

Diss. ETH No. 16742

# **Hardware Virtualization on a Coarse-Grained Reconfigurable Processor**

A dissertation submitted to the  
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZÜRICH

for the degree of  
Doctor of Sciences

presented by

**CHRISTIAN PLESSL**

Dipl. El.-Ing., ETH Zürich, Switzerland

born 1975-02-27

citizen of  
Germany

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner

Prof. Peter Cheung, co-examiner

Prof. Dr. Marco Platzner, co-examiner

2006

Examination date: July 10, 2006





Institut für Technische Informatik und Kommunikationsnetze  
Computer Engineering and Networks Laboratory

---

TIK-SCHRIFTENREIHE NR. 84

Christian Plessl

# Hardware Virtualization on a Coarse-Grained Reconfigurable Processor



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

A dissertation submitted to the  
Swiss Federal Institute of Technology Zurich  
for the degree of Doctor of Sciences

Diss. ETH No. 16742

Prof. Dr. Lothar Thiele, examiner  
Prof. Peter Cheung, co-examiner  
Prof. Dr. Marco Platzner, co-examiner

Examination date: July 10, 2006

## Acknowledgements

I would like to thank:

- my advisor Prof. Lothar Thiele for supporting my research and for providing me with an excellent research environment. I have benefited a lot from his experience and his critical thinking in our discussions.
- Prof. Marco Platzner for his invaluable advice during the Zippy project. I appreciate his support and our continued, fruitful collaboration. I would also like to thank him for serving as co-examiner for my dissertation.
- Prof. Peter Cheung for kindly serving as co-examiner for my dissertation.
- RolfENZler for the close collaboration in the Zippy research project. The results on this work have been the basis for this dissertation.
- my colleagues from the Computer Engineering group for countless inspiring and controversial discussions.
- my friends and my family for their continued support and encouragement.



# Contents

<b>Contents</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Research Topics in Reconfigurable Architectures for Embedded Systems . . . . .	2
1.2 Contributions . . . . .	5
1.3 Application Specification and Execution on the Zippy Architecture . . . . .	7
1.4 Preliminary Work at ETH . . . . .	10
1.5 Thesis outline . . . . .	11
<b>2 Dynamically Reconfigurable Processor Architecture</b>	<b>13</b>
2.1 Design Objectives . . . . .	14
2.2 System Architecture . . . . .	15
2.2.1 Structure . . . . .	15
2.2.2 Embedded CPU core . . . . .	16
2.2.3 Reconfigurable Processing Unit . . . . .	17
2.3 Reconfigurable Processing Unit Architecture . . . . .	18
2.3.1 Reconfigurable Array . . . . .	19
2.3.2 FIFOs . . . . .	24
2.3.3 Register Interface . . . . .	26
2.3.4 Context Sequencer and Synchronization . . . . .	26
2.3.5 Multi-Context Architecture . . . . .	30
2.3.6 Configuration Architecture . . . . .	31
2.3.7 Parameters of the Reconfigurable Processing Unit . . . . .	32
2.4 Summary . . . . .	32
<b>3 Tool-Flow</b>	<b>35</b>
3.1 Tool-Flows for Coarse-Grained Reconfigurable Processors	36
3.2 The Zippy Tool-Flow . . . . .	39
3.3 Hardware Tool-Flow . . . . .	41
3.3.1 Hardware specification . . . . .	41
3.3.2 Architecture modelling . . . . .	43

---

3.3.3	Placement . . . . .	46
3.3.4	Routing . . . . .	49
3.3.5	Configuration Generation . . . . .	53
3.4	Software Tool-Flow . . . . .	55
3.4.1	Hardware- Software Interface . . . . .	55
3.4.2	CPU Simulator and Compilation Tool-Chain . . . . .	56
3.4.3	Extensions to the CPU Simulator . . . . .	56
3.4.4	Compilation Tool-Chain . . . . .	57
3.5	Summary . . . . .	58
<b>4</b>	<b>Performance evaluation</b>	<b>61</b>
4.1	Performance Evaluation for Reconfigurable Processors . . . . .	62
4.1.1	Motivation . . . . .	62
4.1.2	Challenges . . . . .	62
4.1.3	Approaches . . . . .	63
4.2	System-Level Cycle-Accurate Co-Simulation for Zippy . . . . .	64
4.2.1	Architectural assumptions . . . . .	65
4.2.2	CPU Simulation Model . . . . .	65
4.2.3	RPU Simulation Model . . . . .	67
4.2.4	Cosimulation framework . . . . .	67
4.3	Summary . . . . .	71
<b>5</b>	<b>Hardware Virtualization</b>	<b>73</b>
5.1	Introduction to Hardware Virtualization . . . . .	74
5.1.1	Hardware Virtualization Approaches . . . . .	74
5.1.2	Temporal Partitioning . . . . .	76
5.1.3	Virtualized Execution . . . . .	81
5.1.4	Virtual Machine . . . . .	84
5.1.5	Summary . . . . .	85
5.2	Hardware Virtualization on the Zippy Architecture . . . . .	85
5.2.1	Application Specification Model . . . . .	86
5.2.2	Virtualized Execution . . . . .	86
5.2.3	Temporal Partitioning . . . . .	87
5.3	A Novel Method for Optimal Temporal Partitioning . . . . .	90
5.3.1	Outline of the Method . . . . .	90
5.3.2	Models . . . . .	92
5.3.3	Basic Problem Formulation . . . . .	95
5.3.4	Resource Constraints . . . . .	98
5.3.5	Solving the Temporal Partitioning MILP . . . . .	105
5.3.6	Extension to Functional Pipelining . . . . .	105
5.3.7	Example . . . . .	105
5.3.8	Related Work and Discussion . . . . .	107
5.4	Summary . . . . .	110



---

<b>6</b>	<b>Experimental Results</b>	<b>111</b>
6.1	Virtualized Execution of a Digital Filter . . . . .	111
6.1.1	FIR Filter Partitioning and Mapping . . . . .	112
6.1.2	Experimental Setup . . . . .	114
6.1.3	Results and Discussion . . . . .	115
6.2	Temporal Partitioning of an ADPCM Decoder . . . . .	118
6.2.1	Application . . . . .	119
6.2.2	Experiments . . . . .	119
6.2.3	Experimental Setup and Results . . . . .	122
6.3	Summary . . . . .	124
<b>7</b>	<b>Conclusions</b>	<b>125</b>
7.1	Contributions . . . . .	125
7.2	Conclusions . . . . .	127
7.3	Future Directions . . . . .	128
<b>A</b>	<b>Acronyms</b>	<b>131</b>
	<b>Bibliography</b>	<b>133</b>

## Abstract

In this thesis, we propose to use a reconfigurable processor as main computation element in embedded systems for applications from the multi-media and communications domain. A reconfigurable processor integrates an embedded CPU core with a Reconfigurable Processing Unit (RPU). Many of our target applications require real-time signal-processing of data streams and expose a high computational demand. The key challenges in designing embedded systems for these applications is to find an implementation that satisfies the performance goals and is adaptable to new applications, while the system cost is minimized. Implementations that solely use an embedded CPU are likely to miss the performance goals. ASIC-based coprocessors can be used for some high-volume products with fixed functions, but fall short for systems with varying applications.

We argue that a reconfigurable processor with a coarse-grained, dynamically reconfigurable array of modest size provides an attractive implementation platform for our application domain. The computational intensive application kernels are executed on the RPU, while the remaining parts of the application are executed on the CPU. Reconfigurable hardware allows for implementing application specific coprocessors with a high performance, while the function of the coprocessor can still be adapted due to the programmability. So far, reconfigurable technology is used in embedded systems primarily with static configurations, e. g., for implementing glue-logic, replacing ASICs, and for implementing fixed-function coprocessors. Changing the configuration at runtime enables a number of interesting application modes, e. g., on-demand loading of coprocessors and time-multiplexed execution of coprocessors, which is commonly denoted as hardware virtualization. While the use of static configurations is well understood and supported by design-tools, the role of dynamic reconfiguration is not well investigated yet. Current application specification methods and design-tools do not provide an end-to-end tool-flow that considers dynamic reconfiguration. A key idea of our approach is to reduce system cost by keeping the size of the reconfigurable array small and to use hardware virtualization techniques to compensate for the limited hardware resources.

The main contribution of this thesis is the codesign of a reconfigurable processor architecture named ZIPPY, the corresponding hardware and software implementation tools, and an application specification model which explicitly considers hardware virtualization. The ZIPPY architecture is widely parametrized and allows for specifying a whole family of processor architectures. The implementation tools are also parametrized and can target any architectural variant. We evaluate the performance of

the architecture with a system-level, cycle-accurate cosimulation framework. This framework enables us to perform design-space exploration for a variety of reconfigurable processor architectures. With two case studies, we demonstrate, that hardware virtualization on the Zippy architecture is feasible and enables us to trade-off performance for area in embedded systems. Finally, we present a novel method for optimal temporal partitioning of sequential circuits, which is an important form of hardware virtualization. The method based on Slowdown and Retiming allows us to decompose any sequential circuit into a number of smaller, communicating subcircuits that can be executed on a dynamically reconfigurable architecture.

## Zusammenfassung

In dieser Arbeit schlagen wir vor, einen Rekonfigurierbaren Prozessor als Hauptrecheneinheit in eingebetteten Systemen für Anwendungen aus den Bereichen Multimedia und Kommunikation zu verwenden. Ein Rekonfigurierbarer Prozessor integriert einen eingebetteten CPU Core mit einer Rekonfigurierbaren Recheneinheit. Viele unserer Zielanwendungen verlangen nach Signalverarbeitung von Datenströmen in Echtzeit und haben hohe Rechenanforderungen. Die grössten Herausforderungen beim Entwurf eines Eingebetteten Systems für diese Anwendungen ist es, eine Implementierung zu finden, welche den Leistungsanforderungen genügt und auf neue Anwendungen angepasst werden kann, während gleichzeitig die Systemkosten minimiert werden. Implementierungen, welche lediglich eine eingebettete CPU verwenden, können die Leistungsanforderungen in den meisten Fällen nicht erfüllen. Für gewisse Anwendungen mit fester Funktion und hohen Stückzahlen können Coprozessoren basierend auf ASICs verwendet werden. Allerdings sind diese Systeme ungeeignet, wenn die Anwendungen für das System variieren.

Wir legen dar, dass ein Rekonfigurierbarer Prozessor mit einem relativ kleinen, grobgranularen, dynamisch rekonfigurierbaren Array eine attraktive Implementierungsplattform für unsere Anwendungsbereiche ist. Die rechenintensiven Teile der Anwendung, die sogenannten Kernels, werden auf der Rekonfigurierbaren Recheneinheit ausgeführt, während die verbleibenden Teile der Anwendung auf der CPU ausgeführt werden. Die Verwendung von Rekonfigurierbarer Hardware erlaubt die Implementierung von anwendungsspezifischen Coprozessoren mit hoher Rechenleistung, während es die Programmierbarkeit der Hardware erlaubt, die Funktion der Coprozessoren anzupassen. Bisher wurde Reconfigurierbare Hardware in Eingebetteten Systemen vor allem mit statischer Konfiguration eingesetzt, z. B. für die Implementierung von Glue-Logic, den Ersatz von ASICs und für die Implementierung von Coprozessoren mit fixen Funktionen. Das Ändern der Konfiguration zur Laufzeit ermöglicht eine Vielzahl von interessanten Anwendungsmodi, z. B. das Laden von Coprozessoren nach Bedarf und die Verwendung von Zeit-Multiplexing bei der Ausführung von Coprozessoren, was üblicherweise als Hardware Virtualisierung bezeichnet wird. Während die Verwendung von statischen Konfiguration gut etabliert ist und von computergestützten Entwurfswerkzeugen unterstützt wird, wurde die Rolle der dynamischen Rekonfiguration noch nicht ausreichend untersucht. Gegenwärtige Methoden zur Spezifizierung von Anwendungen und die entsprechenden Entwurfswerkzeuge bieten noch keinen durchgängigen Entwurfsprozess an der dynamische Rekonfiguration in Betracht zieht.

Eine Schlüsselidee unseres Ansatzes ist es, die Systemkosten zu reduzieren, indem wir die Grösse der Rekonfigurierbaren Recheneinheit bewusst klein halten und die limitierten Hardware-Ressourcen mittels Hardware Virtualisierung kompensieren.

Der Hauptbeitrag dieser Arbeit ist der gemeinsame Entwurf (Code-sign) einer rekonfigurierbaren Prozessor Architektur (ZIPPY), den entsprechenden Hardware und Software Implementierungswerkzeugen und eines Spezifikationsmodells, welches explizit Hardware Virtualisierung einschliesst. Die ZIPPY Architektur ist weitgehend parametrisiert und erlaubt somit eine ganze Familie von Prozessorarchitekturen zu spezifizieren. Die Implementierungswerkzeuge sind ebenfalls parametrisiert und unterstützen jede Architekturvariante. Die Performance der Architektur wird mit einem Cosimulations Framework ermittelt, das eine systemweite, zyklengenaue Performance-Analyse erlaubt. Dieses Framework ermöglicht uns auch eine Entwurfsraumexploration für verschiedene rekonfigurierbare Prozessorarchitekturen durchzuführen. Mit zwei Fallstudien zeigen wir, dass Hardware Virtualisierung mit der Zippy Architektur möglich ist. Wir zeigen, dass durch Hardware Virtualisierung neue Flächen-Performance Trade-Offs in Eingebetteten Systemen realisieren können. Schliesslich präsentieren wir eine neue Methode zur optimalen Temporalen Partitionierung von sequentiellen Schaltungen. Temporale Partitionierung ist eine wichtige Technik zur Hardware Virtualisierung. Unsere Methode beruht auf Slowdown und Retiming und erlaubt es, jede sequentielle Schaltung in eine Menge von kleineren, kommunizierenden Schaltungen zu zerlegen, welche auf einer dynamisch rekonfigurierbaren Architektur ausgeführt werden können.



# 1

## Introduction

Many embedded systems execute a set of data processing tasks which—apart from the computational requirements—frequently also demand for a guaranteed timing behavior. In recent embedded systems, e. g., Personal Digital Assistants (PDAs), digital music and video players, or set-top-boxes for digital TV, the processing of tasks related to multi-media applications becomes increasingly important. Typically, these tasks expose a high computational demand which is caused by complex signal processing algorithms, e. g., for audio and video decompression, cryptography, or communication protocols.

Given the stringent cost constraints for many embedded systems, finding a cost-effective implementation for these computationally intensive applications, such that the performance requirements are met, is a challenging task. Implementations that rely solely on an embedded CPU are likely to miss the performance goal. For high-volume and fixed-function systems, e. g., smart cell-phones, certain algorithms can be implemented as Application-Specific Integrated Circuits (ASICs) to achieve an optimal trade-off between performance, cost and power-consumption. But for embedded systems with varying applications, low to medium production volumes, or when field upgrades are required, ASIC solution fall short due to their fixed function nature and the high initial cost for an ASIC production.

The introduction of reconfigurable hardware, e. g., Field-Programmable Gate-Arrays (FPGAs) has changed the design space for embedded systems. Reconfigurable devices can be use to implement highly customized, application-specific coprocessors that are able to deliver ASIC-like performance, while at the same time, the functionality can be changed

rapidly. Reconfigurable hardware is also economically attractive since reconfigurable devices are general-purpose devices that are produced in large quantities and can therefore benefit from the economy of scale.

Reconfigurable hardware plays an important role in general purpose and embedded systems and has been applied for many purposes, e. g., logic emulation, replacement of dedicated ASICs and the implementation of runtime-reconfigurable, application specific coprocessors. The latter is the most promising application of reconfigurable technology in embedded systems, since it allows for cost-effective, application specific processing.

While the use of reconfigurable technology with a static configuration is well understood and supported by vendor design-tools, the role of dynamic reconfiguration is not well investigated yet. Current application specification methods and design-tools do not provide an end-to-end tool-flow that considers dynamic reconfiguration.

This thesis advocates the use of a *dynamically reconfigurable processor* as the main computation element in an embedded system. We argue that for embedded systems, the addition of a coarse-grained dynamically reconfigurable array of modest size is beneficial. Fast dynamic reconfiguration enables *hardware virtualization* and can compensate for the limited hardware resources of the reconfigurable processor. In addition to a new hardware architecture, we also introduce an application specification model, and the associated hardware and software tool-flow.

Section 1.1 briefly outlines important research topics related to the use of coarse-grained, dynamic reconfigurable architectures as the main computation units in embedded systems. In Section 1.2 we present an overview of the contributions of this dissertation to the state-of-the art in application and theory of dynamic reconfigurable architectures in embedded computing systems. Section 1.3 presents the general concept of application specification and implementation that underlies this work and outlines the interrelations between the architecture, the design tools and the applications. Section 1.4 discusses the relation of this thesis to preliminary work at ETH. Finally, Section 1.5 outlines the organization of this thesis.

## 1.1 Research Topics in Reconfigurable Architectures for Embedded Systems

Using reconfigurable computing in embedded systems is attractive for a number of reasons. Reconfigurable coprocessors have been shown to achieve high-performance and energy efficient implementations for



many applications [ASI<sup>+</sup>98, MMF98, CH02]. The functionality of the coprocessor can be changed, even in the field after deployment. This feature is a clear advantage over ASIC-based implementations and can be used to fix system malfunctions but also enables to add new functions to the system.

While programmable logic is routinely used in embedded systems for implementing digital-logic, runtime reconfiguration of hardware is not widely applied. Although academic research has shown that reconfigurable computing has a high potential, it is still difficult for the average system designer to exploit the potential of reconfigurable computing. There are a number of open issues related to the reconfigurable architectures, the application specification, and the application implementation that complicate the use of reconfigurable computing in embedded systems.

In the following, we summarize the most important research issues related to the use of dynamic reconfigurable architectures in embedded systems.

### **Reconfigurable Processor Architecture**

From economical and practical aspects its favorable to implement only the kernels of an application on the Reconfigurable Processing Unit (RPU) while the rest of the application runs on the CPU core. Hence, a close integration of CPU and RPU is needed, e. g., as attached coprocessor or Reconfigurable Functional Unit (RFU). Different CPU/RPU integration possibilities have been discussed in [CH02].

