main objective is the investigation of the architectural trade-offs involved in terms of computational performance and chip area.

Reconfigurable architectures promise to be a valuable alternative to conventional computing devices such as processors and application-specific integrated circuits. The hardware of reconfigurable architectures is, in contrast to processors or application-specific integrated circuits, not static but adapted to the applications at hand. Through dynamic hardware customization, reconfigurable architectures potentially achieve a higher efficiency than processors while maintaining a higher level of flexibility than application-specific integrated circuits. This work focuses on hybrid, dynamically reconfigurable processors that couple a standard CPU core with a reconfigurable processing unit.

The development of such technology includes a multitude of design decisions and architectural trade-offs. In order to study these issues, we propose a system-level evaluation methodology that allows a designer to measure the computational performance of hybrid reconfigurable processors and to estimate their chip area. The methodology is based on a hybrid architecture model, a system-wide, cycle-accurate simulation environment, and a parameterized area estimation model. Our approach enables a designer to investigate the system-level impact of specific architectural design features.

In order to characterize the targeted embedded computing domain, we analyze a pool of applications that represents a typical embedded workload. We distinguish between the application groups multimedia, cryptography, and communications. The analysis shows that the selected embedded workload differs significantly from a general-purpose workload. Furthermore, the analysis reveals that the three particular application groups compared to each other also feature distinctive characteristics and hence stress different architectural features of a processor. The consequence is that a reconfigurable processor design must account for the peculiarities of the targeted application domain.

For the class of data-streaming applications, which is part of the multimedia group, we present a hybrid, dynamically reconfigurable processor architecture that couples a multi-context, coarse-grained reconfigurable array as a coprocessor to a CPU core. The CPU is responsible for the data transfer, context loading, and control of the reconfigurable array. The array stores several configurations on-chip and thus allows for fast adaptation of its functionality. The envisioned programming model makes use of hardware virtualization, which allows for the abstraction of limited reconfigurable hardware resources.

In a case study, we implemented large finite impulse response filters and quantitatively investigated various design features of the reconfigurable processor. The experiments show that hardware virtualization is a suitable programming model, and that multi-context devices can successfully be employed in this regard. The results further prove that hybrid multi-context architectures have the potential to yield significant speedups. In our experiments, we achieved speedups of up to an order of magnitude over the stand-alone CPU at the expense of moderate area overheads. Overall, the case study emphasizes the importance of our system-level evaluation approach.

Zusammenfassung

Die vorliegende Arbeit befasst sich mit dem Entwurf und der Evaluation dynamisch rekonfigurierbarer Prozessorarchitekturen im Bereich eingebetteter Systeme. Das Hauptziel der Arbeit besteht dabei in der Untersuchung der Design-Tradeoffs hinsichtlich Rechenleistung und Chipfläche.

Rekonfigurierbare Architekturen stellen eine vielversprechende Alternative zu konventionellen Rechenbausteinen dar, zu denen Prozessoren und applikationsspezifische integrierte Schaltungen gehören. Die Hardware rekonfigurierbarer Architekturen ist im Gegensatz zu Prozessoren oder applikationsspezifischen integrierten Schaltungen nicht statisch, sondern kann dynamisch der aktuellen Applikation angepasst werden. Durch ihre parallelen Hardwarestrukturen erzielen rekonfigurierbare Bausteine eine potenziell höhere Rechenleistung als Prozessoren. Gleichzeitig ermöglicht die dynamische Adaption eine grössere Flexibilität im Vergleich zu anwendungsspezifischen integrierten Schaltungen. Die vorliegende Arbeit konzentriert sich auf hybride, dynamisch rekonfigurierbare Prozessoren, welche eine Standard-CPU mit einer rekonfigurierbaren Recheneinheit koppeln.

Bei der Entwicklung solcher Architekturen sind eine Vielzahl von Designentscheidungen und Tradeoffs zu berücksichtigen. Um die damit verbundenen Fragestellungen zu untersuchen, präsentieren wir eine Evaluationsmethodik auf Systemebene, mit der die Rechenleistung rekonfigurierbarer Prozessoren gemessen sowie deren Chipfläche abgeschätzt werden kann. Die Methodik basiert auf einem hybriden Architekturmodell, einer systemweiten, zyklengenauen Simulationsumgebung und einem parametrisierten Modell für die Flächenabschätzung. Mit Hilfe unseres Ansatzes kann der Einfluss der spezifischen Merkmale einer Prozessorarchitektur auf Systemebene untersucht werden.

Um das Anwendungsgebiet der eingebetteten Systeme zu charakterisieren, analysieren wir eine Reihe von Benchmark-Applikationen, welche eine für eingebettete Systeme typische Arbeitslast darstellen. Wir unterscheiden dabei die Applikationsgruppen Multimedia, Kryptographie und Kommunikation. Die Analyse belegt, dass sich der ausgewählte Applikationsmix deutlich von Applikationen aus dem General-Purpose-Bereich unterscheidet. Darüber hinaus zeigt sich, dass auch die

drei untersuchten Applikationsgruppen im Vergleich zueinander verschiedene Charakteristika aufweisen und folglich unterschiedliche Merkmale eines Prozessors akzentuieren. Die Konsequenz ist, dass beim Entwurf rekonfigurierbarer Prozessoren das Anwendungsgebiet mitberücksichtigt werden muss.

Für die Klasse der Streaming-Applikationen, die eine Teilgruppe der Multimedia-Anwendungen darstellt, entwickeln wir eine rekonfigurierbare Prozessorarchitektur, welche ein rekonfigurierbares, grobgranulares Multikontext-Array als Coprozessor an einen CPU-Kern koppelt. Die CPU ist dabei verantwortlich für den Datentransfer, das Laden der Kontexte und die Kontrolle des rekonfigurierbaren Arrays. Das Array speichert mehrere Konfigurationen auf dem Chip und ermöglicht dadurch eine schnelle Adaption der Funktionalität. Das vorgesehene Programmiermodell basiert auf Hardwarevirtualisierung und erlaubt, limitierte rekonfigurierbare Hardwareressourcen zu abstrahieren.

In einer Fallstudie diskutieren wir die Implementierung von nichtrekursiven Filtern und untersuchen verschiedene Architekturmerkmale rekonfigurierbarer Prozessoren. Die Experimente belegen, dass Hardwarevirtualisierung ein geeignetes Programmiermodell darstellt, und dass Multikontext-Bausteine in diesem Zusammenhang erfolgreich verwendet werden können. Die Resultate zeigen des Weiteren, dass hybride Multikontext-Architekturen das Potenzial haben, die Rechenleistung beachtlich zu erhöhen. In unseren Experimenten erreichen wir Verbesserungen von bis zu einer Grössenordnung im Vergleich zur eigenständigen CPU, bei moderat höherem Flächenbedarf. Insgesamt unterstreicht die Fallstudie die Wichtigkeit unseres Evaluationsansatzes auf Systemebene.

1

Introduction

Designers of digital systems face a fundamental trade-off between flexibility and efficiency when selecting computing elements. The available alternatives span a wide spectrum with general-purpose microprocessors and application-specific integrated circuits (ASICs) at opposite ends. Microprocessors are used in personal computers, in workstations, in servers, as building blocks for contemporary supercomputers, and increasingly in embedded systems. They are flexible due to their versatile instruction sets that allow the implementation of any computation task. ASICs on the other hand are dedicated hardware circuits tuned to a very small number of applications or even to just one task. ASICs are mainly used in high-volume embedded system markets such as telecommunications, consumer electronics, or the automotive industry. For a given task, dedicated circuits execute faster, require less silicon area, and are more power efficient than general-purpose architectures. The drawback of such highly specialized architectures is their lack of flexibility – if the application changes a redesign of the ASIC is required.

In the last decade, the new class of reconfigurable computing devices has emerged, which promises to combine the flexibility of processors with the efficiency of ASICs [155, 168, 171]. The hardware of reconfigurable devices is not static but adapted to each individual ap-

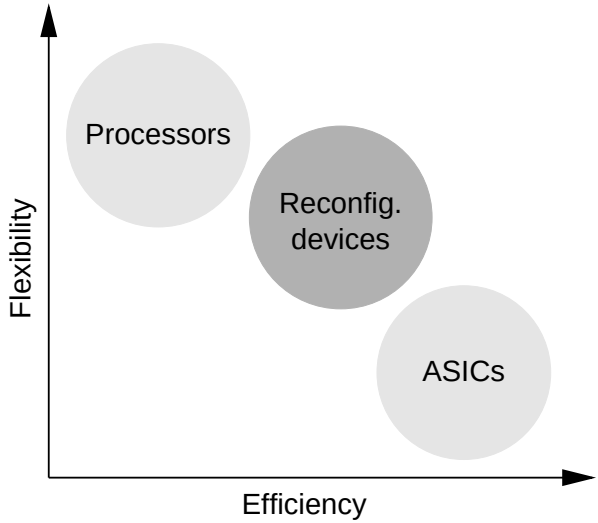
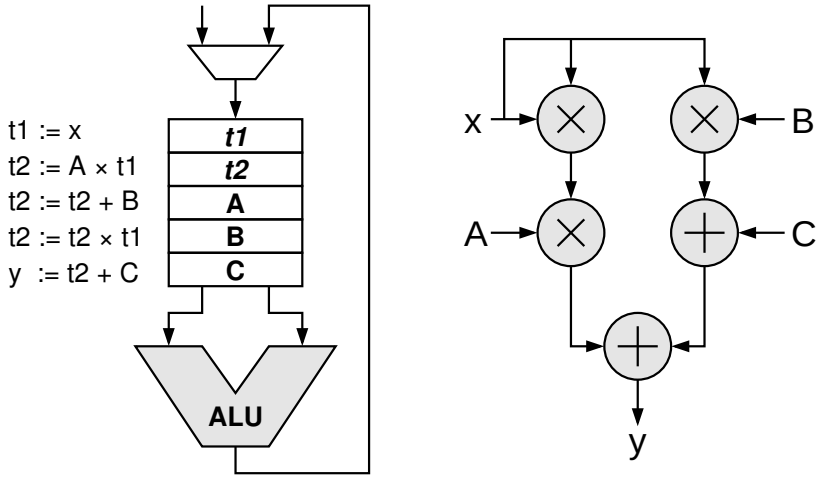


Figure 1.1. Reconfigurable computing devices promise to combine the flexibility of processors with the efficiency of ASICs

plication. Through hardware customization, reconfigurable devices potentially achieve a higher efficiency than microprocessors, while the dynamics of the customization process allow a higher level of flexibility than ASICs. Figure 1.1 outlines the trade-off between flexibility and efficiency as well as the position of reconfigurable devices compared to processors and ASICs.

1.1. Reconfigurable Computing Paradigm

Figure 1.2 sketches the computing paradigms of processors and ASICs, respectively. Processors have a general, fixed architecture that allows tasks to be implemented by *temporally* composing atomic operations, which are provided for example by the arithmetic and logic unit (ALU) or the floating-point unit. In contrast, ASICs implement tasks by *spatially* composing operations, which are provided by dedicated computational units like adders or multipliers. Reconfigurable computing combines these computing paradigms by means of reconfigurable hardware structures, which allow tasks to be implemented both “in time” and “in space”.



(a) Computation in time

(b) Computation in space

Figure 1.2. Computation of the expression $y = Ax^2 + Bx + C$

The characteristics of the computing paradigms are also reflected in the respective system compositions, outlined in Figure 1.3. In the processor case, the instructions (composed into the program code) define the behavior of the computing element. The behavior of a reconfigurable device is specified by its configuration. The behavior of an ASIC is typically hard-wired and does not allow for any dynamic adaptation, except maybe for some adjustable coefficients.

Within the domain of reconfigurable computing two fundamental kinds of (re-)configurability are distinguished [90, 142]:

- *static* or *compile-time reconfiguration (CTR)* – where the configuration of the device is loaded once at the outset, after which it does not change during the execution of the task at hand, and
- *dynamic* or *run-time reconfiguration (RTR)* – where the configuration of the device may change at any arbitrary moment during run time.

This work focuses on dynamically reconfigurable devices. Static reconfiguration is thus not further considered and discussed.

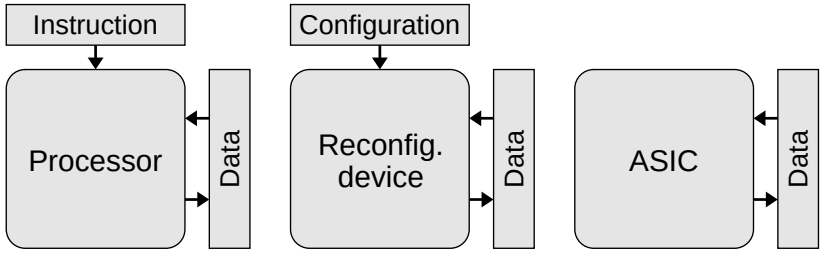


Figure 1.3. System outlines corresponding to the computing paradigms of processors, reconfigurable devices and ASICs

1.2. Reconfigurable Systems

The enabling technology for building reconfigurable systems was the field-programmable gate array (FPGA) [26]. FPGAs were introduced to the market at the high-end of programmable logic devices (PLDs) in the mid 1980s. FPGAs consist of an array of logic blocks, routing channels to interconnect the logic blocks, and surrounding I/O blocks. SRAM-based FPGAs use static RAM (SRAM) cells to control the functionality of the logic, I/O blocks, and routing. They can be reprogrammed in-circuit arbitrarily often by downloading a bitstream of configuration data to the device. Typically, FPGAs are fine-grained architectures that operate on bit-wide data types and use look-up tables (LUTs) as computing elements. While early FPGA generations were quite limited in their capacity, today's devices feature millions of gates of programmable logic, dense enough to host complete computing systems.

Currently, a strong trend towards *hybrid reconfigurable processors* can be observed. Hybrid architectures combine standard CPU cores with arrays of field-programmable elements. Several of these new reconfigurable processors are entering the commercial market and claim to offer three benefits:

- *Functionality on demand* — Typical examples are network processors with hardware modules for tasks such as protocol handling. The modules can be adapted to protocol standards that were not fully specified or not even known at design time. Another example is that of set-top boxes for digital TV and Internet connection that can be equipped with decoding hardware on demand by the user. As there is a strong tendency to network all

kinds of embedded systems, remote and thus fast and inexpensive hardware update is a highly desirable technology.

- *Acceleration on demand* — The reconfigurable processing unit of the hybrid architecture serves as a coprocessor and accelerates the critical parts of an application. This is especially interesting for computationally demanding applications found in areas such as multimedia, cryptography, and communications.
- *Shorter time to market* — Products that eventually target ASIC platforms can be released earlier using reconfigurable hardware. In many market segments, the early market entry compensates for the more expensive and power-hungry nature of the initial product series.

Table 1.1 lists a selection of commercial reconfigurable processors and summarizes their main architectural characteristics. Apart from these, Elixent (www.elixent.com) offers a coarse-grained reconfigurable intellectual property (IP) core called D-Fabrix [160], which is intended for integration with a CPU and further system components such as memory. Further, QuickSilver Technology (www.quicksilvertech.com) has announced the adaptive computing machine (ACM) architecture [152] consisting of heterogeneous, hierarchically interconnected clusters. Each cluster contains different types of computational nodes including CPU cores, coarse-grained arithmetic nodes, fine-grained bit-manipulation nodes, and finite state machine nodes.

1.3. Dynamic Reconfiguration

A crucial parameter for any dynamically reconfigurable computing system is the *reconfiguration time*, i. e. the time it takes to switch the functionality of the reconfigurable device. The reconfiguration time imposes limits on the applications that can be efficiently mapped to the reconfigurable hardware. Commodity reconfigurable devices, in particular FPGAs, show relatively long reconfiguration times in the range of dozens of milliseconds [197]. Consequently, rather long-running application functionality is mapped to such devices. For shorter term functionality the reconfiguration overhead can be significant, negating any performance gain over software. The trend towards ever larger devices aggravates the problem further, because the reconfiguration time is proportional to the amount of configuration data, which grows with the device size.

Table 1.1. Selection of commercial reconfigurable processors

	Atmel FPSLIC www.atmel.com	Triscend E5 www.triscend.com	Triscend A7 www.triscend.com	Altera Excalibur ARM www.altera.com	Altera Excalibur MIPS www.altera.com	Chameleon Systems CS2000 www.chameleonsystems.com	Xilinx Virtex-II Pro www.xilinx.com	PACT SMeXPP www.pactcorp.com	PACT SDRXPP www.pactcorp.com
Year	2000	2000	2001	2001	2001	2001	2002	>2003[1]	>2003[1]
CPU									
Core	AVC	8032	ARM7	ARM9	MIPS32	ARC	PowerPC	ARM7	ARM11
Frequency in MHz	40	40	60	166	166	125	300	104	n/a
Word width in bits	8	8	32	32	32	32	32	32	32
Reconfigurable array									
Granularity	fine	fine	fine	fine	fine	coarse	fine	coarse	coarse
FPGA family	AT40K	pty.[2]	pty.[2]	APEX	APEX	–	Virtex-II	–	–
Max. element count	2304	3200	3200	38 400	38 400	84	50 832	20	48
Max. gate count	40K	40K	40K	1M	1M	–	4M	–	–

[1] announced in 2003; release date undetermined
[2] proprietary technology

The data required to configure a reconfigurable device is commonly denoted as a *context*. Depending on the capabilities of the device, two basic classes are distinguished. *Single-context devices* store exactly one configuration on the chip. Before a new context can execute, the corresponding configuration data has to be loaded from off-chip. Conventional FPGAs fall into this category. *Multi-context devices* hold a set of configurations on-chip. At any given time exactly one configuration is in use: the so-called active context. To execute a new context, the contexts are switched, i. e. a previously inactive, stored context becomes active. Since the configuration data does not have to be loaded from off-chip, the context switch is significantly sped up. Commercial efforts include the SIDSA FIPSOC device [60, 62], the Chameleon Systems CS2000 device [141, 200], and the NEC DRP device [119].

Fast reconfiguration enables models of computation that are not, or only rudimentarily achievable using conventional reconfigurable devices:

- *Time sharing* — Several applications compete for the reconfigurable hardware resources. Each application receives a certain share of the execution time and the applications are sequentially swapped in and out [46, 172].

- *Hardware virtualization* — An algorithm is partitioned into several parts, each implemented by an individual circuit and executed sequentially on the reconfigurable device. This allows a reconfigurable device of arbitrary size to be emulated, e. g. [32, 34, 47, 147, 172].

- *Functionality on demand* — The functionality of the reconfigurable hardware is switched when required, i. e. at arbitrary, not predefined points in time, e. g. [89, 120, 191].

- *Acceleration on demand* — The reconfigurable hardware accelerates critical parts of an algorithm by customizing datapaths and operators and by massive parallel processing, e. g. [11, 23, 71, 116, 128, 135].

- *Dynamic adaptation* — The algorithm is adapted at run time depending on the incoming data. Different scenarios are feasible, from adapting coefficients to dynamically generating whole circuits. Examples are neural networks [20, 31, 54], adaptive filters [44, 51, 183] and constant propagation [186, 192].

Table 1.2. Comparison between RISC, DSP, FPGA and ASIC implementations of the IDEA cryptography algorithm by Mencer et al. [114]

Type / Device	Technology μm	Clock rate MHz	Performance Mbit/s	Power W	Efficiency Mbit/J
RISC SA-110	0.35	200	32.0	1.0	32.0
DSP TMS320C6x	0.25	200	53.1	6.0	8.9
FPGA XC4020XL	0.35	33	528.0	3.2	167.6
ASIC VINCI [42]	1.20	25	177.8	1.5	118.7

1.4. Efficiency of Reconfigurable Systems

Various case studies have shown that field-programmable devices can achieve higher throughput and be more energy efficient than processors, provided that the application matches well the spatial structure of the reconfigurable device and possesses a sufficient degree of parallelism.

The first reconfigurable systems for which remarkable performance was reported were Splash 2 [10, 68] and DECPeRLe-1 [181] in the early 1990s. These systems proved to achieve higher computational performance than any other architecture for applications requiring highly parallel, bit-level operations. The Splash 2 system, for example, outperformed any contemporary supercomputer implementation of genetic string matching by over two orders of magnitude [88]. Similar results were reported for other applications implemented on these systems such as image and signal processing, video compression, and computer vision [135, 178, 181]. More recent applications that exhibit significant speedups using reconfigurable hardware include for example cryptography [53], automatic target recognition [92, 94, 138], Boolean satisfiability [128, 131], and software-defined radio [41, 50, 157].

Mencer et al. [114] compared different implementations of the IDEA cryptography algorithm on a reduced instruction set computer (RISC), a digital signal processor (DSP), an FPGA, and an ASIC. Table 1.2 lists throughput, power dissipation and throughput per power. The FPGA achieved the highest computational performance and energy efficiency. Abnous et al. [2] performed similar studies on finite and infinite impulse response filters (FIR, IIR). The FPGA achieved a better energy efficiency than the embedded RISC CPU. The DSP outperformed the FPGA in terms of energy efficiency, because FIR and IIR filters perfectly match the DSP architecture.

Stitt et al. [162] studied the energy efficiency of hybrid reconfigurable processors and evaluated a set of benchmarks that are relevant to embedded computing. On the Triscend E5 and A7 devices [98, 189], Stitt et al. measured average energy savings of 71% and 53% respectively, by moving application kernels to the FPGA instead of running the applications exclusively on the CPU. The authors estimated that energy savings would increase to 89% and 75% respectively if the Triscend devices supported voltage scaling.

1.5. Research Objectives

Although reconfigurable computing has recently gained increasing attention in the research community, a number of important research issues remain open. The major issues with respect to hybrid, dynamically reconfigurable processors can be classified as falling into the following categories:

- the design of the reconfigurable processing unit (RPU),
- the integration of RPU and CPU into the system architecture,
- the programming model and the compilation technology, and
- the evaluation at the system level.

This work emphasizes the architectural trade-offs involved in the design of hybrid, dynamically reconfigurable processor architectures. The work has evolved out of the research project ZIPPY carried out at the Swiss Federal Institute of Technology (ETH) Zurich [207]. The main novelties compared to existing work in the field relate to the targeted application domain, the system-level evaluation approach, and based on this, the development of a hybrid reconfigurable processor:

- In contrast to many approaches investigating reconfigurable technology, we are aiming not at the general-purpose but the embedded computing domain, in particular handheld and wearable computing. Compared to general-purpose computing, the embedded domain puts more stringent requirements on computing power, energy consumption, costs, weight, volume, etc. and stresses the trade-offs with respect to these objectives. In order to represent a typical workload, we have assembled an application pool with applications from the fields of multimedia, cryptography, and communications. The quantitative characteristics of these application fields directly impact the design of our reconfigurable processor.

- While there already exists a substantial body of work on reconfigurable architectures including hybrid approaches, a system-level evaluation of the performance and the various features of the reconfigurable devices is missing. We propose a methodology that allows for system-wide, cycle-accurate co-simulation of hybrid reconfigurable processors. Together with a parameterized area estimation model this enables us to study the architectural trade-offs involved in the processor design.

- We elaborate a hybrid reconfigurable processor architecture targeting data-streaming applications that map well to macro-pipelines. We construct a reconfigurable hybrid system by coupling a coarse-grained, multi-context reconfigurable array as a coprocessor to a CPU core. The CPU takes care of data I/O, context loading, and control of the reconfigurable array. The reconfigurable array stores several configurations on the chip, which allows it to quickly change functionality. As a programming model we propose and investigate hardware virtualization, which abstracts the limited reconfigurable hardware resources.

With respect to the compilation technology, we rely on a library-based approach to generate code for the CPU and the RPU. This is a common approach used in hybrid reconfigurable systems. Compilation technology and code generation for reconfigurable hardware is a relevant research field in its own right within the area of reconfigurable computing. We have been careful to ensure that our methodology is extensible and that it allows automatic compilation technology to be incorporated later.

1.6. Overview

Chapter 2 reviews the major challenges in developing reconfigurable technology, in particular hybrid reconfigurable processors, and discusses related work.

Chapter 3 presents the proposed evaluation methodology, which allows us to quantitatively investigate the trade-offs involved in the design of reconfigurable processor architectures.

Chapter 4 provides a quantitative characterization of the targeted application domain of embedded systems and discusses the impact on the design of reconfigurable technology.

Chapter 5 elaborates the hybrid, multi-context, reconfigurable processor architecture aimed at data-streaming applications and discusses hardware virtualization as the envisioned programming model.

Chapter 6 presents experiments and evaluation results based on the proposed evaluation methodology as well as the elaborated reconfigurable architecture.

Chapter 7 concludes the work with a summary, a list of achievements, and an outlook.

2

Design Issues of Reconfigurable Processors

Hybrid reconfigurable processors couple reconfigurable elements tightly with a CPU. Major challenges in developing a hybrid reconfigurable processor are the design of the reconfigurable processing unit (RPU), its integration with a CPU into the system architecture, and the programming model of the hybrid processor.

This chapter discusses the design issues and outlines the state of the art in designing reconfigurable processor technology. First, the concepts for integrating reconfigurable hardware into a computing system are presented. Then the crucial architectural design parameters of dynamically reconfigurable arrays are identified and described. Programming models and compilation technology for hybrid reconfigurable processors are broad research areas in their own right. The major issues are outlined and some pointers to further research are provided. Finally,

approaches for evaluating reconfigurable processors at the system level are discussed.

In this chapter, we focus on the relevant topics in the context of this work. For a comprehensive survey and various approaches to classification of reconfigurable systems we refer to [15, 22, 37, 76, 134, 145, 168]. Table 2.1 shows a selection of processors which are prominent in research. The table lists the year of the first major publication, the envisioned application domain, the integrated CPU core, and the status of the processor development. The outlined design features are discussed in this chapter. The status of the projects is either concept, simulation, emulation on FPGA-based logic emulators, or VLSI implementation.

2.1. System Integration

CPU and RPU are two computing units with various possible interactions. The architectural integration of these units concerns

- the coupling between CPU and RPU,
- the way the CPU issues instructions to the RPU,
- the synchronization between CPU and RPU, and
- the way operands are transferred between CPU and RPU.

2.1.1. Coupling

The coupling between the CPU core and the RPU determines the type of applications that benefit most from the hybrid reconfigurable processor. Generally, a tighter coupling leads to a smaller communication overhead. Loose couplings thus require bigger amounts of computation assigned to the RPU in order to operate efficiently. Couplings can be classified into three main categories, which Figure 2.1 illustrates.

- *Reconfigurable functional unit (RFU)* — This is the tightest coupling that can be achieved. The RPU is integrated into the CPU core as a functional unit. Examples are PRISC [136], OneChip [33, 91, 195], and Chimaera [77, 202].

- *Reconfigurable coprocessor* — The RPU is part of the processor and placed next to the CPU core. Examples are Garp [29, 80], NAPA [140], REMARC [116], MorphoSys [105, 153], and OneChip98 [91].

Table 2.1. Selection of dynamically reconfigurable processors in research

	PRISC [136]	OneChip [195]	Chimaera [77, 202]	Garp [29, 80]	NAPA [140]	REMARC [116]	MorphoSys [105, 153]	OneChip98 [91]
Year	1994	1996	1997	1997	1998	1998	1998	1999
Application domain[1]	GP	GP	GP	GP	GP	MM	MM	GP
Status	concept	emulation	concept	simulation	simulation	simulation	VLSI	emulation
System integration								
CPU core	R2000	DLX	MIPS	MIPS-II	CompactRISC	MIPS-II	Tiny RISC[2]	S-DLX
Coupling	RFU	RFU	RFU	coproc.	coproc.	coproc.	coproc.	coproc.
Concurrent operation	inherent	inherent	inherent	yes	yes	yes	no	yes
Data transfer[3]	R	R	R	R/M	R/M/D	R	R/D	R/M
Reconfigurable unit								
Granularity	fine	fine	fine	fine	fine	coarse	coarse	fine
Multiple contexts	no	yes	yes	yes	no	yes	yes	yes
Context fetching[4]	L	P	C/P	P	L	L	P	P

[1] GP – general purpose, MM – multimedia
[2] CPU core by Abnous et al. [1]; not to be confused with TinyRISC by LSI Logic (www.lsilogic.com)
[3] R – register, M – memory hierarchy, D – dedicated memory port
[4] L – load on demand, C – configuration cache, P – prefetch

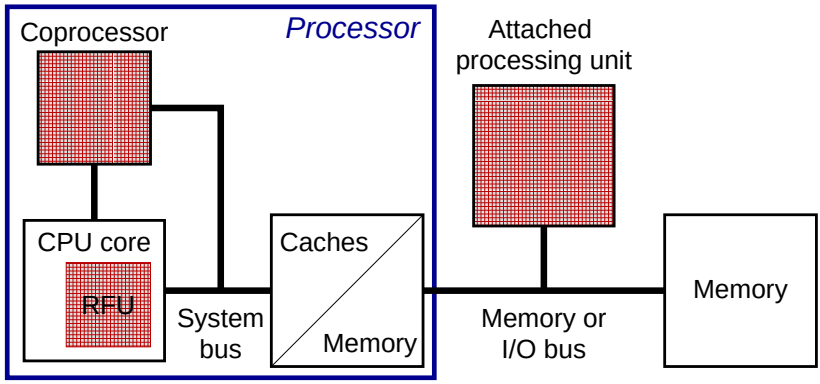


Figure 2.1. Possible couplings between the CPU core and the reconfigurable processing unit

- *Attached reconfigurable processing unit* — This is the loosest method of coupling. The RPU is located outside the processor and connected to a memory or I/O bus, e. g. the PCI bus. Unlike RFUs and coprocessors, the instruction set of the CPU core is not extended. Examples are Splash 2 [10], DECPeRLe-1 [181], PRISM [185], Teramac [7], and MORRPH [52].

The commercial processors listed in Table 1.1 are on the border between attached units and coprocessors. Although they are integrated with a CPU core, caches and memories on a system on a chip (SoC)[1], they are not as tightly coupled to the CPU as coprocessors. There are no instruction set extensions for the RPU.

In the remaining part of this chapter, we concentrate on RFU and coprocessor approaches.

2.1.2. Instruction Set Extension

Both RFU and coprocessor approaches extend the CPU's instruction set with customized instructions. The CPU core fetches and decodes instructions and issues the customized instructions to the corresponding unit. There are several types of such instructions:

[1] In this context sometimes referred to as configurable system on a chip (CSoC) or system on a programmable chip (SoPC).

- For RFUs two types of instructions exist: instructions that start the reconfiguration of the RFU and instructions that actually execute the RFU function.
- Instructions for coprocessors also include reconfiguration and execution instructions, and additionally, instructions that transfer data and synchronize the CPU with the RPU.

2.1.3. Synchronization

Synchronization is required whenever two computing elements operate concurrently. RFUs can operate concurrently with other functional units, because the CPU's control logic synchronizes activities and controls access to the register file. In Table 2.1 this is denoted as "inherent" concurrent operation.

In the case of coprocessors, simple approaches force the CPU to stall until RPU execution has completed. More advanced techniques allow concurrent operation and synchronize by means of status flags, semaphores or interrupt mechanisms.

While the RFU approach delivers the fastest interaction between CPU and RPU, it requires a major redesign of the CPU core. Coprocessors need less core redesign but may require more effort for synchronization.

2.1.4. Operand Transfer

An RFU uses the CPU register file to read and write operand data, the same way as any ordinary functional unit does. Coprocessors can use several options:

1. Data may be transferred between the coprocessor and the CPU via registers.
2. Coprocessors can have access to the same memory hierarchy as the CPU including several levels of caches, on-chip memories, and the external memory interface.
3. To increase the overall memory bandwidth some approaches equip the RPU with dedicated memory ports. While this certainly increases bandwidth, it can also lead to data consistency problems.