Many important multi-media signal-processing algorithms perform arithmetic operations on word sized data, rather than bit-level operations. We present an analysis of important applications in [EPP<sup>+</sup>01]. Since the fine-grained logic elements of FPGAs do not match these data-processing characteristics, a number of coarse-grained reconfigurable architectures have been designed, e. g., [SLL<sup>+</sup>00, MO99, BMN<sup>+</sup>01, SC01]. The functionality of the reconfigurable cells in these architectures differs widely, from simple ALUs up to full-featured, autonomous CPUs with dedicated instruction and data memories.

Multi-context architectures concurrently store multiple configurations (contexts) on-chip and allow for rapidly switching between these contexts. Multi-context architecture have been proposed for both fine-grained FPGA architectures, e. g., [TCJW97, DeH94, SV98], and for coarse-grained architectures, e. g., [MO99, SLL<sup>+</sup>00]. Rapid reconfigurability enables a number of execution modes that depend on fast reconfiguration, for example, hardware virtualization.

## Programming Model

The conventional logic-synthesis centric FPGA tool-flow does not match the expectations of many embedded system designers. So far, no programming and application specification model for applications of coarse-grained reconfigurable CPUs has been widely accepted. There are two main approaches to application specification and implementation for coarse-grained architectures: high-level language compilation and circuit-centric design. High-level compilation for coarse-grained architectures demands for automated hardware/software partitioning and automated hardware synthesis and has been studied for example in [CW02, LKD03, VNK<sup>+</sup>03]. But the design of a high-level compilation tool-flow that generates high-performance implementations for arbitrary applications is still an open research problem. In a circuit centric design-flow an application is specified as a structural signal-flow description. This specification approach is also the basis for block-based application specification tools, e. g., [HMSS01]. The resulting circuits are implemented with design-tools that are similar to FPGA placement and routing tools [BRM99, EMHB95].

Recently, FPGA vendors started to embed CPU cores into their high-end FPGA families [Xil05, Alt02]. The reconfigurable structures in these devices are general purpose and can be reconfigured at runtime. The lack of a clear programming model and tool-flow is certainly one cause, that the reconfigurable logic in these devices is so far primarily used for custom peripherals or static coprocessors.

## Dynamic Reconfiguration

Dynamically reconfigurable architectures allow for modifying the configuration of the device at runtime. Partially reconfigurable architectures even allow for modifying only a part of the configuration, while the other parts are running unaffectedly. Dynamic reconfiguration allows for treating the reconfigurable architecture as a time-shared computing resource and enables a number of interesting application modes, such as rapid instantiation of application-specific coprocessors or time-multiplexed execution of circuits (hardware virtualization). Hardware virtualization is particularly interesting for embedded systems, since the impact of resource limitations of a reconfigurable architecture can be reduced by hardware virtualization. Hence, the reconfigurable architecture can be dimensioned small to reduce system cost.

We have demonstrated a tool-flow and an application of a dynamically reconfigurable coprocessor for decoding compressed audio signals in [DPP02]. Depending on the audio format a suitable coprocessor is loaded on demand at runtime.

Although dynamic reconfiguration has been identified as an interest-

ing research area and a number of architectures support dynamic reconfiguration, e. g., [TCJW97, DeH94, SV98, Xil05, Atm05], dynamic reconfiguration is rarely applied in practice. This fact can be attributed largely to a missing integration into a coherent application specification and implementation tool-flow. Applications of dynamic reconfiguration frequently require hand-crafted designs and custom tools, which is prohibitive for a widespread use.

### **Performance Evaluation**

Accurate evaluation of the performance of a reconfigurable processor is a difficult problem, since the application is running partially on the CPU core and partially on the RPU. The system-level performance depends on the execution time for the hardware and software parts of the application and on the performance of the CPU/RPU communication. While determining the performance of RPU execution is relatively simple, cycle-accurately modelling the CPU performance is much more difficult due to dynamic effects in the CPU, e. g., branch prediction, caching, out-of-order execution, etc.

Performance evaluation of CPUs with fine-grained RFUs have been discussed in [CEC01, YMHB00, LTC<sup>+</sup>03], but these approaches use a simplified, purely functional model of the RFU and do not account for all dynamic effects. Although the importance of system-level performance evaluation for reconfigurable processors is widely recognized, the topic has not gained a lot of attention so far.

## **1.2 Contributions**

In this dissertation we make a number of contributions to the state of the art in reconfigurable processor architectures, design-tools and applications.

The main contribution of this thesis is the co-design of a dynamically reconfigurable processor architecture for embedded systems. The architecture and the design-tools are co-designed to enable an application specification model that explicitly uses hardware virtualization.

Specifically, we make the following contributions:

- We present a new dynamically reconfigurable processor architecture named Zippy. The architecture combines an embedded CPU core with an RPU attached to the CPU's coprocessor interface. The RPU is a multi-context architecture and supports single-cycle, dynamic reconfiguration. An array of coarse-grained reconfigurable cells

and on-chip FIFO buffers tailor the architecture to the processing of streaming multi-media applications.

Since embedded systems frequently have stringent cost (i. e., chip-size) constraints, we aim at keeping the size of the reconfigurable co-processor extension to the CPU small by using a reconfigurable array of modest size. Hardware virtualization allows for executing application specific co-processors of virtually arbitrary size. Hence, we employ hardware virtualization to reduce the impact of the resource constraints imposed by the small array.

The architecture provides dedicated hardware units to efficiently support hardware virtualization, i. e., the time-multiplexed execution of circuits. Specifically, the architecture offers multiple configuration contexts, dedicated context sequencers, and units for efficient inter-configuration communication.

- A suite of design-tools has been developed concurrently with the Zippy architecture. Zippy is not a single concrete architecture, but a widely parametrized, cycle-accurate architecture simulation model. Likewise, the design-tools are also parametrized and can generate software and hardware implementations for all architecture variants. The hardware implementation tools base on iterative placement and routing algorithms, that have been adapted to the coarse-grained architecture.
- The performance of an instance of a Zippy architecture is evaluated by co-simulation. The co-simulation environment combines a cycle-accurate simulator for the CPU core with a the cycle-accurate model of the RPU. This performance evaluation environment allows for bit-exact and cycle-true performance evaluation of Zippy architectures. The parametrized nature of the Zippy model and the design-tools allow for exploring the design-space for a whole family of architectures.
- We provide an application specification model that considers the coarse-grained nature of the reconfigurable cells as well as the dynamic reconfiguration capabilities of the RPU. If circuits exceed the resource supply of the RPU, we use hardware virtualization to decompose the application into a set of smaller, communicating parts, where each part respects the resource constraints. The application specification model is translated to an implementation by an end-to-end tool-flow, which is widely automated.
- Temporal partitioning allows for splitting arbitrary sequential circuit into a set of smaller circuits. When executed cyclically, these

subcircuits implement the same function as the original circuit. We present a novel method for optimal temporal partitioning of sequential circuits. The method bases on formulating the temporal partitioning problem as an optimization problem. In contrast to related work, our approach is more general since we consider structural modifications of the circuit. Additionally, thanks to a Mixed Integer Linear Program (MILP) problem formulation, we can solve the temporal partitioning problem optimally, while the majority of related work uses heuristic approximations.

Zippy is a concrete architecture family, and the methods, algorithms and models in this work have been tailored to this concrete architecture family. However, many methods and ideas, e. g., the method for optimal temporal partitioning, are also relevant in a broader context and can be applied to different, dynamically reconfigurable architectures.

### 1.3 Application Specification and Execution on the Zippy Architecture

In this section, we outline the general concept for application specification, implementation and execution, that forms the basis of this work. The Zippy architecture and the design-tools have been co-designed to enable the implementation of this concept. Figure 1 illustrates the general concept.

#### a) Application specification

An application for a reconfigurable processor architecture is usually not mapped to the RPU as a whole, but only the computational intensive kernels of the application are executed on the RPU. The first step in the implementation tool-flow is thus an analysis phase that identifies the kernels of the application. The example in Fig. 1(a) shows an application that has been decomposed into three communicating tasks. These tasks will be implemented in software or hardware. The semantics of synchronization and communication between the tasks is formally defined with a process network formalism.

#### b) Hardware/Software partitioning

This step performs hardware/software partitioning, which is a manual process in this work. Each task is assigned either to the CPU or to the RPU for execution. Software tasks are specified as C source code. Software libraries for accessing the hardware communication

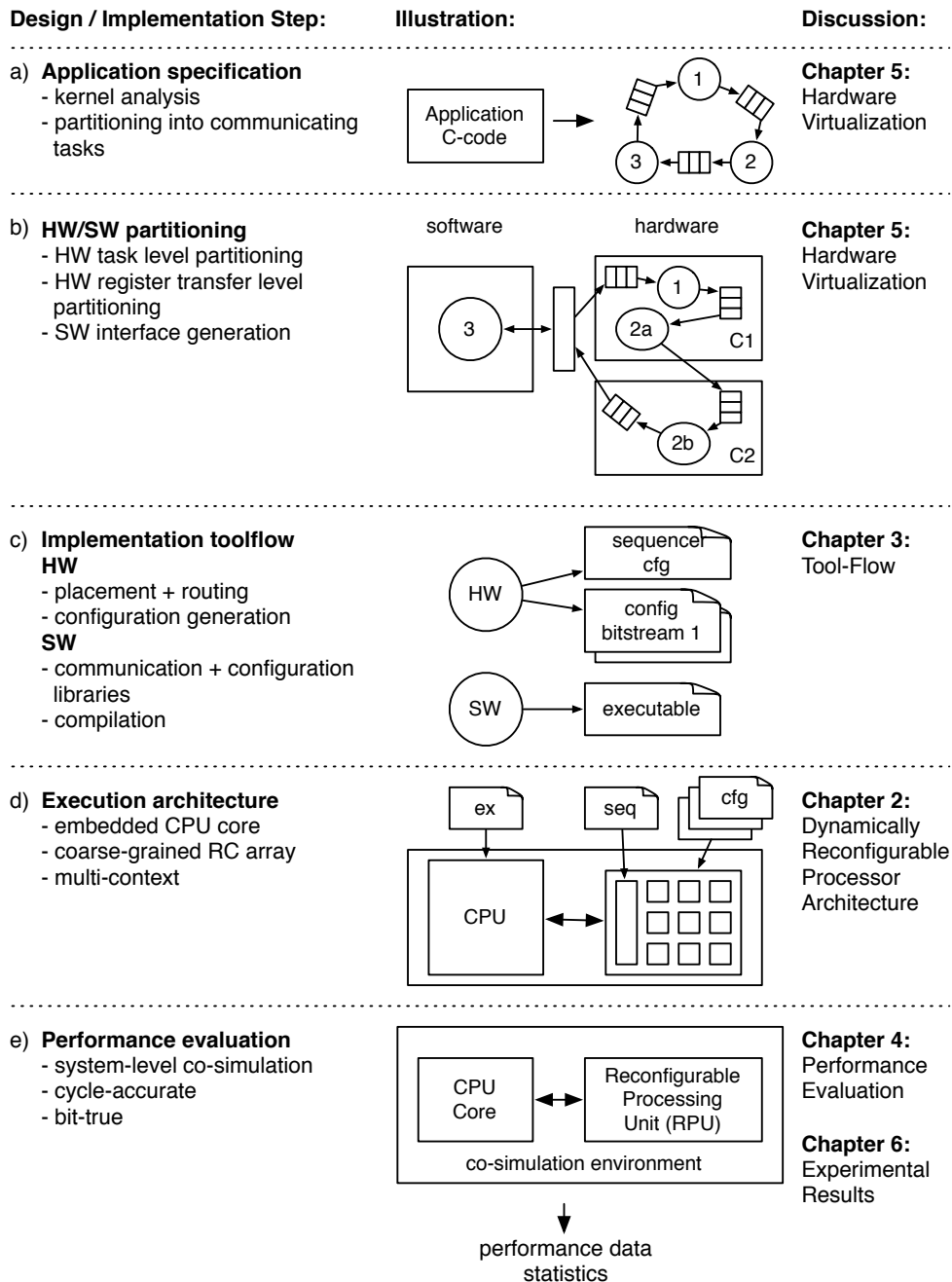


Fig. 1: Overall Concept for Application Specification and Execution

interfaces are linked to the software tasks. If a task is mapped to the RPU (hardware task), the designer has to specify a netlist of a functional equivalent circuit.

While the CPU can execute software tasks of arbitrary size thanks to virtual memory, the size of the hardware tasks is limited by the device capacity.

We use hardware virtualization to overcome this limitation. Hardware virtualization can be applied at the task level or at the register transfer level. At the task level, we decompose tasks into smaller subtasks, which is a manual process in this work. At the register transfer level, tasks cannot be naturally decomposed into subtasks, but an automated temporal partitioning process is used for this purpose.

The partitioning process is applied offline during the design phase and determines a static partitioning and a corresponding execution schedule. At runtime, the partitions are not modified, but the execution schedule can be varied depending on the timing constraints of the application.

Fig. 1(c) shows that tasks 1 and 2 are mapped to hardware, and task 3 is mapped to software. The tasks communicate via FIFO buffers. The hardware tasks use hardware virtualization, task 2 is temporal partitioned into task 2a and task 2b.

### c) **Implementation Tool-Flow**

The implementation tool-flow, see Fig. 1(c), transforms the hardware and software tasks to representations that can be executed on the reconfigurable processor.

The hardware tasks pass through an automated placement and routing process that generates the configuration data for the reconfigurable array. If the hardware tasks use hardware virtualization, the hardware tool-flow also generates a configuration for the context sequencer that controls the activation and execution of contexts.

The software tasks are compiled into executables, which access the RPU with the communication and configuration libraries.

### d) **Execution architecture**

The execution architecture is a reconfigurable processor which is based on an embedded CPU core, see Fig. 1(d). A coarse-grained reconfigurable array is attached to the CPU's coprocessor port. The reconfigurable array supports fast dynamic reconfiguration by providing multiple configuration contexts and a programmable context sequencer.

The reconfigurable processor executes the software executable generated by the software tool-flow and downloads the configuration data and context sequencer configuration to the RPU.

At runtime, the CPU executes the software tasks, while the hardware tasks are running on the RPU. Software and hardware tasks communicate via FIFO buffers, that are accessible from software via the coprocessor port.

In this example, the hardware tasks use temporal partitioning. Thus, during execution, the context sequencer continuously switches between the execution of context 1 (implements task 1 and 2a) and context 2 (implements task 2b).

#### e) Performance evaluation

We use a system-level, cycle-accurate cosimulation for validating the function of the architecture and for generating execution statistics. Our simulation method is execution-based, i. e., the simulation models are sufficiently detailed, to actually execute the code on the CPU core and run the circuits on the RPU.

## 1.4 Preliminary Work at ETH

This thesis is part of the Zippy research project. The Zippy project was a joint project of the Computer Engineering Lab (TIK) and the Electronics Lab (IfE) at ETH Zürich and was running from 2000–2003. The project aimed at studying reconfigurable architectures for embedded computing in handheld and wearable computing applications. In [EPP<sup>+</sup>01, PEW<sup>+</sup>02, PEW<sup>+</sup>03] we have shown the potential of reconfigurable technology in wearable computing for implementing energy efficient, high-performance computing.

Two dissertations have been planned in the context of the project. Rolf Enzler submitted the first of these dissertations [Enz04], for which he obtained his PhD degree in 2004. Enzler's thesis is focused on design-space exploration for a dynamical reconfigurable processor, which is an ancestor of the Zippy architecture. He proposes system-level co-simulation as performance evaluation methodology and presents a corresponding simulation framework. Enzler's thesis concludes with a case-study that evaluates the design trade-offs in terms of performance and estimated chip-size for different variants of the reconfigurable processor architecture.

While this dissertation builds on the infrastructure that has been developed in the course of the Zippy project, namely the co-simulation



framework and the basic reconfigurable architecture, the focus of this work is different from the design-space exploration oriented thesis of Rolf Enzler: This thesis advocates the use of dynamic reconfiguration for embedded systems and proposes an end-to-end design-flow that considers hardware virtualization. To this end, the reconfigurable architecture, the application specification model and the corresponding tool-flow have been co-designed. Additionally, we have developed a novel method for optimal temporal partitioning that can be applied to the Zippy architecture.

## 1.5 Thesis outline

This section outlines the structure of this thesis.

Chapter 2 (Dynamically Reconfigurable Processor Architecture) introduces the Zippy reconfigurable processor architecture, which is the basis for this work. The architecture is defined as a widely parametrized, cycle-accurate architecture model.

Chapter 3 (Tool-Flow) presents the application implementation tool-flow for the Zippy architecture. The software tool-flow uses a library based approach to control the reconfigurable processing unit. The hardware tool-flow implements a placement and routing process that targets all parametrized variants of the Zippy architecture.

In Chapter 4 (Performance evaluation) we present a performance evaluation environment for the simulation of the Zippy architecture. We combine a cycle-accurate CPU simulator with a detailed VHDL model of the reconfigurable processing unit into one co-simulator to provide the developer with cycle-accurate and bit-exact performance results.

In Chapter 5 (Hardware Virtualization) we introduce hardware virtualization and classify different approaches. We discuss the applicability of hardware virtualization in the Zippy architecture and in embedded systems in general. We introduce a novel method for hardware virtualization by temporal partitioning. We use a problem formulation based on mathematical programming, which can be used to solve the problem optimally.

Chapter 6 (Experimental Results) presents two case studies that demonstrate that hardware virtualization on the Zippy architecture is feasible. We use the co-simulation environment for cycle-accurate performance evaluation and we discuss performance-area trade-offs for different implementations.

In Chapter 7 (Conclusions) we summarize the contributions of this thesis and conclude the thesis with an outlook to future work.



# 2

## Dynamically Reconfigurable Processor Architecture

In this chapter we introduce a dynamically reconfigurable processor architecture called *Zippy*, which has been developed for this work.

Section 2.1 presents the design objectives for the *Zippy* architecture, which is a parametrized architecture and simulation model of a reconfigurable processor. We motivate the use of the model as an experimentation framework for reconfigurable processors in embedded systems. We identify hardware virtualization as an important execution model and present the architectural requirements for an efficient implementation of hardware virtualization.