In Table 2.1, these options are denoted as "R" for register, "M" for memory hierarchy, and "D" for dedicated memory ports.

2.2. Reconfigurable Processing Unit

The main design parameters of an RPU are the granularity of the processing elements, the interconnect between the processing elements, and the reconfiguration mechanism.

2.2.1. Operators and Granularity

The granularity of the processing elements can be either fine-grained or coarse-grained:

- *Fine-grained* arrays use logic blocks with 1-bit to 4-bit inputs and single flip-flops. These structures are well suited to implement bit-manipulation operations and random logic. Examples are commercial FPGAs [6, 198] and FPGA-based architectures, e. g. [33, 91, 195]. Not FPGA-based architectures include PRISC [136], DPGA [47, 165], Chimaera [77, 202], and Garp [29, 80].
- *Coarse-grained* architectures accommodate 4-bit to 32-bit ALUs and registers and are better suited to implement regular arithmetic operations on word-sized data, e. g. found in multimedia applications. Examples are CHESS (4-bit datapath) [112], MorphoSys (8- or 16-bit datapath) [105, 153], REMARC (16-bit datapath) [116], and XPP (32-bit datapath) [16].

There is a trade-off involved in selecting the granularity of the processing elements as typical real-world workloads contain both fine-grained and coarse-grained types of applications. Several research groups have performed trade-off studies with respect to granularity. Kouloheris and El Gamal [99] studied the impact of logical block granularity on FPGA performance. Betz and Rose [19] investigated the optimal number of inputs provided by LUT-based logic blocks. Goldstein et al. [70] employed a parameterized model of an ALU array architecture to study key design parameters including the granularity of the processing element.

Configuration size

A parameter strongly related to the granularity is the configuration size. Given a certain silicon area for the RPU, many fine-grained elements or fewer coarse-grained elements can be implemented. A large number of

fine-grained elements requires more configuration data than a smaller number of coarse-grained elements.

The configuration size of the fine-grained Xilinx Virtex-II Pro family for example ranges between 1.24 Mbits and 41.58 Mbits [198], while the configuration size of the coarse-grained Chameleon Systems CS2000 device totals less than 50 Kbits [141].

Multi-granular processing elements

Research groups also investigate multi-granular elements that are well suited to implement bit-manipulation operators, but can also be efficiently arranged to suit word-level operations.

The RAW microprocessor [182] is composed of multiple identical tiles. Each tile incorporates an ALU and fine-grained reconfigurable logic. Waingold et al. claim that the small amount of reconfigurable logic in each tile – for which software can select the datapath width – allows a RAW processor to support multi-granular operation. However, the mechanism how the ALUs cooperate with the reconfigurable logic is not elaborated. In more recent research, the reconfigurable logic has been disregarded [166].

The CHESS architecture [112] is an ALU array aimed to be integrated into an ASIC or a processor datapath. The CHESS array features a chessboard-like floorplan alternating ALUs and switchboxes. The switchboxes can be converted into 16-word by 4-bit RAMs allowing them to be used as 4-input, 4-output LUTs for operations that do not map well onto the ALUs. This same feature can also be used to make arbitrary interconnections at the bit level, which are not supported by CHESS' 4-bit wiring. The commercial successor D-Fabrix [160] employs multiplexers for control and bit-level data operations.

Other research projects investigate heterogeneous arrays, i. e. arrays that incorporate different types of processing elements. Work on heterogeneous arrays can be classified into arrays that

- integrate functional blocks such as multipliers or digital signal processing blocks, e. g. [6, 81, 198],
- use two or more different sizes of LUTs, e. g. [39, 82],
- employ embedded RAMs as supplementary logic when not used as memory, e. g. [6, 38, 198], or
- contain both LUTs and product term arrays, e. g. [83, 97].

2.2.2. Interconnect

The structure of the reconfigurable array is not only determined by the granularity of the processing elements, but also by their interconnect. Reconfigurable elements are typically placed in 2-dimensional arrays [37]. The simplest interconnect scheme connects each element to its four neighbors horizontally and vertically. Additional buses may exist that for example connect all elements in a row and in a column.

Many interconnect architectures are hierarchically structured, e. g. [4, 173, 176, 204]. The reconfigurable array is divided into compounds, which themselves consist of several processing elements. Both the compounds that form the array and the elements that form a compound use their own interconnect. Compounds can also contain specialized resources, e. g. embedded memory blocks [6, 198].

The interconnect constitutes a dominant component in many reconfigurable devices in terms of area requirements as well as energy consumption. From an empirical review of conventional FPGA devices DeHon [48, 49] concludes that in FPGAs about 90% of the chip area is dedicated to the interconnect. By implementing a set of benchmark netlists onto a Xilinx XC4003 FPGA, Kusse and Rabaey [101] have found that 65% of the energy consumption is devoted to the interconnect. These results are also discussed by George and Rabaey [66].

2.2.3. Reconfiguration Mechanism

A crucial parameter of dynamically reconfigurable processors is the time it takes to reconfigure the device. The *reconfiguration time* depends on the configuration size and the location from which the configuration data is loaded. The goal is single-cycle reconfiguration, i. e. the whole reconfigurable array can be reprogrammed within a single clock cycle. This requires the configuration data to be stored on the processor, near the reconfigurable elements.

For *single-context devices*, i. e. devices that store one configuration on the chip, several improvements to the reconfiguration mechanism have been proposed, including partial reconfiguration [86, 151], configuration compression [78, 79, 206], configuration prefetching [75], pipeline reconfiguration [146, 147], configuration caching [205], and wormhole reconfiguration [21].

Multi-context devices represent another approach to overcoming the limitations imposed by long reconfiguration times. Instead of storing a single configuration, multi-context devices concurrently hold a set of

configurations on the chip. Holding several contexts on-chip has various advantages. Switching to another context is fast – potentially a single clock cycle. Furthermore, the latency of loading a context onto the device can often be hidden, partially or entirely, by writing a context while another context is active. The optimizations used in single-context devices can also be applied in multi-context devices. Research architectures include WASMII [107], DPGA [46, 47, 165], time-multiplexed FPGA [172], CSRC [106, 132, 144], DRLE [63, 64], and MorphoSys [153]. Commercial efforts include FIPSOC [60–62], CS2000 [141, 200], and DRP [119].

Depending on the capabilities of a reconfigurable device, different context fetching mechanisms can be applied. Table 2.1 distinguishes three techniques: load on demand ("L"), where the configuration is loaded at the time it is required; configuration caching ("C"), where the configuration memory is used as a cache that holds recently used contexts; and prefetching ("P"), where a context can be prefetched concurrently to the execution of the active context.

2.3. Programming Models and Compilers

Programming models for reconfigurable processors have not yet received sufficient attention. This must change, as the success of reconfigurable architectures strongly depends on reasonable programming models that allow for the construction of automated code generation tools. Consequently, a significant amount of research is dedicated to compilation technology and code generation for reconfigurable hardware, constituting a research field of its own within the reconfigurable computing domain [8, 37, 111, 190]. A comprehensive discussion of this topic is beyond the scope of this work. We confine the discussion to an outline of the state of the art and provide some pointers to existing research efforts.

Currently, commercial programming environments for reconfigurable systems typically consist of separate tool flows for the software and the hardware. Processor code and configuration data for the RPU are hand-crafted and wrapped into library functions that are linked with the user code. Library-based approaches are also commonly employed in research, e. g. [23, 45, 104, 109, 110, 113].

The next step is compilers that automatically generate code and configurations from a general-purpose programming language such as C. Such a compiler constructs a control flow graph from the source

program and then decides which operations will go into the RPU. Generally, inner loops of programs are good candidates for reconfigurable fabrics. For general-purpose code this leads to several problems: First, it is quite difficult to extract a set of operations with matching granularity at a sufficient level of parallelism. Second, inner loops of general-purpose programs often contain excess code, i. e. code that must be run on the CPU such as exceptions, function calls, and system calls. Efforts aimed at automatic code generation for reconfigurable architectures include [29, 69, 70, 95, 115, 121, 175, 182, 203].

Similar problems are also being faced and tackled by researchers in the fields of hardware/software codesign and compiler construction for very long instruction word (VLIW) architectures. RFUs have recently gained interest for VLIW architectures, where optimized compilers extract instruction-level parallelism and schedule customized functional units at compile time, e. g. [5, 28, 139, 145]. VLIW techniques have also been proposed to map program loops of general-purpose programs onto reconfigurable coprocessors [29, 30].

In this work, we rely on a library-based approach to generate code for the hybrid reconfigurable processor. However, we are careful that our methodology and environment are extensible in the way that more complex compiler layers can be constructed on top. This allows us to incorporate automatic compilation technology and to account for advances made in this research field.

2.4. System-Level Evaluation

While there already exists a substantial body of work on reconfigurable architectures, few attempts have been made with respect to system-level evaluation of computational performance. System-level simulation for performance evaluation has been used only for CPUs that integrate RFUs into their datapath. RFU instructions are integrated with CPU simulators by extending the simulated instruction set. The new instructions are then scheduled to the RFU.

Carrillo Esparza and Chow [33] describe the third-generation OneChip architecture as well as their simulation approach. To evaluate the OneChip architecture, an extension to the SimpleScalar CPU simulator [13] is used. The RFU is not modeled explicitly but treated as a black-box, i. e. no detailed micro-architecture simulation of the RFU is performed. Instead, the input/output behavior and the execution latency of each particular RFU configuration are specified. The func-

tional model of a particular RFU configuration comprises three main parameters:

- the operation latency, i. e. the number of execution cycles until the result is computed,
- the issue latency, i. e. the number of cycles before another operation can be issued on the same RFU resource, and
- a C code fragment, which specifies the functional behavior of the RFU configuration.

The separation of the execution latency into operation and issue latency allows the specification of pipelined configurations. An RPU configuration can for example take 20 cycles to complete, but – since the configuration is pipelined – a new computation can start every four cycles. The RFU is assumed to be optimal in the sense that the correlation between a configuration's functionality and its execution latency is neglected.

La Rosa et al. [102] propose a similar approach for the XiRisc architecture [108]. Similar to OneChip, the RFU configurations are characterized by their execution latency and their behavior, which is also specified by a C code fragment. To evaluate the performance of the XiRisc architecture, the RISC CPU simulator provided within the standard GNU project debugger (GDB) distribution is extended with the additional architectural features of XiRisc and with the support for measuring the execution cycles.

Ye et al. [202] describe a simulation approach for the Chimaera architecture [77]. As for OneChip, the SimpleScalar CPU simulator is extended and used for simulation. The C compiler they developed, which automatically maps groups of instructions onto RFU operations, is discussed in [203]. The operations mapped onto the RFU are also modeled by specifying the execution latency of each particular RFU configuration. Several latency models are presented, which model the execution latency in terms of CPU cycles. The simpler models are based on counting the number of original instructions that are replaced by the RFU operation. The more complex models are based on hand-mapping the RFU operations onto the reconfigurable array and measuring the number of transistor levels in the critical path. The simulation takes the overhead that is caused by loading the RFU configurations into account.

In contrast to these RFU-based architectures, the current work targets coprocessor architectures where the RPU is attached to the CPU's coprocessor interface. The CPU is responsible for the data transfer, context loading, and control of the RPU. The long execution latencies of the RPU together with data-dependent processing times as well as dynamic effects of the CPU pipeline and the caches render functional RPU models unrealistic. Thus, we apply a cycle-accurate co-simulation approach that explicitly models the RPU. The functionality of the RPU is determined by a configuration bitstream.

3

System-Level Evaluation Methodology

The main factors affecting the performance of a hybrid reconfigurable processor are the architecture of the CPU and the reconfigurable processing unit, the system integration mechanism, the compilation technology for CPU and RPU, and the applications to be executed. To quantitatively measure and evaluate the architectural trade-offs involved in the design of reconfigurable systems, a methodology is required that incorporates these factors.

This chapter proposes an evaluation methodology that allows cycle-accurate performance statistics and area estimates to be gathered at the system level. The computational performance is evaluated by means of a simulation environment, which has first been proposed in [56] and is also discussed in [58]. The area requirements are estimated by constructing an analytical area model.

A further important criterion, particularly in the embedded systems domain, is the energy consumption. However, the presented evaluation methodology does currently not incorporate an energy estimation tool. This topic must be seen as a viable extension. Some starting points are discussed.

3.1. Methodology Outline

Figure 3.1 outlines the proposed evaluation methodology aimed at hybrid reconfigurable processor architectures. The evaluation methodology consists of

- the *architecture model*, which specifies the structure and the architectural assumptions of the hybrid reconfigurable processors that we investigate;
- the *simulation environment*, which enables us to implement and compile applications onto the modeled reconfigurable architecture and to gather cycle-accurate performance statistics by simulation; and
- the *analytical area model*, which combines parameterized area models of architectural building blocks, such as SRAM memory, with data from VLSI synthesis of the RPU's processing elements in order to estimate the overall system area.

Based on the performance statistics and the area estimates, the architecture model can be improved and the applications under consideration can be optimized with respect to the architecture model. This leads to a stepwise refinement process.

To achieve performance evaluation at the system level, we integrate two cycle-accurate simulators into a system-wide co-simulation framework. The requirements for the CPU simulator are high efficiency, cycle accuracy, and the availability of a robust code-generation framework for compiling benchmarking applications. The requirements for the RPU simulator are cycle accuracy and the ability to specify the RPU architecture at various levels of abstraction including register–transfer level and structural level.

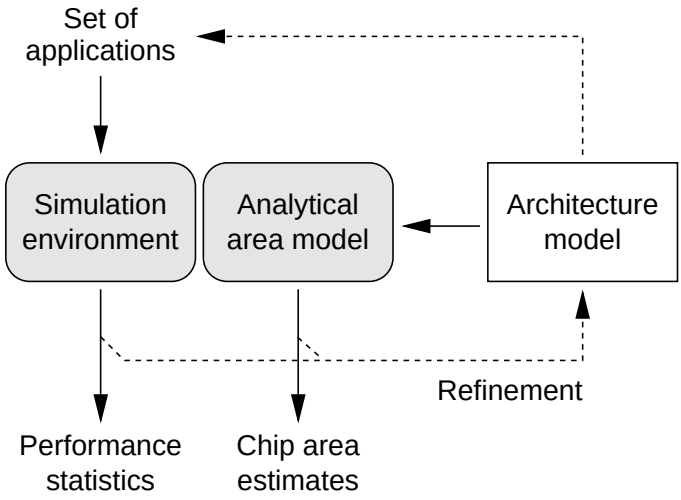


Figure 3.1. Methodology outline for evaluating the computational performance and the area requirements of hybrid reconfigurable processors

3.2. Architecture Model

Figure 3.2 outlines the architecture model of the hybrid reconfigurable processor. The architecture model is rather general since the only assumption is that the RPU is attached to a conventional RISC CPU core via a dedicated coprocessor interface. Data transfers, configuration loading, and execution control (synchronization of RPU and CPU) are performed using the coprocessor interface. To this end the RPU provides a number of coprocessor registers.

3.2.1. CPU Core

For the simulation of the CPU core and the memory architecture we use the *SimpleScalar* simulation tool suite [13]. SimpleScalar is written in the C language and provides a widely parameterized CPU model, which we incorporate into our architecture model.

The SimpleScalar CPU model is based on a 32-bit RISC architecture with a MIPS-IV like instruction set called PISA [27]. The instructions are extended to 64 bits in order to allow for experimental instruction set extensions. The parameters of the superscalar execution core include the number of functional units (integer and floating-point ALUs and

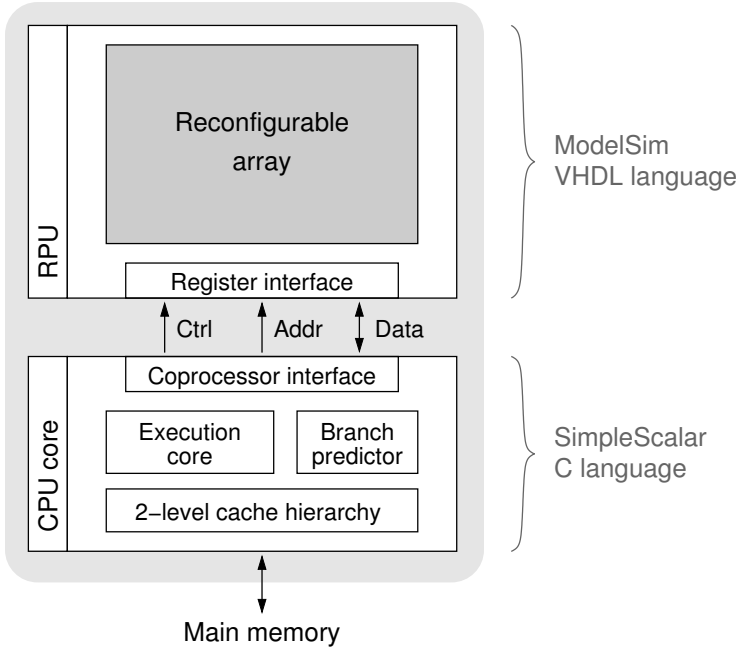


Figure 3.2. Architecture model and simulation outline of the hybrid reconfigurable processor architecture relying on a coprocessor coupling between CPU and RPU

multipliers), the sizes of the instruction fetch queue (IFQ), the register update unit (RUU), and the load/store queue (LSQ). The model supports several kinds of branch predictors, parameterized decode, issue and commit bandwidths, and optional in-order or out-of-order instruction issuing. The memory architecture can be customized by specifying a two-level cache hierarchy and the main memory access times. Overall, these parameterization facilities allow for performance evaluation of a broad spectrum of CPUs – ranging from small, embedded CPUs to high-end, superscalar CPUs.

Since our architecture model relies on a coprocessor coupling between CPU and RPU, we add a coprocessor interface to SimpleScalar and extend the PISA instruction set with appropriate coprocessor instructions. The coprocessor instructions allow the CPU to access the RPU's coprocessor register file.

3.2.2. Reconfigurable Processing Unit

The RPU is treated as a black box in the simulation framework. Basically, the RPU can be modeled at any level of abstraction, as for example described with the Y-chart by Gajski [65] or the system design model by Teich [167]. The only requirement is that the RPU model is specified by a formal, cycle-accurate description. We use the VHDL language for this purpose.

Figure 3.3 illustrates the levels of abstraction at which the RPU can be modeled. Although viable, our framework does not target the algorithmic level. There are more efficient approaches for this abstraction level. Examples are the OneChip [33] and Chimaera [202] projects, which integrate functionally modeled RFUs into the SimpleScalar CPU simulator. These functional models merely specify the input/output behavior and the required number of execution cycles of each particular RFU configuration. Our framework, in contrast, is well suited for the register–transfer level (RTL) and allows the VHDL model to be stepwise refined in order to develop a fully synthesizable RPU description. We assume in the following that the architecture of the RPU is modeled at least at the RTL level and that the RPU functionality is specified by the configuration (via the configuration memory) – similarly to SRAM-based FPGAs.

The simulation of the RPU is performed on the powerful *ModelSim* VHDL simulator. A conventional VHDL testbench can be used in standalone simulation mode to verify the RPU's functional correctness before integrating the model into the co-simulation environment. The decision to use VHDL as a specification language was driven by VHDL's mature development, simulation and synthesis tools. For the future, we consider the SystemC language [72, 124, 126] will be an interesting alternative, once the tool support for SystemC is on a comparable level to that currently available for VHDL.

3.3. Performance Simulation Environment

To enable a system-level performance evaluation, we integrate the two cycle-accurate simulators SimpleScalar and ModelSim into one co-simulation environment. This allows us to use the appropriate simulation tool for each simulation task, as indicated in Figure 3.2.

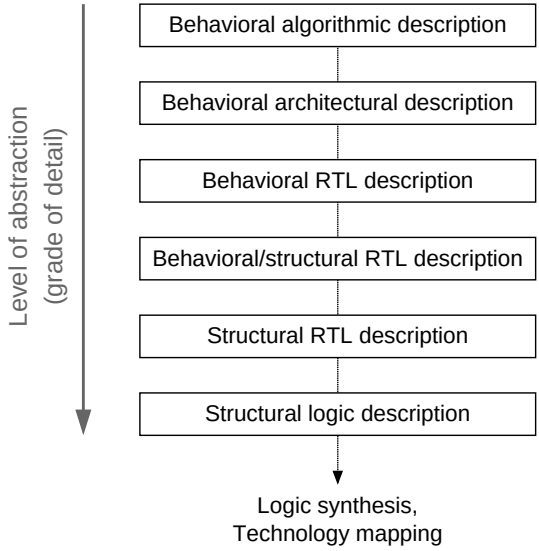


Figure 3.3. Principal refinement process of the RPU architecture model

3.3.1. Extended CPU Simulator

For the simulation of the CPU core and the memory architecture, we rely on the popular SimpleScalar tool suite [13]. SimpleScalar is an open source project available in C source code and allows modifications for academic purposes. As Figure 3.4 outlines, the tool set provides six simulators, which vary in their focus of architectural detail versus simulation speed. They range from a functional simulator (*sim-safe*) to a cycle-accurate simulator with a detailed microarchitectural timing model (*sim-outorder*). An important feature of SimpleScalar is its extensibility, which allows the adaptation or extension of the simulated processor architecture as well as the derivation of customized simulators. In this work, we use the original simulators, which are based on a MIPS-IV like instruction set called PISA. Only recently, the tool set has been enhanced to support several more instruction sets, including Alpha, PowerPC, x86, and ARM.

Since we are interested in cycle-accurate execution figures, we use the cycle-accurate *sim-outorder* simulator. SimpleScalar provides a C cross-compiler to compile the applications. The *sim-outorder* simulator then gathers detailed, cycle-accurate execution statistics by executing

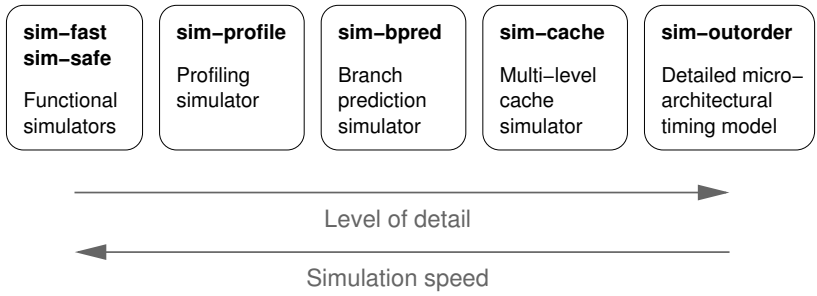


Figure 3.4. The various SimpleScalar simulators vary in their focus of architectural detail versus simulation speed

the compiled application binary on the specified CPU architecture. The CPU model is widely parameterized, as discussed in Section 3.2.1.

To support our coprocessor architecture model, we have extended the *sim-outorder* simulator to support a coprocessor interface and the corresponding coprocessor instructions. The coprocessor interface is modeled as an additional functional unit of the CPU with a dedicated I/O bus and its own I/O address space. This allows concurrent access on the memory and the coprocessor bus.

3.3.2. Co-simulation

For co-simulating the complete hybrid reconfigurable processor, SimpleScalar must be able to control the RPU simulation in VHDL. ModelSim supports the extension of its VHDL simulator through the so-called *foreign language interface* [117]. User-defined shared libraries can be loaded at startup of the VHDL simulator and provide access to the simulation kernel. Making use of this feature, we have implemented an interface that exposes complete control over the RPU simulation in ModelSim to the SimpleScalar simulator.

Technically, ModelSim and SimpleScalar run as two parallel processes. They communicate via commands exchanged using a shared memory area and semaphores which coordinate the accesses to the shared memory [161]. A VHDL testbench is built comprising the VHDL model of the RPU as well as an interface driver entity. By means of the shared memory interface, SimpleScalar accesses the ports of the interface driver and controls the ModelSim simulation task.

Whenever SimpleScalar encounters one of the coprocessor instructions, it relays the corresponding command to the ModelSim VHDL simulator, where the request is processed. The results are then sent back to SimpleScalar. For performance reasons we allow the simulation time on the ModelSim VHDL simulator to lag behind the SimpleScalar simulation time. On every communication event, the simulation times of SimpleScalar and ModelSim are re-synchronized.[1]

3.3.3. Stand-Alone Simulation Modes

Besides the co-simulation mode, in which the SimpleScalar and ModelSim simulators run concurrently, each simulator can run individually in stand-alone mode:

- SimpleScalar can run stand-alone simulating a "CPU only" model without attached coprocessor. This is convenient to gather CPU performance figures for reference purposes.
- ModelSim can run with a conventional VHDL testbench to simulate an "RPU only" model. This method is used for the verification of the RPU architecture model.

3.3.4. Application Mapping and Compilation

Mapping an application to the hybrid reconfigurable processor starts with partitioning the application into the parts that run on the CPU and the parts that run on the RPU. So far, this hardware/software partitioning step is performed manually. The same applies for the functional specification of the RPU configurations. The subsequent steps of generating the configuration bitstreams and compiling the application program are automated.

Figure 3.5 gives an overview of the co-simulation environment. SimpleScalar requires three input files: the CPU model parameters, the cross-compiled application binary, and the configuration bitstreams for the RPU. ModelSim requires the VHDL model of the RPU as input.

[1] The simulation time refers to the time representation within the simulation model and is distinctly different to the CPU time used in executing the simulation model or the "real world" time of the person running the simulation model.

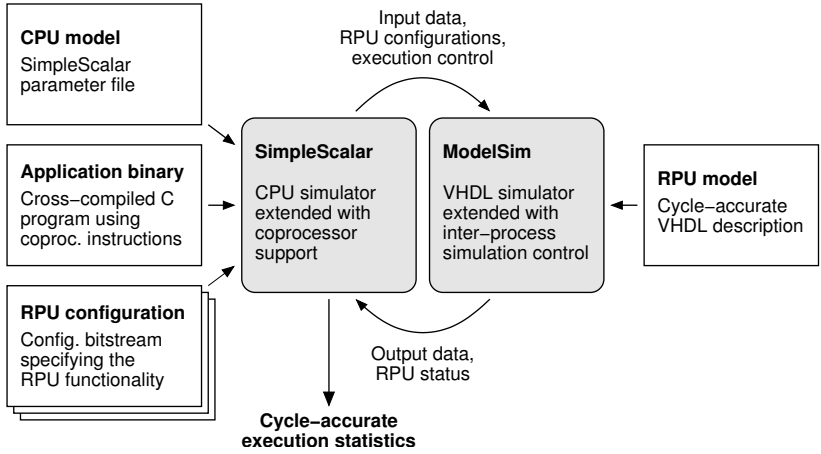


Figure 3.5. Co-simulation environment combining the SimpleScalar and ModelSim simulators

Software tool flow

The application parts that run on the CPU are implemented in the C language and compiled with the GCC-based C cross-compiler provided by the SimpleScalar tool suite. The application code is responsible for downloading the configuration bitstreams to the RPU and for controlling the execution of the configurations.

In order to support the coprocessor instructions that we have added to the PISA instruction set, we have to adapt the compilation chain. Although SimpleScalar allows easy modification of the simulated instruction set, the corresponding tools – in particular the compiler and assembler – cannot be generated automatically. Thus, to avoid the tedious task of modifying the compiler and assembler, we have decided to add instead some intermediate steps to the compilation flow.

The coprocessor instructions are accessed by using so-called *pseudo-assembler instructions*. We provide a library that facilitates the access to these pseudo-assembler instructions within a C application program. The library contains functions for transferring data between CPU and RPU, for downloading configurations from the CPU to the RPU, and for switching between the configurations. Figure 3.6 outlines the software tool flow and the implementation of the library functions. The figure illustrates as an example how the `get_RPUreg` coprocessor instruction,

which we have added to PISA, can be used within a C application program:

1. In `macros.h`, a GCC inline assembler macro is used to define a function-like wrapper for the underlying pseudo-assembler instruction.
2. Within the application program `app.c`, the function wrapper is called like any ordinary C function.
3. The compiler translates the function call into the `get_RPUreg` pseudo-instruction. The compiler takes care of inserting the correct register names into the inline assembler macro.
4. Before passing the intermediate file `app.i.s` to the assembler, we replace the pseudo-instruction by its binary instruction coding.
5. The unmodified assembler and linker then process the assembler file `app.s` and generate the binary executable `app.ss`, which can be run on the extended *sim-outorder* simulator.

Hardware tool flow

The functionality of the RPU is determined by its configuration. For convenience, the programmer can specify an RPU configuration by means of a VHDL record, which represents a hierarchical, structured view of the configuration. The VHDL record comprises all control signals of the RPU that are configuration controlled. The VHDL model of the RPU uses the configuration record to set the functionality of the RPU's processing elements and their interconnect. The VHDL configuration record is transformed by the VHDL simulator to a C readable configuration bitstream, which the application program can read in and download to the RPU. Figure 3.7 illustrates the tool flow for generating an RPU configuration bitstream:

1. The library file `configLib.vhd` contains the collection of RPU configurations which were developed. A configuration is accessed by means of a function that returns the appropriate configuration record.
2. The package file `auxPkg.vhd` provides the auxiliary procedure `gen_hfile()`, which transforms a VHDL configuration record into a C language array and writes it into a C header file.

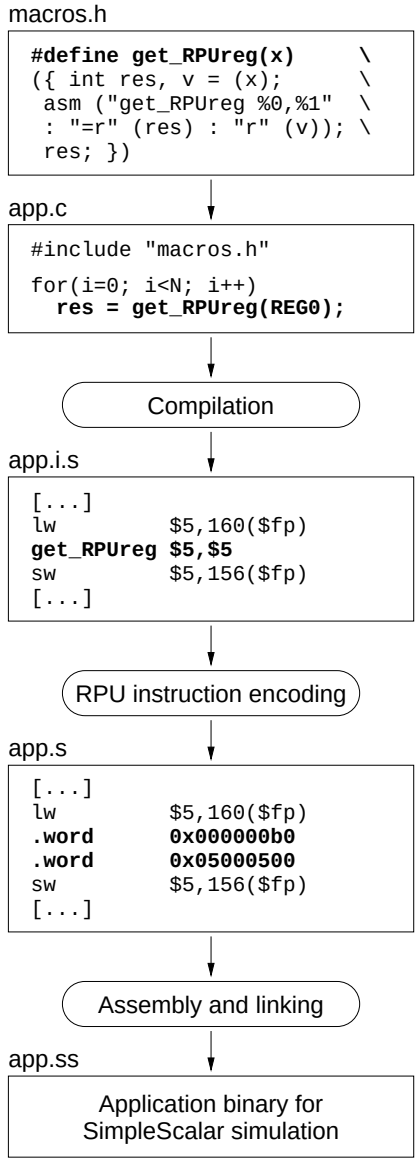


Figure 3.6. Software tool flow making use of pseudo-assembler instructions

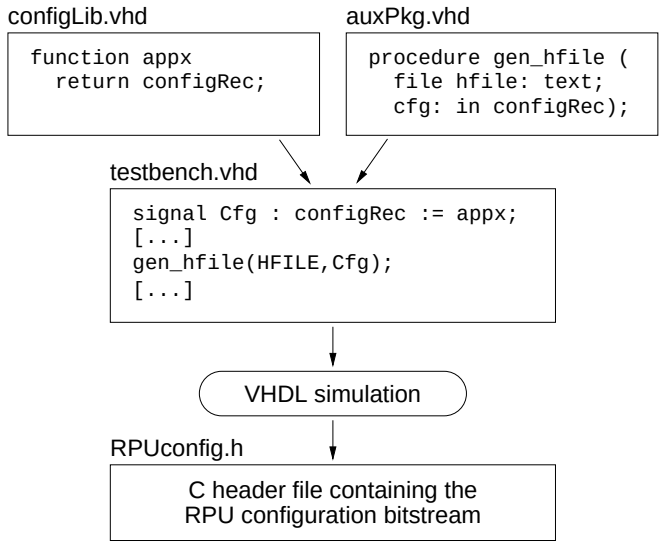


Figure 3.7. Hardware tool flow to generate RPU configuration bitstreams that can be included by the C application program

3. The VHDL testbench (`testbench.vhd`) loads the desired configuration record and then calls the procedure `gen_hfile()` with the name of the output file and the loaded configuration record as arguments.

4. The result of simulating the VHDL testbench is a C header file, denoted `RPUconfig.h`, containing the RPU configuration bitstream, which the C application program can include.

During the subsequent CPU/RPU co-simulation, the application program running on the CPU can download the configuration bitstream to the RPU. The concept of automatically generating the configuration bitstream ensures the consistency of the configuration data throughout the whole (co-)simulation flow.

3.3.5. Simulation Speed

A basic characteristic of a simulation environment is its simulation speed. Austin et al. [13] report the simulation speed of SimpleScalar in *simulated instructions per second*. For the *sim-outorder* simulator, they

measured a simulation speed of 300k instructions per second. In our case study experiments, which are discussed in Section 6, we achieve for a rather simple, embedded CPU model in the SimpleScalar stand-alone mode, i. e. without simulating the attached RPU, an average simulation speed of 132k instructions per second. The experiments were performed on a 900-MHz SunBlade 1000. For SimpleScalar's default, superscalar CPU model, we achieve a simulation speed of 167k instructions per second.

For comparison purposes with the hybrid reconfigurable processor model, we prefer the metric *simulated cycles per second* rather than instructions per second. In CPU stand-alone mode, we achieve a simulation speed of 226k cycles per second using the embedded CPU model. In the co-simulation case, i. e. when SimpleScalar and ModelSim run concurrently, we achieve an average simulation speed of 4.2k cycles per second. It is important to notice that the simulation speed depends both on the load balancing between CPU and RPU, which is determined by the application at hand, and on the CPU and RPU models.

3.4. Chip Area Estimation

Since we target not the general-purpose but rather the embedded computing domain, the computational performance is not our only optimization objective. In particular, we intend considering the costs that have to be increased in order to achieve a certain performance improvement. In order to relate the computational performance figures to the dedicated costs, we apply an area model that allows us to estimate the chip area of an architecture. We assume that silicon area is a rather good indicator for the chip costs.

3.4.1. Area Model

The first priority for the area estimation model is not to achieve a high accuracy for the overall total chip area. Rather, we want to observe the trend of the impact on the chip area for the particular design features under investigation. This allows us to evaluate the efficiency of the design features in terms of performance and costs.

The RPU and CPU parts of the architecture are considered separately. In order to have technology independent area figures, we follow the common approach to normalize the area data to λ^2, where λ denotes half the minimum feature size of the CMOS process technology.

Table 3.1. Technology and area figures for a selection of 32-bit processors. The area is normalized to λ^2, where λ denotes half the minimum feature size of the CMOS process technology

Processor core		Year	Technology μm	Die size mm^2	Area Mλ^2
ARM940T	[149]	1997	0.35	13	425
StrongARM 110	[118]	1996	0.35	50	1 600
MIPS 3900	[163]	1997	0.40	64	1 600
StrongARM 1500	[143]	1998	0.28	60	3 061
MIPS 10000	[74]	1995	0.50	298	4 768
PowerPC 620	[74]	1995	0.50	311	4 976
Alpha 21264	[67]	1997	0.35	314	10 300
Pentium4	[87]	2001	0.18	217	26 800

CPU core

For the CPU core, we rely on data published in the literature. Table 3.1 lists a number of processors with their respective technology and area figures. While the first four processors target the embedded domain, the second group aims at the desktop domain.

Reconfigurable processing unit

Regarding the RPU we consider a twofold approach: we combine data from VLSI synthesis with parameterized, analytical area models of architectural building blocks, such as registers or SRAM memory. The starting point is a block diagram of the RPU. To determine the overall area requirements, the area figures for the individual building blocks are accumulated, as Figure 3.8 illustrates.

We assume that the routing does not require a significant amount of chip area, since today's technology provides several metal layers to implement the routing. A certain routing overhead implied by the fact that the building blocks cannot be perfectly placed together, for example due to the escaping, is taken into account through an overhead factor based on empirical data – a typical value being 20% [93]. This is clearly a very simplified model. However, the chosen approach allows more accurate models to be integrated in a straight-forward way.

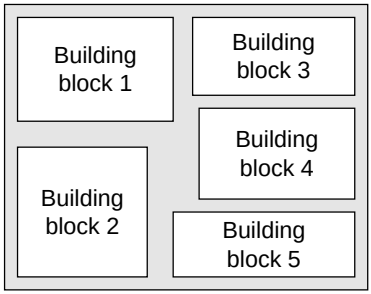


Figure 3.8. The area figures of the individual building blocks are accumulated to determine the overall area requirements of the RPU. An empirical area factor represents the routing overhead (gray area)

Thus, the overall chip area of the RPU results in

$$A_{\mathrm{RPU}} = a_{\mathrm{rout}} \cdot \sum_{\forall b} A_{\mathrm{BB}_b} \quad , \tag{3.1}$$

where A_{BB} denotes the area of the individual building blocks and a_{rout} the routing area overhead. If available, area data from VLSI synthesis can be employed to gain more accurate area figures. A prerequisite for this is a synthesizable description of the relevant building blocks for example in VHDL. Parts of the RPU model can be refined towards a synthesizable description while leaving other parts in a behavioral description.

3.4.2. Basic Building Blocks

Figure 3.9 depicts empirical area data collected by Kaeslin [93] for registers (built from D-type flip-flops) and on-chip SRAM. For registers built from latches instead of flip-flops[2], we assume that a latch requires 64% of the flip-flop area [93].

Based on this empirical data, we construct parameterized, analytical area models for these three basic building blocks. To this end, we define the functions $\alpha_{\mathrm{FF}}()$, $\alpha_{\mathrm{Latch}}()$, and $\alpha_{\mathrm{SRAM}}()$, which implement the respective correspondence between storage capacity and silicon area. To estimate the not explicitly given design points, the register area is

[2]We follow the terminology that latches are level-sensitive bistable elements, while flip-flops are edge-triggered bistable elements.

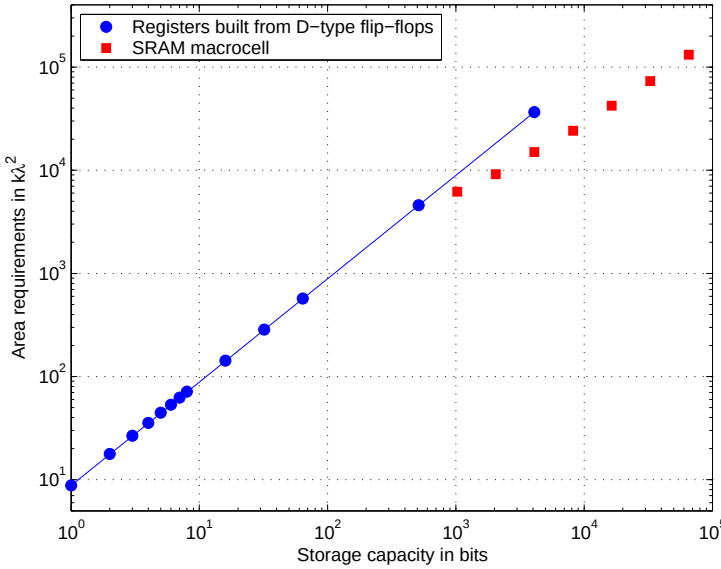


Figure 3.9. Area figures of registers and on-chip SRAM based on empirical data collected by Kaeslin [93]

linearly approximated and the SRAM data is fitted by a cubic polynomial with respect to the double-logarithmic capacity–area space. The polynomial coefficients are determined by means of Matlab. Thus, the functions are defined as

$$\alpha_{\mathrm{FF}}(s) := 8.93\, s - 0.112 \ , \tag{3.2}$$

$$\alpha_{\mathrm{Latch}}(s) := 0.64 \cdot \alpha_{\mathrm{FF}}(s) \ , \tag{3.3}$$

$$\alpha_{\mathrm{SRAM}}(s) := 1.91 \cdot 10^{-10}\, s^3 + 2.36 \cdot 10^{-5}\, s^2 + 2.68\, s + 3800 \ , \tag{3.4}$$

where s denotes the storage capacity. The area requirements are computed in $\mathrm{k}\lambda^2$.

The construction of analytical area models for further building blocks, such as ALUs, multipliers, etc., is feasible. However, we have opted for another approach; we use synthesis data for the processing cells of the reconfigurable array, which yields more accurate area figures for these rather complex structures. Since this alternative approach makes the area models specific to the RPU architecture, we discuss them in Chapter 6.

3.5. Energy Consumption

The evaluation methodology proposed in this work enables a designer to evaluate reconfigurable architectures in terms of computational performance and chip area. For the targeted domain of embedded computing systems, however, further optimization objectives are of interest, the most obvious one being the energy consumption.

A particularly interesting option for reconfigurable architectures is to trade off the gain in computational performance against the energy consumption, as for example proposed in [12, 35, 133]. The achieved speedup cannot solely be employed for faster execution of the application at hand, but also to lower the execution frequency and potentially the supply voltage [17]. The latter is especially relevant because the switching power in CMOS circuits depends quadratically on the supply voltage [18].

Consequently, the integration of a high-level energy or power estimation tool can be seen as a valuable extension of our evaluation methodology. Some starting points are given in the following sections.

3.5.1. High-Level Estimation Approaches for CPUs

A couple of approaches to the estimation of CPU power have been proposed in the literature. Landman [103] provides an overview of high-level estimation approaches and discusses the levels of abstraction that the various techniques target. In our context, two interesting approaches are instruction-level estimation and simulation-based techniques.

Instruction-level estimation

Instruction-level estimation relies on an instruction power profiling of the CPU instruction set [154, 169]. Each instruction has a power cost associated, which is determined by measuring the current that the CPU draws as it repeatedly executes a sequence of this same instruction. To account for the power overhead due to the change of the circuit state when two different instructions are sequentially processed, the inter-instruction effect is profiled for each pair of consecutive instructions in a similar way. Other effects such as pipeline stalls or cache misses can be incorporated as well. The idea is basically the same: programs, where the desired effect occurs, are run repeatedly and the current that is drawn by the CPU is measured.

Microarchitecture-level simulation

A number of approaches propose the usage of microarchitecture-level CPU simulators, which include detailed, cycle-accurate power models of the major CPU units. An obvious option in our case are extensions of the SimpleScalar simulator like Wattch [24], PowerAnalyzer [122] or SimplePower [177, 201]. The extended SimpleScalar simulators keep track of which CPU units are accessed per cycle and record the total energy consumed for an application. However, the proposed simulators are currently in a development stage and, in our experience, do not yet operate reliably.

Examples of simulation approaches at the microarchitecture level that do not rely on the SimpleScalar infrastructure are presented in [25, 40, 180].

3.5.2. Overall Reconfigurable Processor

In order to study the architectural trade-offs in the overall hybrid, reconfigurable processor, a power model for the attached RPU must be developed. As in the CPU case, such a power model can aim at various levels of abstraction, from coarse activity models to cycle-accurate switching models.

While integrating an RPU power model into a simulation framework is desirable and would allow for evaluation studies at the system level, there exist simpler approaches if the problem can be restricted. For example to determine the energy consumption of a particular architecture, logic synthesis and technology mapping can be applied. However, this does not allow for any trade-off studies.

For FPGAs – as a particular case of a reconfigurable processing unit – power estimation has recently become a topic of interest, e. g. [188, 199]. Researchers have also begun to investigate low-power FPGA structures, e. g. [66, 101, 184].

4

Workload Characterization

In contrast to many approaches investigating reconfigurable technology, we target not the general-purpose but rather the embedded computing domain, in particular handheld and wearable computing. In order to represent a typical workload for this computing domain, we have assembled an application pool with applications from the fields of multimedia, cryptography and communications.

This chapter deals with the quantitative characterization of this application pool. The analysis focuses on two main issues: first, the characteristics of the three application fields representing the embedded computing domain and, second, the differences between the embedded and the general-purpose domains. Based on the analysis results, we discuss the impact on the design of reconfigurable processor technology. The selection of representative benchmarks and the analysis approach have first been proposed in [55].

4.1. Benchmarks

There exist several benchmarks for general-purpose computing of which the SPEC benchmark suites by the Standard Performance Evaluation Corporation (www.spec.org) are the most popular ones.

Benchmarking embedded systems is generally more difficult than benchmarking general-purpose systems, mainly due to two reasons. First, system specialization leads to several application fields in the embedded domain with rather diverse applications and constraints. The formulation of a *unique* embedded benchmark would thus not be very meaningful. Second, compiler technology for embedded processors is far from being as mature as for general-purpose processors. Many embedded applications are still written in assembly language to exploit special features of the embedded processor at hand. A benchmarking procedure that relies exclusively on out-of-the-box code and thus includes the compiler quality in the evaluation does not reflect the likely use case for embedded processors.

Benchmarking efforts in the embedded and reconfigurable areas can be classified into the following categories:

- *Kernel-oriented benchmarks* — Benchmarks for embedded processors typically focus on computation-intensive kernels rather than on full applications. The underlying assumption is that embedded applications spend most of their run time executing these kernels. The kernel performance is then representative of the overall application. Examples for kernel-oriented, embedded benchmarks are BDTImark [59] and EEMBC [187].

- *Application-oriented benchmarks* — Collections of complete applications have been used to model workloads in multimedia and communications. These benchmarks are oriented towards SPEC and target high-performance processors with multimedia instruction set extensions and VLIW architectures. Examples are MediaBench [36], MiBench [73] and CommBench [196].

- *Benchmarks for reconfigurable computers* — Recently, benchmarks specifically for the area of reconfigurable computing have emerged. Two examples are ACS and RAW. The adaptive computing systems (ACS) [100] benchmark suite evaluates the architecture and tools of a configurable computing system. ACS proposes benchmarks that focus on a specific characteristic of a configurable system such as versatility, capacity, timing sensitivity,

scalability or interfacing. The reconfigurable architecture workstation (RAW) [14] benchmark suite consists of twelve programs representing general-purpose algorithms. Each benchmark can be parameterized to derive instances of different problem sizes.

For embedded systems that target the application domain of handhelds and wearables there exist currently no widely accepted benchmarks. The existing application-oriented benchmark suites come close to this domain from a functional point of view. However, they do not explicitly target embedded systems or even reconfigurable systems. ACS and RAW are explicitly designed for evaluating the performance of a given reconfigurable system in various aspects, but they do not focus on our application domain.

In summary, current benchmarks are not well suited for our purpose of determining the optimal reconfigurable architecture for a specialized domain. Hence we assemble our own application pool reflecting a characteristic workload for the targeted application domain. Within our evaluation methodology, we use hand-crafted code fragments mapped to the RPU in order to speed up the hot spots of an application. The application programs written in C will be linked with a library of such code fragments. For the workload characterization, we therefore concentrate on *complete applications* and *out-of-the-box code*, i. e. unoptimized C code that has not been target-specifically optimized.

4.2. Application Pool

Our envisioned workload for the handheld and wearable computing domain emphasizes applications from the fields multimedia (MM), cryptography (CRY), and communications (COM).

As the benchmark suites MediaBench [36], MiBench [73] and CommBench [196] cover parts of the targeted application fields, we have selected some of their programs and have added the cryptography standard RIJNDAEL [43], and the text-to-speech synthesizer SVOX [170]. The result are 32 programs, which constitute our application pool. We denote this set of programs as MCCmix.

Table 4.1 lists the set of applications. Many of these applications actually consist of two programs: encoding/decoding or encryption/decryption. The SUSAN package contains programs to recognize corners and edges as well as a program for image smoothing. More details on the applications that are part of one of the benchmark suites

Table 4.1. Application pool – denoted MCCmix – divided into the three application groups multimedia, cryptography, and communications

Multimedia (MM)	
JPEG	Lossy image compression [36, 73, 196]
EPIC	Lossy image compression based on wavelets [36]
MPEG2	Lossy video compression [36]
ADPCM	Adaptive differential pulse code modulation [36, 73]
GSM	European standard for speech transcoding [36, 73]
G.721	CCITT voice compression [36]
SUSAN	Image recognition [73]
RASTA	Speech recognition [36]
SVOX	Text-to-speech synthesizer [170]
Cryptography (CRY)	
RIJNDAEL	Advanced encryption standard (AES) [73, 123]
PGP	Cryptography program that uses IDEA for encryption and RSA for key management [36, 73]
PEGWIT	Elliptic curve cryptography algorithm [36]
CAST	DES-like cryptography algorithm [196]
Communications (COM)	
DRR	Deficit round robin fair scheduling algorithm [196]
FRAG	IP packet fragmentation with checksum computation [196]
REED	Reed–Solomon forward error correction [196]
RTR	Radix-tree routing table lookup [196]
ZIP	Data compression based on the Lempel-Ziv (LZ77) algorithm [196]

can be found in the referenced literature. The additional two applications are briefly described below:

RIJNDAEL [43] is the winner of the advanced encryption standard (AES) effort [123] initiated by the U.S. National Institute of Standards and Technology (NIST) in order to find a successor for the data encryption standard (DES) [148, 156]. RIJNDAEL is a block cipher with variable block length and key length of 128, 192 or 256 bits. RIJNDAEL is also part of the MiBench suite, but the implementation is different from the one used in this work.

SVOX [170] is a commercial text-to-speech program for the German language and consists of three steps: word and sentence analysis, prosody control, and voice synthesis. The word analysis step applies phonetic lexica to yield the phonetic representation of each single word in the text. Prosody control then determines speech melody and sound durations based on statistical models, i. e. neural networks and others. Finally, the speech signal is synthesized by concatenation of small units extracted from natural human speech. SVOX applies special signal processing methods to modify these units such that they achieve the previously determined pitch and duration values.

4.3. Evaluation Setup

The goal of the application analysis is to characterize the workload rather than to model the processor architecture. Thus, we simplify the simulation in the sense that we do not co-simulate the attached RPU, but reduce the evaluation infrastructure to the stand-alone CPU simulator. We particularly use the SimpleScalar simulators *sim-safe*, *sim-profile* and *sim-outorder*:

- *sim-safe* is a minimal, functional simulator, which emulates only the instruction set. We use *sim-safe* to verify the functional correctness of the benchmarking applications that we simulate.

- *sim-profile* gathers profiling data for the application under investigation. The *sim-profile* simulator does not rely on a particular processor architecture, i. e. the generated profiling data depends only on the instruction set, but not on the CPU model.

Table 4.2. Embedded CPU model for the processor dependent part of the application characterization

CPU parameter	Setup
Integer units	1 ALU, 1 multiplier
Floating-point units	1 ALU, 1 multiplier
Instruction fetch queue (IFQ) size	1 instruction
Register update unit (RUU) size	4 instructions
Load/store queue (LSQ) size	4 instructions
Decode bandwidth	1 instruction
Issue bandwidth	2 instructions
Commit bandwidth	2 instructions
Instruction issuing	In-order
Branch prediction	Static (always "not taken")
1st-level instruction cache	32-way 16 Kbytes
1st-level data cache	32-way 16 Kbytes
2nd-level cache	None
Memory bus width	32 bits
Memory ports	1

- *sim-outorder* is the cycle-accurate simulator that we use within our co-simulation environment. The *sim-outorder* simulator relies on a parameterized CPU model and allows detailed execution statistics to be gathered for the application being simulated.

For the architecture dependent simulations with *sim-outorder*, the SimpleScalar CPU model was set up to resemble modern embedded processors such as StrongARM [194] or TinyRISC [127]. Table 4.2 summarizes the main parameters of the CPU model used.

All applications have been simulated using out-of-the-box code, i. e. C code that is not target specifically optimized. The optimization level for the SimpleScalar cross-compiler was set to "O2". This means that the cross-compiler performs the supported optimizations that do not involve a space–speed trade-off.

Since the original data sets of the application programs are often small and intended only for test purposes, we have chosen our own input data sets to represent a reasonable workload. Table 4.3 reports the sizes of the benchmark programs in terms of committed instructions.

Table 4.3. Size of the MCCmix benchmark programs in terms of committed instructions

Application group	Benchmark program	Instruction count
MM	JPEG decode	156 442 354
	JPEG encode	235 509 725
	EPIC decode	98 975 971
	EPIC encode	482 568 099
	MPEG2 decode	182 888 391
	MPEG2 encode	1 133 718 151
	ADPCM decode	277 157 028
	ADPCM encode	329 225 126
	GSM decode	161 614 916
	GSM encode	511 706 316
	G721 decode	304 731 239
	G721 encode	316 152 916
	SUSAN corners	61 854 613
	SUSAN edges	221 187 438
	SUSAN smoothing	959 423 107
	RASTA	341 365 633
	SVOX	506 732 113
CRY	RIJNDAEL decrypt	32 928 121
	RIJNDAEL encrypt	32 230 571
	PGP decrypt	100 027 981
	PGP encrypt	170 489 536
	PEGWIT decrypt	34 115 554
	PEGWIT encrypt	54 790 190
	CAST decrypt	50 289 806
	CAST encrypt	50 289 752
COM	DRR	213 093 381
	FRAG	217 174 137
	REED decode	451 744 149
	REED encode	226 375 969
	RTR	503 135 868
	ZIP decode	68 643 501
	ZIP encode	400 358 085

4.4. Workload Analysis

In this section, we follow a number of conventions with regard to the notation.

1. A set of applications is denoted as $\mathbb{A}$. Examples are the multimedia application group or the MCCmix.

2. The value $n_{\mathbb{A}}$ refers to the number of applications in $\mathbb{A}$. The multimedia application group for example comprises 17 applications, the MCCmix 32 in total.

3. Let x represent a characteristic of an application, e. g. the instruction count. For a set of applications $\mathbb{A}$, the respective characteristics are organized in a vector $\mathbf{x}$ having $n_{\mathbb{A}}$ elements:

$$\mathbf{x} := \left(x^{(1)}, x^{(2)}, \ldots, x^{(n_{\mathbb{A}})} \right) \quad ,$$

   where $x^{(i)}$ refers to the measured value for application i.

4. The average value of the characteristic x over all the applications in $\mathbb{A}$, denoted $\overline{\mathbf{x}}$, is defined as the arithmetic mean of the elements contained in the vector $\mathbf{x}$:

$$\overline{\mathbf{x}} := \frac{1}{n_{\mathbb{A}}} \sum_{\mathbb{A}} x^{(i)} = \frac{1}{n_{\mathbb{A}}} \sum_{i=1}^{n_{\mathbb{A}}} x^{(i)} \quad .$$

5. If a characteristic is represented by a function $f(x)$ instead of a scalar, the respective characteristics of the applications in $\mathbb{A}$ are organized as a vector

$$\mathbf{f}(x) := \left(f^{(1)}(x), f^{(2)}(x), \ldots, f^{(n_{\mathbb{A}})}(x) \right) \quad .$$

6. The average value of such a characteristic for a determined x is then defined as

$$\overline{\mathbf{f}}(x) := \frac{1}{n_{\mathbb{A}}} \sum_{\mathbb{A}} f^{(i)}(x) = \frac{1}{n_{\mathbb{A}}} \sum_{i=1}^{n_{\mathbb{A}}} f^{(i)}(x) \quad .$$

4.4.1. Instruction Class Mix

The instruction class mix characterizes applications according to the frequency of the different instruction groups during execution. These frequencies depend only on the application, the processor's instruction set and the compiler, but not on any architectural parameters of the processor such as the number of functional units, cache sizes, etc. We use the instruction class mix as an indicator of the operations that an application uses. We classify the instructions according to their functionality into eight instruction classes:

1. load,
2. store,
3. branch,
4. integer arithmetic,
5. logic,
6. shift,
7. floating-point (FP) arithmetic,
8. miscellaneous.

In a different context, another classification scheme may be applied. For example in [9], the instructions are classified into seven classes stressing the difference between computationally "cheap" and "expensive" instructions. In [129] and [130] the same classification pattern is used as in this work to characterize an ASIC-on-demand prototype, which incorporates reconfigurable hardware into autonomous wearable computing nodes.

By means of the *sim-profile* simulator we generate the instruction profile of each application revealing how many times each instruction type is issued during execution. The *relative instruction type frequency* of the instruction type i is defined as

$$f_{\text{itype}}(i) := \frac{n_{\text{itype}}(i)}{n_{\text{inst}}} \ , \tag{4.1}$$

where $n_{\text{itype}}(i)$ is the number of times the instruction type i is issued, and n_{inst} the total number of instructions executed.

Arranging the gathered results into the predetermined instruction classes, the *relative instruction class frequency* of instruction class c is given by

$$f_{\text{iclass}}(c) := \sum_{i \in \mathbb{I}_c} f_{\text{itype}}(i) = \frac{1}{n_{\text{inst}}} \sum_{i \in \mathbb{I}_c} n_{\text{itype}}(i) \ , \tag{4.2}$$

where $\mathbb{I}_c$ denotes the set of instructions forming the instruction class c. Figure 4.1 depicts the distribution of the instruction classes over the

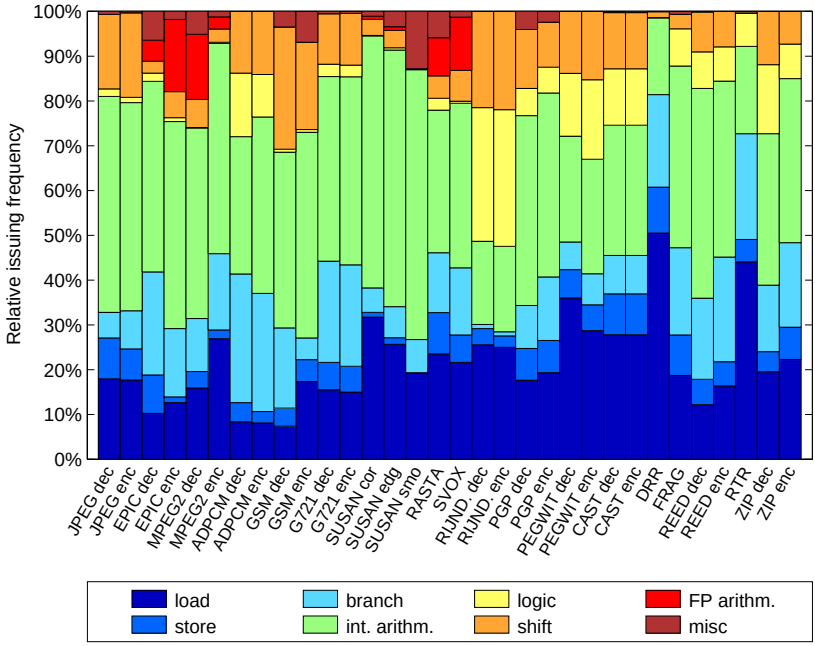


Figure 4.1. Instruction class mix for the MCCmix applications depicting the relative issuing frequencies of the eight determined instruction classes

MCCmix applications. Figure 4.2 shows the results in more detail by displaying the relative issuing frequencies for each instruction class individually.

In order to compare the different application groups, we compute for each group the average relative instruction class frequencies

$$\mu_{\text{iclass}}(c) := \overline{\mathbf{f}}_{\text{iclass}}(c), \quad \text{for all } c \ . \tag{4.3}$$

Figure 4.3(a) displays the relative instruction class averages for the individual MCCmix application groups. For comparison purposes, we have also gathered the instruction class mix of the integer SPEC95 benchmarks (CINT95) [137, 158] as a representative of a general-purpose workload. Figure 4.3(b) shows the results for MCCmix overall and CINT95. Table 4.4 provides the results quantitatively.

Observations

The comparative analysis between the MCCmix application groups leads to a number of observations.

- *Multimedia* applications require a significant larger number of integer arithmetic instructions compared to the MCCmix average. This observation is not astonishing since most multimedia applications perform arithmetic-intensive signal processing relying on fixed-point data. Logic operations, on the other hand, are used only rarely.
- *Cryptography* algorithms are characterized by regular code operating on bit-level data. The regularity of these algorithms reflects in a small number of branch instructions. Since the PISA instruction set does not provide special instructions for bit manipulation, the bit-level operations are performed using repeated shift and logic operations, which is evidenced by a high percentage of shift and logic instructions.
- *Communication* applications are characterized by using a high number of branch instructions. This results from the control-flow dominated nature of communication applications. Arithmetic and shift operations, in contrast, are of less significance for this application group.

The results underline the importance of dividing the benchmark applications into several application groups. Each group shows its particular characteristics and hence emphasizes different architectural features of a processor.

Another observation is that only eight out of 32 programs require floating-point operations. Given the fact that the results were generated with out-of-the-box code, we can expect optimized programs to use floating-point operations even less.

Comparing the overall results of the MCCmix to the general-purpose CINT95, we observe major differences. The instruction classes store, integer arithmetic, logic, and shift show the most significant differences. Particularly notable is the small average percentage of store instructions in the MCCmix.