Section 2.2 outlines the system architecture of the reconfigurable processor and introduces the two main components: the CPU core and the Reconfigurable Processing Unit.

In Section 2.3 the architecture of the Reconfigurable Processing Unit is described in detail. In particular, the section elaborates on the dedicated hardware support for hardware virtualization.

While this chapter focuses exclusively on the hardware architecture, *Zippy* has been co-designed with a number of supporting design-tools and programming models. A hardware and software tool-flow for the *Zippy* architecture is presented in Chapter 3. Chapter 4 introduces a framework for system-level performance evaluation. *Zippy* has been designed to support execution models for hardware virtualization, the

related models, tools and algorithms are presented in Section 5.

## 2.1 Design Objectives

The *Zippy architecture* is not a single, concrete architecture but an architectural *simulation model* of a dynamically reconfigurable processor. The model integrates an embedded CPU core with a coarse-grained Reconfigurable Processing Unit (RPU). Such architectures are commonly known as Reconfigurable Processors or hybrid CPUs [CH02].

Zippy was created to provide an experimentation framework to study the use of coarse-grained, dynamically reconfigurable CPUs and the associated design tools in the embedded system domain. The architectural model is widely parametrized and can be configured to resemble whole families of reconfigurable CPUs. Zippy architectures are modeled at a level of detail that is sufficient for system-wide cycle-accurate simulation. The goal of this simulation-based approach is to explore different design alternatives and to evaluate the impact of the various architectural parameters. An associated co-simulation framework for performance evaluation will be presented in Chapter 4.

In contrast to many approaches studying reconfigurable technology, we aim not at the general-purpose but at the embedded computing domain. The embedded domain puts more stringent requirements on computing power, energy consumption, cost, 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.

The applications we are targeting with our architecture are mainly digital-signal processing applications for multi-media, communications and cryptography, as they occur in handheld and wearable applications. Our application analysis [EPP<sup>+</sup>01] has shown that these applications are characterized by a high demand for arithmetic and logic operations on word-sized operands. Typically the computational kernels are rather small and work on streaming data. Consequently, we have designed our architecture as a coarse-grained reconfigurable array rather than a typical, fine-grained FPGA-like architecture.

A specific goal of this work is to investigate dynamic reconfiguration in the context of embedded systems. In particular, we are interested in studying *hardware virtualization*. Hardware virtualization is an implementation technique that partitions a circuit into a number of smaller sub-circuits at compile-time. At run-time, these sub-circuits are executed on a reconfigurable hardware device in a time-multiplexed way. The sub-circuits perform the same function as the original circuit using less

hardware resources, but at the expense of reduced performance. The foundations of hardware virtualization and its application to the Zippy architecture will be discussed in more detail in Chapter 5.

Although hardware virtualization can be used with any reconfigurable architecture, an efficient implementation requires support by dedicated hardware components in the reconfigurable architecture, namely: 1) fast reconfiguration, 2) fast, repeated activation of a fixed sequence of configurations, and 3) efficient data-transfers between configurations.

The Zippy architecture supports these operations with the following architectural features:

- 1) *multi-context configuration architecture*, i. e., several configurations are stored on-chip concurrently. The activation of a stored configuration happens within a single cycle
- 2) *dedicated context sequencers* that autonomously activate a programmable sequence of configurations for a given time, and
- 3) *data-transfer register-files* that are shared between different configurations for communicating data within the reconfigurable array (see Section 2.3.1), and *FIFO memory-queues* for communication between configurations and with the reconfigurable processor's CPU core

Important architectural foundations for this work, e. g., multi-context architectures, dynamic reconfiguration, have been treated in related work, but have not been incorporated in a consistent framework for application specification, implementation and execution.

The contribution of this thesis in the context of reconfigurable processor architecture is the design of a coarse-grained reconfigurable processor architecture, that is explicitly co-designed with a programming model and design-tools for the application of hardware virtualization techniques.

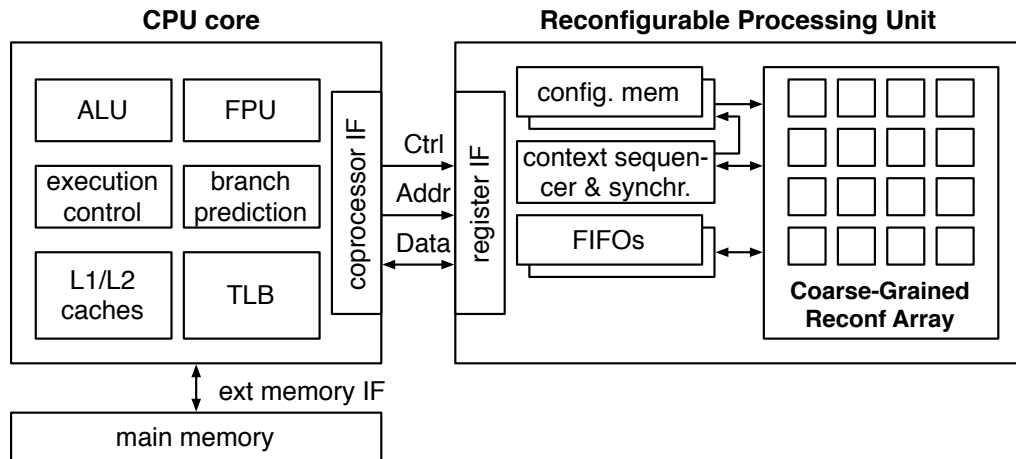
## 2.2 System Architecture

This section presents an overview of the Zippy system architecture which consists of a CPU core and a Reconfigurable Processing Unit (RPU). The RPU itself will be presented in more detail in Section 2.3.

### 2.2.1 Structure

Zippy is a reconfigurable processor composed of two main units: the *CPU core* and the *Reconfigurable Processing Unit (RPU)*. Figure 2 presents a schematic drawing of the Zippy system architecture.

The RPU acts as a coprocessor to the CPU. i. e., the CPU's coprocessor interface is attached to the RPU's register interface. All data-transfers



**Fig. 2:** The Zippy system architecture comprises a CPU core and a Reconfigurable Processing Unit (RPU)

between the CPU and the RPU are performed via this coprocessor interface. Additionally, the various functions of the RPU (e. g., configuration loading, context sequencer programming or synchronization of CPU and RPU) are exposed to the CPU via read and write operations on the register interface. We assume that the CPU and the RPU use the same clock.

Our target applications work on data-streams rather than on single data values and typically do not need random data access. CPU and RPU can operate concurrently and synchronize only on demand. Hence attaching the RPU to the coprocessor port is appropriate, since only infrequent communication is required. Attaching the RPU as a coprocessor is used in many reconfigurable processor architectures, e. g., in the GARP [HW97], OneChip [WC96], or REMARC [MO99] architecture.

Alternatively to a dedicated coprocessor interface, the RPU could also be attached to the memory interface of the CPU. We have studied this mechanism in [PP03b] where we interface an Field-Programmable Gate-Array (FPGA)-based coprocessor to the system memory bus in a standard PC. Using the memory bus as a coprocessor interface is attractive, because it provides a high-bandwidth and low-latency communication interface and doesn't require modifying the CPU. On the other hand, using state-of-the-art memory interfaces for memory mapped IO becomes increasingly complex since CPUs use caching and complex memory-access protocols for SDRAM access.

## 2.2.2 Embedded CPU core

We use the *SimpleScalar CPU simulator* for modeling the CPU core in the Zippy architecture. Thus the CPU core is not a real CPU core, but a cycle-

accurate simulation model of SimpleScalar’s architecture [ALE02]. SimpleScalar is a well established tool for CPU architecture research because it provides cycle-accurate simulation of a highly configurable architecture and has an extensible instruction set. SimpleScalar has been also used for simulating the CPU core in the OneChip [CEC01] architecture.

SimpleScalar models a parametrized 32-bit super-scalar RISC CPU architecture with a MIPS-like instruction set. The principal parameters of the CPU core are:

- the number of computation units (integer and floating-point ALUs and multipliers),
- decode, issue, and commit bandwidths
- the sizes of the instruction fetch queue, the register update unit, and the load/store queue,
- in-order or out-of-order execution, and
- the branch prediction mode.

Further, the architecture of the on-chip data and instruction caches can be configured. At maximum, two cache-levels are supported, and caches can be split or unified. The cache size, the number of sets, the associativity, and the replacement strategy can be configured.

These configuration parameters allow for customizing the CPU model to resemble a broad range of architectures, from small low-end CPUs with a single integer ALU and small caches to powerful super-scalar CPU architectures with multi-level cache hierarchies. Table 1 summarizes the parameter settings for two CPU architectures, that we have used in our studies. The *Embedded CPU* settings approximate the Intel StrongARM architecture, which is a CPU that is frequently used in embedded systems. This model is used as the default CPU configuration of the Zippy architecture. The *Desktop CPU* configuration is the default configuration for the SimpleScalar architecture and simulates a 4-way super-scalar CPU. We use this model for comparing the impact of the CPU architecture on the system performance.

The original SimpleScalar architecture does not have a coprocessor interface. For interfacing the RPU to the CPU core we have extended SimpleScalar with a new functional unit that implements the coprocessor interface. The extensible instruction set allowed us to add new instructions for co-processor access.

### 2.2.3 Reconfigurable Processing Unit

The main computational resource on the RPU is the coarse-grained *reconfigurable array*. To provide the reconfigurable array with configurations and data, the RPU offers memories for storing the configurations and FIFO memory queues for data-transfers between CPU and RPU. The

Parameter	Embedded CPU	Desktop CPU
Integer units	1 ALU, 1 Multiplier	4 ALU, 1 Multiplier
Floating point units	1 ALU, 1 Multiplier	4 ALU, 1 Multiplier
L1 I-cache	32-way 16k	1-way 16k
L1 D-cache	32-way 16k	4-way 16k
L2 cache	none	4-way 256k (unified)
Memory bus width	32 bit	64 bit
Memory ports	1	2
Instruction fetch queue size	1	4
Register update unit size	4	16
Load/store queue size	4	8
Decode width	1	4
Issue width	2	4
Commit width	2	4
Execution order	in-order	out-of-order
Branch prediction	static (not-taken)	bi-modal

**Tab. 1:** CPU configurations of embedded and desktop CPU models.

RPU model is implemented as a cycle-accurate VHDL simulation model. This modelling style allows us to use different levels of abstraction in the RPU model. We specify complex behavior or computations with behavioral VHDL constructs on an abstract level, while the discrete event semantic of VHDL allows us to preserve overall cycle accuracy.

Just as the CPU core, the RPU architecture model is also highly configurable. This enables us to explore and evaluate a large family of reconfigurable processors.

## 2.3 Reconfigurable Processing Unit Architecture

This section discusses the design of Zippy's RPU. A schematic diagram of the Reconfigurable Processing Unit is shown in Figure 3.

Zippy is a multi-context architecture, i.e., several configurations can be stored concurrently on-chip in the *configuration memory*. The RPU can switch rapidly between these configurations. The activation and sequencing of configurations is controlled by the *context sequencer*. The *FIFOs* are accessible by both, the reconfigurable array and the CPU core and are used to pass input data and results between the CPU core and the reconfigurable array and also between different configurations (contexts) of the RPU. The *register interface* provides the CPU with access to the RPU



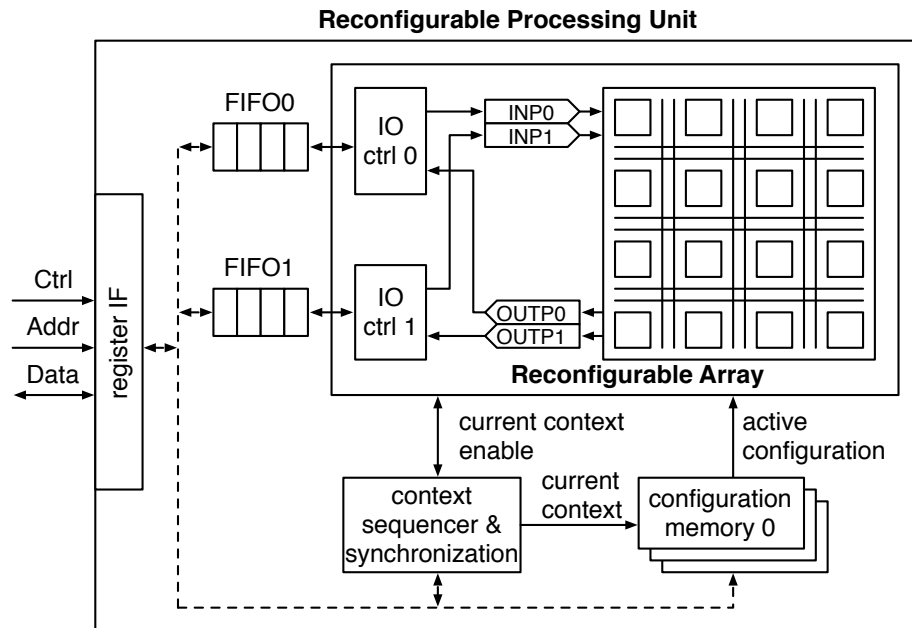


Fig. 3: Reconfigurable Processing Unit Architecture

function blocks.

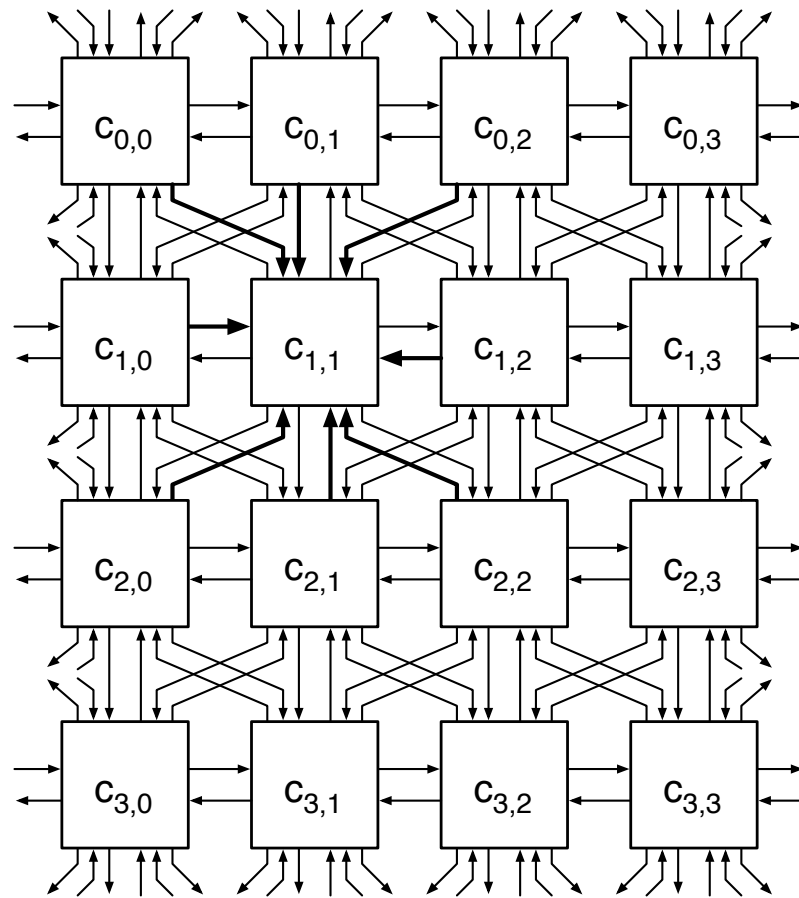
### 2.3.1 Reconfigurable Array

The *reconfigurable array* is the core unit of the Zippy architecture. It is organized as an array of uniform, coarse-grained reconfigurable cells, which are connected by two programmable interconnection structures: a local interconnect and a bus interconnect. The reconfigurable array has two input and two output ports (INP0/1) and (OUTP0/1) that connect the internal buses to the IO-controllers on the RPU.

The bit-width of the ALU in the cells, the interconnection wires, and the FIFOs can be configured. Typically, the bit-width is set to 16 bit or 24 bit, which are common bit-widths for many fixed-point signal processing algorithms.

#### Interconnection network

The Zippy architecture uses two interconnect structures: a *local interconnect* between neighboring cells, and a *bus interconnect*, between cells in the same row or column. The local interconnect is shown in Figure 4. Each cell can read the output data from all of its 8 immediate neighbors. Having a fully uniform interconnect allows for writing simpler placement and routing algorithms. To this end, the array's interconnect is cyclically continued at the edges of the array to make it fully symmetric and uni-



**Fig. 4:** Reconfigurable Array: Local Interconnect

form. For example, the connection of cell  $c_{1,0}$  pointing westward connects to cell  $c_{1,3}$ .

Figure 5 shows the bus interconnect for a  $4 \times 4$  cell instance of the Zippy array. Programmable routing switches are indicated by small bus-driver symbols at the crossing of wires. There are three types of horizontal buses: the horizontal north buses ( $hbus_n$ ) that connect cells in *adjacent rows*, the horizontal south buses ( $hbus_s$ ) that connect cells in the *same row*, and the memory buses ( $hbus_mem$ ) that connect all cells in a row to an *on-chip memory block*. Additionally, the vertical east buses ( $vbus_e$ ) provide connectivity between the cells in the *same column*. The example in Fig. 5 shows an architecture with 2 horizontal north, 2 horizontal south and 2 vertical east buses. The number of buses can be chosen independently and is fully parametrized. The bit-width of the interconnect corresponds to the data-width of the ALU.