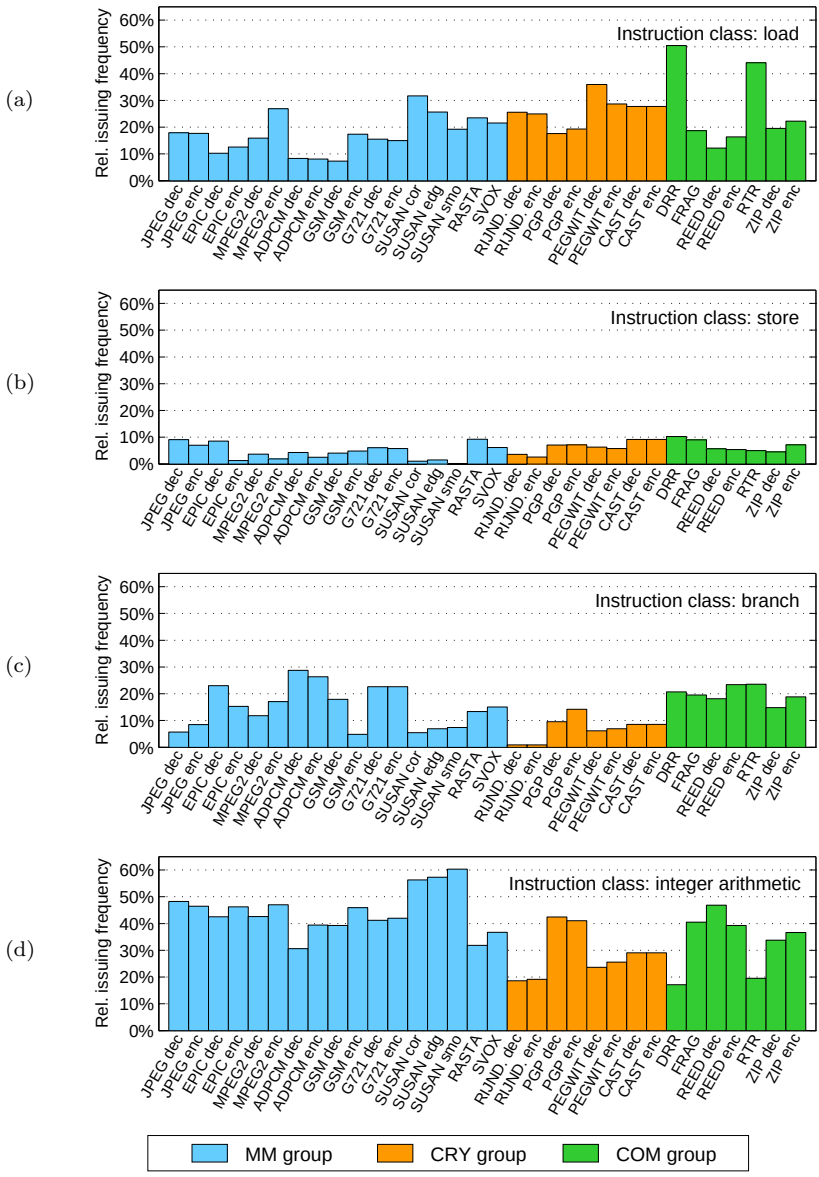
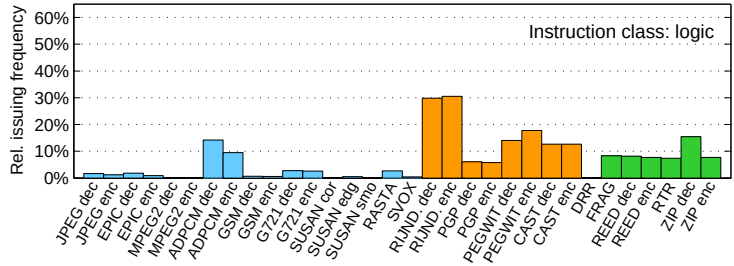
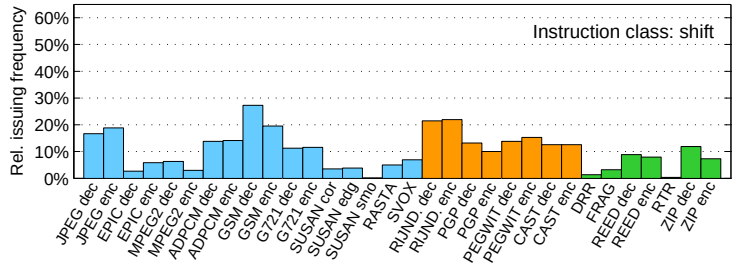


Figure 4.2. Separated instruction class mix, showing the instruction classes individually. *(continued on the next page)*

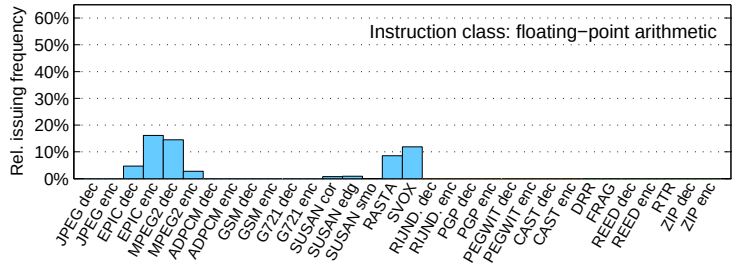
(e)



(f)



(g)



(h)

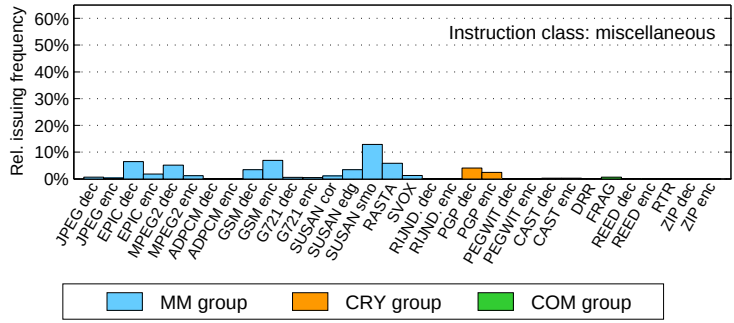
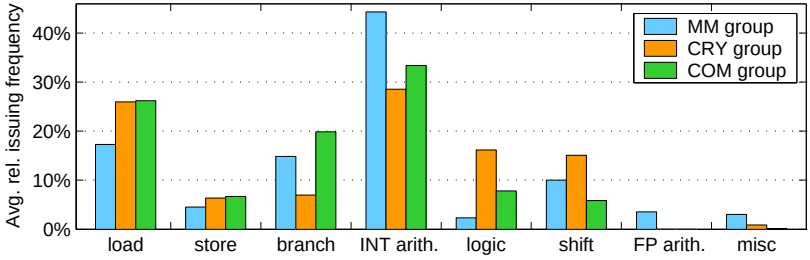
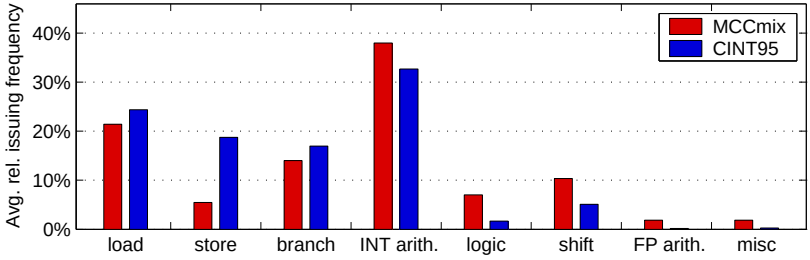


Figure 4.2. *(continued)*



(a) MCCmix application groups



(b) MCCmix and CINT95

Figure 4.3. Average relative issuing frequencies of the eight determined instruction classes

Table 4.4. Average relative issuing frequencies of the eight instruction classes for the MCCmix application groups individually, the MCCmix overall, and CINT95

Application group	Average relative issuing frequency of instruction class in %							
	load	store	branch	integer	logic	shift	FP	misc
MM	17.3	4.5	14.9	44.3	2.3	10.0	3.5	3.0
CRY	26.0	6.4	7.0	28.6	16.2	15.1	0.0	0.9
COM	26.2	6.7	19.9	33.4	7.8	5.8	0.0	0.1
MCCmix	21.4	5.5	14.0	38.0	7.0	10.4	1.9	1.9
CINT95	24.4	18.8	17.0	32.7	1.7	5.1	0.2	0.3

4.4.2. Memory Requirements

With respect to the memory requirements of the applications, we differentiate between program text, data size, and dynamic memory.

- *Program text* refers to the size of the executable code.
- *Data size* is the size of data (DATA and BSS segments). The initialized data section is also incorporated in the program text.
- *Dynamic memory* is the amount of memory that is allocated by the application at run time. This number has to be treated as an upper bound because the SimpleScalar simulators do not report on memory deallocation.

Figure 4.4 and Table 4.5 show the average memory requirements of the three MCCmix application groups individually, the MCCmix overall, and CINT95. The RASTA benchmark includes an 8-Mbyte static data structure, which is most likely not always fully used. In order to have a realistic comparison, the results are hence reported with and without including RASTA.

Observations

The MCCmix applications have rather small memory requirements. The data size requirements of CINT95 exceeds the requirements of MCCmix by a factor of 24. If the RASTA benchmark is omitted from MCCmix, a factor of 132 results. In Figure 4.4 the results excluding the RASTA benchmark are plotted with a dashed line. The average program text of CINT95 is four times larger than the average program text of MCCmix. The upper bound for dynamic memory is between seven and eight times higher for CINT95.

4.4.3. Cache Miss Rates

A cache miss occurs when the CPU does not find a data item or instruction in the accessed cache. The *cache miss rate* is defined as the fraction of the cache accesses that result in a miss [85]:

$$r_{\text{cmiss}} := \frac{n_{\text{cmiss}}}{n_{\text{caccess}}} \quad , \tag{4.4}$$

where n_{cmiss} denotes the number of cache misses and n_{caccess} the total number of cache accesses.

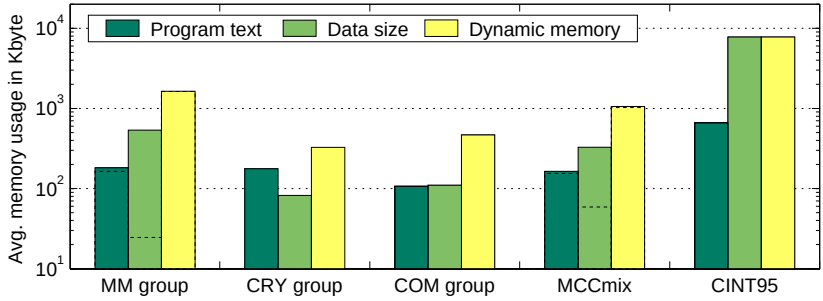


Figure 4.4. Average memory usage for the MCCmix application groups individually, the MCCmix as a whole, and CINT95. The dotted lines indicate the results if RASTA is omitted

Table 4.5. Average memory usage for the MCCmix application groups individually, the MCCmix as a whole, and CINT95 (with and without RASTA)

Application group	Average memory usage in Kbyte		
	Program text	Data size	Dynamic memory
MM w/ RASTA	181.9	535.9	1635.5
MM w/o RASTA	164.2	24.6	1627.9
CRY	177.2	82.1	327.2
COM	107.5	110.4	466.4
MCCmix w/ RASTA	164.5	329.3	1052.7
MCCmix w/o RASTA	154.8	58.8	1029.9
CINT95	661.0	7766.2	7748.1

The *sim-outorder* simulator reports the miss rates of the involved caches. For the simulation runs we used the CPU model outlined in Table 4.2, which incorporates first-level instruction and data caches, but no second-level cache. Figure 4.5(a) and Figure 4.5(b) show the measured cache miss rates. For comparison purposes, the average cache miss rates, i. e.,

$$\mu_{\text{cmiss}} := \overline{\mathbf{r}}_{\text{cmiss}} \quad , \tag{4.5}$$

for both MCCmix and CINT95 are plotted in the form of horizontal lines.

Observations

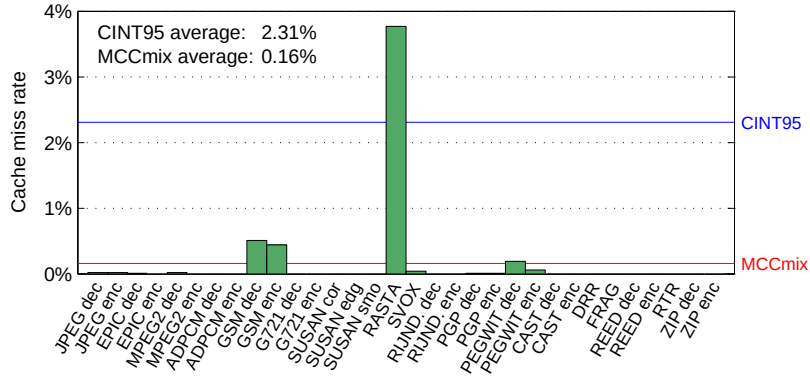
MCCmix has lower miss rates for both instruction and data caches, but the difference is more significant for the instruction cache. This confirms the assumption that many embedded applications spend most of their run time in rather small code sections – the kernels. The MCCmix applications contain compute-intensive code parts with high data locality. Thus, processors for the MCCmix application domain can work efficiently with smaller instruction caches compared to general-purpose processors.

4.4.4. Function Breakdown

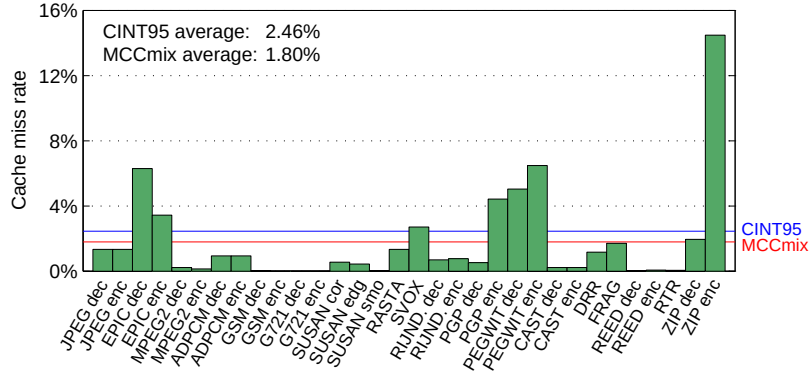
In order to investigate the structure of the MCCmix and CINT95 programs and to identify the most compute-intensive program parts, we generate the function breakdown for each application. The function breakdown reports the fraction on the application's run time taken by each function in terms of execution cycles. The *relative run time* spent in function f is thus defined as

$$t_{\text{fct}}(f) := \frac{n_{\text{fct}}(f)}{n_{\text{cycles}}} \quad , \tag{4.6}$$

where $n_{\text{fct}}(f)$ refers to the number of cycles spent in function f, and n_{cycles} refers to the total number of execution cycles. To determine the number of execution cycles spent in each function, we merge address information from the disassembled program binaries with run-time information delivered by the *sim-outorder* simulator.



(a) Instruction cache



(b) Data cache

Figure 4.5. Instruction and data cache miss rates for the MCCmix applications. For comparison purposes, the overall MCCmix and CINT95 averages are drawn as horizontal lines

Figure 4.6 shows the function breakdown of the MCCmix and CINT95 applications. The three left-most segments of each horizontal bar represent the percentages for the three most compute-intensive (or dominating) program functions – denoted f_1, f_2 and f_3, respectively. The fourth segment represents the accumulated percentages of all the remaining functions – denoted $f_{\overline{123}}$. The run-time distribution of *MPEG2 encoding* is for example:

$$\begin{array}{lll} f_1 & \text{– function } \texttt{dist1}: & t_{\text{fct}}(f_1) = 67.8\%\ , \\ f_2 & \text{– function } \texttt{fdct}: & t_{\text{fct}}(f_2) = 11.1\%\ , \\ f_3 & \text{– function } \texttt{fullsearch}: & t_{\text{fct}}(f_3) = 5.7\%\ , \\ f_{\overline{123}} & \text{– remaining functions}: & t_{\text{fct}}(f_{\overline{123}}) = 15.4\%\ . \end{array}$$

In order to compare the run time spent in the program functions, we consider the *cumulative relative run time*, denoted $T_{\text{fct}}(f)$, of the dominating functions. Figure 4.7 displays the average cumulative relative run times of the dominating functions for MCCmix and CINT95, which are computed as

$$\mu_{\text{T}1} := \overline{\mathbf{T}}_{\text{fct}}(f_1) = \overline{\mathbf{t}}_{\text{fct}}(f_1)\ , \tag{4.7}$$

$$\mu_{\text{T}2} := \overline{\mathbf{T}}_{\text{fct}}(f_2) = \overline{\mathbf{t}}_{\text{fct}}(f_1) + \overline{\mathbf{t}}_{\text{fct}}(f_2)\ , \tag{4.8}$$

$$\vdots$$

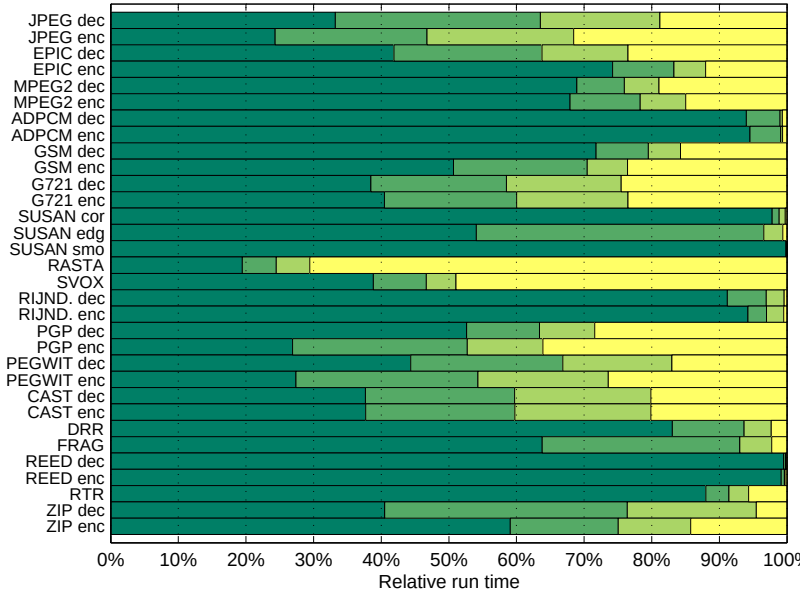
$$\mu_{\text{T}i} := \overline{\mathbf{T}}_{\text{fct}}(f_i) = \overline{\mathbf{t}}_{\text{fct}}(f_1) + \overline{\mathbf{t}}_{\text{fct}}(f_2) + \ldots + \overline{\mathbf{t}}_{\text{fct}}(f_i)\ . \tag{4.9}$$

Table 4.6 provides some quantitative results of the cumulative function breakdown.

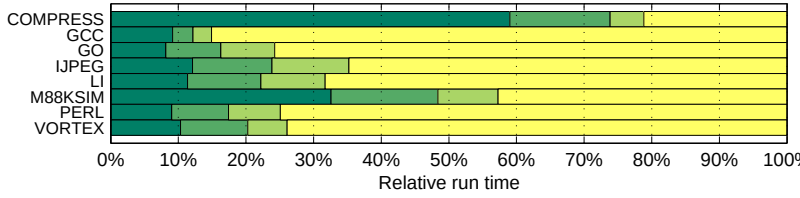
Observations

The function breakdown facilitates the identification of the compute-intensive parts of the applications – the kernels. The quantitative information about the MCCmix kernels is essential for the design of a reconfigurable processor since the kernels form the greatest potential for improvements in terms of performance and energy consumption.

The results point out that MCCmix and CINT95 differ strongly. The quantitative numbers show that the MCCmix applications are strongly kernel oriented, while the general-purpose CINT95 applications consist of many, smaller sized functions. While in the MCCmix 61.1% of the run time is spent in the most compute-intensive function and 90.9% in the five dominating functions, the corresponding figures in case of CINT95 are only 19.0% and 47.9%, respectively.



(a) MCCmix



(b) CINT95

Figure 4.6. Function breakdown based on the execution cycles

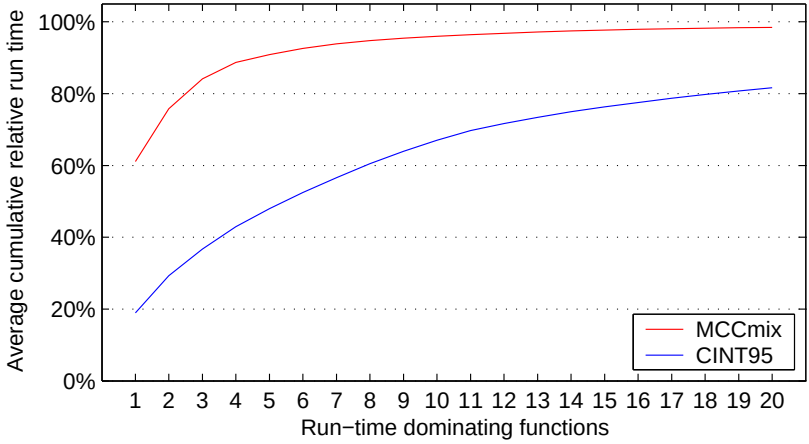


Figure 4.7. Cumulative function breakdown, revealing the average cumulative relative run time of the dominating functions

Table 4.6. Quantitative results of the cumulative function breakdown

Avg. cumulative relative run time	MCCmix %	CINT95 %
μ_{T1}	61.1	19.0
μ_{T2}	75.8	29.3
μ_{T3}	84.1	36.7
μ_{T5}	90.9	47.9
μ_{T10}	95.9	67.0
μ_{T20}	98.4	81.6

4.4.5. Key Results

In summary, based on the application analysis we draw the following qualitative conclusions:

- The embedded applications are strongly kernel oriented.
- The memory requirements are one to two orders of magnitude lower compared to CINT95.
- The MCCmix applications achieve a higher cache performance than the CINT95 applications employing a rather small instruction cache.
- Floating-point operations are used moderately in MCCmix. Only 8 out of the 32 MCCmix applications use floating-point operations at all.
- The three application groups distinguished in MCCmix show quite differing characteristics:
  - multimedia applications are dominated by integer arithmetic operations,
  - cryptography applications require a large number of logic and shift operations, and
  - communication applications rely on many branch operations.

4.5. Impact on Reconfigurable Processor Design

Each particular application group shows its distinctive characteristics and hence emphasizes different architectural features of a processor. Multimedia applications stress integer arithmetic on word-organized data and thus benefit from a coarse-grained architecture. The control-oriented communication applications, on the other hand, benefit from fine-grained structures, which allow finite state machines to be efficiently implemented. Cryptography applications use a large number of bit-level operations, which due to the absence of bit-operators in the PISA instruction set results in many logic and shift operations. Consequently, the RPU features, e. g. the supported operations and their granularity, are subject to a trade-off which depends on the priorities of the applications at hand.

We consider the insights revealed by the workload analysis in the design of the processor architecture described in the following chapter:

- The applications are highly kernel-dominated, which supports our approach of attaching the RPU to the standard CPU via a dedicated coprocessor interface. The reconfigurable coprocessor is responsible for the bulk of the data processing, while the CPU handles the main control flow and the operations that do not map well onto the reconfigurable array.

- The analysis reveals that small caches perform well. This supports the notion of incorporating a rather small CPU core that is well suited for the embedded systems domain.

- Since we target particularly data-streaming applications, we emphasize word-oriented integer arithmetic operators and develop a coarse-grained reconfigurable array. Additionally, we integrate some fine-grained reconfigurable resources that allow us to adapt the data access controller according to the memory access patterns of the application at hand.

- Data-streaming applications show less complex data flow patterns than general-purpose applications. For this reason, rather moderate interconnect facilities are provided compared to general-purpose devices, in particular FPGAs.

- Since very few floating-point operations are required, the reconfigurable array does not provide any floating-point operators. If required, the CPU takes care of the floating-point operations.

5

Reconfigurable Processor Architecture

Based on the proposed evaluation methodology and on the results of the workload characterization, this chapter elaborates a hybrid reconfigurable processor architecture. We form a reconfigurable processor by coupling a coarse-grained, multi-context reconfigurable processing unit to a CPU core. The RPU stores several configurations on the chip allowing for fast switching of its functional behavior. The CPU is responsible for the data transfer, context loading, and control of the multi-context RPU. We target the embedded domain, in particular data-streaming applications that map well to macro-pipelines. To this end, we introduce hardware virtualization as a programming model. Hardware virtualization is supported by the multi-context features of the RPU. The programming model as well as parts of the reconfigurable processor architecture were first published in [57].

5.1. Programming Model

A drawback of current reconfigurable devices – in particular FPGAs – is the lack of appropriate programming models. Applications are compiled (or synthesized in this context) to given, fixed-size hardware. The resulting configuration bitstream cannot be reused to program a device of different type or size, not even inside the same device family. Consequently, in order to leverage advances in VLSI technology – i. e. increased transistor count and higher clock rates – a complete recompilation process needs to be performed. The key to overcome this limitation is hardware virtualization [16, 34, 147, 172].

5.1.1. Hardware Virtualization

In order to achieve hardware virtualization, we define a set of basic operators that a hardware device can execute. Together with a description of the data flow (the communication paths between the operators) and the control flow (the sequencing of the operators), we establish a hardware programming model that compilers can target.

In comparison, processors use a well-established form of hardware virtualization by defining an instruction set architecture, which decouples the compiler from the actual hardware organization. This allows the processor architecture to change as long as the defined instruction set is still supported. Programs can be used on the new processor without a recompilation process.

Achieving virtualization of reconfigurable hardware is more complex. Reconfigurable hardware excels when computations are organized spatially. The basic operators will thus have greater complexity than processor instructions and the number of possible operators is very large. Further, the reconfigurability allows many basic operators to be implemented with one type of hardware block.

5.1.2. Macro-Pipelining and Contexts

We consider data-streaming applications that map well to macro-pipelines, where a pipeline stage is implemented by one basic hardware block. We assume the basic hardware block to be a reconfigurable array. The inputs and outputs of the array connect to first-in, first-out (FIFO) buffers in order to facilitate data streaming. One configuration of the array is denoted as a *context*. Applications are organized by pipelining several *logical context* executions.

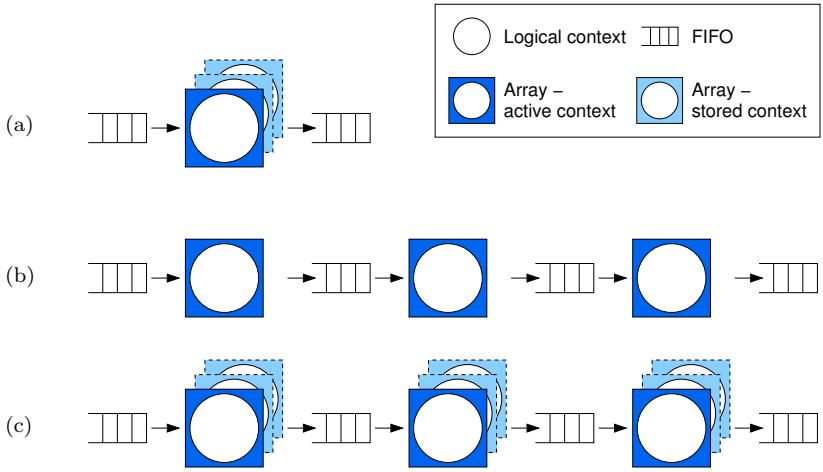


Figure 5.1. Models of virtualized macro-pipelining representing different trade-offs with respect to multiple contexts and physical pipelining

A reconfigurable array can hold one or more contexts on the chip. Single-context devices hold exactly one configuration on the device. Before a new logical context can be executed, the corresponding configuration has to be loaded from off-chip. Multi-context devices hold a set of configurations on-chip, denoted as *physical contexts*. At any given time there is exactly one configuration in use, the so-called *active context*. Figure 5.1 illustrates different models of virtualized macro-pipelining.

- Figure 5.1(a) shows one multi-context array that is reconfigured to execute logical contexts as required. The array stores multiple physical contexts to minimize or even hide the reconfiguration time.
- Figure 5.1(b) shows several single-context arrays arranged in a pipelined fashion. The arrays are still reconfigured to execute different logical contexts. However, the configuration data of the inactive contexts is not stored on-chip, which leads to a longer reconfiguration time. On the other hand, since several contexts run in parallel, the throughput increases.
- Multiple contexts and physical pipelining can be combined, which is illustrated in Figure 5.1(c).

All these architectures achieve hardware virtualization as they provide the logical pipeline of context executions as the programming model, but differ in their potential computational performance and hardware requirements. Although this model is rather restrictive, it is amenable to true hardware virtualization and targets the important domain of data-streaming applications.

In the following, we focus on an architecture with one coarse-grained, multi-context reconfigurable array. We form a hybrid system by coupling the reconfigurable array to a standard CPU.

5.2. System Integration

Unlike many other approaches studying reconfigurable computing systems, we aim not at the general-purpose but rather at the embedded computing domain. The embedded domain puts more stringent requirements on computing power, energy consumption, costs, weight, volume, etc. and stresses the trade-offs with respect to these objectives. Consequently, our goal is to employ limited reconfigurable hardware resources in an efficient way, rather than using devices of arbitrarily large size.

5.2.1. Basic Design Features

As the application analysis of the embedded workload has revealed, the specific domain of data-streaming applications already points to some desirable basic design features. First, streaming applications typically make extensive use of integer arithmetic and show a high degree of parallelism which favors coarse-grained reconfigurable arrays. Second, the rather simple memory access patterns as well as the designated programming model indicate the need for FIFO buffers in the RPU.

Figure 5.2 shows the block diagram of the reconfigurable processing unit attached to the coprocessor interface of the CPU core. The RPU architecture comprises the coprocessor register interface, two FIFO buffers for data transfer, the configuration memory, the context sequencer, and the reconfigurable processing fabric – an array of coarse-grained reconfigurable cells. Data to and from the RPU is transferred via the two FIFO buffers. In order to communicate with the CPU, the RPU provides a set of coprocessor registers, listed in Table 5.1. To access a FIFO for example, the CPU reads from or writes to the corresponding FIFO coprocessor register.

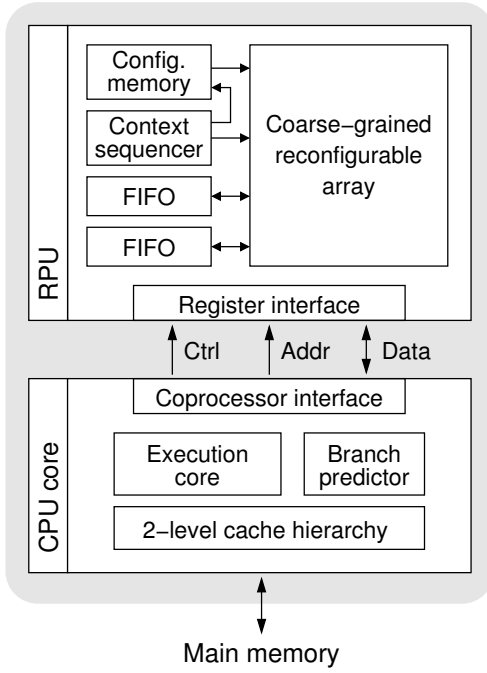


Figure 5.2. System outline showing the reconfigurable processing unit attached to the coprocessor interface of the CPU core

Table 5.1. Coprocessor registers provided by the RPU with the corresponding read (R) and write (W) CPU access options

RPU coprocessor register	CPU access
RPU reset	W
FIFO $\{1, 2\}$	R/W
FIFO $\{1, 2\}$ – fill level	R
Configuration memory: context $\{1, \ldots, c\}$	W
Configuration memory pointer: context $\{1, \ldots, c\}$	W
Context select – clearing context state	W
Context select – holding context state	W
Cycle count	R/W
Context sequencer – start	W
Context sequencer – status	R
Context sequencer – program store: instruction $\{1, \ldots, s\}$	W

Two important design features that we investigate are multiple contexts and register replication. If the cells of the reconfigurable array contain registers in their datapath, then a context carries state. Upon a context switch, the state information, i. e. the content of the datapath registers, must be saved such that it can be restored on the next invocation of the same logical context. In general, there are three ways to achieve this:

- the content of the datapath register is transferred to a memory from which it can be read back;
- the state is not explicitly saved and then restored, but it is recomputed; or
- the datapath registers are replicated such that each context accesses its own, dedicated register set, i. e. the state remains in the corresponding registers.

Our RPU architecture model concentrates on the latter two alternatives.

Recapitulating, the architectural model of the RPU incorporates several parameterized design features in order to represent different RPU variants:

- the datapath width (up to 32 bits),
- the size of the FIFO buffers,
- the number of physical contexts that the configuration memory can store, and
- the number of register planes, if the datapath registers within the reconfigurable cells are replicated.

5.2.2. CPU Core

As discussed in Chapter 3, we build on the CPU model provided by SimpleScalar, which is based on a 32-bit RISC processor architecture with a MIPS-like instruction set called PISA. The data and control path of the CPU as well as the memory architecture are highly parameterized. The main architectural units of which the CPU is comprised can be divided into the execution core, the branch predictor, and the two-level cache hierarchy. The following list gives the major CPU parameters involved.

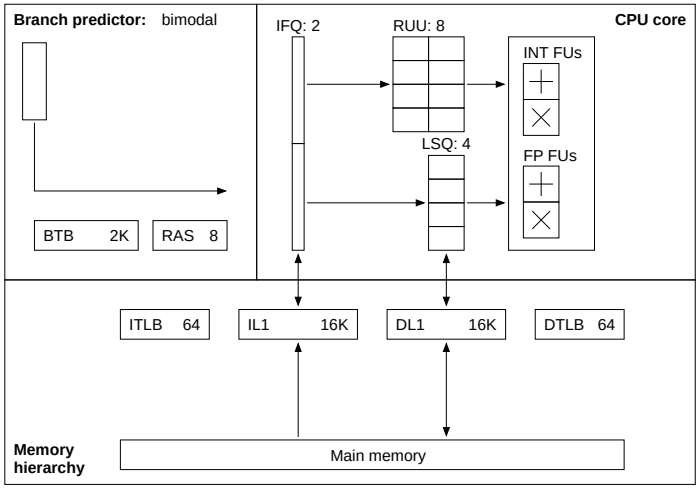
- Execution core:
  - Number of integer and floating-point functional units, i. e. ALUs and multipliers
  - size of the instruction fetch queue (IFQ)
  - size of the register update unit (RUU)
  - size of the load/store queue (LSQ)
  - decode, issue, and commit bandwidths
  - instruction issuing (in-order or out-of-order)
- Branch predictor:
  - type and configuration
  - size of the return address stack (RAS)
  - configuration of the branch target buffer (BTB)
- Memory hierarchy:
  - types of the first-level (L1) and second-level (L2) caches
  - configuration and access time of the caches
  - configuration of the instruction and data translation look-aside buffers (ITLB and DTLB)
  - main memory bus width and access time

Burger and Austin [27] provide an in-depth discussion of the CPU model and its parameters.

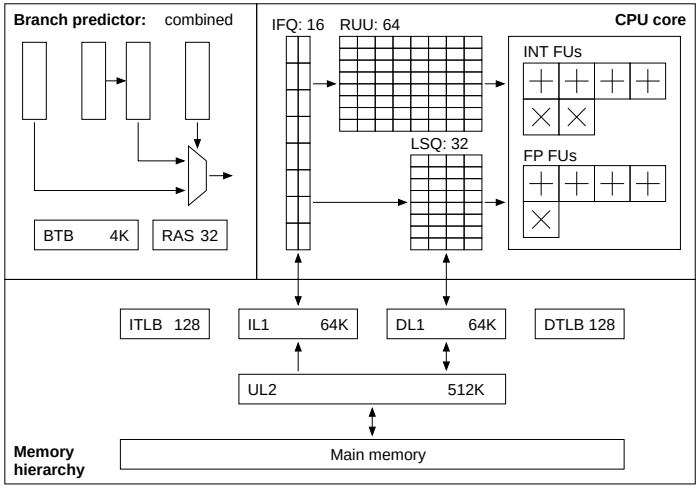
Five branch predictor types are supported: two simple, static types (*always "taken"*, *always "not taken"*) and three more complex, dynamic types (*bimodal*, *2-level*, *combined*). The dynamic branch predictors have further configuration options, in particular the sizes of their branch history tables.

The two-level cache hierarchy supports L1 and L2 instruction (ILx), data (DLx), and unified (ULx) caches. The configuration parameters of the caches include the number of sets, block size, associativity, and replacement policy.

Due to the rich configuration options, the CPU model can be set up to resemble a broad spectrum of architectures, from small low-end to powerful high-end CPUs. Figure 5.3 illustrates two simplified example setups of the CPU model. The first setup corresponds to a typical low-end CPU, the second setup to a high-end CPU.



(a) Low-end CPU



(b) High-end CPU

Figure 5.3. Two example setups of the CPU model illustrating the major parameterized architecture features of the model

5.2.3. Coprocessor Registers

CPU and RPU communicate by means of the coprocessor register file (CRF). Table 5.1 lists the coprocessor registers that the RPU provides. To perform a certain action, the CPU reads from or writes to the appropriate coprocessor register.

For example to initiate a reset on the RPU, the CPU writes to the *RPU reset* register. The coprocessor interface controller of the RPU is responsible for handling the request and for taking care of the control signals within the RPU. The CPU accesses the FIFO buffers, the configuration memory, the *cycle count* and *context select* registers, as well as the context sequencer registers in a similar way.

Some of the coprocessor registers are virtual registers, i. e. they are not physically implemented as registers in hardware. The *RPU reset* or the *context sequencer start* registers for example just initiate a certain action when they are written, while the *FIFO* register corresponds to an SRAM-based FIFO buffer implementation.

5.2.4. FIFO Buffers

Data is transferred between the CPU and RPU via the two FIFO buffers that the RPU incorporates. Both FIFOs are readable and writable by the CPU as well as by the RPU. The FIFOs also serve as data buffers between consecutive executions of two RPU contexts.

In a typical scenario the CPU first writes a data block to one of the FIFOs. The samples are then processed by the active RPU context and written to the second FIFO. The context may then be switched and the new active context processes the samples again, writing the results back to the first FIFO. In the same way, several more context switches may be initiated. Finally, the CPU reads the data block back from the appropriate FIFO.

To arbitrate between the accesses of the reconfigurable array and the CPU (via the coprocessor interface), each FIFO has a small control unit associated with it. The control unit is responsible for routing the datapath to the FIFO and for setting the FIFO control signals. To this end, the reconfigurable array provides read and write enable signals, which are controlled by the RPU configuration. The control unit makes sure that these enable signals only take effect while a computation is active on the reconfigurable array. The control unit prioritizes the reconfigurable array over the CPU, i. e. if the reconfigurable array requests access to a FIFO, the CPU access is blocked.

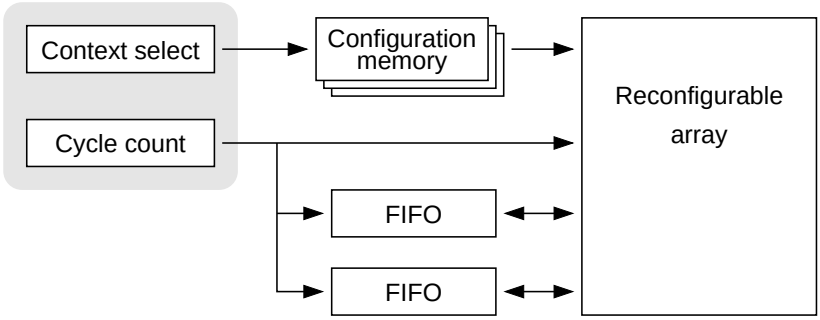


Figure 5.4. Interface provided by the RPU for controlling the execution of the reconfigurable array

5.2.5. Configuration Memory

The configuration memory holds one or more entire configurations (the contexts) for the reconfigurable array. In Table 5.1, the number of physical contexts is denoted as *c*. For each physical context, the coprocessor interface provides a *configuration memory* register. The configuration bitstream is written from the CPU to the RPU via the corresponding *configuration memory* register in 32-bit chunks. The coprocessor interface controller on the RPU is responsible for inserting the 32-bit configuration chunks at the correct physical location in the configuration memory.

The RPU supports the download of full and partial configurations for any of the physical contexts. To support partial reconfiguration, the coprocessor interface provides an additional *configuration memory pointer* register for each physical context. To download a partial configuration, the CPU first writes the start address of the partial bitstream to the *configuration memory pointer* register, followed by writing one or more 32-bit configuration chunks to the *configuration memory* register.

5.2.6. Synchronization and Context Scheduling

In order to control the execution of the reconfigurable array, the RPU incorporates an interface comprising the *cycle count* register and the *context select* register. Figure 5.4 illustrates the control interface.

The CPU selects the active RPU context by means of the *context select* register. Two modes are provided for the CPU writing into the

context select register, either the context state is cleared or kept. This means that the datapath registers of the reconfigurable array are either reset upon a context switch or the registers hold their state, respectively.

The CPU starts the RPU execution by writing the number of clock cycles the reconfigurable array is to execute to the *cycle count* register. In every clock cycle, the *cycle count* register is decremented by one. The execution of the reconfigurable array stops when the *cycle count* register reaches zero. This synchronization mechanism between CPU and RPU is similar to the mechanism proposed by Hauser and Wawrzynek [80]. As our hybrid processor model does not so far support interrupts, the synchronization of CPU and RPU execution is done by polling the *cycle count* register. Furthermore, the output of the *cycle count* register is used to communicate to the FIFO control units that the reconfigurable array is active.

The application program running on the CPU is responsible for scheduling and activating the contexts. There are two modes for how the CPU can control the execution of the contexts:

1. The CPU activates a context on the RPU for execution by writing the context ID to the *context select* register. The RPU context is immediately switched and the CPU can trigger the execution of the reconfigurable array by writing the desired number of execution cycles to the *cycle count* register.

2. The CPU makes use of the hardware context scheduler, the *context sequencer*, that is incorporated into the RPU. The context sequencer allows a sequence of contexts to be autonomously executed by the RPU without the intervention of the CPU. At the beginning of an application, the CPU downloads the program for the sequencer. Afterwards, the CPU can trigger the context sequencer by accessing the *context sequencer start* register. The sequencer then autonomously executes the programmed context sequence. After termination, the sequencer activates the *sequence status* register, which the CPU can poll.

5.2.7. Context Sequencer

As Figure 5.5 outlines, the context sequencer consists of the sequence program store and some control facilities. It attaches to the execution interface depicted in Figure 5.4.

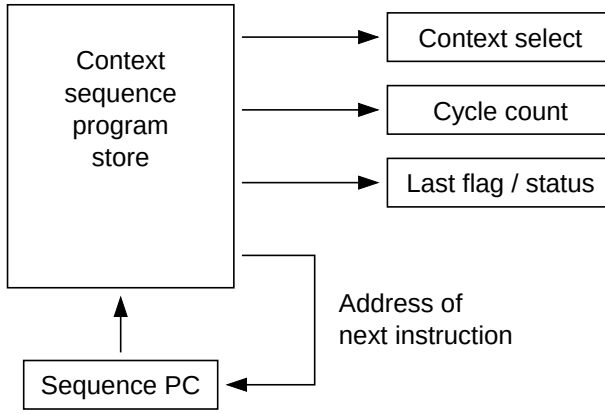


Figure 5.5. Context sequencer, which allows the RPU to execute a sequence of contexts autonomously, i. e. without intervention of the CPU

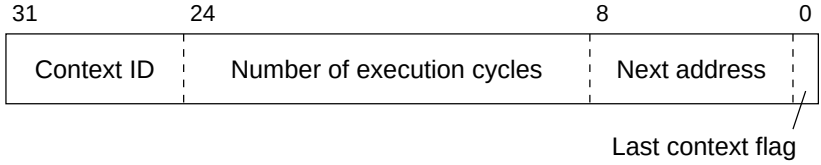


Figure 5.6. Instruction word for the context sequencer held in the context sequence program store

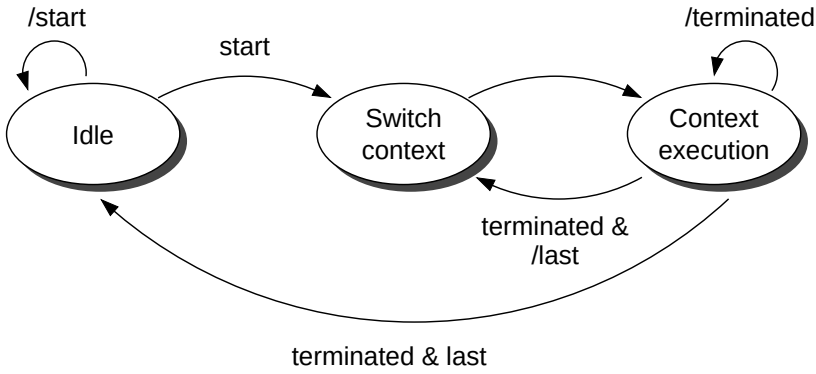


Figure 5.7. Control state machine of the context sequencer which controls the autonomous execution of a context sequence

The sequence program store holds a number of sequencer instructions, which control the execution of the contexts. In Table 5.1 the number of sequencer instructions is denoted as s. Figure 5.6 shows the format of a sequencer instruction consisting of four fields: the ID of the context to be activated, the number of execution cycles, the address of the next instruction to be processed, and the *last context* flag, which determines whether the current context is the last one in the sequence. The lengths of the individual instruction fields are parameterized, the figure shows an example. The sequence program counter (SPC) is used to supply the next sequencer instruction.

Upon termination of a context execution, a small control state machine, depicted in Figure 5.7, loads the SPC either with the address of the next instruction or actives the sequencer status register in order to signal to the CPU that the context sequence has finished, depending on the *last context* flag.

The concept of the context sequencer is similar to the microprogramming in some general-purpose CPUs [84]. However, the functionality is fundamentally different. While microinstructions are simple instructions used in sequences to implement a more complex instruction set, we use the sequencer instructions to control the execution of complex array contexts. Nevertheless, the advancements achieved in microprogramming [84], such as more complex next-address mechanisms, could be adapted in order to extend the functionality of the context sequencer.

5.3. Reconfigurable Array

Since we are targeting the embedded domain, our goal is to employ limited hardware resources in an efficient way. Hence, we start with a rather small reconfigurable array that features relatively few routing facilities. The reconfigurable array is comprised of a 4×4 array of homogeneous, coarse-grained, reconfigurable cells, which are interconnected by a two-level network.

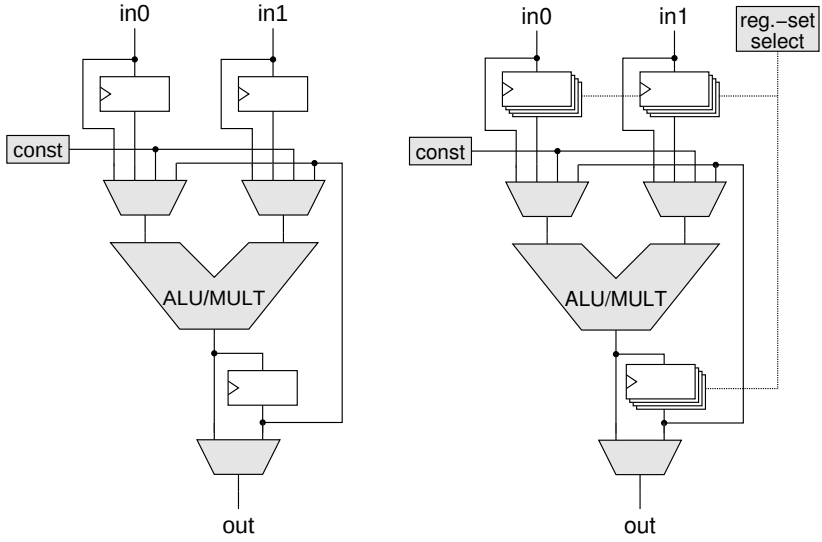
5.3.1. Reconfigurable Cells

As the workload analysis has revealed, data-streaming applications make extensive use of integer arithmetic on word-sized data. Thus, we favor coarse-grained over fine-grained processing elements. The logical and arithmetic operations can either be implemented by an ALU or a LUT. Since ALUs are more efficient for coarse-grained operators, we prefer this alternative. Figure 5.8 depicts the datapath of a reconfigurable cell consisting of a fixed-point ALU, datapath multiplexers, and input and output registers.

The control signals for the ALU and the multiplexers are part of the configuration. The configuration also contains a constant operand, which can be routed to either ALU input. The ALU implements the common arithmetic and logic operations (addition, subtraction, shift, OR, NOR, NOT, etc.) as well as multiplication.

An important design feature is register replication. Figure 5.8(a) sketches a reconfigurable cell with a single set of registers. Consequently, all contexts share the same registers. The registers are reset upon a context switch, i. e. their state cannot be stored. Figure 5.8(b) displays a reconfigurable cell that replicates the registers. This allows the register state to be stored over several context switches. Alternatively, the registers can also be reset upon a context switch. If the array provides sufficient registers sets, each context can have its own, dedicated register set assigned. The configuration determines on which register plane the current context operates.

The reconfigurable cells connect to the local interconnect network and the global buses. Figure 5.9 outlines the routing facilities within a reconfigurable cell. The cell's inputs are routed by means of multiplexers. The cell's output is directly connected to the local interconnects and via tristate buffers to the global buses.



(a) Single register set (b) Replicated register sets

Figure 5.8. Datapath of a reconfigurable cell with a single register set and with replicated register sets, respectively. The shaded parts are controlled by the configuration

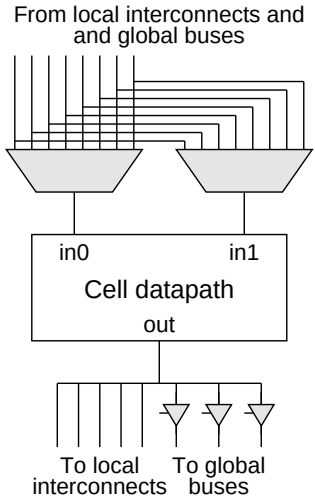


Figure 5.9. Routing facilities within a reconfigurable cell. The shaded parts are controlled by the configuration

5.3.2. Two-Level Interconnect

The cell interconnect reflects the mainly forward streaming nature of the data flow in data-streaming applications. The interconnect network consists of two routing levels:

1. local interconnects between adjacent reconfigurable cells, shown in Figure 5.10, and
2. global buses between cell rows, depicted in Figure 5.11.

The local interconnect is cyclically continued at the array borders. Each reconfigurable cell can route the outputs of five of its neighbors to its inputs via the local connections. The inputs can also be connected to any of the three global buses located between the cell rows. Two of the global buses can be driven by the cells from the row above. The third bus can be driven by the cells in the same row, which allows for feed back of the outputs from the same row.

The reconfigurable array features two input and two output ports (IPx, OPx), which are connected to the FIFO buffers of the RPU. Inside the array, the connections to the I/O ports are routed via the global buses.

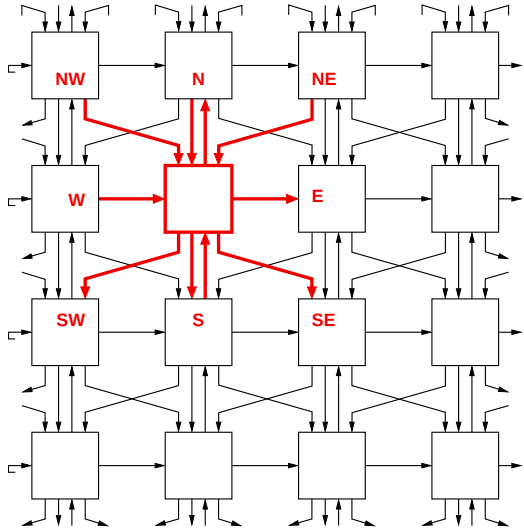


Figure 5.10. Local interconnects between adjacent reconfigurable cells; for visibility, the connections of one cell are highlighted

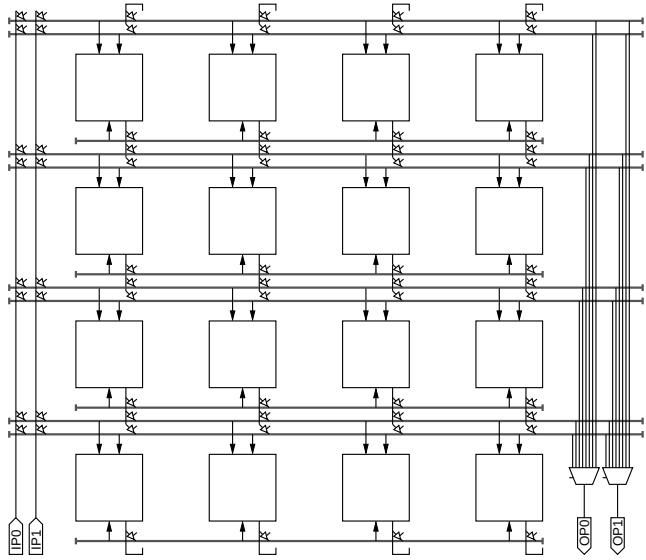


Figure 5.11. Global bus interconnect and I/O ports (IPx, OPx)

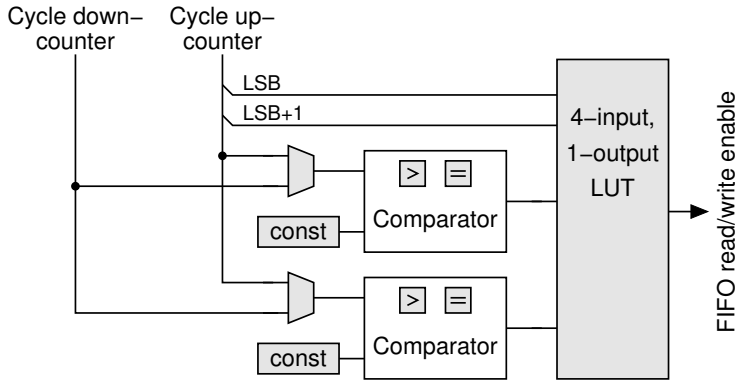


Figure 5.12. I/O port controller providing a FIFO read/write enable signal. The shaded parts are controlled by the configuration

5.3.3. Input/Output Ports

To access the FIFO buffers, the reconfigurable array has to provide their control signals, i. e. read and write enables. To this end, each I/O port has a configurable, fine-grained controller associated with it, which is responsible for controlling the data access.

Figure 5.12 outlines the architecture of an I/O port controller, consisting mainly of two comparators and a 4-input, 1-output LUT. The inputs to the I/O port controller are the values of two execution cycle counters, which count up and down the execution cycles computed on the reconfigurable array. The down-counter coincides with the *cycle count* register used for the synchronization of the CPU and RPU. Each comparator compares one of the cycle counters with a constant value provided by the configuration. The comparators can be configured to operate in either "greater than" or "equal" mode. The outputs of the comparators together with the two least significant bits (LSBs) of the up-counter form the input to the LUT. The functionality of the LUT is specified by the configuration.

This scheme allows us to generate moderately complex rules for the enable signals of the FIFOs. Examples are setting the enable signals after a certain number of execution cycles, or setting them a certain number of cycles before the end of a computation. Other examples are setting the enables every second or every forth execution cycle.

Table 5.2. RPU configuration size for a selection of datapath widths

Datapath width bits	Configuration size bits	Datapath width bits	Configuration size bits
4	734	20	990
8	798	24	1054
12	862	28	1118
16	926	32	1182

5.3.4. Configuration

To summarize, the configuration of the reconfigurable array is responsible for

- the functionality of the reconfigurable cells and the I/O port controllers,
- in the case of replicated datapath registers, the selection of the active register plane, and
- the routing of the datapath between the reconfigurable cells, from the input ports to the reconfigurable cells, and from the reconfigurable cells to the output ports.

The configuration memory stores one or more entire configurations.

Since the configuration incorporates constant values, which can be used as inputs to the processing elements in the reconfigurable cells, as depicted in Figure 5.8, the number of configuration bits depends on the datapath width, which is parameterized. Table 5.2 lists some of the design points. Given a datapath width of 16 bits, the configuration size for the reconfigurable array will be 926 bits. For comparison with a fine-grained architecture, the configuration size of the Xilinx Virtex-II Pro family ranges between 1.24 Mbits and 41.58 Mbits [198]. Compared to our coarse-grained architecture, the configuration size of the Virtex-II Pro devices is three to four orders of magnitude larger.

6

Experiments and Results

As a case study we discuss the partitioning and mapping of finite impulse response (FIR) filters of arbitrary order onto RPUs with limited hardware resources. We make use of the hardware virtualization programming model introduced in the previous chapter. We perform several experiments by implementing the same filter on different architectural variants of our parameterized reconfigurable processor model. We determine the computational performance gains for the reconfigurable hybrid over the stand-alone CPU, depending on a number of design parameters such as the number of physical on-chip contexts, the FIFO buffer size, and the ability to restore the state of a previous context. We also measure the impact on the CPU load for the various system configurations. Further, we estimate the chip area requirements of the different system designs and study their area–speed trade-offs.

Table 6.1. CPU model resembling an embedded CPU

CPU parameter	Setup
Integer units	1 ALU, 1 multiplier
Floating-point units	1 ALU, 1 multiplier
Instruction fetch queue (IFQ) size	1 instruction
Register update unit (RUU) size	4 instructions
Load/store queue (LSQ) size	4 instructions
Decode bandwidth	1 instruction
Issue bandwidth	2 instructions
Commit bandwidth	2 instructions
Instruction issuing	In-order
Branch prediction	Static (always "not taken")
1st-level instruction cache	32-way 16 Kbytes
1st-level data cache	32-way 16 Kbytes
2nd-level cache	None
Memory bus width	32 bits
Memory ports	1

6.1. Experimental Setup

We have set up our hybrid processor architecture to study the following system configurations:

- stand-alone CPU without attached RPU,
- CPU with attached single-context RPU, and
- CPU with attached multi-context RPU holding 2, 4 or 8 contexts.

We assume that CPU and RPU operate at the same clock frequency. This assumption is realistic for two reasons: first, we aim at the embedded computing domain, where maximal clock speed is not the sole and major optimization criteria; and second, we use a coarse-grained reconfigurable array allowing for higher clock speeds than fine-grained FPGAs.

We consider reconfigurable arrays with one shared set of registers as well as replicated register sets. Additionally, we regard RPUs with and without integrated hardware context sequencer (hardware scheduler). We use the same CPU model as for the workload characterization resembling a low-end, embedded CPU. Table 6.1 lists the main CPU parameters.

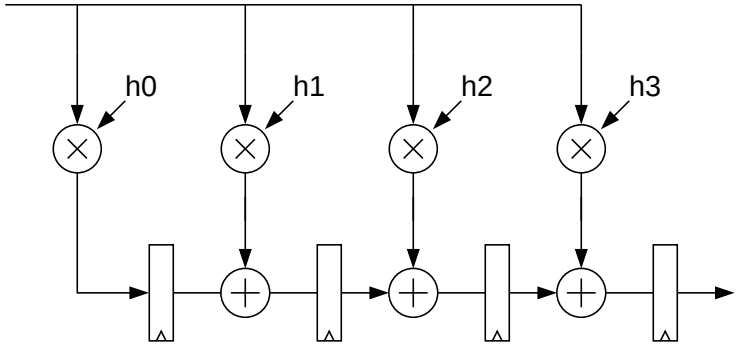


Figure 6.1. Transposed direct form realization of a 4-tap FIR filter

6.2. FIR Filter Virtualization

6.2.1. Partitioning and Mapping

The response $Y(z)$ of an FIR filter given by its transfer function $H(z)$ to an input signal $X(z)$ can be computed as $Y(z) = H(z) \cdot X(z)$. The transfer function $H(z)$ is a polynomial in z^{-i} given by

$$H(z) = h_0 + h_1 \cdot z^{-1} + \ldots + h_m \cdot z^{-m} = \sum_{i=0}^{m} h_i \cdot z^{-i} \quad . \tag{6.1}$$

In the FIR filter case, the transfer function $H(z)$ can be factorized into first and second order polynomials $H_i(z)$ [125]. That is, we can represent $H(z)$ as

$$H(z) = H_1(z) \cdot H_2(z) \cdot \ldots \cdot H_n(z) \quad . \tag{6.2}$$

This relation allows us to split up an FIR filter into a cascade of FIR subfilter sections of smaller order, which are sequentially executed.

In our case study, we implement a 56th-order FIR filter as a cascade of eight subfilters, each of 7th order. The subfilter sections are implemented in transposed direct form [125]. Figure 6.1 illustrates an example with four taps. Each section is mapped to an individual logical RPU context. Fig. 6.2 shows a simplified schematic of the mapping of one section onto the reconfigurable array. The filter coefficients h_i are part of the RPU configuration.

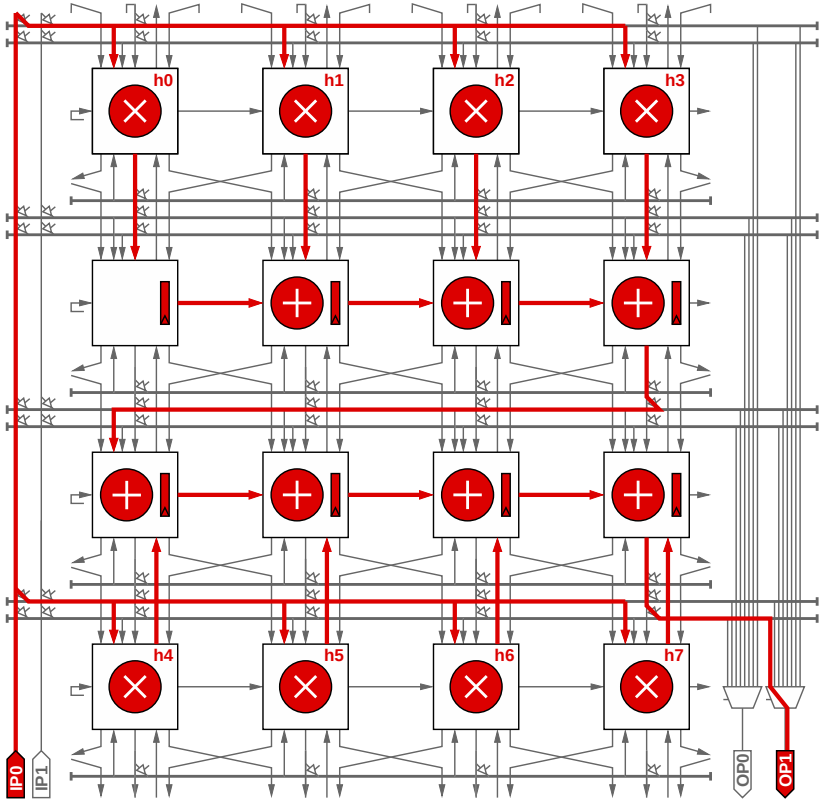


Figure 6.2. One 8-tap FIR subfilter section mapped onto the reconfigurable array, constituting a logical RPU context

It is important to notice the difference between a cascaded and a non-cascaded implementation of a FIR filter, which is in our case of order 56:

- The cascaded filter implementation employs 8 subfilter sections of order 7, resulting in an overall filter order of 8 times 7, i. e. 56. Each 7th-order subfilter section features 8 taps, resulting overall in 8 times 8, i. e. 64, taps.
- In the non-cascaded case on the other hand, where the filter is not partitioned into subfilter sections but constructed directly, the implementation requires 57 taps.

Thus, the partitioning of the FIR filter creates a certain computing overhead due to the higher number of taps that have to be processed.

6.2.2. Context State

Each FIR subfilter section requires delay registers in the datapath which form the state of the corresponding RPU context. The state information must be restored before the same context is executed again. Depending on the capabilities of the reconfigurable array, there are two ways to achieve this:

1. If all contexts of the array share the same set of registers, the state cannot be saved and restored, but must be recomputed. We achieve this by overlapping subsequent data blocks, according to the *overlap-save method* by Oppenheim and Schafer [125]. Overall, this results in an execution overhead, since the overlapped samples are processed twice.
2. If the array provides a dedicated register plane for *each* logical context, which we denote as full register replication, the state can be kept automatically in the datapath registers and no overlapping of data blocks is required. Thus, there is no execution overhead.

6.2.3. Context Scheduling

In each experiment, 64K samples organized in data blocks are processed. The size of the data block depends on the size of the FIFO buffers available on the RPU. We vary the size of the FIFOs between 64 and 1K words.

In the system configurations that make use of an RPU coprocessor, a data block is written to the RPU, processed sequentially by the eight FIR subfilter sections (the eight logical contexts), and then read back.

At the beginning of the application computation, the program running on the CPU loads as many logical contexts as fit onto the RPU. If not all logical contexts fit, the contexts are loaded on demand. Each time a filter context is required that is not present on the RPU, the CPU control task performs the download by overwriting a physical RPU context. To switch between logical contexts that are available on the RPU, the CPU has two possibilities: either to explicitly initiate a switch by writing to the *context select* register and then to start the execution of the reconfigurable array, or to make use of the hardware context sequencer.