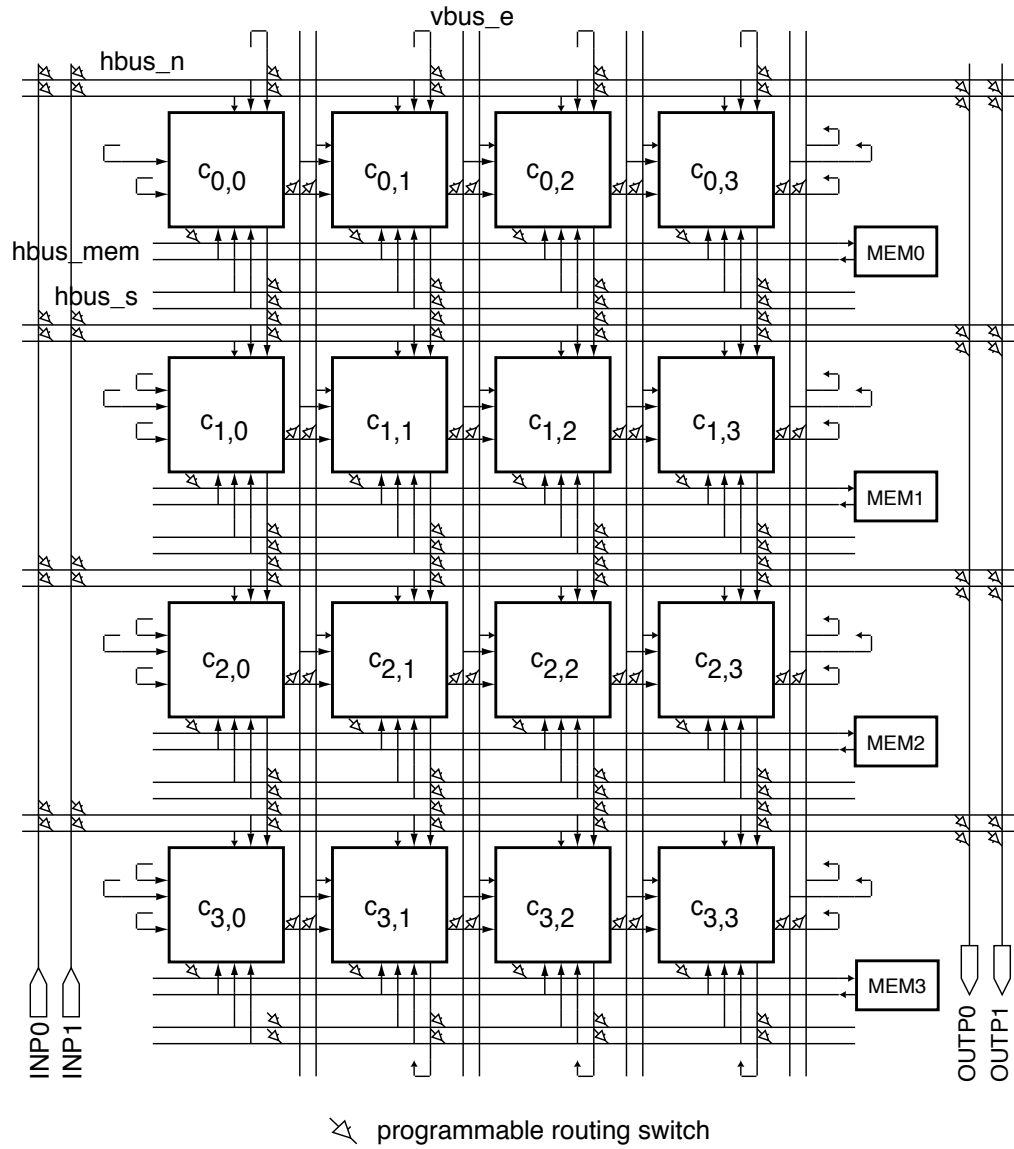
### Reconfigurable Cell

The *reconfigurable cell* is composed of three main structures: a versatile input structure with overall three inputs, an operator block, and an output structure. Figure 6 presents a detailed schematic of the cell. The function of all shaded parts is controlled by the configuration. As Zippy is a multi-context architecture every context has a distinct configuration for these parts. Additionally to the inputs, the current context selector is also fed into the cell to control the input multiplexers of the input and output register-files.

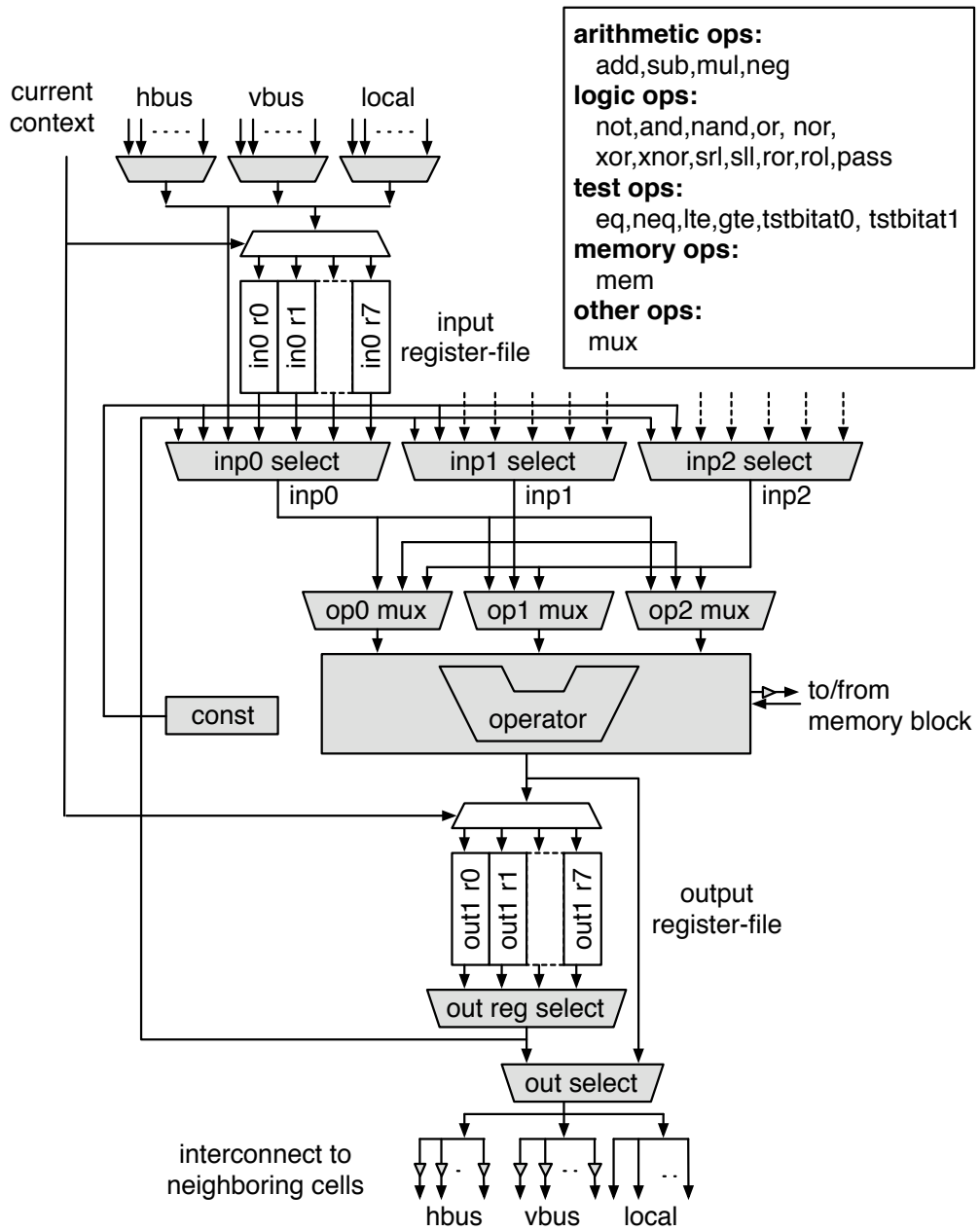
The input multiplexer ( $inpX\ select$ ) connects each input ( $inpX$ ) to either of six sources: to one of the cell's inputs (horizontal bus, vertical bus, any local neighbor), to a configurable constant, to the cell's output register (feedback path), or to one of the registers of the input register-file. The input register-file provides a dedicated register per input and context, which stores the selected bus or local input. The input register-files can be used for transferring data between the contexts, since the input multiplexer ( $inX\ select$ ) has also access to input registers that have been written in all different contexts.

The operator block bases on a fixed-point ALU and performs the cell computation. The operator takes up to three inputs and computes one output value. Figure 6 includes a table of supported cell operations.

Most arithmetic and logical operations are self-explaining. The *pass* operation directs the unmodified input value to the output, what can be useful for routing purposes. The *testbitat* operations are used for bit-tests.  $testbitat0(value,mask)$  takes an input value and a mask and returns 1 if all bits that are set in *mask* are set to 0 in *value*, otherwise the operator returns 0. The  $testbitat1$  operator works analogously for testing whether bits are



**Fig. 5:** Reconfigurable Array: Bus Interconnect



**Fig. 6:** Reconfigurable Processing Unit: Cell architecture (shaded parts are controlled by the configuration)

set to 1. The  $\text{mux}(sel,a,b)$  operator is a ternary operator that forwards input  $a$  or input  $b$  to the output, depending on the value of the least-significant bit of  $sel$ . This operator is used for implementing control-flow operations, i.e., data-dependent processing. Each row of cells has access to a shared ROM memory block, see Fig. 5. The  $\text{rom}(addr)$  operation of a cell reads the contents at address  $addr$  of the ROM associated with this cell.

Like the input register-file, the output register-file provides a dedicated register per context. The output of the operator block is stored in a dedicated register of the output register-file. The output of the cell can be selected either as the combinational output of the operator block, or as the contents of an output register. Since the output multiplexer has access to all output registers it can also be used for data-transfers between different contexts.

The structure and function of the register files is similar to the micro-registers in the TMFPGA multi-context FPGA architecture [TCJW97]. In contrast to Zippy, TMFPGA provides only an output register file (but no input register files), and the cell output provides both, the combinational output and the registered output.

In general, the use of either input or output registers in a reconfigurable cell is sufficient to implement arbitrary static circuits. Hence, most architectures provide only output registers to reduce the total register count. The Zippy architecture offers both, input and output register files, to enable studies on the usage of input and output registers in applications of dynamic reconfiguration, e. g., hardware virtualization.

### Memory Blocks

Each row of the reconfigurable array has an associated ROM memory block. The depth of the ROM is an architecture parameter, the content of the ROM is defined by the configuration.

### 2.3.2 FIFOs

The RPU provides two *FIFO memory queues*. The depth of the FIFOs is parametrized. The FIFOs are used for transferring input data from the CPU to the RPU and for reading the results back to the CPU after processing. As the FIFOs are accessible from any context they can be also used for passing data between contexts. This is particularly important for hardware virtualization through virtualized execution where FIFOs are used to pass intermediate data from one context to the next (see Section 5.2.2).

The control signals for the FIFOs are generated by the unit that accesses the FIFOs, i.e., either the register interface, or the IO controller if the

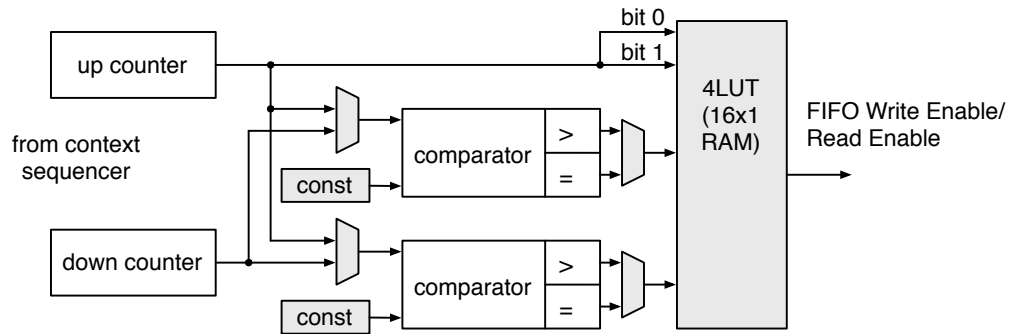


Fig. 7: IO port controller (shaded parts are controlled by the configuration)

reconfigurable array accesses the FIFO.

### IO Ports and IO Controllers

For controlling the access from the reconfigurable array to the FIFOs on the RPU, the Zippy architecture provides dedicated IO controllers. These simple but effective controllers are a unique feature of the Zippy architecture and allow for generating many important FIFO activation sequences.

The controllers can generate repetitive activation sequences, e. g., activating the FIFO in every fourth cycle. Additionally, the controllers can generate more complex activation schemes that depend on the number of execution cycles without fixing this number at circuit compilation time. For example, writing to a FIFO can be stopped a given number of cycles before the end of the last cycle. This activation mode is important for pipelined circuits when the transient data-samples, caused by filling and clearing the pipeline, shall be discarded.

The reconfigurable array accesses the FIFO contents via input and output buses that are connected to *IO ports*, i. e., input ports (INP0/1) and output ports (OUTP0/1), as well as to the bus interconnect, see Fig. 5. Each IO port is equipped with a configurable controller (see Fig. 7) that generates the read and write enable signals for accessing the FIFO buffers. The controller comprises two comparators, two configurable constants, a number of programmable multiplexers, and a 4 input look-up table (LUT). The controller reads two inputs named *up counter* and *down counter* that are generated by one of the context sequencers. In the case of the *cycle-counter context sequencer*, the up counter denotes the number of execution cycles the array has performed in the current execution phase, while the down counter denotes the remaining cycles in the current execution phase. For more details on the context sequencers we refer to Section 2.3.4.

Each comparator compares one of the counters to a configurable con-

RPU coprocessor register	CPU access
FIFO {1,2}	R/W
FIFO {1,2} level	R
configuration memory {1...n}	W
RPU reset	W
cycle count	R/W
context select	W
context sequencer mode	W
context sequencer temporal partitioning contexts	W
context sequencer start	W
context sequencer status	R
context sequence store {1...s}	W

**Tab. 2:** Register Interface: Commands

stant and passes the result (either “greater than” or “equal”) to the LUT. The two least significant bits of the up-counter determine the remaining two inputs of the LUT. The configuration of the 4 input LUT allows for defining an arbitrary boolean function of all four inputs.

### 2.3.3 Register Interface

The *register interface* implements the interface from the CPU’s coprocessor port to the RPU and vice versa. It is accessed with the coprocessor instructions that we have added to SimpleScalar. Table 2 provides an overview of the commands supported by the register interface.

The FIFO functions offer FIFO read and write access to the CPU, further the FIFO’s fill-level can be queried. The *configuration memory command* initiates the upload of a configuration to a context memory. The *cycle count* command sets the number of execution cycles if the cycle-counter context sequencer is used. This register is also polled by the CPU for detecting the end of the execution. *Context select* activates a context for execution. Optionally, the registers in the cell’s register-files that are associated with the selected context can be reset on activation. The remaining *context sequencer commands* are used for selecting the context sequencer mode and setting the sequencer’s parameters.

### 2.3.4 Context Sequencer and Synchronization

Hardware virtualization requires the execution of a sequence of configurations (contexts) where each context is executed for a predefined number of cycles. Efficient context sequencing is a precondition for efficient hard-



ware virtualization, but has so far not been explicitly targeted in related work. The requirements on the efficiency depend on the hardware virtualization approach (see Chapter 5). Hardware virtualization by “temporal partitioning” requires a context switch in every cycle and hence demands for a context sequencer with low timing overhead. Although temporal partitioning has been proposed for the fine-grained TMFPGA [Tri98, TCJW97] and DPGA [DeH96a] architectures, these architectures do not provide the required context sequencer for an efficient implementation.

If the “virtualized execution” hardware virtualization method is used, the execution can tolerate longer context switching overheads, since context switches occur less frequently. Hence, the context sequencing could be controlled either by the reconfigurable processors CPU core or a dedicated sequencer. However, a dedicated sequencer as it is provided in the Zippy architecture increases the performance.

The Zippy architecture offers a choice of *three different context sequencers* that are designed to match the requirements for hardware virtualization:

- a) the cycle-counter sequencer,
- b) the virtualized execution sequencer, and
- c) the temporal partitioning sequencer.

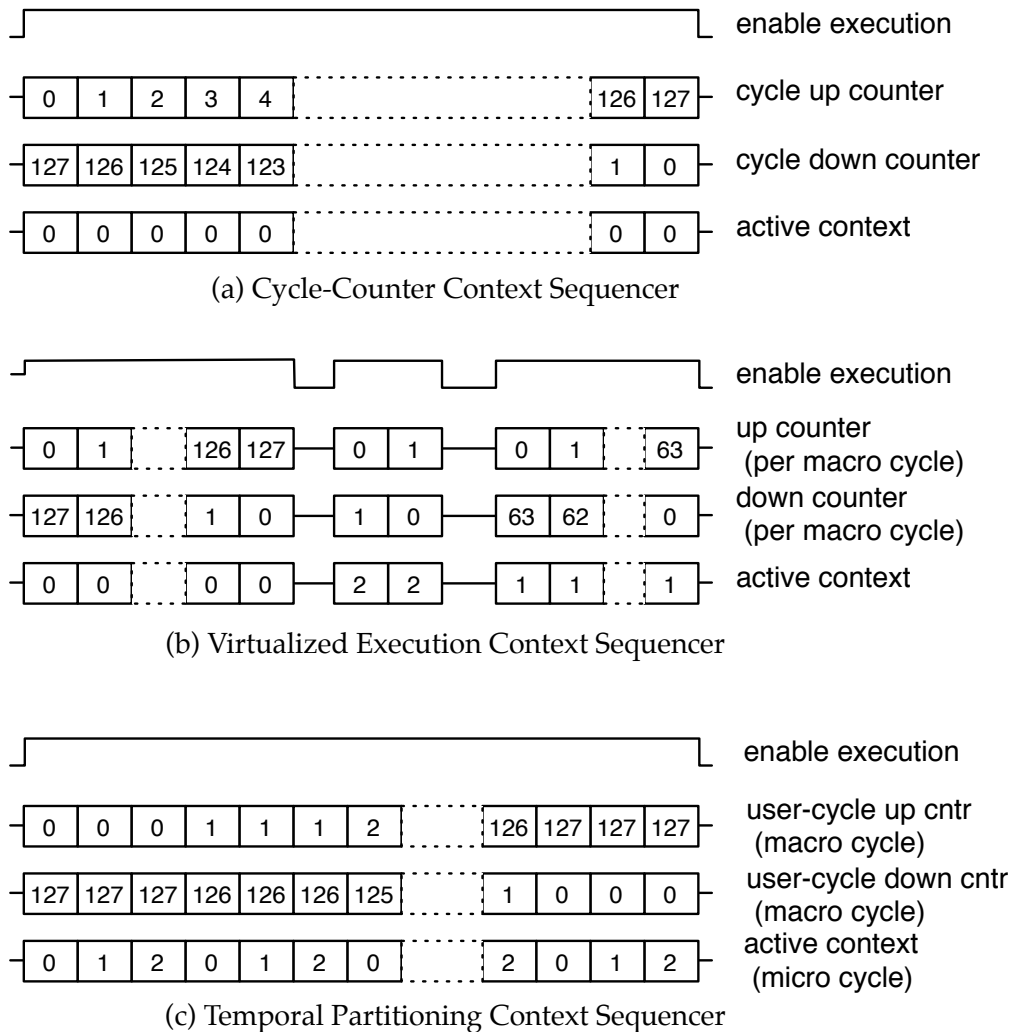
The context sequencers handle the activation and switching of contexts without CPU intervention and thus relieve the CPU from this task. A limitation of SimpleScalar, which is consequently inherited by the Zippy architecture, is the lack of interrupt support. The CPU core thus cannot be asynchronously notified of events on the RPU. Hence, the synchronization of the CPU and the RPU, i. e., detecting the termination of a sequencer is implemented with polling. To avoid the inherent overheads involved with polling, we have designed the context sequencers to work largely autonomously without CPU intervention.