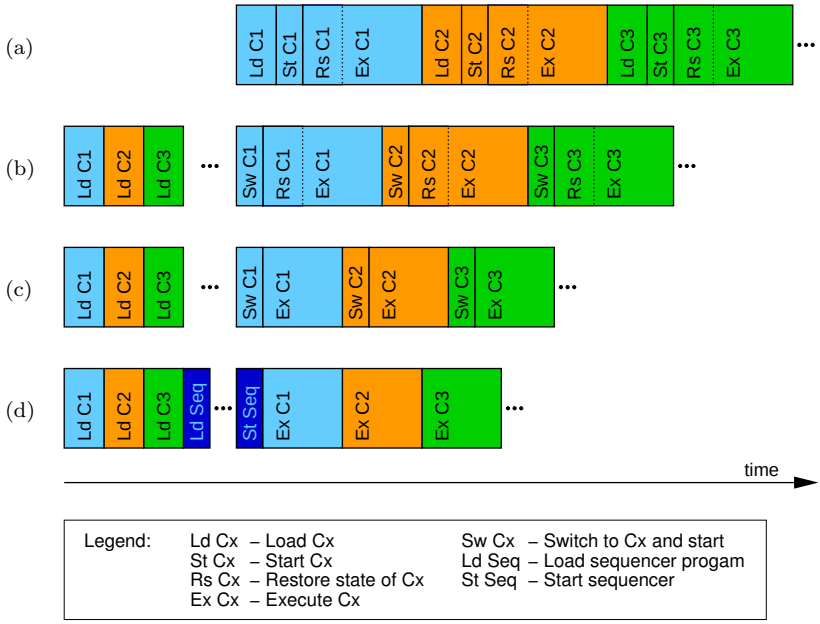
Figure 6.3 illustrates the execution flow of one data block for various system configurations considering a simplified application with only three logical contexts.

(a) 1 physical context, shared register set, no sequencer.
If only one physical context is present on the RPU, each logical context must be loaded on demand. After loading the desired context onto the RPU ("Ld Cx"), the CPU starts the execution of the context ("St Cx"). Then, the state of the last activation of this context needs to be restored ("Rs Cx"). Finally, the context executes ("Ex Cx"). This procedure is repeated for all logical contexts (and, accordingly, for all data blocks).

(b) 3 physical contexts, shared register set, no sequencer.
If the configuration memory provides sufficient physical contexts to hold all logical contexts on-chip, the logical contexts are loaded only once at the beginning of the computation. The overhead for activating a new context reduces to switching to the desired context and starting it ("Sw Cx"), plus restoring its state.

(c) 3 physical contexts, dedicated registers, no sequencer.
If the datapath registers are fully replicated, i. e. each logical context operates on its dedicated register set, the context state is kept automatically and no longer needs to be restored.

(d) 3 physical contexts, dedicated registers, sequencer.
The on-chip context sequencer reduces the overhead to initiating the start of every sequence ("St Seq"). The sequencer is programmed once at the beginning of the computation ("Ld Seq").



(a) 1 physical context, shared register set, no context sequencer
(b) 3 physical contexts, shared register set, no context sequencer
(c) 3 physical contexts, dedicated registers, no context sequencer
(d) 3 physical contexts, dedicated registers, context sequencer

Figure 6.3. Execution flow of one data block for various system setups considering a simplified application example with three logical contexts

6.3. Computational Performance

6.3.1. Results

For reference purposes, we have determined the performance results of the stand-alone CPU without attached RPU. The execution time of the FIR filter program processing the 64K samples has been measured as 110.65 million cycles. In this case, we use the non-cascaded implementation of the 56th-order FIR filter, which requires 57 taps.

Figures 6.4, 6.5, 6.6 and 6.7 illustrate the results of our experiments with the hybrid processor incorporating an RPU. The results are plotted as a function of the FIFO buffer size (shown on the horizontal axis) and the system configuration. Figures 6.4 and 6.5 show the speedups achieved relative to the execution time of the stand-alone CPU. The results shown in Figures 6.4 do not include the hardware context sequencer, those shown in Figures 6.5 do. Figures 6.6 and 6.7 show the CPU load normalized to the stand-alone CPU, again without and with making use of the context sequencer, respectively. That is, the load of the stand-alone CPU serves as a reference point, representing the 100% mark. Additionally, Tables 6.2 and 6.3 provide the detailed speedup figures, Tables 6.4 and 6.5 the detailed CPU load figures.

6.3.2. Discussion

In general, making use of the RPU for an application function shows two benefits:

- First, we accelerate the application function, which is measured by the speedup in the Figures 6.4 and 6.5 (Tables 6.2 and 6.3).
- Second, the CPU is relieved from some operations and can devote the free capacity to other functions. We measure this effect by the relative CPU load in the Figures 6.6 and 6.7 (Tables 6.4 and 6.5).

We make the following observations, with reference to the system configurations that use the hardware context sequencer:

- Using an RPU we achieve significant speedups, ranging up to a factor of 9.5 for an 8-context RPU with dedicated register sets. The performance deteriorates with decreasing FIFO size due to the imposed communication overhead.

- Enlarging the FIFOs increases the performance, but at the same time increases the filter delay. Practical applications could limit these potential gains by imposing delay constraints. For instance, a 2-context, shared-register-set RPU using FIFOs of 1K words instead of 128 words improves the speedup by a factor of 2.7, while increasing the latency by a factor of 8.

- Full register replication greatly benefits our application as we totally avoid the overlapping of data blocks. For an 8-context RPU with 128-word FIFOs, the speedup increases by a factor of 2. Additionally, the speedup compared to the stand-alone CPU becomes almost independent of the FIFO size because no context reloading is required.

- The RPU use lowers the CPU load significantly. For a single-context RPU with a shared register set, the CPU load drops from 100% to 28.3% for 128-word FIFOs and to 6.4% for 1K-word FIFOs. Increasing the number of physical contexts and providing dedicated register sets, the load approaches the asymptotic value of 4.7%. The CPU task reduces to transferring data and starting the context sequencer.

- As a rather surprising result we have found that the impact of using the on-chip context sequencer is moderate. For an 8-context, dedicated-register-sets RPU with 64-word FIFOs, the speedup improves by 8.2% and the CPU load drops by 18.3%. However, for fewer physical RPU contexts and with increasing FIFO size the improvement is not so large. The reason is that our coarse-grained architecture requires only a small amount of configuration data. With increasing FIFO size the context switch overhead becomes marginal.

Overall, the results emphasize the importance of our system-level, cycle-accurate simulation approach for architectural evaluation and optimization. As an example, Figure 6.5 shows that for most FIFO sizes a single-context RPU with dedicated register sets performs similarly to or even better than an 8-context RPU with a shared register set – a finding that is not obvious. Nevertheless, the speedup results show that incorporating multiple contexts on the RPU is valuable. For the FIR filter example in our case study though, register replication can be more beneficial.

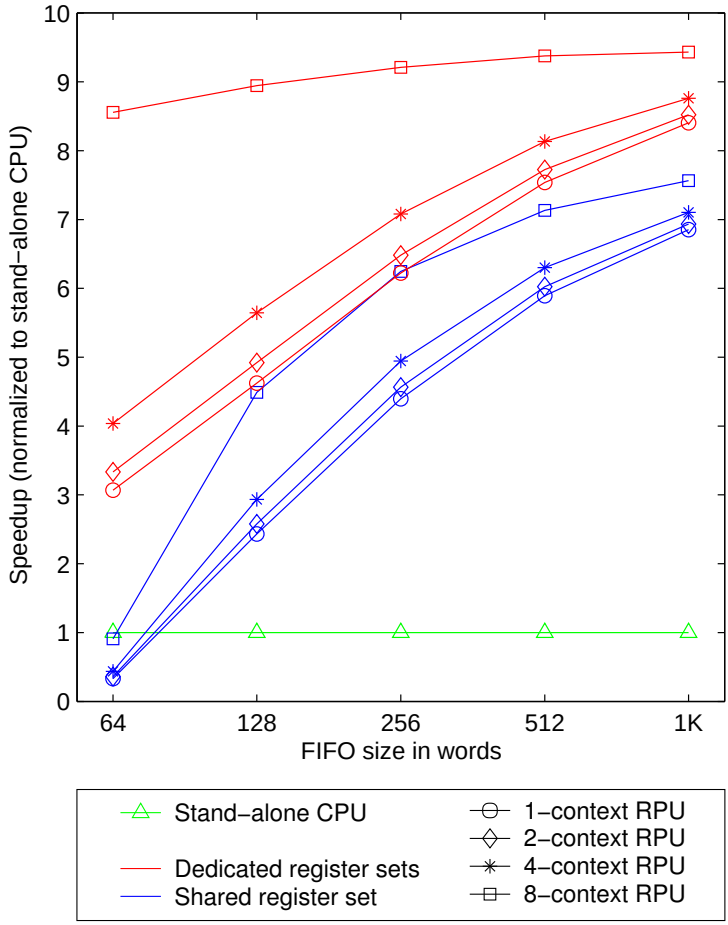


Figure 6.4. Speedup compared to the stand-alone CPU for RPUs *without* a hardware context sequencer

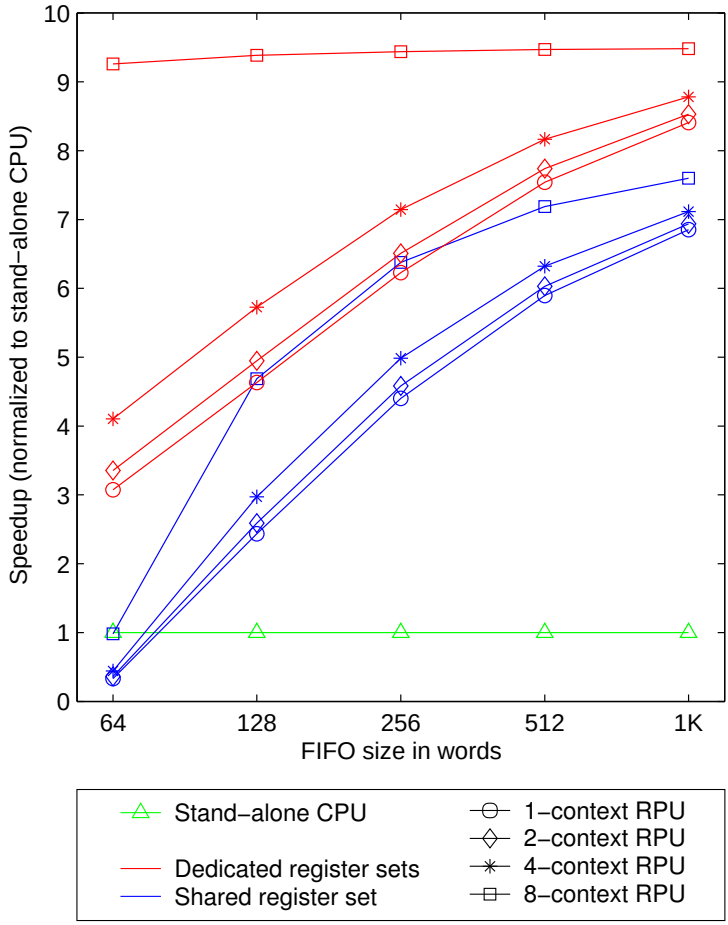


Figure 6.5. Speedup compared to the stand-alone CPU for RPUs *with* a hardware context sequencer

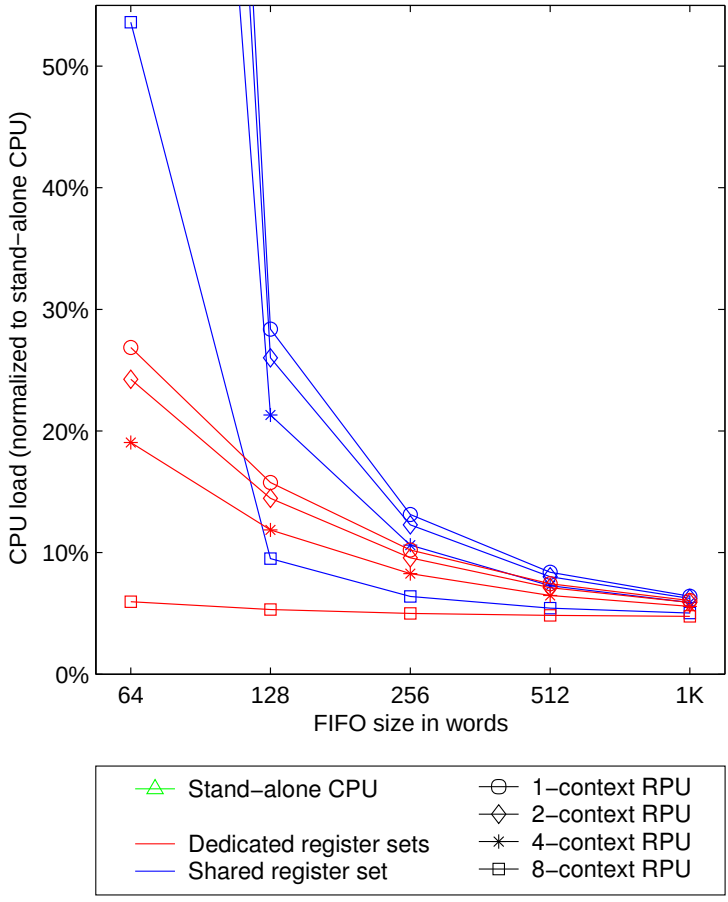


Figure 6.6. CPU load normalized to the stand-alone CPU for RPUs *without* a hardware context sequencer (only the region of interest is shown)

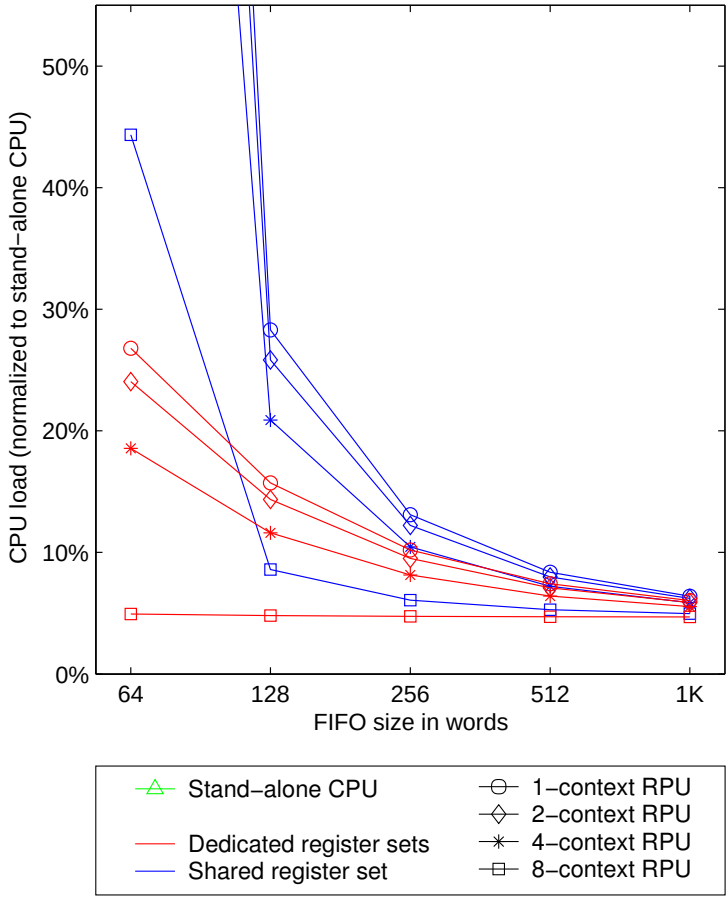


Figure 6.7. CPU load normalized to the stand-alone CPU for RPUs *with* a hardware context sequencer (only the region of interest is shown)

Table 6.2. Speedup compared to the stand-alone CPU for RPUs *without* a hardware context sequencer (cf. Figure 6.4)

Register set	RPU contexts	Speedup				
		FIFO size in words				
		64	128	256	512	1024
Shared	1	0.33	2.43	4.40	5.89	6.85
	2	0.36	2.58	4.57	6.02	6.93
	4	0.44	2.93	4.94	6.30	7.10
	8	0.91	4.49	6.25	7.13	7.57
Dedicated	1	3.07	4.62	6.22	7.54	8.41
	2	3.33	4.92	6.48	7.73	8.52
	4	4.04	5.64	7.08	8.13	8.76
	8	8.56	8.95	9.21	9.38	9.43

Table 6.3. Speedup compared to the stand-alone CPU for RPUs *with* a hardware context sequencer (cf. Figure 6.5)

Register set	RPU contexts	Speedup				
		FIFO size in words				
		64	128	256	512	1024
Shared	1	0.33	2.43	4.40	5.90	6.85
	2	0.36	2.59	4.58	6.03	6.94
	4	0.44	2.97	4.98	6.32	7.12
	8	0.98	4.69	6.38	7.19	7.60
Dedicated	1	3.07	4.63	6.23	7.54	8.41
	2	3.35	4.95	6.51	7.74	8.53
	4	4.10	5.73	7.14	8.16	8.78
	8	9.26	9.39	9.44	9.47	9.48

Table 6.4. CPU load normalized to the stand-alone CPU for RPUs *without* a hardware context sequencer (cf. Figure 6.6)

Register set	RPU contexts	CPU load in %				
		FIFO size in words				
		64	128	256	512	1024
Shared	1	245.0	28.4	13.1	8.4	6.4
	2	221.1	26.0	12.3	8.0	6.2
	4	173.4	21.3	10.6	7.3	5.9
	8	53.6	9.5	6.4	5.4	5.0
Dedicated	1	26.9	15.8	10.2	7.4	6.1
	2	24.3	14.5	9.6	7.1	5.9
	4	19.0	11.9	8.3	6.5	5.6
	8	6.0	5.3	5.0	4.8	4.8

Table 6.5. CPU load normalized to the stand-alone CPU for RPUs *with* a hardware context sequencer (cf. Figure 6.7)

Register set	RPU contexts	CPU load in %				
		FIFO size in words				
		64	128	256	512	1024
Shared	1	244.3	28.3	13.1	8.4	6.4
	2	219.2	25.8	12.2	8.0	6.2
	4	169.0	20.9	10.5	7.2	5.9
	8	44.4	8.6	6.1	5.3	5.0
Dedicated	1	26.8	15.7	10.2	7.4	6.0
	2	24.1	14.4	9.5	7.1	5.9
	4	18.6	11.6	8.1	6.4	5.5
	8	4.9	4.8	4.7	4.7	4.7

6.4. Chip Area

6.4.1. Estimation Model

For the area requirements of the RPU we consider the area contributions of the following building blocks:

- the reconfigurable array (A_{Array}),
- the configuration memory (A_{CMem}),
- the two FIFO buffers ($2 \cdot A_{\text{FIFO}}$),
- the hardware context sequencer (A_{CSeq}), and
- the coprocessor register file (A_{CRF}).

The area taken up by the I/O port controllers and the various small hardware controllers integrated on the RPU is currently not taken into account. Thus, considering (3.1) the area of the RPU follows

$$A_{\text{RPU}} = a_{\text{rout}} \cdot \left(A_{\text{Array}} + A_{\text{CMem}} + 2A_{\text{FIFO}} + A_{\text{CSeq}} + A_{\text{CRF}}\right) \ , \quad (6.3)$$

where a_{rout} denotes the overall area overhead of the routing. Based on an empirical value of 20% [93], we choose a slightly increased routing overhead of 25% in order to compensate for the routing between the building blocks as well as for the neglected parts of the RPU such as the hardware controllers.

6.4.2. Building Block Area Models

For the building blocks, we define parameterized area models represented by the equations (6.4) through (6.8). The individual area models are discussed in detail below.

$$A_{\text{Array}} = n_{\text{cell}} \cdot \left(A_{\text{cell}} + \alpha_{\text{FF}}\left((f_{\text{repl}} - 1) \cdot n_{\text{reg}} \cdot w_{\text{reg}}\right)\right) \ , \quad (6.4)$$

$$A_{\text{CMem}} = c \cdot \alpha_{\text{Latch}}(s_{\text{cfg}}) \ , \quad (6.5)$$

$$A_{\text{FIFO}} = \text{minarea}(s_{\text{FIFO}} \cdot w_{\text{FIFO}}) \ , \quad (6.6)$$

$$A_{\text{CSeq}} = \text{minarea}(n_{\text{CStore}} \cdot w_{\text{CStore}}) + \alpha_{\text{FF}}(w_{\text{SPC}}) \ , \quad (6.7)$$

$$A_{\text{CRF}} = \alpha_{\text{FF}}(w_{\text{CC}}) + \alpha_{\text{FF}}(w_{\text{CS}}) + \alpha_{\text{FF}}(w_{\text{CSS}}) \ . \quad (6.8)$$

We make use of the area functions $\alpha_{\text{FF}}()$, $\alpha_{\text{Latch}}()$, and $\alpha_{\text{SRAM}}()$ introduced in (3.2), (3.3), and (3.4), respectively. Furthermore, we define

Table 6.6. Area for various configurations of the reconfigurable cell determined through synthesis using a standard cell library for a 0.25 μm UMC CMOS process maintained by Virtual Silicon

Register replication	Area of reconfigurable cell in Mλ^2			
	Datapath width in bits			
f_{repl}	4	8	16	32
1	1.754	3.622	9.077	25.931
2	2.064	4.303	9.477	26.407
4	2.787	5.416	12.680	31.309
8	3.950	8.512	17.930	42.526

the auxiliary function minarea(s), which chooses the more area-efficient implementation of either registers or SRAM memory for a given storage capacity s, i. e.

$$\text{minarea}(s) := \min\left(\alpha_{\text{FF}}(s), \alpha_{\text{SRAM}}(s)\right) \ . \tag{6.9}$$

Reconfigurable array

The reconfigurable array is composed of a 4×4 array of reconfigurable cells. The number of reconfigurable cells (n_{cell}) is therefore 16. We have determined the area of an individual cell featuring a single register set (A_{cell}) through RTL synthesis using a standard cell library for a 0.25 μm UMC CMOS process maintained by Virtual Silicon [179]. For the synthesis, we employed the Synopsys Design Compiler tool suite [164]. If the datapath registers are replicated, the additional area depends on the replication factor (f_{repl}), the number of registers within an individual cell (n_{reg}), and the register width in bits (w_{reg}).

We have synthesized various configurations of the reconfigurable cell varying the datapath width and the number of replicated datapath registers. Table 6.6 lists the synthesis results. If we compare the synthesis results with the area model, we observe that for a replication factor of 2 the results are accurate, while for higher replication factors the model underestimates the chip area. The difference can be explained by the neglected routing overhead for the additional register planes, which is more significant for higher replication factors.

Configuration memory

For simplicity reasons, the configuration memory is considered as a self-contained building block. In an actual silicon implementation, however, the configuration memory is most likely to be distributed over the reconfigurable array in order to have the configuration bits located physically near the reconfigurable cells. The configuration memory holds a certain number of physical contexts (c). The size of a configuration in bits is denoted s_{cfg}. Since we deal with rather small configuration sizes, we assume that the configuration memory is implemented with latches rather than in SRAM technology.

FIFO buffer

The area of a FIFO buffer depends on the FIFO size in words (s_{FIFO}) and the data width (w_{FIFO}), which ranges in our experiments from 4 to 32 bits. Consequently, we have to cover a rather large range in terms of storage capacity and it is not a priori clear, which implementation is most area efficient (registers or SRAM). For this reason, we make use of the minarea() function, which selects the more area-efficient solution of either registers (built from flip-flops) or SRAM.

Hardware context sequencer

The hardware context sequencer consists of the program store and the sequence program counter (SPC). The program store is characterized by the number of sequencer instructions it holds (n_{CStore}) and the instruction width in bits (w_{CStore}). As with the FIFO buffers we choose the more area-efficient solution comparing register and SRAM implementations. The SPC is implemented as a register of a certain bit width (w_{SPC}), which corresponds to the number of instructions in the program store. We assume a program store holding 64 32-bit instructions and, thus, employ a 6-bit wide SPC.

Coprocessor register file

We also incorporate the area of the coprocessor registers that are physically implemented and not covered by other area models. We consider the bit widths of the *cycle count* register (w_{CC}), the *context select* register (w_{CS}) and the *context sequencer status* register (w_{CSS}). We assume a 32-bit *cycle count* and a 16-bit *context select* register. The *context sequencer status* register consists of a single bit.

6.4.3. Results and Discussion

Figures 6.8 and 6.9 depict the results of the area estimation as a function of the FIFO buffer size (shown on the horizontal axis) and the system configuration. Figure 6.8 assumes an 8-bit RPU datapath, Figure 6.9 a 16-bit RPU datapath. Tables 6.7 and 6.8 list the detailed area figures.

The area figures show the quantitative impact of the design parameters such as FIFO buffer size, register replication and multiple contexts. We make the following observations, particularly with reference to the RPUs with a 16-bit datapath:

- Integrating eight physical contexts instead of a single context results in an additional area requirement of 46.6 $M\lambda^2$. Comparing the single-context, shared-register-set RPU featuring 128-word FIFO buffers with its 8-context counterpart reveals an area increase of 21%.

- Full register replication, i. e. in this case eight times replicated registers, results in an additional area requirement of 60.0 $M\lambda^2$. For a single-context RPU with 128-word FIFO buffers this constitutes an area increase of 27%.

- Increasing the FIFO buffer size from 128 to 1K words, results in an additional area requirement of 82.4 $M\lambda^2$. For a single-context RPU featuring a shared register set this represents an area increase of 37%.

For a selection of RPU architectures, Figure 6.10 illustrates the area breakdown, which reveals the relative area contributions of the major building blocks. The four selected RPUs all feature a 16-bit datapath and incorporate 512-word FIFOs. In all cases, the reconfigurable array is the major area contributor, followed by the FIFO buffers. The contribution of the coprocessor registers, on the other hand, can be neglected in the current RPU implementations. The pie charts also illustrate the relative increase in the RPU area for

1. employing an 8-context RPU instead of a single-context RPU (left versus right diagrams), as well as

2. fully replicating the datapath registers (upper versus lower diagrams).

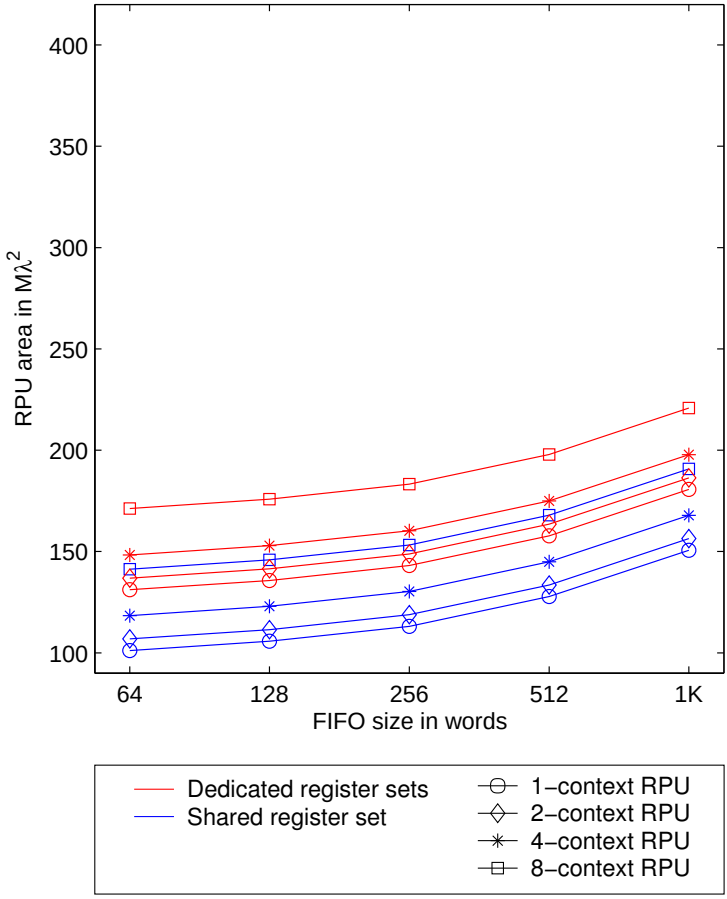


Figure 6.8. Area of the 8-bit datapath RPUs

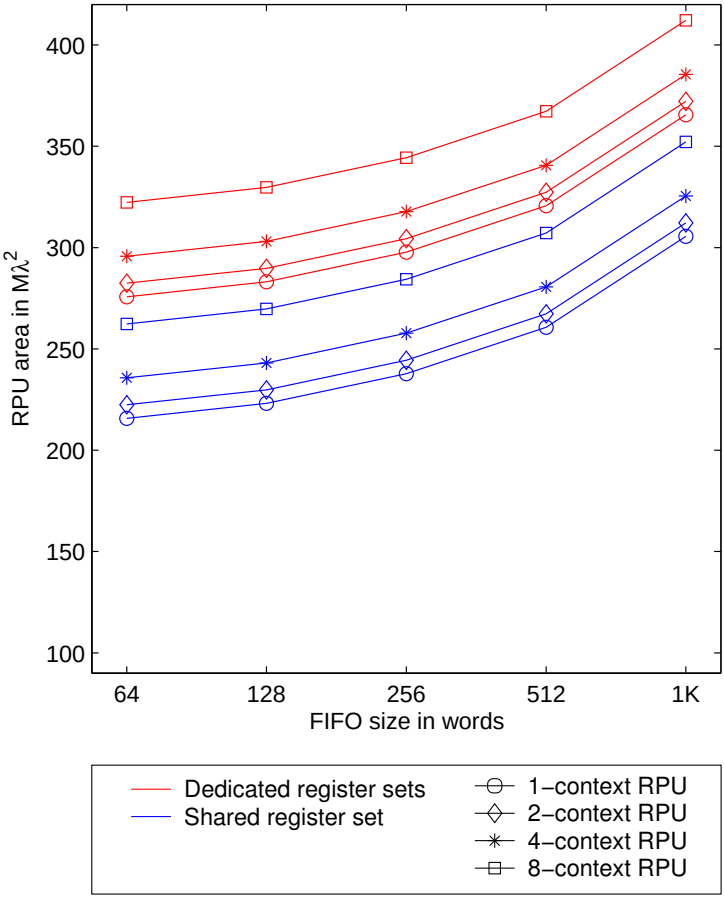


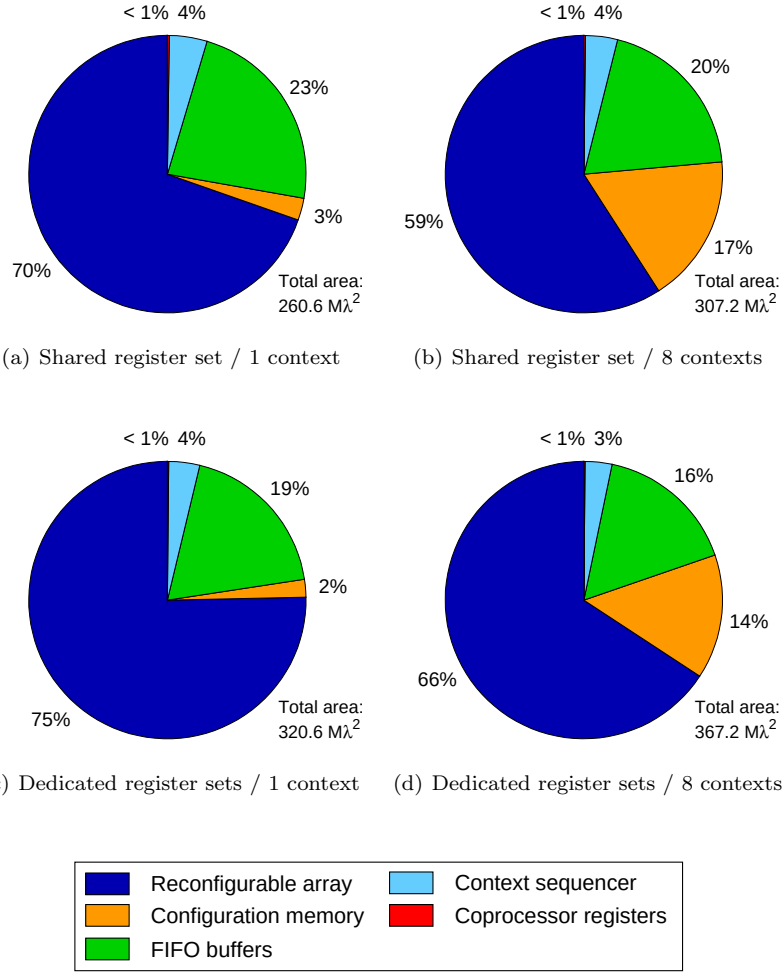
Figure 6.9. Area of the 16-bit datapath RPUs

Table 6.7. Area figures of the 8-bit datapath RPUs (cf. Figure 6.8)

Register set	RPU contexts	RPU area in $M\lambda^2$				
		FIFO size in words				
		64	128	256	512	1024
Shared	1	101.1	105.7	113.1	127.8	150.6
	2	106.9	111.5	118.8	133.5	156.4
	4	118.3	122.9	130.3	145.0	167.8
	8	141.3	145.9	153.2	167.9	190.8
Dedicated	1	131.1	135.7	143.1	157.8	180.6
	2	136.9	141.5	148.8	163.5	186.4
	4	148.3	152.9	160.3	175.0	197.8
	8	171.3	175.9	183.2	197.9	220.8

Table 6.8. Area figures of the 16-bit datapath RPUs (cf. Figure 6.9)

Register set	RPU contexts	RPU area in $M\lambda^2$				
		FIFO size in words				
		64	128	256	512	1024
Shared	1	215.8	223.1	237.8	260.6	305.5
	2	222.4	229.8	244.4	267.3	312.2
	4	235.7	243.1	257.8	280.6	325.5
	8	262.3	269.7	284.4	307.2	352.1
Dedicated	1	275.7	283.1	297.8	320.6	365.5
	2	282.4	289.8	304.4	327.3	372.2
	4	295.7	303.1	317.8	340.6	385.5
	8	322.3	329.7	344.4	367.2	412.1



(a) Shared register set / 1 context

(b) Shared register set / 8 contexts

(c) Dedicated register sets / 1 context

(d) Dedicated register sets / 8 contexts

Figure 6.10. Area breakdown for a selection of 16-bit datapath RPUs (with 512-word FIFOs), revealing the relative area contributions of the major building blocks

The results show again the importance of the system-level evaluation approach. As an example, if our starting point is a single-context, shared-register-set RPU with 128-word FIFOs and we can afford to employ some more chip area, say 60 Mλ^2, Figure 6.9 shows that we can then either switch to

- an RPU with fully replicated registers,
- an 8-context RPU, or
- a 4-context RPU with 256-word FIFOs.