Figure 8 shows the activation sequences that are generated by the sequencers with timing-diagrams.

#### a) Cycle-Counter Context Sequencer

The *Cycle-Counter Context sequencer* is the basic form of a context sequencer. It executes a single context for a configurable number of cycles, after which the execution is stopped. This sequencer is very similar to the mechanism proposed by Hauser to control the execution of the GARP architecture [Hau97].

Figure 8(a) illustrates the generated activation patterns. In this example, context 0 is activated for 128 cycles. The CPU starts the RPU execution for a given number of cycles by first activating the desired context (*context select* command) and then programming the Cycle-Counter Sequencer with the desired number of clock cycles (*cycle*



**Fig. 8:** Context sequencers

*count* command). The sequencer generates two counter values, the *cycle up counter* and the *cycle down counter*, that are also passed to the IO controllers. After the initialization, the sequencer starts execution by enabling the reconfigurable array (*enable execution* signal). The cycle down counter outputs the remaining execution cycles and is decremented by one in every cycle. When the cycle down counter reaches zero the execution is stopped. The CPU detects the end of the execution phase by polling the *cycle down* register for reaching zero.

#### b) Virtualized Execution Context Sequencer

The *Virtualized Execution Context Sequencer* executes a programmable sequence of contexts. After a context has been executed, the next context is activated and executed, until the whole sequence of contexts has been processed. The duration of the execution can be configured per context.

Figure 8(b) shows an example for a virtualized execution sequencer with 3 contexts: context 0 is executed for 128 cycles, then context 2 is executed for 2 cycles, and finally, context 1 is executed for 64 cycles.

After enabling the virtualized execution mode (*context sequencer mode* command) all required configurations and the desired schedule are loaded to the RPU (*context sequence store* commands). When the CPU triggers the sequencer with a *context sequencer start* command the sequencer starts to execute the programmed schedule autonomously until the last context has finished. Switching between contexts takes 3 cycles. The execution of the reconfigurable array is disabled during this switching phase. The termination of the sequence is detected by polling the *context sequencer status* register.

#### c) Temporal Partitioning Context Sequencer

The *Temporal Partitioning Context Sequencer* is specifically designed for hardware virtualization through temporal partitioning (Temporal partitioning and its application to the Zippy architecture will be introduced in detail in Chapter 5). Temporal partitioning requires a very specific, cyclic execution sequence. Each context is executed for a single cycle, after which the next context is activated and executed. After the last context, the sequence is cyclically repeated. Due to the frequent context switches, a context sequencer with low switching overhead is mandatory for efficient temporal partitioning. Our design allows for switching contexts in a single cycle, i. e., without any timing overhead.

The Temporal Partitioning Context Sequencer requires only little configuration: the number of contexts (programmed with the *context sequencer temporal partitioning contexts* command) and the total number

of macro-cycles (*cycle count* command). The number of macro-cycles determine, how many times the sequencer repeats the basic schedule (activating each context once).

Figure 8(c) shows an example with 3 contexts that are repeated 128 times. Note, that this sequencer has no overhead for context switching in contrast to the virtualized execution sequencer (cf. Fig. 8(b)).

### 2.3.5 Multi-Context Architecture

An important design goal for Zippy is the support of execution modes that rely on fast, dynamic reconfiguration. To this end, we have designed Zippy as a multi-context architecture. Multi-context architectures concurrently store several configurations—denoted as contexts—on-chip and allow for switching rapidly between these configurations. We even speed-up this switching process in our architecture with dedicated context sequencers. While multi-context devices do not decrease the time for initial loading a configuration, they can dramatically reduce the *reconfiguration* time because they allow for fast switching between preloaded contexts. All contexts share the computation elements in the data-path, but each context has its own set of registers. This allows us to store and share intermediate results generated by a context until the next context invocation and eliminates the need to use memory structures to store these data and time-consuming context store and restore phases.

DeHon showed for FPGAs [DeH96b] that adding multiple contexts does not only reduce reconfiguration time, but also increases the computational density, while the impact on the chip-size is only moderate. A couple of fine-grained multi-context architectures have been developed, e. g., DPGA [DeH94], TMFPGA [TCJW97], WASMII [XA95], or CSRC [SV98]. Along with these architectures, application modes that exploit dynamic reconfiguration have been proposed, e. g., for TMFPGA [Tri98], DPGA [TCE<sup>+</sup>95], and DRLE a successor of the WASMII architecture [FFM<sup>+</sup>99, SUA<sup>+</sup>00].

There exists also a number of coarse-grained, multi-context devices. While many architectures bear similarities in the architecture of the reconfigurable cells and and partly also in the interconnect, the execution models for these architectures differ a lot. Like the Zippy architecture, all of these coarse-grained architectures include a CPU core to form a reconfigurable processor.

The MorphoSys architecture [SLL<sup>+</sup>00] specifically targets image and video processing. The architecture features a reconfigurable array of 8×8 cells that base on 16bit ALUs. The array follows a SIMD model of computation, that is, all cells in a row or column perform the same

operations. The architecture provides 32 configuration planes (contexts), which determine the function of the rows or columns.

The REMARC [MO99] architecture also provides an array of  $8 \times 8$  coarse-grained cells, but it uses a VLIW-like execution model. REMARC's cells feature a nano-processor, as small CPU core with dedicated instruction and data memories and registers. While not explicitly stated by the authors, REMARC also provides multi-context support. The local instruction memories can be thought of contexts which are activated by the Global Control Unit.

Chameleon Systems CS2000 [Cha00] architecture is tailored to wireless communication systems and features a coarse-grained 32-bit fabric coupled to a processor. CS2000 provides two contexts, whereas the background context can be loaded while the active context is in use.

PACT's XPP device [BMN<sup>+</sup>01] is a coarse-grained architecture, that implements a data-flow oriented execution model directly in hardware. Data transfers between the cells use a handshake protocol to ensure self-timed execution that satisfies data dependencies. The architecture uses a hierarchical configuration management that loads and activates configurations on demand.

### 2.3.6 Configuration Architecture

The configuration architecture of Zippy is similar to the configuration architecture of fine-grained FPGAs. The configurations are stored in the configuration memory (SRAM) and determine the function of the cells and the IO controllers, and configure the interconnection network. If the active configuration is switched, e. g., by the context sequencer, the configurations of all cells change at once.

The configuration bitstream is uploaded from the CPU to the RPU via the register interface. The RPU supports the download of full and partial configurations for any of the contexts. A partial reconfiguration can be used to make small changes to a configuration and prevents the overhead of loading a full configuration.

The size of a configuration depends on the architecture parameters, e. g., the size of the array, the bit-width, the size of the on-chip memory blocks, etc. Given an array instance with  $4 \times 4$  cells, a data-width of 24 bits, 2 horizontal north buses, 2 horizontal south buses, 2 vertical east buses and a 128x24bit ROM per row, the configuration size of a context is 1784 bytes.

parameter	description	typical value
DATAWIDTH	width of the data-path	24bit
FIFODEPTH	depth of the FIFOs	4096 words
N_CONTEXTS	number of contexts	8
N_ROWS	number of rows	4
N_COLS	number of columns (cells per row)	4
N_IOP	number of input and output ports	2
N_HBUSN	number of horizontal north buses	2
N_HBUSS	number of horizontal south buses	2
N_VBUSE	number of vertical east buses	2
N_MEMDEPTH	depth of memory blocks	128 words
N_CELLINPS	number of inputs to a cell	3
N_LOCALCON	number of local connections of a cell	8

**Tab. 3:** Reconfigurable Processing Unit: configurable architecture parameters

### 2.3.7 Parameters of the Reconfigurable Processing Unit

As it has been mentioned before, the Zippy architecture is widely parametrized. Table 3 summarizes the configurable architecture parameters of the RPU. Additionally to these parameters, the reconfigurable cell can also be easily modified and extended, for example, by adding new operators.

## 2.4 Summary

In this chapter we have introduced the Zippy architecture. The Zippy architecture is a simulation model of a novel reconfigurable processor which is specifically tailored to streaming multi-media applications in the embedded systems domain.

We propose hardware virtualization as a promising implementation technique for applications in embedded systems because it allows for keeping the reconfigurable structures small while still enabling the execution of large circuits. The Zippy architecture was specifically designed with hardware virtualization in mind. We have discussed the requirements for an efficient hardware virtualization implementation and we have shown how these requirements are met in the Zippy architecture. The new, architectural contributions to support the hardware virtualization execution model, are the configurable IO controllers, the context sequencers, and the input and output-register files.

We have shown that Zippy is not a single concrete architecture, but a widely parametrized architecture model, that allows for configuring the

architecture parameters for the CPU and the Reconfigurable Processing Unit (RPU).

Concurrently with the hardware architecture, a set of hardware and software design-tools, a performance evaluation framework, and a programming model have been co-designed. These design-tools also support the same set of parameters as the Zippy hardware architecture.

This configurability of hardware and software allows us to use Zippy as an experimentation framework for coarse-grained reconfigurable processors. To the best of our knowledge, Zippy is the only coarse-grained reconfigurable architecture that provides a parametrized hardware architecture as well as parametrized design-tools.





# 3

## Tool-Flow

This chapter presents the application implementation tool-flow for the Zippy architecture.

Section 3.1 introduces two major approaches to a tool-flow for coarse-grained reconfigurable processors: a high-level compilation, and a circuit-centric tool-flow.

Section 3.2 presents the application execution model and an overview of Zippy's circuit-centric tool-flow.

Section 3.3 presents the hardware tool-flow in detail. We present the application specification formalism and introduce the routing architecture modelling. The core of the hardware tool-flow is an iterative placement and routing process. We use a placer that uses a stochastic search method for iteratively improving the placement. The router uses an adaption of the Pathfinder [EMHB95] routing algorithm. The result of the hardware tool-flow are two representations of a configuration for the Reconfigurable Processing Unit (RPU): a structured configuration representation in VHDL which is used for simulation purposes, and a configuration bitstream that is used to program the configuration memory.

The software-toolflow is introduced in Section 3.4. We introduce the extensions that have been added to the CPU simulator to support a co-processor interface. The software-toolflow bases on a conventional C compilation tool-chain (GNU C compiler) which is augmented with additional processing steps.

## 3.1 Tool-Flows for Coarse-Grained Reconfigurable Processors

In this section we discuss the two major approaches to an application implementation tool-flow for a coarse-grained reconfigurable processor. We introduce the high-level compilation approach and the circuit-centric approach, which is used in the Zippy architecture.

An implementation tool-flow for a coarse-grained reconfigurable processor transforms an application specification into an implementation where parts of the application run on the CPU core, while the remaining parts run on the RPU.

The challenge for a tool-flow is to provide a good compromise between abstraction and ease of specification on the one hand, and efficiency of the implementation on the other hand. A well-designed tool-flow enables an application specification that fits the application domain, while it uses the computing resources efficiently.

There are two major approaches to a tool-flow for coarse-grained reconfigurable processors:

- the *compilation-centric approach* which tries to derive an implementation directly from a specification in a high-level language (for example C code), and
- the *circuit-centric approach*, that requires the developer to explicitly specify a circuit and a software application which communicate by a defined hardware-software interface.

### High-Level Compilation Tool-Flow

The *compilation-centric* approach bases on advanced, high-level compilation technology. High-level compilation for *fine-grained* reconfigurable logic has been studied since the emergence of Field-Programmable Gate-Arrays (FPGAs) [PL91, GG97, CHW00]. But compiling for FPGAs differs significantly from compiling for coarse-grained reconfigurable architectures. Since fine-grained architectures can implement virtually any circuit, the compiler has not only to extract the data-flow graph from the application and then wire fixed arithmetic units, but the compiler must also perform circuit generation, i. e., data-path, architecture, and control-path synthesis.

For coarse-grained architectures, the cells of the reconfigurable array already predetermine the set of the arithmetic units that are targeted by the compilation process. Essentially, the compiler performs parallel compilation for a multi-processor array in order to generate a set of configurations for a coarse-grained array. The result of the compilation process is comparable to a program for a Very Long Instruction Word (VLIW) pro-

cessor. Each VLIW instruction can be compared to a configuration for the reconfigurable array. The sub-instructions within the VLIW instruction define the function of each cell and the routing. Evidently, the peculiarities of the data-transfers and the data-storage in the reconfigurable array are different from the shared register-file found in VLIW processors.

Usually, only the runtime-intensive kernels (inner loops) of an application are mapped to hardware. The compiler identifies the kernels, performs hardware-software partitioning to split the hardware and software parts of the application, and finally generates an optimized hardware implementation for the kernels. Recently, a number of researchers started investigating the use of high-level compilation for coarse-grained reconfigurable architectures [CW02, LKD03, VNK<sup>+</sup>03].

The sole reliance on compilation makes this approach well suited for general-purpose use, i. e., when nothing is known about the characteristics of the application. The disadvantage of the compilation-centric tool-flow is that the performance of the resulting implementation is generally suboptimal. The high performance of hardware-accelerated computation can be largely attributed to parallel execution, pipelining and the use of custom data-paths [CH02]. Many application domains show regular communication and computation patterns, but it is difficult for a compiler to detect and exploit this domain knowledge when compiling from arbitrary high-level code.

So far, most high-level compilation tool-flows support only a subset of the specification language. Hence, there are also practical limitations of this approach and at least parts of the specification's code have to be adapted.

### **Circuit-Centric Tool-Flow**

In the *circuit centric* tool-flow, the developer explicitly specifies the application as a software part for execution on the CPU and a hardware circuit for execution on the reconfigurable array. Like in the compilation-centric approach, the initial specification is given in a high-level language. The designer identifies the kernels of the application and performs a manual hardware-software partitioning process.

The designer is required to generate a functional equivalent hardware implementation (circuit) of the kernel. This circuit can be specified in any formalism, e. g., as a netlist or in a hardware description language, and is then implemented with conventional Computer-Aided Design (CAD) tools. The designer needs to replace the kernel in the initial application with calls to the kernel's hardware implementation by using the software-hardware communication primitives. Typically, a reconfigurable processor supports hardware-software communication patterns that are tailored

to the characteristics of the application domain, e. g., FIFO communication channels for streaming data processing, or shared memory areas (frame-buffers) for image processing [SLL<sup>+</sup>00].

A circuit-centric tool-flow is used for many reconfigurable computing systems that attach a reconfigurable coprocessor to an external IO bus of a host computer. It has been also successfully used for tightly integrated, fine-grained and coarse-grained reconfigurable processor architectures [MO99, SLL<sup>+</sup>00].

The prime advantage of this approach is that it allows for using conventional CAD tools for circuit implementation and for reusing optimized circuits. Reusing existing circuits is attractive because there exist optimized implementations for algorithms from many application domains, e. g., digital filters. These implementations heavily rely on parallel execution, algorithmic and arithmetic optimization and pipelining to achieve high-throughput. It is unlikely, that a general-purpose compilation-based approach will be able to match the performance of such an optimized implementation.

The disadvantage of this approach is that the designer must provide a hardware implementation for the kernel. There is no support for automatically generating a functionally correct but presumably suboptimal kernel implementation.

A noteworthy difference between the circuit-centric and the compilation-centric approach is that the circuit-centric approach does not inherently include the notion of a reconfiguration or “instruction-sequencing”. Most models of circuits assume that the *entire* circuit is continuously executing in parallel, while the compilation-centric view suggests an execution model that provides both, parallel and sequential execution. That is, an application is defined as a sequence of VLIW-like instructions, but within each instruction the execution is parallel. Instructions thus represent configurations and sequencing between instructions represents a reconfiguration process. The compilation-centric execution model hence naturally treats the reconfigurable array as a dynamic resource, which can also execute applications that do not fit into a single configuration.

The circuit-centric approach can also be extended to support dynamic reconfiguration of the array, if the circuit execution model is extended with *temporal-partitioning*. Temporal partitioning is a *hardware virtualization* technique, that allows for executing circuits of arbitrary size. We discuss hardware virtualization, temporal partitioning and its application to the Zippy architecture in Chapter 5.

## 3.2 The Zippy Tool-Flow

This section presents an overview of the Zippy tool-flow and briefly introduces the application execution model. The hardware tool-flow will be treated in detail in Section 3.3, the software tool-flow will be discussed in Section 3.4.

The main application domain for Zippy is streaming multi-media signal-processing. Applications from this domain continuously process data from a buffer and usually require only infrequent communication between the CPU and the RPU. The Zippy architecture supports the communication and computation demands of these applications by providing FIFO buffers for intermediate storage of data-streams, low overhead context sequencers for hardware execution without CPU intervention, and a coarse-grained reconfigurable array that supports important arithmetic operations.

We use a *circuit-centric* implementation tool-flow for the Zippy architecture, because it matches the intended application domain well. Most kernels in digital-signal processing are rather small and there exist numerous optimized circuits. Hence, a manual hardware-software partitioning process to extract the rather small kernels is appropriate.

For the purpose of presenting the hardware implementation tool-flow, we assume that circuits fit as a whole onto the reconfigurable array. We will give up this constraint with the introduction of *hardware virtualization* in Chapter 5.

### Application Execution Model

The CPU core coordinates the execution of the whole reconfigurable processor. The prime responsibility of the CPU is to execute the software computation tasks which are integral parts of the application itself. Apart from this computation tasks, the CPU is also responsible for a variety of other control and communication tasks:

- The CPU initiates all data IO operations (to/from main memory and between CPU and RPU) because the RPU is a pure communication slave to the CPU and has no direct memory interface.
- The CPU manages the configuration of the RPU. This includes loading the configurations to the configuration memory and programming the context sequencers.
- The CPU handles the execution of the RPU, i. e., it transfers data between CPU and the FIFO buffers, triggers the context sequencers and polls the context sequencer status to detect the termination of an execution cycle.

The function of the reconfigurable array is determined by the configuration data. The configuration also specifies the settings for the IO controllers and the context sequencer (see Section 2.3). The IO controllers manage the access of the reconfigurable array to the FIFOs. The context sequencer defines the length and—in the case of multiple configurations—the sequence of context (configuration) executions. As soon as the CPU triggers the context sequencer, the RPU executes the respective contexts autonomously until completion without further CPU intervention.

The configuration for the RPU is either directly included in the software executable for the CPU, or loaded from the main memory at application startup. The hardware and software parts communicate via the CPU's coprocessor port that is attached to the RPU's register interface. A communication and configuration library provides the software with a programming interface to the functions exposed by the register interface. After a configuration has been downloaded, the CPU transfers the initial input data to the FIFOs and programs the context sequencers appropriately. During the runtime of the application the CPU is responsible for polling the RPU for the termination of the context sequencer, for transferring input data and results between the CPU and the RPU via the FIFOs, and for triggering new execution cycles of the context sequencer.

An application thus consists of three main components, which need to be generated by the design-tools during the implementation tool-flow:

1. a *software executable* for the CPU core (uses a communication library for communicating with the RPU),
2. *configuration data* (also called configuration bitstream) for the RPU that implements the hardware part of the application, and
3. a *configuration for the context sequencer* of the RPUs.

## Outline of the Zippy Tool-Flow

Figure 9 presents a graphical outline of the complete implementation tool-flow. The starting point of the tool-flow is an application specification as C source code. In a manual codesign process the application is decomposed into a hardware and a software part and a context sequencer configuration. The software part is specified as C source code and uses a communication and reconfiguration library for accessing the RPU. The hardware part is a circuit that uses the resources of the reconfigurable array and is specified as a netlist. In the subsequent hardware implementation tool-flow, an automated placement and routing process determines a valid implementation of the circuit on the reconfigurable array. This implementation is transformed into configuration data that is downloaded to the RPU. In a final step, the software part of the application is compiled into an executable with a C compiler.

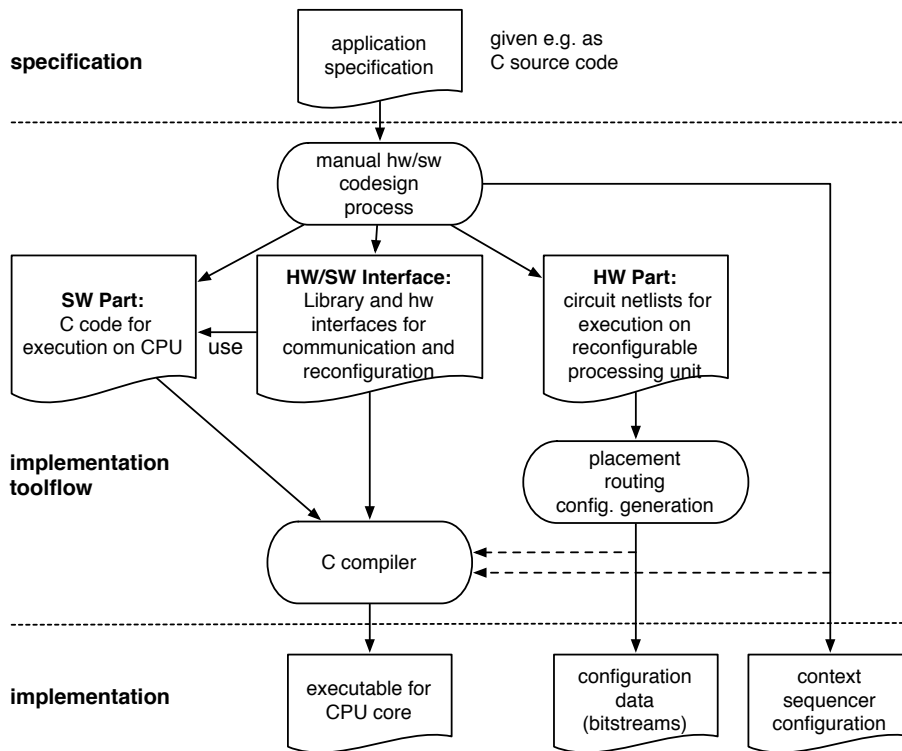


Fig. 9: Outline of the application implementation tool-flow.

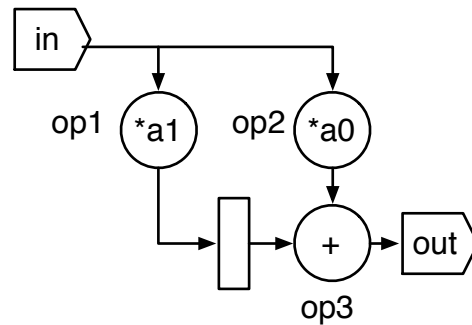
### 3.3 Hardware Tool-Flow

This section presents the hardware tool-flow which transforms the specification of a circuit to an implementation on the reconfigurable array. The circuit is defined as a netlist of operators that can be implemented by the cells of the reconfigurable array. The circuit's netlist is processed by a chain of design-tools that determine a feasible placement and routing. The routing algorithm bases on the Pathfinder algorithm [EMHB95]. The design-tools use a parametrized model of the reconfigurable CPU's routing architecture.

The result of the place and route process is a hierarchical configuration description represented as a VHDL data-structure. This data-structure defines the settings for all programmable switches and memories in the reconfigurable architecture. In a last step, a flattened, binary representation of the configuration is generated. This representation is loaded to the context memories to configure the function of the RPU.

#### 3.3.1 Hardware specification

The hardware parts of applications for the Zippy architecture are specified as netlists of coarse-grained operators and registers. We have chosen this



**Fig. 10:** Netlist specification (signal-flow diagram) of a first order FIR filter.

specification style since application specification with a netlist of high-level, coarse-grained computing elements is familiar to DSP algorithm developers. In this application domain, the netlists are typically referred to as *signal-flow diagrams*. A similar block-based specification approach is also used by a number CAD tools for developing signal processing algorithms, such as Ptolemy [Lee01], Simulink [Sim06], or the Xilinx System Generator [HMSS01]. These tools could be extended to support the development of applications for the Zippy architecture.

A circuit's netlist is a directed (possibly cyclic) graph composed of the following elements: *primary inputs*, *primary outputs*, *coarse-grained cells*, and *registers*. The netlist elements are connected by nets, which are multi-terminal connections. Each net a) connects a primary input to a cell, b) connects an output of a cell to a primary output, or c) connects the output of a cell to the inputs of other cells. If the netlist is cyclic, each cycle must include at least one register to prevent combinational feedback loops.

Figure 10 presents an informal graphical representation of the netlist of a first order FIR filter. This circuit will serve as an example throughout this section to illustrate the tool-flow from specification to implementation. These signal-flow diagram representations of netlists are well suited to provide humans with an intuitive overview of an application's structure.

To provide the placement and routing tools in the hardware implementation tool-flow with a well-defined netlist specification, a textual netlist format named *Zippy Netlist Format (ZNF)* has been defined.

Listing 3.1 shows the complete ZNF description for the example circuit from Fig. 10. The ZNF file consists of three sections that define the primary inputs and outputs, the cells, and the nets of the circuit. Lines 5 and 6 define a primary input named *in* and a primary output named *out*. Both ports have a fixed placement (*p.in0*, *p.out0*). Lines 10–12 declare the cells of the netlist. Each cell has a name, type, a placement constraint, and a list of attributes. All cells in this circuit are of type *std* and do not have any



---

```

1 znf 0.1 fir    # first order FIR filter
2
3 # define input and output ports
4 # col2=name, col3=placement (fixed placement)
5 i in  p.in0:f
6 o out p.out0:f
7
8 # define cells
9 # col2=name, col4=placement, col5=function and IO configuration
10 c op1  std * f=alu_multlo , i.0=noreg , i.1=const , const=32,o.0=noreg
11 c op2  std * f=alu_multlo , i.0=noreg , i.1=const , const=16,o.0=noreg
12 c op3  std * f=alu_add , i.0=noreg , i.1=reg , o.0=noreg
13
14 # define nets
15 # col2=netname, col3=source, col4=list of sinks
16 n nin      in      op1.i.0,op2.i.0
17 n n1      op1.o.0  op3.i.1
18 n n2      op2.o.0  op3.i.0
19 n n3      op3.o.0  out

```

---

**Lst. 3.1:** Netlist definition for the first order FIR example in Zippy Netlist Format (ZNF) format.

placement constraint (indicated by a \*). The cell's attributes are used to set the function of the cell and to configure its inputs and outputs. The list of attributes can be extended arbitrarily and is handed to the placement and routing tools for interpretation. Finally, lines 16–19 define the nets in the circuit. Each net has a *source* (primary input, or the output of a cell) and one or many *sinks* (primary output, input of a cell).

ZNF supports three types of placement constraints: *fixed placement* (which ties cells or primary inputs/outputs to a fixed location), *initial placement* (used to pass an initial placement to the placer), or *free placement* (where an optimal placement is determined by the placer).

The primary input and output ports connect the circuit to the IO controllers (see Sec. 2.3.1), for example, input port *INP0* at location *p.in0* is connected to the output of *FIFO0*.

### 3.3.2 Architecture modelling

Since the Zippy architecture is not a single concrete architecture but a parametrized family of architectures, the hardware implementation tools are also parametrized to support any instance of a Zippy architecture. To permit the use of the same set of tools for many different architectures the design-tools rely only on an abstract graph model of the reconfigurable unit's routing-architecture, the *routing-resource graph*. Zippy's routing-resource graph is generated on-the-fly during the implementation phase

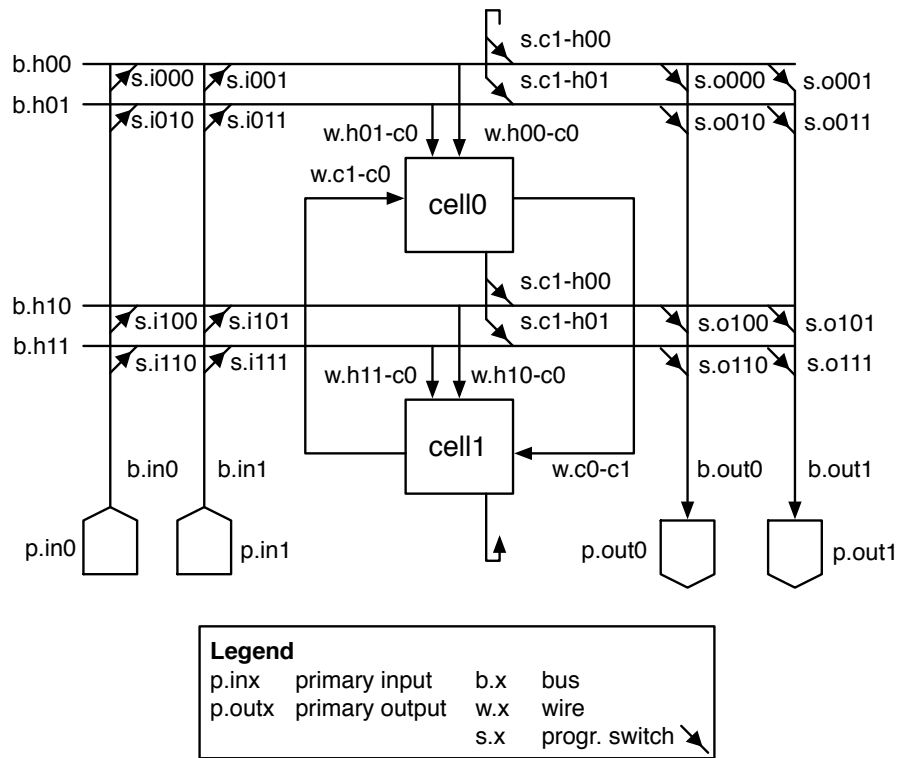


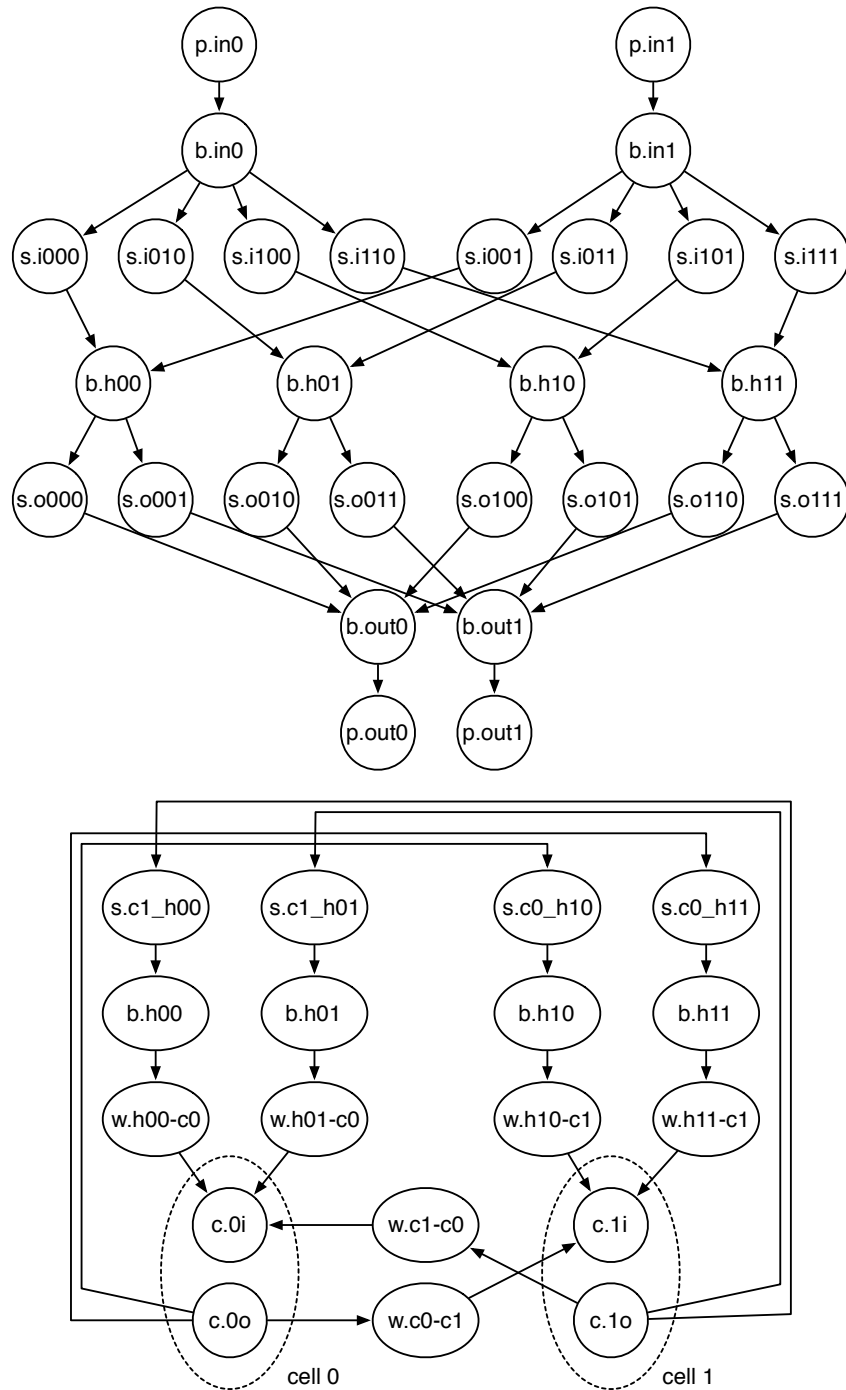
Fig. 11: Simplified reconfigurable array architecture

according to the architecture parameters. This approach is also used in the parametrized VPR design tools [BRM99].

Since the routing-resource graph for a real Zippy architecture is too complex for illustration, we use a simplified architecture for explaining the hardware tool-flow. Figure 11 introduces the model of a reconfigurable array that comprises the same resource types as the Zippy architecture (input and output ports, buses, wires, programmable switches, and cells), but is reduced to two cells and a few buses.

The routing-resource graph model is derived from the architecture by identifying interconnection resources of the architecture with routing-resource nodes in the routing-resource graph. Each interconnection resource is modelled as a node in the routing-resource graph. The connections between routing resources are modeled as directed edges in the routing-resource graph.

Figure 12 shows the routing-resource graph for the architecture of Fig. 11. To enhance the readability of the figure, Fig. 12 is split into two parts that are actually connected via the nodes representing the horizontal buses (b.h00–b.h11). The upper part of the figure illustrates the bus interconnect and the primary inputs and outputs. The lower part of the figure illustrates the connections between the cells and shows how the



**Fig. 12:** Routing-resource graph for the simplified architecture from Fig. 11

cell outputs drive the horizontal buses via switches.

The cells themselves are not actual routing-resources, because the cell input and output are not connected. Hence, the cell's inputs and outputs are represented by distinct cell input nodes (c.xi) and cell output nodes (c.xo). For the sake of simplicity, the figure shows only a single input node per cell, a complete model would include a dedicated cell input node for every cell input.

The routing-resource graph allows us to formulate *placement and routing* as *graph problems*. Finding a placement for a circuit corresponds to finding a mapping from the elements of a circuit's netlist to the primary input/output and cell nodes in the routing-resource graph. After placement, all nets of the circuit need to be routed. Finding a feasible route for a net between a primary input/output and a cell or between cells corresponds to finding a directed path in this routing-resource graph. Each routing resource can be used for only one single net, thus finding a feasible routing for all nets in a circuit is equivalent with finding a set of paths in the routing-architecture graph, such that each routing resource node is used only once, i. e., all paths are disjoint.