Comparing the results for the 8-bit datapath RPUs with the results for the 16-bit datapath RPUs, the picture changes. The proportional impact of providing multiple contexts and replicating the datapath registers changes in comparison to each other. In contrast to the 16-bit case, the single-context, dedicated-register-sets RPUs with 8-bit datapath show lower area requirements than their 8-context, shared-register-set counterparts. Consequently, we observe that the datapath width is an important design parameter as well and cannot be neglected in the trade-off considerations.

6.5. Area–Speed Trade-offs

In order to compare the various processor architectures in terms of computational power and area requirements, it is helpful to be able to relate the performance and area figures to one another. To perform such comparisons at the system level, besides the RPU we also need to consider the CPU core. To this end, we assume a hypothetical embedded CPU core operating at 100 MHz with an area requirement of 1500 Mλ^2 (cf. Table 3.1).

Figure 6.11 illustrates the evaluation results as a function of the execution time (on the horizontal axis) and the overall area of the hybrid architectures with an attached 16-bit RPU. Results are shown for the same RPU architectures as in the previous performance and area figures. Table 6.9 provides the detailed area and execution time numbers.

The figure allows the Pareto-optimal design points to be determined. In this context, a design point is considered Pareto-optimal if there exists no other design point that performs better on both criteria, i. e. area requirements and execution time. Assuming no further constraints, such as for example a maximal latency, we make the following observations:

- The 8-context RPUs with dedicated registers are all Pareto-optimal.
- Many of the dedicated-register-sets RPUs are dominated by shared-register-set RPUs. This holds in particular true for decreasing FIFO sizes.
- The single-context RPUs with shared register set become Pareto-optimal for FIFO sizes smaller than 512 words.

A common metric for the *efficiency* of a computing architecture is the area–time product [49, 150, 193]. This metric is similar to the (inverse) throughput but incorporates the area costs as well. Figure 6.12 shows the results of the area–time product for the architectures with 16-bit RPU as a function of the FIFO size (on the horizontal axis). Table 6.10 lists the detailed results. We make the following observations:

- The architectures incorporating an 8-context RPU with dedicated register sets perform the best for any FIFO buffer size.
- For these architectures, the RPU incorporating 128-word FIFO buffers is the optimum with respect to the area–time product. For all other cases, the optimum is not apparent and lies higher than 1K-word FIFOs.
- The single-context, dedicated-register-sets RPUs perform better than most multi-context, shared-register-set RPUs. The only exceptions are the 8-context RPUs incorporating FIFOs of size 128 and 256. This emphasizes the observation again that for the FIR filter example in our case study, register replication is in many cases more beneficial than using multiple contexts.

In comparison, the area–time product for the stand-alone hypothetical CPU is $1659.8\,\mathrm{M}\lambda^2\mathrm{s}$. Thus, the coupling of an RPU to the CPU improves the area–time product up to a factor of 7.7.

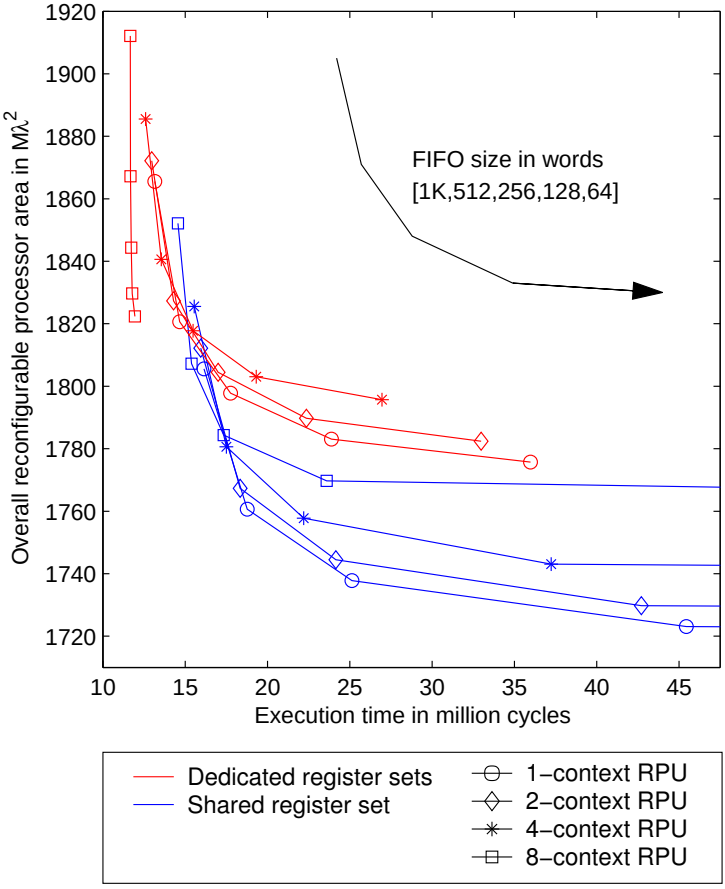


Figure 6.11. Area requirements versus execution time for processor architectures considering a hypothetical embedded CPU core and an attached 16-bit RPU (only the region of interest is shown)

Table 6.9. Area requirements and execution time figures for processor architectures with a hypothetical embedded CPU core and an attached 16-bit RPU (cf. Figure 6.11)

Register set	RPU contexts	Area in $M\lambda^2$ / Execution time in million cycles				
		FIFO size in words				
		64	128	256	512	1024
Shared	1	1715.8 / 332.64	1723.1 / 45.45	1737.8 / 25.14	1760.6 / 18.77	1805.5 / 16.14
	2	1722.4 / 305.21	1729.8 / 42.71	1744.4 / 24.15	1767.3 / 18.35	1812.2 / 15.95
	4	1735.7 / 250.02	1743.1 / 37.24	1757.8 / 22.20	1780.6 / 17.50	1825.5 / 15.55
	8	1762.3 / 112.72	1769.7 / 23.60	1784.4 / 17.35	1807.2 / 15.39	1852.1 / 14.56
Dedicated	1	1775.7 / 35.99	1783.1 / 23.89	1797.8 / 17.77	1820.6 / 14.67	1865.5 / 13.16
	2	1782.4 / 32.99	1789.8 / 22.37	1804.4 / 17.00	1827.3 / 14.30	1872.2 / 12.97
	4	1795.7 / 26.96	1803.1 / 19.32	1817.8 / 15.49	1840.6 / 13.55	1885.5 / 12.60
	8	1822.3 / 11.95	1829.7 / 11.79	1844.4 / 11.72	1867.2 / 11.68	1912.1 / 11.67

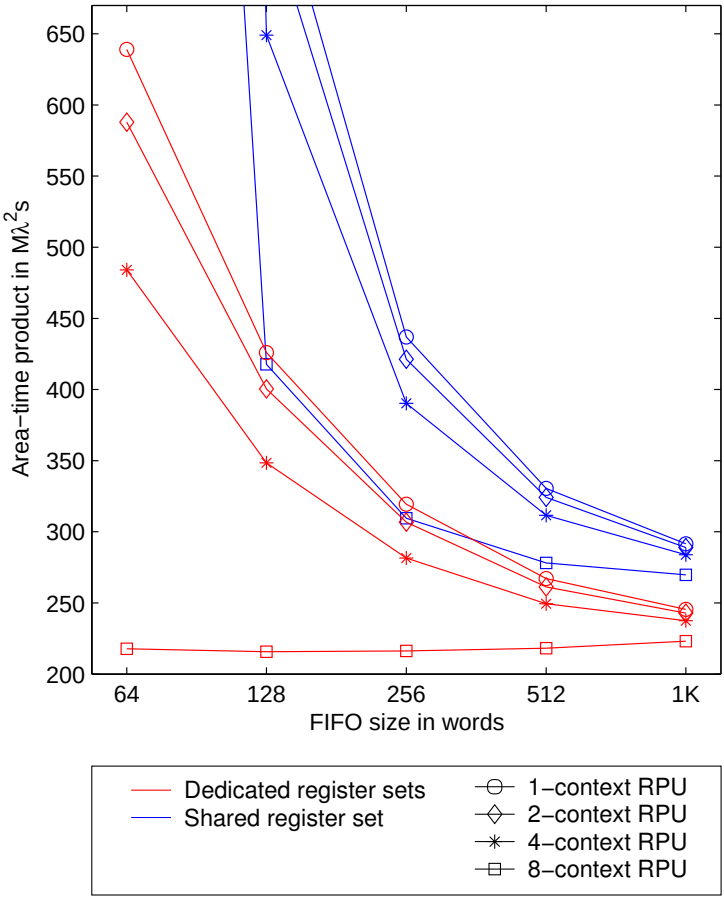


Figure 6.12. Area–time product for processor architectures with a hypothetical embedded CPU core and an attached 16-bit RPU operating at 100 MHz (only the region of interest is shown)

Table 6.10. Area–time product for processor architectures with a hypothetical embedded CPU core and an attached 16-bit RPU operating at 100 MHz (cf. Figure 6.12)

Register set	RPU contexts	Area–time product in $M\lambda^2 s$				
		FIFO size in words				
		64	128	256	512	1024
Shared	1	5707.3	783.1	436.9	330.4	291.5
	2	5256.9	738.8	421.3	324.2	289.0
	4	4339.6	649.1	390.2	311.6	283.8
	8	1986.5	417.7	309.6	278.1	269.7
Dedicated	1	639.0	425.9	319.4	267.1	245.5
	2	588.0	400.4	306.8	261.3	242.8
	4	484.1	348.4	281.5	249.4	237.5
	8	217.8	215.7	216.2	218.2	223.1

7

Conclusion

7.1. Summary and Achievements

In the last decade, reconfigurable computing has gained increasing interest in the research community. Reconfigurable architectures promise to become a valuable alternative to conventional computing devices such as processors and ASICs. This work focuses on hybrid, dynamically reconfigurable processors that combine a CPU core with a reconfigurable processing unit. The objective of this work is the quantitative evaluation of the architectural trade-offs involved in the design of such reconfigurable architectures – in particular with respect to computational performance and area requirements. The major contributions are the following:

- We have proposed a system-level evaluation methodology for hybrid reconfigurable processors. The methodology consists of an architecture model, a cycle-accurate co-simulation environment and a parameterized area model. The evaluation approach enables us to study the system-level impact of the architectural design features on the computational performance and the area requirements.

- To measure the computational performance, we have developed a co-simulation framework for hybrid reconfigurable processors that attach a reconfigurable processing unit to the coprocessor port of a standard CPU core. The co-simulation framework integrates a C and a VHDL simulator and allows us to gather cycle-accurate performance statistics for the overall reconfigurable system.

- To estimate the area requirements of the hybrid reconfigurable processor architectures, we have elaborated a parameterized area model. For the RPU, we combine area models of architectural building blocks, such as registers or SRAM memory, with data from VLSI synthesis. Together with data found in the literature regarding CPU area, this approach allows us to estimate the overall chip area in a rather straight-forward way.

- In contrast to most work on reconfigurable processors, we aim at the embedded and not the general-purpose computing domain. To represent a typical embedded workload, we have assembled an application pool denoted MCCmix from the fields of multimedia, cryptography and communications. We have analyzed the applications in order to quantitatively characterize the application domain and to draw conclusions for the design of reconfigurable processor technology.

- We have presented a hybrid, dynamically reconfigurable processor architecture that aims at data-streaming applications. The architecture couples a multi-context, coarse-grained reconfigurable array to a CPU core as a coprocessor and allows for fast switching of the active context. We have described hardware virtualization which abstracts limited reconfigurable hardware resources as the envisioned programming model.

- As a case study, we have implemented FIR filters of arbitrary order on different configurations of the proposed RPU architecture. To this end, we have made use of hardware virtualization. We have evaluated different system configurations and investigated a number of RPU design features – such as multiple contexts, register replication, FIFO buffer size and context scheduling in hardware – in terms of computational performance and chip area.

7.2. Conclusions

We draw the following conclusions:

1. The results achieved illustrate the importance of the system-level evaluation approach. Without a system-level methodology, the architectural trade-offs involved in reconfigurable processor design could not be satisfactorily analyzed. The cycle-accurate co-simulation framework developed has shown itself to be a valuable tool to investigate the computational performance of hybrid reconfigurable architectures. The estimation approach for the overall chip area has proven convenient for gathering system-level area figures. Further investigations of both the currently neglected parts of the system and the adjustment of the model to the results of VLSI synthesis would allow us to increase the accuracy.

2. The application analysis of the assembled embedded workload has revealed that the general-purpose and embedded computing domains feature significantly different characteristics. Within the embedded domain, the three application groups considered (multimedia, cryptography, and communications) show distinctive requirements as well and hence emphasize different architectural features of a reconfigurable processor. Consequently, it is important to account for the peculiarities of the targeted application field when designing reconfigurable processor technology.

3. The results from the case study show that hardware virtualization is a powerful concept and that multi-context features can successfully support this programming model. The results show furthermore that hybrid multi-context architectures have the potential to yield significant speedups. In the FIR filter case, we have achieved speedups of up to an order of magnitude over the stand-alone CPU by employing an RPU that comprises a 4×4 array of reconfigurable cells. Specifically, we have seen that in most cases register replication delivers more benefit than multiple contexts. Another insight is that the size of the FIFO buffers used for transferring data between CPU and RPU has a major impact. A rather surprising result is that the benefit of using a hardware context sequencer is rather moderate. The reason for this is that our coarse-grained architecture requires only a small amount of configuration data and, thus, the overhead of a context switch is rather small.

7.3. Outlook

There are several starting points for further research, which we briefly discuss in the following paragraphs.

The architecture model presented makes the assumption that the RPU is attached to the CPU via a coprocessor interface. Our experiments have revealed that the integration of a dedicated RPU memory port to the main memory promises to be a valuable extension. This would relieve the CPU load and increase the bandwidth. To achieve such a system integration, the co-simulation framework has to be extended and an appropriate memory controller integrated. Furthermore, the potential data consistency problem which might arise has to be addressed.

Regarding the RPU, a number of design features are worth further investigation, including larger reconfigurable arrays consisting for example of 8×8 or 16×16 cells, the consideration of context prediction and prefetching techniques, and the implementation of applications that require more complex context sequences and hence require the RPU to provide control flow features. A further promising option is the incorporation of more memory as well as a programmable memory controller into the RPU. This would allow for more complex memory access patterns and open new application fields besides data-streaming applications. In a larger context, the potential of the hardware virtualization programming model could be explored further by extending the RPU to several physical arrays arranged in a macro-pipeline.

The proposed methodology enables the designer to evaluate reconfigurable architectures in terms of computational performance and chip area. For the targeted domain of embedded computing systems, however, further optimization objectives are of interest such as the energy consumption. Consequently, the integration of an energy or power simulator would be a valuable extension to the evaluation methodology. On the CPU side, a number of approaches exist. An obvious option in our case are SimpleScalar extensions like Wattch, PowerAnalyzer or SimplePower. However, these simulators are currently in a development stage and, in our experience, do not yet operate reliably. On the RPU side, a power model would have to be developed. Approaches at several levels of abstractions are viable, from coarse activity models to detailed, cycle-accurate switching models.

The simulation framework developed relies so far on a library-based approach to generate code for the CPU and reconfigurable processing unit. Basic functions have been constructed that exploit the reconfigurable array. These functions are provided as a library to the programmer. This represents a common approach in reconfigurable systems. However, based on our simulation framework, more advanced techniques could be explored. A currently very active research topic is compilers that automatically extract runtime-intensive functions from high-level language programs and synthesize code that targets the hybrid reconfigurable processor.

The problem of finding an optimal architecture in terms of computational power, chip area, energy consumption, etc. can be seen as a multi-objective optimization problem. As a long-term vision, we would like to apply more formal approaches by using stochastic search procedures, e. g. evolutionary algorithms, in order to find Pareto-optimal design points. For stand-alone CPUs the first approaches in this direction already exist [3, 96, 159]. To achieve an automated framework for the overall hybrid reconfigurable processor, however, advances in automatic code generation, i. e. compilers, and appropriate simulator or estimation extensions, such as a power simulator, are required.

Glossary

Symbols

$\alpha_{\mathrm{FF}}(s)$	area of flip-flop-based registers with storage capacity s
$\alpha_{\mathrm{Latch}}(s)$	area of latch-based registers with storage capacity s
$\alpha_{\mathrm{SRAM}}(s)$	area of on-chip SRAM with storage capacity s
λ	half the minimum feature size of a process technology
μ_{cmiss}	average cache miss rate over a set of applications
$\mu_{\mathrm{iclass}}(c)$	average relative issuing frequency of instruction class c over a set of applications
$\mu_{\mathrm{T}i}$	average cumulative relative run time of function f_i over a set of applications
a_{rout}	routing area factor of the RPU
$\mathbb{A}$	set of applications
A_{Array}	area of the reconfigurable array
A_{BB_b}	area of building block b
A_{cell}	area of a reconfigurable cell without replicated registers
A_{CMem}	area of the configuration memory
A_{CRF}	area of the coprocessor register file
A_{CSeq}	area of the hardware context sequencer
A_{FIFO}	area of a FIFO buffer
A_{RPU}	area of the RPU
c	number of contexts the configuration memory can hold
f_1, f_2, f_i	most, 2nd most, ith most compute-intensive (dominating) program function
$f_{\overline{123}}$	all program functions except the three dominating ones
$f_{\mathrm{iclass}}(c)$	relative issuing frequency of instruction class c
$f_{\mathrm{itype}}(i)$	relative issuing frequency of instruction type i
f_{repl}	replication factor of the datapath registers
h_i	filter coefficients
$H(z)$	filter transfer function
$\mathbb{I}_c$	set of instructions forming instruction class c

$\text{minarea}(s)$	area requirement of the most efficient technology for storage capacity s
$n_{\mathbb{A}}$	number of applications in $\mathbb{A}$
n_{caccess}	total number of cache accesses
n_{cell}	number of cells constituting the reconfigurable array
n_{cmiss}	number of cache misses
n_{CStore}	number of instructions the program store can hold
n_{cycles}	total number of execution cycles
$n_{\text{fct}}(f)$	number of cycles spent in program function f
n_{inst}	total instruction count of an application
$n_{\text{itype}}(i)$	absolute issuing frequency of instruction type i
n_{reg}	number of datapath registers in a reconfigurable cell
r_{cmiss}	cache miss rate
s_{cfg}	configuration size in bits
s_{FIFO}	size of a FIFO buffer in words
$t_{\text{fct}}(f)$	relative run time spent in program function f
$T_{\text{fct}}(f)$	cumulative relative run time of program function f
w_{CC}	width of the *cycle count* register in bits
w_{CS}	width of the *context select* register in bits
w_{CSS}	width of the *context sequencer status* register in bits
w_{CStore}	width of a context sequencer instruction in bits
w_{FIFO}	width of a FIFO buffer in bits
w_{reg}	width of the datapath registers in bits
w_{SPC}	width of the sequence program counter (SPC) in bits

Acronyms and Abbreviations

ADPCM	adaptive differential pulse code modulation
AES	advanced encryption standard
ALU	arithmetic and logic unit
ASIC	application-specific IC
BTB	branch target buffer
CCITT	International Consultative Committee on Telecommunications and Telegraphy
CINT95	SPEC95 integer benchmark suite
CMOS	complementary metal oxide semiconductor
COM	communications
CPU	central processing unit
CRF	coprocessor register file
CRY	cryptography
CSoC	configurable system on a chip
CTR	compile-time reconfiguration
DES	data encryption standard
DL1	L1 data cache
DL2	L2 data cache
DSP	digital signal processor or processing
DTLB	data TLB
EEMBC	Embedded Microprocessor Benchmark Consortium (www.eembc.org)
FIFO	first-in, first-out
FIR	finite impulse response
FP	floating point
FPGA	field-programmable gate array
FU	functional unit
G	standard prefix for billion (10^9) or giga
GCC	GNU C compiler
GDB	GNU project debugger
GNU	recursive acronym for "GNU's Not Unix"; the Free Software Foundation's project to provide a freely distributable replacement for Unix (www.gnu.org)

GP	general purpose
GSM	global system for mobile communications
IC	integrated circuit
ID	identification
IDEA	international data encryption algorithm
IFQ	instruction fetch queue
IIR	infinite impulse response
IL1	L1 instruction cache
IL2	L2 instruction cache
I/O	input/output
IP	[1] Internet protocol; [2] intellectual property
IPx	input port x
ISA	instruction set architecture
ITLB	instruction TLB
JPEG	Joint Photographic Experts Group (www.jpeg.org)
k	decimal thousand (1000)
K	binary thousand (1024)
Kbit	kilobit
Kbyte	kilobyte
L1	first-level or primary (cache)
L2	second-level or secondary (cache)
LSB	least significant bit
LSQ	load/store queue
LUT	look-up table
M	standard prefix for million or mega
MCCmix	set of embedded benchmark programs from the fields of multimedia, cryptography, and communications
MIPS	[1] million instructions per second; [2] MIPS Technologies Inc. (www.mips.com)
MM	multimedia
MPEG	Moving Picture Experts Group (www.chiariglione.org/mpeg)
Mbit	megabit
Mbyte	megabyte

NIST	U.S. National Institute of Standards and Technology (www.nist.gov)
OPx	output port x
PCI	peripheral component interconnect
PGP	pretty good privacy; cryptography program
PISA	portable instruction set architecture
PLD	programmable logic device
RAM	random access memory
RAS	return address stack
RFU	reconfigurable functional unit
RISC	reduced instruction set computer
RPU	reconfigurable processing unit
RSA	cryptography algorithm named after Ronald Rivest, Adi Shamir and Leonard Adleman
RTL	register–transfer level
RTR	run-time reconfiguration
RUU	register update unit
SPC	sequence program counter
SPEC	Standard Performance Evaluation Corporation (www.spec.org)
SPEC95	general-purpose benchmark suite by SPEC
SRAM	static RAM
SoC	system on a chip
SoPC	system on a programmable chip
TLB	translation look-aside buffer
VHDL	VHSIC hardware description language
VHSIC	very high speed integrated circuit
VLIW	very long instruction word
VLSI	very large-scale integration

Bibliography

[1] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, and N. Bagherzadeh. VLSI design of the Tiny RISC microprocessor. In *Proc. 14th IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 30.4.1–30.4.5, 1992. 15

[2] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, and J. Rabaey. Evaluation of a low-power reconfigurable DSP architecture. In *Proc. 5th Reconfigurable Architectures Workshop (RAW)*, vol. 1388 of *Lecture Notes in Computer Science*, pp. 55–60. Springer-Verlag, 1998. 8

[3] S. Agarwal, E. Chan, B. Liblit, and C. J. Lin. Processor characteristic selection for embedded applications via genetic algorithms. CS252 semester project report, EECS, UC Berkeley, Dec. 1998. Available at http://www.cs.berkeley.edu/~liblit/darwin/. 121

[4] A. A. Aggarwal and D. M. Lewis. Routing architectures for hierarchical field programmable gate arrays. In *Proc. 12th Int. Conf. on Computer Design (ICCD)*, pp. 475–478, 1994. 20

[5] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. Determining the optimum extended instruction-set architecture for application specific reconfigurable VLIW CPUs. In *Proc. 12th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pp. 50–56, 2001. 22

[6] Altera. *Stratix Device Handbook (Vol. 1–3)*, Oct. 2003. Available at http://www.altera.com/. 18, 19, 20

[7] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac – configurable custom computing. In *Proc. 3rd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pp. 32–38, 1995. 16

[8] D. Andrews, D. Niehaus, and P. Ashenden. Programming models for hybrid CPU/FPGA chips. *IEEE Computer*, 37(1):118–120, Jan. 2004. doi:10.1109/mc.2004.1260732. 21

[9] U. Anliker, J. Beutel, M. Dyer, R. Enzler, P. Lukowicz, L. Thiele, and G. Tröster. A systematic approach to the design of distributed wearable systems. *IEEE Trans. on Computers*, to be published. 51

[10] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pp. 316–322, 1992. doi:10.1145/140901.141896. 8, 16

[11] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3): 11–18, Mar. 1993. doi:10.1109/2.204677. 7

[12] E. Atzori, S. M. Carta, and L. Raffo. 44.6% processing cycles reduction in GSM voice coding by low-power reconfigurable co-processor architecture. *Electronics Letters*, 38(24):1524–1526, Nov. 2002. doi:10.1049/el:20021019. 41

[13] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2): 59–67, Feb. 2002. doi:10.1109/2.982917. 22, 27, 30, 36

[14] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The RAW benchmark suite: Computation structures for general purpose computing. In *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 134–143, 1997. 45

[15] F. Barat and R. Lauwereins. Reconfigurable instruction set processors: A survey. In *Proc. 11th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pp. 168–173, 2000. 14

[16] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP – A self-reconfigurable data processing architecture. *Journal of Supercomputing*, 26(2):167–184, Sept. 2003. doi:10.1023/a:1024499601571. 18, 68

[17] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 8(3): 299–316, June 2000. doi:10.1109/92.845896. 41

[18] L. Benini, G. De Micheli, and E. Macii. Designing low-power circuits: Practical recipes. *IEEE Circuits and Systems Mag.*, 1 (1):6–25, 2001. doi:10.1109/7384.928306. 41

[19] V. Betz and J. Rose. How much logic should go in an FPGA logic block? *IEEE Design & Test of Computers*, 15(1):10–15, Jan.–Mar. 1998. doi:10.1109/54.655177. 18

[20] J.-L. Beuchat, J.-O. Haenni, and E. Sanchez. Hardware reconfigurable neural networks. In *Proc. 5th Reconfigurable Architectures Workshop (RAW)*, vol. 1388 of *Lecture Notes in Computer Science*, pp. 91–98. Springer-Verlag, 1998. 7

[21] R. Bittner and P. Athanas. Wormhole run-time reconfiguration. In *Proc. 5th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 79–85, 1997. doi:10.1145/258305.258315. 20

[22] K. Bondalapati and V. K. Prasanna. Reconfigurable computing systems. *Proceedings of the IEEE*, 90(7):1201–1217, July 2002. doi:10.1109/jproc.2002.801446. 14

[23] B. Bosi, G. Bois, and Y. Savaria. Reconfigurable pipelined 2-D convolvers for fast digital signal processing. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 7(3):299–308, Sept. 1999. doi:10.1109/92.784091. 7, 21

[24] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. 27th Int. Symp. on Computer Architecture (ISCA)*, pp. 83–94, 2000. doi:10.1145/339647.339657. 42

[25] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, Nov./Dec. 2000. doi:10.1109/40.888701. 42

[26] S. Brown and J. Rose. FPGA and CPLD architectures: A tutorial. *IEEE Design & Test of Computers*, 13(2):42–57, 1996. doi:10.1109/54.500200. 4

[27] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report #1342, Computer Sciences Dept., Univ. of Wisconsin–Madison, June 1997. 27, 73

[28] N. G. Busa and C. R. Sala. A run-time word-level reconfigurable coarse-grain functional unit for a VLIW processor. In *Proc. 15th Int. Symp. on System Synthesis (ISSS)*, pp. 44–49, 2002. 22

[29] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, Apr. 2000. doi:10.1109/2.839323. 14, 15, 18, 22

[30] T. J. Callahan and J. Wawrzynek. Instruction-level parallelism for reconfigurable computing. In *Proc. 8th Int. Workshop on Field Programmable Logic and Applications (FPL)*, vol. 1482 of *Lecture Notes in Computer Science*, pp. 248–257. Springer-Verlag, 1998. 22

[31] H. C. Card, G. K. Rosendahl, D. K. McNeill, and R. D. McLeod. Competitive learning algorithms and neurocomputer architecture. *IEEE Trans. on Computers*, 47(8):847–858, Aug. 1998. doi:10.1109/12.707586. 7

[32] J. M. P. Cardoso. On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures. *IEEE Trans. on Computers*, 52(10):1362–1375, Oct. 2003. doi:10.1109/tc.2003.1234532. 7

[33] J. E. Carrillo Esparza and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proc. 9th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 141–150, 2001. doi:10.1145/360276.360328. 14, 18, 22, 29

[34] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). In *Proc. 10th Int. Conf. on Field Programmable Logic and Applications (FPL)*, vol. 1896 of *Lecture Notes in Computer Science*, pp. 605–614. Springer-Verlag, 2000. 7, 68

[35] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data driven signal processing: An approach for energy efficient computing. In *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED)*, pp. 347–52, 1996. 41

[36] Chunho Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. 30th Int. Symp. on Microarchitecture (MICRO-30)*, pp. 330–335, 1997. 44, 45, 46

[37] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002. doi:10.1145/508352.508353. 14, 20, 21

[38] J. Cong and S. Xu. Technology mapping for FPGAs with embedded memory blocks. In *Proc. 6th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 179–188, 1998. doi:10.1145/275107.275138. 19

[39] J. J. Cong and S. Xu. Performance-driven technology mapping for heterogeneous FPGAs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(11):1268–1281, Nov. 2000. doi:10.1109/43.892851. 19

[40] T. M. Conte, K. N. Menezes, S. W. Sathaye, and M. C. Toburen. System-level power consumption modeling and tradeoff analysis techniques for superscalar processor design. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 8(2):129–137, Apr. 2000. doi:10.1109/92.831433. 42