### 3.3.3 Placement

The *placer* assigns the components of the circuit's netlist (primary inputs/outputs and cells) to concrete locations, denoted as *sites*, of the reconfigurable architecture. The netlist, the circuit's placement, and the routing-resource graph are the inputs of the router. Since the placement determines which routing-resources can be used, the placement and routing processes are not independent but influence each other. Hence, placement and routing are executed as iterative processes and are repeated until a solution which satisfies all implementation constraints has been reached.

#### Initial placement

An initial placement is needed to create a starting point for the subsequent iterative improvement process. Two algorithms for the creating an initial placement have been devised in this work: a *random placement algorithm* and a *heuristic placement algorithm*. The random placer starts with placing all cells and primary inputs/outputs with fixed or initial placement constraints. Then, the remaining netlist elements are placed to a random unused site, see Algorithm 1.

The *heuristic placement algorithm* aims at placing cells close to each other if they are connected in the netlist. After placing all cells with placement constraints, the algorithm starts exploring the netlist topology starting from nodes that have been placed, but whose successors or predecessors

---

**Algorithm 1** Random placer for creating an initial placement
 

---

```

1: procedure RANDOMPLACEMENT(netlist)
2:   place all cells with fixed or initial placement constraints
3:   place all primary inputs/outputs with fixed or initial placement
     constraints
4:   while unplaced cells (or primary input/output) exist do
5:     place cell (or prim. input/output) to a random unused site
6:   end while
7: end procedure

```

---



---

**Algorithm 2** Heuristic placer for creating an initial placement
 

---

```

1: procedure HEURISTICPLACEMENT(netlist)
2:   place primary inputs
3:   place cells with fixed or initial placement constraints
4:   while unplaced cells exist do
5:     for each cell c that is placed but has unplaced successors or
       predecessors do
6:       for all unplaced sinks in sinks(c) do
7:         if |sinks(op)| > 2 then
8:           place sinks in the next adjacent row
9:         else |sinks(op)| ≤ 2
10:          place sink to preferred neighbor sites
11:        end if
12:       for all unplaced sources in sources(c) do
13:         place source to preferred neighbor site
14:       end for
15:     end for
16:   end while
17:   place primary outputs
18: end procedure

```

---

have not been placed yet. The heuristic favors the use of bus interconnect for connecting nets with many sinks (high fan-out). For nets with a fan-out of more than 2, the sinks are placed to the next available rows and are thus accessible via horizontal buses. Low fan-out nets are preferably routed via local connections and are placed to a *preferred neighbor* site. The placer treats all cells that can be reached with a direct connection as preferred neighbors. If there are no unused sites in the list of preferred neighbors, the cell is placed to a random unused site.

While random placement leads to routable initial placements of acceptable quality for FPGA architectures [BRM99], choosing a random initial placement on the Zippy architecture frequently leads to an unroutable placement. This can be largely attributed to the scarceness of routing resource on the Zippy architecture in comparison to routing-resource rich FPGA architectures. Experiments have shown, that despite of its simplicity, the heuristic placer that has been developed for the Zippy architecture creates significantly better initial placements. This helps the subsequent iterative placement and routing process to arrive faster at a feasible implementation.

### Iterative Placement

To find a routable implementation of the circuit, the initial placement is iteratively improved with a *stochastic search procedure*. Since simulated-annealing-based search algorithms have been successfully used for cell placement in FPGAs and Application-Specific Integrated Circuits (ASICs), we also rely on a *simulated-annealing-based iterative placer*.

Algorithm 3 outlines the pseudo code of the algorithm, which is based on the generic simulated-annealing-based placer in [BRM99]. The placer takes a reference placement  $P$  and randomly exchanges the placement of two nodes. The quality of the new placement  $P'$  is denoted as *cost* and is determined by routing the circuit with the new placement.

Whenever the cost of the new placement is smaller than the cost of the reference placement, the new placement becomes the new reference placement.

To reduce the greediness of this local search strategy, and thus to allow the algorithm to escape from local optima, the algorithm also accepts a deterioration in solution quality with a certain probability. This acceptance probability is controlled by the amount of the deterioration ( $\Delta C$ ) and a temperature parameter  $T$ . The probability of accepting an inferior solution decreases with lower temperature and higher cost difference  $\Delta C$ . As  $T$  is decreased (according to a function denoted as *annealing schedule*), the stochastic search process becomes increasingly greedy over time. For our iterative placer we use a simple annealing schedule that geometrically

**Algorithm 3** Simulated-Annealing-Based Iterative Placer

---

```

1: procedure PLACESIMULATEDANNEALING( $P, T, \lambda, \text{maxOuter}, \text{maxInner}$ )
2:   for  $\text{outer} \leftarrow 1..\text{maxOuter}$  do
3:     for  $\text{inner} \leftarrow 1..\text{maxInner}$  do
4:        $P' \leftarrow \text{randomMove}(P)$ 
5:        $\Delta C \leftarrow \text{cost}(P') - \text{cost}(P)$   $\triangleright$  cost is determined by the router
6:       return if implementation constraints satisfied
7:        $r \leftarrow \text{random}(0, 1)$ 
8:       if  $r < e^{-\Delta C/T}$  then
9:          $P \leftarrow P'$ 
10:      end if
11:    end for
12:     $T \leftarrow \lambda \cdot T$ 
13:  end for
14: end procedure

```

---

decreases  $T$  by a factor  $\lambda$  for every iteration of the outer loop.

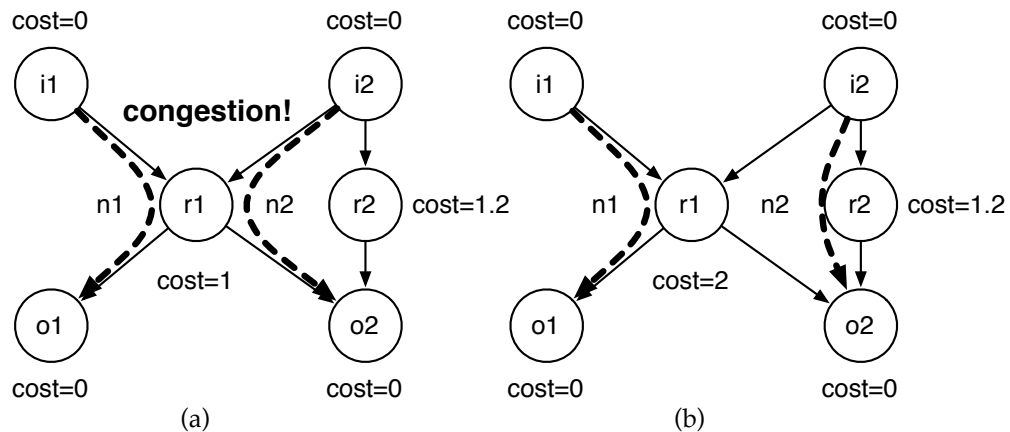
Note, that in contrast to [BRM99], we do not adapt the size of a *randomMove* step. Zippy arrays have typically a rather small number of cells, for example  $4 \times 4$  or  $8 \times 8$  cells. We found that starting with large step sizes (i. e., swapping multiple cells in one step) destroys much of the structure of the initial placement, which has been created by the heuristic placer. We have found that using a constant step size and swapping the placement of two cells in each move leads to good results.

### 3.3.4 Routing

For routing we use an adaption of the Pathfinder pure congestion-based algorithm [EMHB95]<sup>1</sup>. The basic idea of the Pathfinder routing algorithm is that the algorithm does not try to avoid congestion from the beginning, but tries to resolve congestion by repeated routing and rip-up cycles. Each routing resource in the routing-resource graph is assigned a cost value which expresses the cost of using that resource. A net is routed between two nodes in the routing-resource graph by finding a minimum cost route between the source and sink nodes. If congestion occurs after all nets have been routed, i. e., several nets use the same routing resource, a new routing iteration is started. To avoid future congestion, the cost of all nodes that have been overused is increased. In the next routing iteration all nets are ripped-up one after the other and rerouted using

---

<sup>1</sup>Since Zippy is an architecture model and has not been implemented as a chip, we do not have delay information. Hence we use the basic algorithm that optimizes for routability by reducing congestion instead of the delay driven extension to the algorithm.



**Fig. 13:** Congestion control in the Pathfinder algorithm. The nodes are annotated with their total cost  $c_n$ .

the updated resource costs. Hence, the Pathfinder algorithm resolves congestion by adapting the cost of routing resources based on historic and present congestion information.

Figure 13 sketches the concept of the algorithm. Two nets  $n_1 = (i_1, o_1)$  and  $n_2 = (i_2, o_2)$  have to be routed. In the initial situation (see Fig. 13(a)) both nets are routed via routing resource  $r_1$ , which results in congestion of  $r_1$ . Consequently, the cost of  $r_1$  is increased. In the next routing iteration, the congestion is resolved:  $n_1$  is routed via  $r_1$  while  $n_2$  is routed via  $r_2$  (see Fig. 13(b)).

The Pathfinder algorithm does not only include historical cost information, but assigns each node three cost values that are aggregated into a total cost value  $c_n$ :

$$c_n = (b_n + h_n) \cdot p_n$$

$b_n$  is the base cost of a resource. It can be used to favor the usage of a specific resource type, e. g., local connections or fast connections (for delay driven routing). Since we do not have delay information, we use  $b_n = 1$ .  $h_n$  is the historic congestion information. It is increased after every routing iteration if resource  $n$  is congested. The present cost  $p_n$  expresses the current congestion of resource  $n$ . If  $n$  is not congested:  $p_n = 1$ .

Algorithm 4 outlines the pseudo-code of the Pathfinder algorithm. The minimum cost routing step in line 8 is performed with Dijkstra's shortest-path algorithm.

### Application to the Zippy architecture

We have adapted the Pathfinder routing algorithm to the Zippy architecture. In an iterative placement and routing process, the router tries to find



---

**Algorithm 4** Pathfinder Pure Congestion-Based Routing Algorithm [EMHB95]

---

```

1: procedure ROUTE(circuit, routinggraph)
2:   Initialize: reset routing, resource usage, and cost values
3:   repeat
4:     while exists congestion (overused resources) do
5:       start a new routing iteration
6:       for all nets  $n_i$  do
7:         rip-up  $n_i$  and update affected  $p_n$  cost
8:         establish minimum cost route  $r_i$  for  $n_i$  in routing graph
9:         update affected  $p_n$  cost
10:      end for
11:      compute congestion
12:      update historical cost  $h_n$ 
13:    end while
14:  until routed
15: end procedure

```

---

a feasible routing for the placement proposed by the placer. If the router finds a feasible routing within a bounded number of routing iterations, the placement and routing process is successfully completed.

If the router cannot find a feasible routing, the router computes a cost value for the current implementation, which is used to direct the simulated annealing process in the placer. There are two reasons, why no feasible routing can be found for a given placement:

- a) the placement is unroutable because of congestion, or
- b) the placement is generally unroutable because there is no path in the routing-resource graph for certain sources and sinks of nets

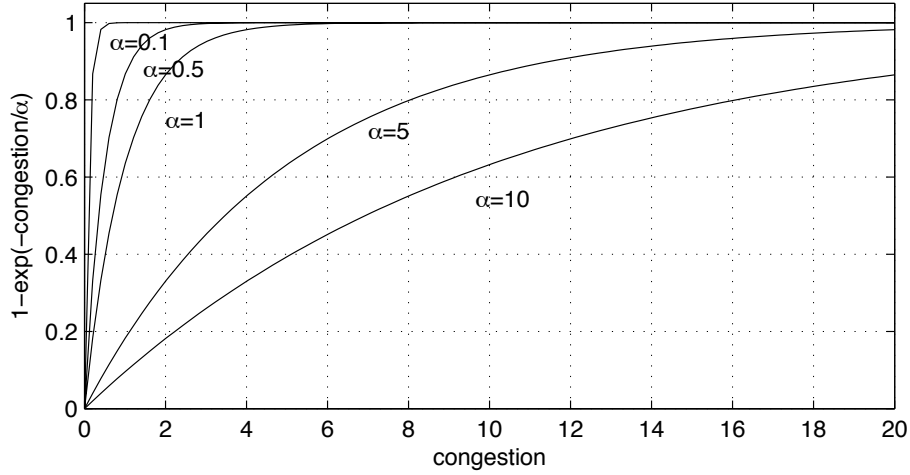
In case a) the router reports the amount of overusage of the routing-resources, in case b) the router reports the number of unroutable nets to the placer.

Since coarse-grained architectures like Zippy have scarcer routing resources, case b) is much more likely as with routing-resource rich fine-grained FPGA architectures. To increase routability we configure each unused cell as feed-through cell.

For routing circuits on the Zippy architecture, we have chosen to logarithmically increase  $p_n$  with the number of nets (*users*) that use routing resource  $n$ :

$$p_n = \begin{cases} 1 & \text{if } users \leq 1 \\ 1 + \beta_p \log(users - 1) & \text{if } users > 1 \end{cases}$$

At the end of each routing iteration  $h_n$  is adapted with the following



**Fig. 14:** Congestion cost in the implementation cost function

function:

$$h_n = \begin{cases} h_n + \beta_h & \text{if } p_n > 1 \\ h_n & \text{otherwise} \end{cases}$$

For providing the simulated annealing process in the iterative placer (see Alg. 3) with a cost metric we use the following cost function to evaluate the quality of a routing:

$$\begin{aligned} \text{cost} &= \text{cost\_unroutability} + \text{cost\_congestion} \\ &= \#\text{unroutable\_nets} + \left(1 - \exp\left(-\frac{\text{total\_congestion}}{\alpha}\right)\right) \end{aligned}$$

$\#\text{unroutable\_nets}$  is the number of nets that could not be routed by the router due to unreachability. The congestion of a routing-resource is measured by the present cost  $p_n$ . The  $\text{total\_congestion}$  parameter sums up all  $p_n$  and is a metric for the total overusage caused by a given placement and routing.  $\alpha$  is a sensitivity parameter that specifies how strong resource overusage is weighted. Since  $\text{cost\_congestion} < 1$  the cost metric always favors implementations with fewer unrouted nets. If two implementations have the same number of unrouted nets, the implementation with smaller congestion has smaller cost. Figure 14 illustrates the congestion cost function in subject to parameter  $\alpha$ . For a feasible implementation without congestion the cost value reaches it's optimal value of 0.

The optimal settings for the parameters of the placer and the router are architecture and also problem dependent and have not been extensively studied. Initial experiments have shown, that the parameter settings given in Table 4 lead to good results.

<b>Placer</b>			
inner iterations	$maxInner$		20
outer iterations	$maxOuter$		100
initial temperature	$T$		0.05
temperature update	$\lambda$		0.95
<b>Router</b>			
overusage sensitivity	$\beta_p$		0.5
historical cost update	$\beta_h$		0.2
congestion sensitivity	$\alpha$		0.5

**Tab. 4:** Typical parameter settings for the placement and routing algorithm (taken from the case-study on the temporal-partitioned implementation of an ADPCM decoder on a  $7 \times 7$  Zippy array, see Sec. 6.2)

### 3.3.5 Configuration Generation

The *configuration* of the reconfigurable array is determined by the results of the placement and routing process. The placement determines the locations to which each netlist element is mapped and the settings for the programmable switches, multiplexers, and cell input/output blocks.

The configuration for Zippy is specified as a *hierarchical VHDL data-structure* which can be directly used by the VHDL model of the Zippy architecture. We have chosen VHDL as the native configuration format because it is human readable and allows for manual modification. Since VHDL does not support automated conversion of VHDL data structures to a flattened, binary format, Zippy's VHDL model includes methods for converting between the structured VHDL configuration data and a compact binary representation. This binary configuration data (frequently called *configuration bitstream*) is the data that is finally stored in the configuration memory.

Listing 3.2 presents the complete VHDL configuration data-structure that has been generated by placing and routing the netlist of the FIR filter example (see Fig. 10) on a  $2 \times 2$  cell Zippy array. The main part of the data-structure is a two dimensional array (*gridConf*) with an element for each cell. Each array element specifies the configuration of the processing part of a cell (*procConf*) and the routing connections to the buses (*routConf*). The data-structure specifies also the configuration for the input and output buses, and the IO controllers.

Figure 15 presents the result of the implementation of the FIR filter example on a  $2 \times 2$  cell Zippy architecture.