[41] M. Cummings and S. Haruyama. FPGA in the software radio. *IEEE Communications Mag.*, 37(2):108–112, Feb. 1999. doi:10.1109/35.747258. 8

[42] A. Curiger, H. Bonnenberg, R. Zimmermann, N. Felber, H. Kaeslin, and W. Fichtner. VINCI: VLSI implementation of the new secret-key block cipher IDEA. In *Proc. 15th IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 15.5.1–15.5.4, 1993. 8

[43] J. Daemen and V. Rijmen. *The Design of Rijndael, AES – The Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2. 45, 47

[44] A. Dandalis and V. K. Prasanna. Signal processing using reconfigurable system-on-chip platforms. In *Proc. 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 36–42. CSREA Press, 2001. 7

[45] D. Davis, M. Barr, T. Bennett, S. Edwards, J. Harris, I. Miller, and C. Schanck. A Java development and runtime environment for reconfigurable computing. In *Proc. 5th Reconfigurable Architectures Workshop (RAW)*, vol. 1388 of *Lecture Notes in Computer Science*, pp. 43–48. Springer-Verlag, 1998. 21

[46] A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proc. 2nd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pp. 31–39, 1994. 7, 21

[47] A. DeHon. DPGA utilization and application. In *Proc. 4th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 115–121, 1996. doi:10.1145/228370.228387. 7, 18, 21

[48] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, 1996. Available at http://www.cs.caltech.edu/~andre/abstracts/dehon_phd.html. 20

[49] A. DeHon. The density advantage of configurable computing. *IEEE Computer*, 33(4):41–49, Apr. 2000. doi:10.1109/2.839320. 20, 111

[50] C. Dick, F. Harris, and M. Rice. FPGA implementation of carrier synchronization for QAM receivers. *Journal of VLSI Signal Processing*, 36(1):57–71, Jan. 2004. doi:10.1023/b:vlsi.0000008070.30837.e1. 8

[51] C. Dick and F. J. Harris. Configurable logic for digital communications: Some signal processing perspectives. *IEEE Communications Mag.*, 37(8):107–111, Aug. 1999. doi:10.1109/35.783133. 7

[52] T. H. Drayer, W. E. King IV, J. G. Tront, and R. W. Conners. A modular and reprogrammable real-time processing hardware, MORRPH. In *Proc. 3rd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pp. 11–19, 1995. 16

[53] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(4):545–557, Aug. 2001. doi:10.1109/92.931230. 8

[54] J. G. Eldredge and B. L. Hutchings. Run-time reconfiguration: A method for enhancing the functional density of SRAM-based FPGAs. *Journal of VLSI Signal Processing*, 12(1):67–86, Jan. 1996. 7

[55] R. Enzler, M. Platzner, C. Plessl, L. Thiele, and G. Tröster. Reconfigurable processors for handhelds and wearables: Application analysis. In *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications III*, vol. 4525 of *Proceedings of SPIE*, pp. 135–146, 2001. doi: 10.1117/12.434376. 43

[56] R. Enzler, C. Plessl, and M. Platzner. Co-simulation of a hybrid multi-context architecture. In *Proc. 3rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 174–180. CSREA Press, 2003. 25

[57] R. Enzler, C. Plessl, and M. Platzner. Virtualizing hardware with multi-context reconfigurable arrays. In *Proc. 13th Int. Conf. on Field Programmable Logic and Applications (FPL)*, vol. 2778 of *Lecture Notes in Computer Science*, pp. 151–160. Springer-Verlag, 2003. 67

[58] R. Enzler, C. Plessl, and M. Platzner. System-level performance evaluation of reconfigurable processors. *Microprocessors and Microsystems Journal*, to be published. 25

[59] J. Eyre and J. Bier. Independent DSP benchmarks: Methodologies and results. In *Proc. Int. Conf. on Signal Processing Applications and Technology (ICSPAT)*, 1999. 44

[60] J. Faura, M. A. Aguirre, J. N. Moreno, P. van Duong, and J. M. Insenser. FIPSOC: A field programmable system on a chip. In *Proc. 12th Int. Conf. on Design of Circuits and Integrated Systems (DCIS)*, pp. 597–602, 1997. 7, 21

[61] J. Faura, C. Horton, P. van Duong, J. Madrenas, M. A. Aguirre, and J. M. Insenser. A novel mixed signal programmable device with on-chip microprocessor. In *Proc. 19th IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 103–106, 1997. 21

[62] J. Faura, J. N. Moreno, M. A. Aguirre, P. van Duong, and J. M. Insenser. Multicontext dynamic reconfiguration and real-time probing on a novel mixed signal programmable device with on-chip microprocessor. In *Proc. 7th Int. Workshop on Field Programmable Logic and Applications (FPL)*, vol. 1304 of *Lecture Notes in Computer Science*, pp. 1–10. Springer-Verlag, 1997. 7, 21

[63] T. Fujii, K.-i. Furuta, M. Motomura, M. Nomura, M. Mizuno, K.-i. Anjo, K. Wakabayashi, Y. Hirota, Y.-e. Nakazawa, H. Itoh, and M. Yamashina. A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture. In *46th IEEE Int. Solid-State Circuits Conf. (ISSCC), Dig. Tech. Papers*, pp. 364–365, 1999. 21

[64] K. Furuta, T. Fujii, M. Motomura, K. Wakabayashi, and M. Yamashina. Spatial–temporal mapping of real applications on a dynamically reconfigurable logic engine (DRLE) LSI. In *Proc. 22nd IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 151–154, 2000. 21

[65] D. D. Gajski, ed. *Silicon Compilation*. Addison-Wesley, 1988. ISBN 0-201-09915-2. 29

[66] V. George and J. M. Rabaey. *Low-Energy FPGAs: Architecture and Design*. Kluwer Academic Publishers, 2001. ISBN 0-7923-7428-2. 20, 42

[67] B. A. Gieseke, R. L. Allmon, D. W. Bailey, B. J. Benschneider, S. M. Britton, J. D. Clouser, H. R. Fair III, J. A. Farrell, M. K. Gowan, C. L. Houghton, J. B. Keller, T. H. Lee, D. L. Leibholz, S. C. Lowell, M. D. Matson, R. J. Matthew, V. Peng, M. D. Quinn, D. A. Priore, M. J. Smith, and K. E. Wilcox. A 600 MHz superscalar RISC microprocessor with out-of-order execution. In *44th IEEE Int. Solid-State Circuits Conf. (ISSCC), Dig. Tech. Papers*, pp. 176–177,451, 1997. 38

[68] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, 24(1):81–89, Jan. 1991. doi:10.1109/2.67197. 8

[69] M. B. Gokhale, J. M. Stone, and E. Gomersall. Co-synthesis to a hybrid RISC/FPGA architecture. *Journal of VLSI Signal Processing*, 24(2/3):165–180, Mar. 2000. doi:10.1023/a:1008141305507. 22

[70] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, Apr. 2000. doi:10.1109/2.839324. 18, 22

[71] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. 26th Int. Symp. on Computer Architecture (ISCA)*, pp. 28–39, 1999. 7

[72] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, May 2002. ISBN 1-4020-7072-1. 29

[73] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. 4th Workshop on Workload Characterization (WWC)*, pp. 3–14, 2001. 44, 45, 46

[74] L. Gwennap. MIPS R10000 uses decoupled architecture. *Microprocessor Report*, 8(14):18–22, Oct. 1994. 38

[75] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proc. 6th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 65–74, 1998. doi: 10.1145/275107.275121. 20

[76] S. Hauck. The roles of FPGA's in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615–638, Apr. 1998. doi:10.1109/5.663540. 14

[77] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 87–96, 1997. 14, 15, 18, 23

[78] S. Hauck and W. D. Wilson. Runlength compression techniques for FPGA configurations. In *Proc. 7th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 286–287, 1999. 20

[79] S. Hauck, Zhiyuan Li, and E. J. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 138–146, 1998. 20

[80] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 12–21, 1997. 14, 15, 18, 77

[81] S. D. Haynes, A. B. Ferrari, and P. Y. K. Cheun. Flexible reconfigurable multiplier blocks suitable for enhancing the architecture of FPGAs. In *Proc. 21st IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 191–194, 1999. 19

[82] J. He and J. Rose. Advantages of heterogeneous logic block architecture for FPGAs. In *Proc. 15th IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 7.4.1–7.4.5, 1993. 19

[83] F. Heile and A. Leaver. Hybrid product term and LUT based architectures using embedded memory blocks. In *Proc. 7th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 13–16, 1999. doi:10.1145/296399.296415. 19

[84] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990. ISBN 1-55860-069-8. 79

[85] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2002. ISBN 1-55860-596-7. 57

[86] J.-P. Heron and R. F. Woods. Accelerating run-time reconfiguration on FCCMs. In *Proc. 7th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 260–261, 1999. 20

[87] G. Hinton, M. Upton, D. J. Sager, D. Boggs, D. M. Carmean, P. Roussel, T. I. Chappell, T. D. Fletcher, M. S. Milshtein, M. Sprague, S. Samaan, and R. Murray. A 0.18-μm CMOS IA-32 processor with a 4-GHz integer execution unit. *IEEE Journal of Solid-State Circuits*, 36(11):1617–1627, Nov. 2001. doi:10.1109/4.962281. 38

[88] D. T. Hoang. Searching genetic databases on Splash 2. In *Proc. 1st IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pp. 185–191, 1993. 8

[89] Huesung Kim, A. K. Somani, and A. Tyagi. A reconfigurable multifunction computing cache architecture. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(4):509–523, Aug. 2001. doi:10.1109/92.931228. 7

[90] B. L. Hutchings and M. J. Wirthlin. Implementation approaches for reconfigurable logic applications. In *Proc. 5th Int. Workshop on Field Programmable Logic and Applications (FPL)*, vol. 975 of *Lecture Notes in Computer Science*, pp. 419–428. Springer-Verlag, 1995. 3

[91] J. A. Jacob and P. Chow. Memory interfacing and instruction specification for reconfigurable processors. In *Proc. 7th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 145–154, 1999. doi:10.1145/296399.296446. 14, 15, 18

[92] J. Jean, Xuejun-Liang, B. Drozd, K. Tomko, and Yan-Wang. Automatic target recognition with dynamic reconfiguration. *Journal of VLSI Signal Processing*, 25(1):39–53, May 2000. doi:10.1023/a:1008173519198. 8

[93] H. Kaeslin. Personal communication. Microelectronics Design Center, Swiss Federal Institute of Technology (ETH) Zurich, Oct.–Dec. 2003. 38, 39, 40, 102

[94] Kang-Ngee Chia, Hea Joung Kim, S. Lansing, W. H. Mangione-Smith, and J. Villasensor. High-performance automatic target recognition through data-specific VLSI. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 6(3):364–371, Sept. 1998. doi:10.1109/92.711308. 8

[95] B. Kastrup, J. Trum, O. Moreira, J. Hoogerbrugge, and J. van Meerbergen. Compiling applications for ConCISe: An example of automatic HW/SW partitioning and synthesis. In *Proc. 10th Int. Conf. on Field Programmable Logic and Applications (FPL)*, vol. 1896 of *Lecture Notes in Computer Science*, pp. 695–706. Springer-Verlag, 2000. 22

[96] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *IEEE Computer*, 35(9):39–47, Sept. 2002. doi:10.1109/mc.2002.1033026. 121

[97] A. Kaviani and S. Brown. Hybrid FPGA architecture. In *Proc. 4th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 3–9, 1996. doi:10.1145/228370.228371. 19

[98] S. Knapp and D. Tavana. Field configurable system-on-chip device architecture. In *Proc. 22nd IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 155–158, 2000. 9

[99] J. L. Kouloheris and A. El Gamal. FPGA performance versus cell granularity. In *Proc. 13th IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 6.2.1–6.2.4, 1991. 18

[100] S. Kumar, L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai, and H. Spaanenburg. A benchmark suite for evaluating configurable computing systems – status, reflections, and future directions. In *Proc. 8th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 126–134, 2000. doi:10.1145/329166.329193. 44

[101] E. Kusse and J. Rabaey. Low-energy embedded FPGA structures. In *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED)*, pp. 155–160, 1998. doi:10.1145/280756.280873. 20, 42

[102] A. La Rosa, L. Lavagno, and C. Passerone. A software development tool chain for a reconfigurable processor. In *Proc. 4th Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 93–98, 2001. 23

[103] P. Landman. High-level power estimation. In *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED)*, pp. 29–35, 1996. 41

[104] D. Lau, A. Scheider, M. D. Ercegovac, and J. Villasenor. A FPGA-based library for on-line processing. *Journal of VLSI Signal Processing*, 28(1/2):129–143, May/June 2001. doi:10.1023/a:1008119407508. 21

[105] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, and V. Castro Alves. Design and implementation of the MorphoSys reconfigurable computing processor. *Journal of VLSI Signal Processing*, 24(2/3):147–164, Mar. 2000. doi:10.1023/a:1008189221436. 14, 15, 18

[106] D. I. Lehn, K. Puttegowda, J. H. Park, P. M. Athanas, and M. T. Jones. Evaluation of rapid context switching on a CSRC device. In *Proc. 2nd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 209–215. CSREA Press, 2002. 21

[107] X.-P. Ling and H. Amano. WASMII: A data driven computer on a virtual hardware. In *Proc. 1st IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pp. 33–42, 1993. 21

[108] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri. A VLIW processor with reconfigurable instruction set for embedded applications. *IEEE Journal of Solid-State Circuits*, 38(11):1876–1886, Nov. 2003. doi:10.1109/jssc.2003.818292. 23

[109] R. Maestre, F. J. Kurdahi, M. Fernández, R. Hermida, N. Bagherzadeh, and H. Singh. A formal approach to context scheduling for multicontext reconfigurable architectures. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(1):173–185, Feb. 2001. doi:10.1109/92.920831. 21

[110] R. Maestre, F. J. Kurdahi, M. Fernández, R. Hermida, N. Bagherzadeh, and H. Singh. A framework for reconfigurable computing: Task scheduling and context management. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(6):858–873, Dec. 2001. doi:10.1109/92.974899. 21

[111] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, and H. A. E. Spaanenburg. Seeking solutions in configurable computing. *IEEE Computer*, 30(12):38–43, Dec. 1997. doi:10.1109/2.642810. 21

[112] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proc. 7th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 135–143, 1999. doi:10.1145/296399.296444. 18, 19

[113] O. Mencer. PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs. In *Proc. 10th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 67–76, 2002. 21

[114] O. Mencer, M. Morf, and M. J. Flynn. Hardware software tri-design of encryption for mobile communication units. In *Proc. 23rd IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 5, pp. 3045–3048, 1998. 8

[115] O. Mencer, M. Platzner, M. Morf, and M. J. Flynn. Object-oriented domain specific compilers for programming FPGAs. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(1):205–210, Feb. 2001. doi:10.1109/92.920835. 22

[116] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. *IEICE Trans. on Information and Systems*, E82-D(2):389–397, Feb. 1999. 7, 14, 15, 18

[117] Model Technology. *ModelSim Foreign Language Interface*, ver. 5.5f, Aug. 2001. Available at http://www.model.com/. 31

[118] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stehpany, and S. C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, Nov. 1996. 38

[119] M. Motomura. A dynamically reconfigurable processor architecture. Microprocessor Forum, Oct. 2002. 7, 21

[120] M. Motomura, Y. Aimoto, A. Shibayama, Y. Yabe, and M. Yamashina. An embedded DRAM-FPGA chip with instantaneous logic reconfiguration. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 264–266, 1998. 7

[121] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8):63–69, Aug. 2003. doi:10.1109/mc.2003.1220583. 22

[122] Nam Sung Kim, T. Austin, T. Mudge, and D. Grunwald. Challenges for architectural level power modeling. In R. Graybill and R. Melhem, ed., *Power Aware Computing*, Series in Computer Science, pp. 317–337. Kluwer Academic/Plenum Publishers, June 2002. ISBN 0-306-46786-0. 42

[123] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the development of the Advanced Encryption Standard (AES). Technical report, National Institute of Standards and Technology (NIST), Oct. 2000. 46, 47

[124] Open SystemC Initiative (OSCI). *SystemC 2.0 User's Guide*, 2002. Available at http://www.systemc.org/. 29

[125] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, int'l edition, 1998. ISBN 0-13-216771-9. 89, 91

[126] P. R. Panda. SystemC: A modeling platform supporting multiple design abstractions. In *Proc. 14th Int. Symp. on Systems Synthesis (ISSS)*, pp. 75–80, 2001. 29

[127] J. Panfil. The TinyRISC CPU – an area efficient CPU core optimized for embedded applications. The European Microprocessor and Microcontroller Conf., Nov. 1996. 48

[128] M. Platzner. Reconfigurable accelerators for combinatorial problems. *IEEE Computer*, 33(4):58–60, Apr. 2000. doi:10.1109/2.839322. 7, 8

[129] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, and L. Thiele. Reconfigurable hardware in wearable computing nodes. In *Proc. 6th Int. Symp. on Wearable Computers (ISWC)*, pp. 215–222, 2002. 51

[130] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, L. Thiele, and G. Tröster. The case for reconfigurable hardware in wearable computing. *Personal and Ubiquitous Computing*, 7(5):299–308, Oct. 2003. doi:10.1007/s00779-003-0243-x. 51

[131] C. Plessl and M. Platzner. Instance-specific accelerators for minimum covering. *Journal of Supercomputing*, 26(2):109–129, Sept. 2003. doi:10.1023/a:1024443416592. 8

[132] K. Puttegowda, D. I. Lehn, P. Athanas, and M. Jones. Context switching in a run-time reconfigurable system. *Journal of Supercomputing*, 26(3):239–257, Nov. 2003. doi:10.1023/a:1025694914489. 21

[133] J. M. Rabaey. Reconfigurable processing: The solution to low power programmable DSP. In *Proc. 22nd IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 275–278, 1997. 41

[134] B. Radunović and V. Milutinović. A survey of reconfigurable computing architectures. In *Proc. 8th Int. Workshop on Field Programmable Logic and Applications (FPL)*, vol. 1482 of *Lecture Notes in Computer Science*, pp. 376–385. Springer-Verlag, 1998. 14

[135] N. K. Ratha and A. K. Jain. Computer vision algorithms on reconfigurable logic arrays. *IEEE Trans. on Parallel and Distributed Systems*, 10(1):29–43, Jan. 1999. doi:10.1109/71.744833. 7, 8

[136] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. 27th Int. Symp. on Microarchitecture (MICRO-27)*, pp. 172–180, 1994. doi:10.1145/192724.192749. 14, 15, 18

[137] J. Reilly. SPEC describes SPEC95 products and benchmarks. *SPEC Newsletter*, Sept. 1995. URL http://www.spec.org/osg/news/articles/news9509/cpu95descr.html. 52

[138] M. Rencher and B. L. Hutchings. Automated target recognition on SPLASH 2. In *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 192–200, 1997. 8

[139] D. Rizzo and O. Colavin. A video compression case study on a reconfigurable VLIW architecture. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pp. 540–546, 2002. 22

[140] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale. The NAPA adaptive processing architecture. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 28–37, 1998. 14, 15

[141] B. Salefski and L. Caglar. Re-configurable computing in wireless. In *Proc. 38th Design Automation Conf. (DAC)*, pp. 178–183, 2001. doi:10.1145/378239.378459. 7, 19, 21

[142] E. Sanchez, M. Sipper, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, and A. Perez-Uribe. Static and dynamic configurable systems. *IEEE Trans. on Computers*, 48(6):556–564, June 1999. doi:10.1109/12.773792. 3

[143] S. Santhanam, A. J. Baum, D. Bertucci, M. Braganza, K. Broch, T. Broch, J. Burnette, E. Chang, Kwong-Tak Chui, D. Dobberpuhl, P. Donahue, J. Grodstein, Insung Kim, D. Murray, M. Pearce, A. Silveria, D. Souydalay, A. Spink, R. Stepanian, A. Varadharajan, V. R. van Kaenel, and R. Wen. A low-cost, 300-MHz, RISC CPU with attached media processor. *IEEE Journal of Solid-State Circuits*, 33(11):1829–1839, Nov. 1998. doi:10.1109/4.726584. 38

[144] S. M. Scalera and J. R. Vázquez. The design and implementation of a context switching FPGA. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 78–85, 1998. 21

[145] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A quick safari through the reconfiguration jungle. In *Proc. 38th Design Automation Conf. (DAC)*, pp. 172–177, 2001. doi:10.1145/378239.378404. 14, 22

[146] H. Schmit. Incremental reconfiguration for pipelined applications. In *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 47–55, 1997. 20

[147] H. H. Schmit, S. Cadambi, M. Moe, and S. C. Goldstein. Pipeline reconfigurable FPGAs. *Journal of VLSI Signal Processing*, 24 (2/3):129–146, Mar. 2000. doi:10.1023/a:1008137204598. 7, 20, 68

[148] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2nd edition, 1996. ISBN 0-471-12845-7. 47

[149] S. Segars. The ARM9 family – high performance microprocessors for embedded applications. In *Proc. 16th Int. Conf. on Computer Design (ICCD)*, pp. 230–235, 1998. 38

[150] C. L. Seitz. Concurrent VLSI architectures. *IEEE Trans. on Computers*, C-33(12):1247–1265, Dec. 1984. 111

[151] S. Sezer, J. Heron, R. Woods, R. Turner, and A. Marshall. Fast partial reconfiguration for FCCMs. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 318–319, 1998. 20

[152] R. Shoup. Heterogeneous architectures and adaptive computing. Presentation at Dagstuhl Seminar on Dynamically Reconfigurable Architectures, July 2003. Available at http://www.dagstuhl.de/03301/Proceedings/. 5

[153] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. on Computers*, 49(5):465–481, May 2000. doi:10.1109/12.859540. 14, 15, 18, 21

[154] A. Sinha and A. P. Chandrakasan. JouleTrack – A web based tool for software energy profiling. In *Proc. 38th Design Automation Conf. (DAC)*, pp. 220–225, 2001. doi:10.1145/378239.378467. 41

[155] M. Sipper and E. Sanchez. Configurable chips meld software and hardware. *IEEE Computer*, 33(1):120–121, Jan. 2000. doi:10.1109/2.963133. 1

[156] M. E. Smid and D. K. Branstad. Data Encryption Standard: Past and future. *Proceedings of the IEEE*, 76(5):550–559, May 1988. doi:10.1109/5.4441. 47

[157] S. Srikanteswara, R. C. Palat, J. H. Reed, and P. Athanas. An overview of configurable computing machines for software radio handsets. *IEEE Communications Mag.*, 41(7):134–141, July 2003. doi:10.1109/mcom.2003.1215650. 8

[158] Standard Performance Evaluation Corporation. SPEC CINT95 benchmarks. Online, 1995–2003. URL http://www.spec.org/cpu95/CINT95/. 52

[159] T. J. Stanley and T. Mudge. Systematic objective-driven computer architecture optimization. In *Proc. 16th Conf. on Advanced Research in VLSI (ARVLSI)*, pp. 286–300, 1995. 121

[160] T. Stansfield. Using multiplexers for control and data in D-Fabrix. In *Proc. 13th Int. Conf. on Field Programmable Logic*

and Applications (FPL), vol. 2778 of *Lecture Notes in Computer Science*, pp. 416–425. Springer-Verlag, 2003. 5, 19

[161] R. Stevens. *UNIX Network Programming, Volume 2: Interprocess Communications*. Prentice Hall, 2nd edition, 1999. ISBN 0-13-081081-9. 31

[162] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid. Using on-chip configurable logic to reduce embedded system software energy. In *Proc. 10th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 143–151, 2002. 9

[163] K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, and T. Kuroda. A 300 MIPS/W RISC core processor with variable supply-voltage. In *Proc. 19th IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 587–590, 1997. 38

[164] Synopsys. Design Compiler. Technology Backgrounder, May 2002. Available at http://www.synopsys.com/. 103

[165] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon. A first generation DPGA implementation. In *Canadian Workshop on Field-Programmable Devices (FPD)*, pp. 138–143, 1995. 18, 21

[166] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, Mar./Apr. 2002. doi: 10.1109/mm.2002.997877. 19

[167] J. Teich. *Digitale Hardware/Software-Systeme. Synthese und Optimierung*. Springer-Verlag, 1997. ISBN 3-540-62433-3. 29

[168] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28(1/2):7–27, May 2001. doi:10.1023/a:1008155020711. 1, 14

[169] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2/3):223–238, Aug. 1996. 41

[170] C. Traber. Syntactic processing and prosody control in the SVOX TTS system for German. In *Proc. 3rd European Conf. on Speech Communication and Technology (Eurospeech)*, pp. 2099–2102, 1993. 45, 46, 47

[171] N. Tredennick and B. Shimamoto. Go reconfigure. *IEEE Spectrum*, 40(12):36–40, Dec. 2003. doi:10.1109/mspec.2003.1249977. 1

[172] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 22–28, 1997. 7, 21, 68

[173] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. HSRA: High-speed, hierarchical synchronous reconfigurable array. In *Proc. 7th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 125–134, 1999. doi:10.1145/296399.296442. 20

[174] Univ. of Chicago Press, ed. *The Chicago Manual of Style*. Univ. of Chicago Press, 15th edition, 2003. ISBN 0-226-10403-6.

[175] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm, and J. Hammes. Automatic compilation to a coarse-grained reconfigurable system-on-chip. *ACM Trans. on Embedded Computing Systems*, 2(4):560–589, Nov. 2003. 22

[176] Vi Cuong Chan and D. M. Lewis. Area-speed tradeoffs for hierarchical field-programmable gate arrays. In *Proc. 4th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 51–57, 1996. doi:10.1145/228370.228378. 20

[177] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. 27th Int. Symp. on Computer Architecture (ISCA)*, pp. 95–106, 2000. doi:10.1145/339647.339659. 42

[178] J. Villasenor and B. Hutchings. The flexibility of configurable computing. *IEEE Signal Processing Mag.*, 15(5):67–84, Sept. 1998. doi:10.1109/79.708541. 8

[179] Virtual Silicon Technology. *UMC L250 Standard Cell Library Databook*, 2002. Available at http://www.virtual-silicon.com/. 103

[180] T. Šimunić, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proc. 36th Design Automation Conf. (DAC)*, pp. 867–872, 1999. doi: 10.1145/309847.310090. 42

[181] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 4(1):56–69, Mar. 1996. doi:10.1109/92.486081. 8, 16

[182] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, Sept. 1997. doi:10.1109/2.612254. 19, 22

[183] M. Wan, Hui Zhang, M. Benes, and J. Rabaey. A low-power reconfigurable data-flow driven DSP system. In *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 191–200, 1999. 7

[184] M. Wan, Hui Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, and J. Rabaey. Design methodology of a low-energy reconfigurable single-chip DSP system. *Journal of VLSI Signal Processing*, 28(1/2):47–61, May 2001. doi:10.1023/a:1008159121620. 42

[185] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II compiler and architecture. In *Proc. 1st IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pp. 9–16, 1993. 16

[186] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):234–248, Feb. 2001. doi:10.1109/43.908452. 7

[187] A. R. Weiss. The standardization of embedded benchmarking: Pitfalls and opportunities. In *Proc. 17th Int. Conf. on Computer Design (ICCD)*, pp. 492–498, 1999. 44

[188] K. Weiß, C. Oetker, I. Katchan, T. Steckstor, and W. Rosenstiel. Power estimation approach for SRAM-based FPGAs. In *Proc. 8th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 195–202, 2000. doi:10.1145/329166.329207. 42

[189] S. Winegarden. Bus architecture of a system on a chip with user-configurable system logic. *IEEE Journal of Solid-State Circuits*, 35(3):425–433, Mar. 2000. doi:10.1109/4.826825. 9

[190] N. Wirth. Hardware compilation: Translating programs into circuits. *IEEE Computer*, 31(6):25–31, June 1998. doi:10.1109/2.683004. 21

[191] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In *Proc. 3rd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pp. 99–107, 1995. 7

[192] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time constant propagation. In *Proc. 5th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 86–92, 1997. doi:10.1145/258305.258316. 7

[193] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time circuit reconfiguration. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 6(2):247–256, June 1998. doi:10.1109/92.678880. 111

[194] R. Witek and J. Montanaro. StrongARM: A high-performance ARM processor. In *Proc. 41st IEEE Int. Computer Conf. (COMPCON)*, pp. 188–191, 1996. 48

[195] R. D. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In *Proc. 4th IEEE Symp. on FPGAs for Custom Computing Machines (FCCM)*, pp. 126–135, 1996. 14, 15, 18

[196] T. Wolf and M. Franklin. CommBench – a telecommunications benchmark for network processors. In *Proc. IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pp. 154–162, 2000. 44, 45, 46

[197] Xilinx. *System ACE MPM Solution (DS087)*, ver. 2.2, June 2003. Available at http://www.xilinx.com/. 5

[198] Xilinx. *Xilinx Virtex-II Pro Platform FPGAs: Complete Data Sheet (DS083)*, Oct. 2003. Available at http://www.xilinx.com/. 18, 19, 20, 85

[199] Xilinx. Power tools. Online, 2004. URL http://www.xilinx.com/ise/power_tools/. 42

[200] Xinan Tang, M. Aalsma, and R. Jou. A compiler directed approach to hiding configuration latency in Chameleon processors. In *Proc. 10th Int. Conf. on Field Programmable Logic and Applications (FPL)*, vol. 1896 of *Lecture Notes in Computer Science*, pp. 29–38. Springer-Verlag, 2000. 7, 21

[201] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Proc. 37th Design Automation Conf. (DAC)*, pp. 340–345, 2000. doi:10.1145/337292.337436. 42

[202] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. 27th Int. Symp. on Computer Architecture (ISCA)*, pp. 225–235, 2000. doi:10.1145/339647.339687. 14, 15, 18, 23, 29

[203] Z. A. Ye, N. Shenoy, and P. Banerjee. A C compiler for a processor with a reconfigurable functional unit. In *Proc. 8th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 95–100, 2000. doi:10.1145/329166.329187. 22, 23

[204] Yen-Tai Lai and Ping-Tsung Wang. Hierarchical interconnection structures for field programmable gate arrays. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 5(2):186–196, June 1997. doi:10.1109/92.585219. 20

[205] Zhiyuan Li, K. Compton, and S. Hauck. Configuration caching management techniques for reconfigurable computing. In *Proc. 8th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 22–36, 2000. 20

[206] Zhiyuan Li and S. Hauck. Don't care discovery for FPGA configuration compression. In *Proc. 7th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pp. 91–98, 1999. doi:10.1145/296399.296435. 20

[207] ZIPPY: A novel dynamically reconfigurable embedded processor. TH Research Project, Swiss Federal Institute of Technology (ETH) Zurich, 2000–2003. URL http://www.zippy.ethz.ch/. 9

Curriculum Vitae

Personal Information

Rolf Enzler
Born 30 September 1971, Thalwil ZH, Switzerland
Citizen of Walchwil ZG, Switzerland

Education

1998–2004	PhD studies in information technology and electrical engineering (Dr. sc. techn.) at ETH Zurich, Switzerland
1991–1997	MSc studies in information technology and electrical engineering (Dipl. El.-Ing. ETH) at ETH Zurich, Switzerland
1984–1991	Gymnasium (Matura, Typus C) at Kantonsschule Zug, Switzerland
1978–1984	Primary school in Horgen and Walchwil, Switzerland

Professional Experience

1997–2004	Research and teaching assistant at Electronics Laboratory, ETH Zurich, Switzerland
1997	Teaching substitution in mathematics and computer science at Kantonsschule Zug, Switzerland
1994	Internship with ESEC SA, Cham, Switzerland: Development of ESEC Autoline® monitoring software
1992	Internship with Landis & Gyr AG, Zug, Switzerland: Course on basic engineering skills