---

```

1  -- c_0_0 op3
2  cfg.gridConf(0)(0).procConf.AluOpXS := alu_add;
3  -- i.0 (no register, read from SE neighbor)
4  cfg.gridConf(0)(0).procConf.OpMuxS(0) := I_NOREG;
5  cfg.gridConf(0)(0).routConf.i(0).LocalxE(LOCAL_SE) := '1';
6  -- i.1 (register, read rom S neighbor)
7  cfg.gridConf(0)(0).procConf.OpMuxS(1) := I_REG_CTX_THIS;
8  cfg.gridConf(0)(0).routConf.i(1).LocalxE(LOCAL_S) := '1';
9  -- o.0 (no register, drive output to b.h10)
10 cfg.gridConf(0)(0).procConf.OutMuxS := O_NOREG;
11 cfg.gridConf(0)(0).routConf.o.HBusNxE(0) := '1';
12
13 -- c_1_0 op1
14 cfg.gridConf(1)(0).procConf.AluOpXS := alu_multlo;
15 -- i.0 (no register, read from bus b.h11)
16 cfg.gridConf(1)(0).procConf.OpMuxS(0) := I_NOREG;
17 cfg.gridConf(1)(0).routConf.i(0).HBusNxE(1) := '1';
18 -- i.1 (use constant input)
19 cfg.gridConf(1)(0).procConf.OpMuxS(1) := I_CONST;
20 cfg.gridConf(1)(0).procConf.ConstOpxD := i2cfgconst(32);
21 -- o.0
22 cfg.gridConf(1)(0).procConf.OutMuxS := O_NOREG;
23
24 -- c_1_1 op2
25 cfg.gridConf(1)(1).procConf.AluOpXS := alu_multlo;
26 -- i.0 (no register, read from bus b.h11)
27 cfg.gridConf(1)(1).procConf.OpMuxS(0) := I_NOREG;
28 cfg.gridConf(1)(1).routConf.i(0).HBusNxE(1) := '1';
29 -- i.1 (use constant input)
30 cfg.gridConf(1)(1).procConf.OpMuxS(1) := I_CONST;
31 cfg.gridConf(1)(1).procConf.ConstOpxD := i2cfgconst(16);
32 -- o.0
33 cfg.gridConf(1)(1).procConf.OutMuxS := O_NOREG;
34
35 -- IO buses: route INP0 to b.h11, and b.h10 to OUTP0
36 cfg.inputDriverConf(0)(1)(1) := '1';
37 cfg.outputDriverConf(0)(1)(0) := '1';
38
39 -- IO port controller (always active)
40 cfg.inportConf(0).LUT4FuncxD := CFG_IOPORT_ON;
41 cfg.outportConf(0).LUT4FuncxD := CFG_IOPORT_ON;

```

---

**Lst. 3.2:** Configuration for the first order FIR filter as hierarchical VHDL data-structure

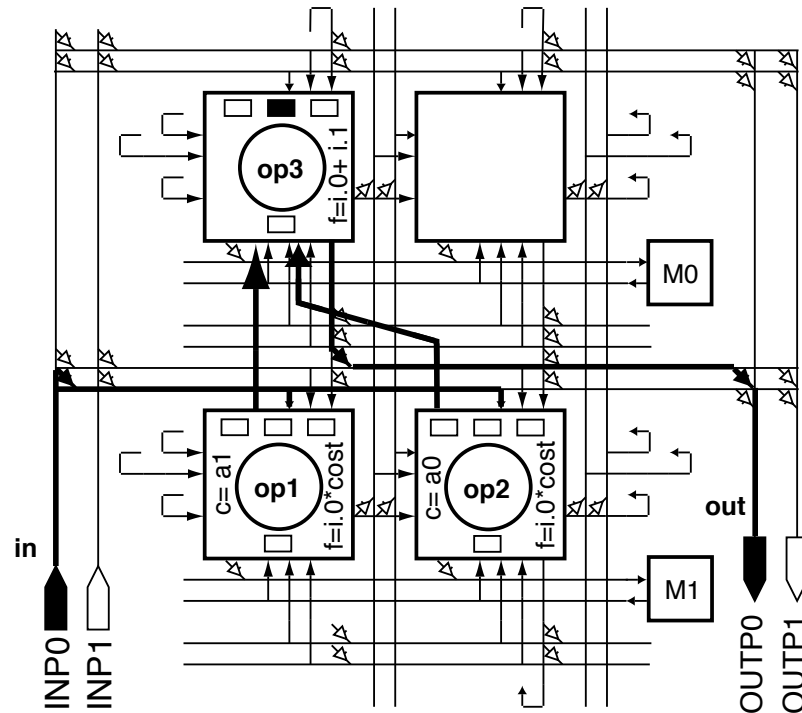


Fig. 15: Implementation of the first order FIR filter on a 2x2 reconfigurable array

### 3.4 Software Tool-Flow

This section introduces the software tool-flow for the Zippy architecture. The result of the software tool-flow is the executable for the reconfigurable processor's CPU core. The executable incorporates the software parts of the application, the configuration for the context sequencers, and the configuration bitstreams that have been created by the preceding hardware tool-flow. An overview of the complete application implementation tool-flow is presented in Figure 9 on page 41.

The software tool-flow bases primarily on a standard C compilation tool-chain. To avoid modification of the compiler and assembler, the software tool-flow augments the compilation tool-chain with intermediate pre and post-processing steps.

#### 3.4.1 Hardware- Software Interface

The Zippy architecture integrates the RPU as a coprocessor to the CPU core (as opposed to a direct integration in the CPU's data-path). This style of coupling is motivated by streaming signal-processing applications where the reconfigurable coprocessor processes a whole buffer of input data without the need for CPU intervention.

The RPU exposes all functions (data-transfer, programming of the configuration memory and the context sequencer, etc.) via the register interface (see Section 2.3.3). Hence, each interaction between CPU and RPU is actually a read or write operation on one of the coprocessor's registers.

### 3.4.2 CPU Simulator and Compilation Tool-Chain

We use the SimpleScalar [ALE02] CPU simulator for simulating the CPU core of the Zippy architecture. Apart from the simulator itself, the SimpleScalar tool-suite also provides a complete C compilation tool-chain. The tool-chain bases on a GNU C cross-compiler that targets SimpleScalar's MIPS-like instruction set called PISA. The tool-chain further includes an assembler and a linker based on the GNU binutils.

The software parts of an application for Zippy are obtained by a code-design process, which is so far performed manually by the developer. The software part is implemented in C code, compiled, assembled and linked with the SimpleScalar cross-compilation tool-chain.

### 3.4.3 Extensions to the CPU Simulator

SimpleScalar allows for easily extending and modifying the instruction set. SimpleScalar even supports the addition of new functional units, whereas the concept of functional units is not restricted to pure arithmetic units. We use this flexibility in defining new functional units for adding a coprocessor interface to the CPU core, which is not available in the default architecture. This adds a dedicated IO interface with its own IO address space to the CPU model and enables concurrent accesses on the memory bus and the coprocessor interface. SimpleScalar takes care of integrating this new functional unit correctly with the rest of the data and control-path of the out-of-order execution, super-scalar CPU architecture.

For accessing the newly defined coprocessor interface we have added two new instructions to the PISA instruction set, see Table 5. The *RU\_setreg*

<code>RU_setreg \$1,\$2,\$3</code>	store value in register \$3 to coprocessor register denoted by register \$2 (store optional result in register \$1)
<code>RU_getreg \$1,\$2</code>	read coprocessor register denoted by register \$2 and store result in register \$1

**Tab. 5:** Coprocessor instructions added to SimpleScalar

and *RU\_getreg* instructions provide the application with low-level access

to the coprocessor port.

Although SimpleScalar allows for easy modification of the instruction set and the parameters of the architecture that is simulated, the corresponding tools (compiler, assembler) cannot be generated automatically from the instruction set. That is—without modification—the cross-compiler will never issue one of the new instructions.

### 3.4.4 Compilation Tool-Chain

In order to access the coprocessor port but to avoid modification of the compiler and assembler, we have decided to use the unmodified compiler and assembler, but to augment the compilation tool-chain with additional processing steps.

To avoid a compiler modification, we make use of the fact that the C compiler passes arbitrary inline assembler instructions directly to the assembler without modification. The developer can thus use the new coprocessor instructions by using inline-assembler statements. The GNU C compiler simplifies this task by supporting powerful *inline-assembler macros*. These macros can create high-level wrappers around inline-assembler commands that can be used like ordinary C functions. The assembler code within these macros can interact with surrounding C code and can access variables with their symbolic names.

The application thus never calls a coprocessor instruction directly, but always uses the function wrappers. The function wrappers are the base of the *communication and configuration* library which allows the application to conveniently access the RPU's functions.

While the proposed method enables the C compiler to issue the new coprocessor instructions, the assembler still cannot process them, because they are not part of the PISA instruction set. We solve this problem by splitting the compilation process into two phases: the compilation phase and the assembly phase. In an intermediate processing step (instruction encoding) the coprocessor instructions are replaced by instructions that are understood by the assembler.

The coprocessor instructions in the application's assembly codes serve as pure *pseudo-instructions* that cannot be directly executed. But, although the instructions are not executable, the inline-assembler has determined the register allocation for the instruction's operands. The *instruction encoding* step removes these pseudo-instructions from the assembly and replaces them with the binary instruction encoding for the instructions and their register operands. The instruction encoding is specified with a *.word* assembler directive, which can insert arbitrary data in the assembled object code.

A similar method for code generation is used in the software tool-flow

for the OneChip [CEC01] and, except for the function wrappers, for the REMARC [MO99] architecture.

Figure 16 depicts the augmented software tool-flow and presents the implementation of the function *RU\_readfifo* which reads the contents of a FIFO on the RPU. The communication library *comlib.h* defines the *RU\_readfifo(x)* function wrapper, which is implemented as an inline-assembler macro. The macro translates the function-like *RU\_readfifo* command into the corresponding coprocessor instruction *ru\_readreg*.

The function can be called within *app.c* like any ordinary C function, but is translated by the compiler to a *ru\_getreg* pseudo-instruction. The compiler determines also the register allocation for the operands of the pseudo-instruction. Before assembling the intermediate assembler file *app.i.s*, the instruction encoding process replaces the pseudo-instruction with its instruction coding, that is inserted with a *.word* directive. Finally, the unmodified assembler and linker transform the resulting assembler file (with all coprocessor instructions encoded) into an executable.

### 3.5 Summary

In this chapter we have introduced an application implementation tool-flow for the Zippy architecture. We have discussed the two major tool-flows that have been proposed for coarse-grained reconfigurable CPUs.

Zippy uses a circuit-centric tool-flow because this tool-flow matches the characteristics of Zippy's application domain well. The circuits are specified with the textual ZNF netlist format.

The hardware tool-flow uses an iterative placement and routing process, which is driven by a simulated-annealing-based stochastic search procedure, to find a feasible circuit implementation. The routing algorithm bases on the well-known Pathfinder algorithm. While starting the placement and routing process with a random initial placement is successful with many FPGA architectures, it leads frequently to unroutable placements on the Zippy architecture, because the coarse-grained Zippy architecture has only sparse routing resources. We have developed a new heuristic placement algorithm that improves the routability of the initial by placing adjacent cells in the netlist to adjacent sites on the reconfigurable array. The Pathfinder routing algorithm and the stochastic search procedures have been adapted by defining architecture-specific cost functions.

Finally, we have developed a software tool-flow based on a C compilation tool-chain. We have extended the SimpleScalar CPU simulator (which is used to simulate the CPU core) with a coprocessor interface and corresponding coprocessor instructions. To enable the use of copro-



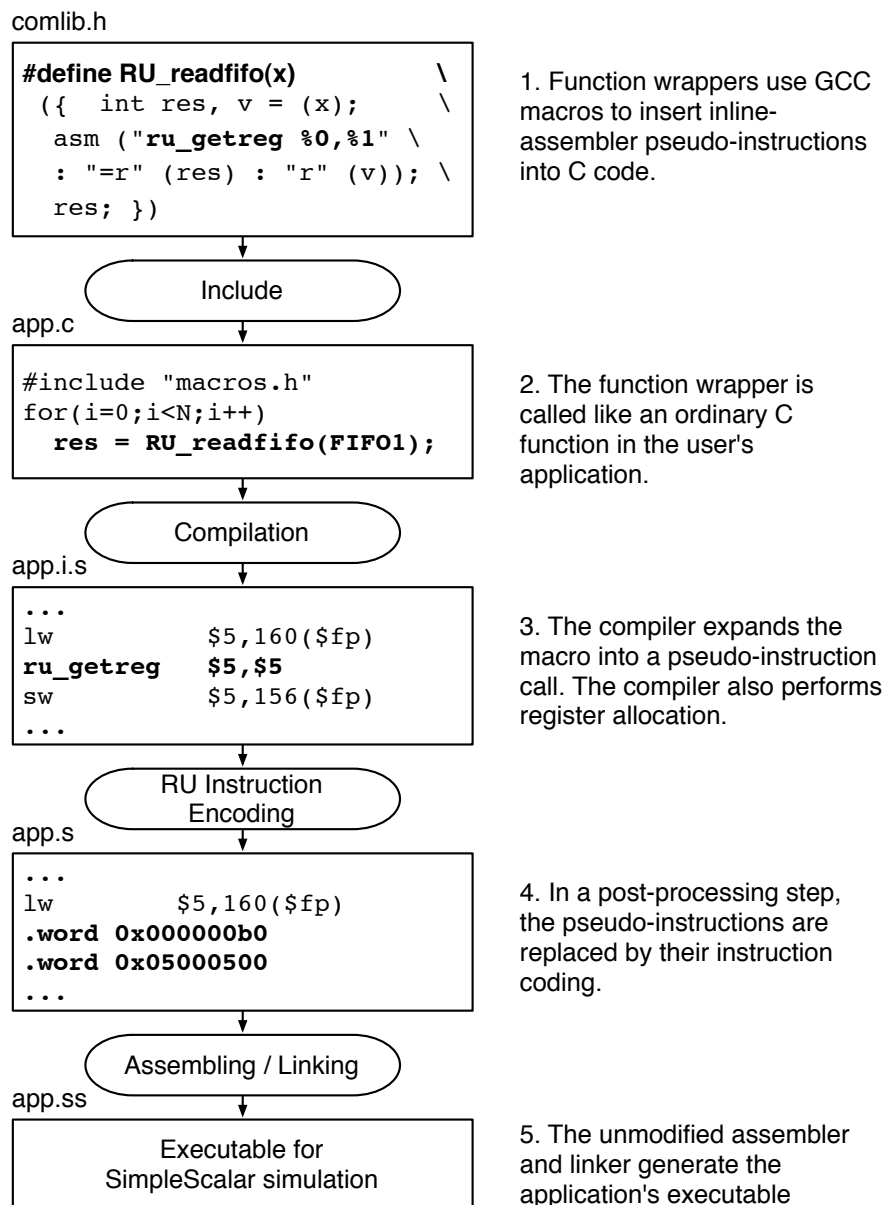


Fig. 16: Tool-Flow for generating software executables running on the CPU

cessor instructions but to avoid a modification of the compiler and the assembler we have augmented the compilation process with intermediate processing steps that perform instruction encoding. We make the coprocessor access functionality available to the application developer via a convenient communication and configuration library.

The proposed implementation tool-flow allows a designer who is familiar with block-based design of streaming signal-processing algorithms to develop an application for the Zippy architecture. The circuit centric tool-flow allows for achieving high-performance implementations by reusing existing circuits, while the communication and configuration process is simplified with a communication and configuration library.

# 4

## Performance evaluation

For evaluating the performance of an instance of the Zippy reconfigurable processor architecture we use a cycle-accurate cosimulation. To this end, we combine the cycle-accurate SimpleScalar CPU simulator with the ModelSim VHDL simulator into a system-level cosimulator for the complete architecture. The VHDL simulator executes a cycle-accurate VHDL model of the Zippy architecture.

Many simulators for reconfigurable processors rely on pure functional simulation of the Reconfigurable Processing Unit (RPU) and thus frequently trade reduced accuracy in timing-behavior and bit-exact computation for a decrease in simulation time. In contrast, our execution-based cosimulation approach allows for cycle-accurate performance evaluation on the system-level and provides bit-exact results.

Section 4.1 motivates the need for system-level performance evaluation. Two approaches for performance evaluation of reconfigurable processors are discussed and compared.

Section 4.2 introduces the execution-based performance evaluation framework for the Zippy architecture. We discuss the modeling and simulation of the CPU core and the RPU, and discuss the integration of the two simulators into a common cosimulation environment.

## 4.1 Performance Evaluation for Reconfigurable Processors

In this section we discuss the challenges in building a performance evaluation framework for reconfigurable processors. We compare two different approaches and motivate our choice of an execution-based cosimulation approach.

### 4.1.1 Motivation

An accurate method for performance evaluation of a reconfigurable processor is needed for assessing the quality of a given reconfigurable processor implementation. But performance evaluation is also of interest during the design-phase of new reconfigurable architectures. For parametrized architectures, like the Zippy architecture, accurate performance evaluation plays an important role in the design-space exploration process for comparing design alternatives.

Performance is frequently identified with the pure computational performance and is measured via the execution time of a benchmark application. We have defined and analyzed a set of benchmark applications, called *MCCmix*, that are significant for our target domain [EPP<sup>+</sup>01]. This benchmark set includes a choice of multi-media, cryptography and communications applications that are representative for handheld and wearable computing applications.

Apart from the *computational performance*, there are two additional important performance metrics: *chip area* and *power consumption*. Since many embedded systems are cost and power sensitive, i. e., chips size and power consumption should be minimized, these additional metrics are also relevant for embedded systems. There is an inherent trade-off between these three performance metrics, e. g., an increase in computational performance entails an increase in chip area or power consumptions.

While Enzler has shown in his dissertation [Enz04] that a parametrized chip area model for a precursor of the Zippy architecture is feasible and power models for a parametrized CPU core have been discussed in [BTM00, BBS<sup>+</sup>00], the derivation of a high-level power model for a complete reconfigurable processor remains an open research issue. For this work, we focus exclusively on a evaluation framework for determining the computational performance.

### 4.1.2 Challenges

Determining the performance of a reconfigurable processor is difficult, mainly due to dynamic effects in the CPU core. State-of-the-art CPU

cores use a number of dynamic techniques to increase the performance, for example, pipelining, caching and out-of-order execution. Since these techniques are applied during runtime and rely on decisions based on the execution history, the execution behavior of a program is difficult to predict. But also the interfaces of the CPU core to its environment, i. e., the memory subsystem and IO buses are becoming increasingly complex and involve varying access latencies.

Thus, cycle-accurate simulation is the only viable method for accurately determining the runtime behavior of the CPU core in a reconfigurable processor.

Performance evaluation for the RPU requires essentially discrete event simulation. Generally, the RPU uses less dynamic techniques than the CPU core. Still, the design of a high-level simulation model that is cycle-accurate and functionally equivalent (bit-exact) is a challenging task because the RPU executes an arbitrary digital-circuit, e. g., with data-dependent processing times, rounding effects due to fixed-point arithmetic, etc.

### 4.1.3 Approaches

Two alternatives for integrating the simulation model of an RPU with a CPU simulator have been proposed, the *functional simulation* approach, and the *execution-based cosimulation* approach.

#### Functional Simulation

In the *functional simulation* approach the reconfigurable architecture is not explicitly modelled, but *one particular configuration* of the RPU is treated as an additional CPU instruction. A static functional and timing behavior which is determined by the RPU configuration is assigned to this additional instruction. For the purpose of simulation, the execution of the RPU is replaced by a software component, that implements the functional behavior and also accounts for the latency of the RPU execution. During simulation, the CPU simulator schedules the RPU operations to this new functional unit. This modelling style is frequently used for RPUs that are tightly integrated with the processors data-path (pipeline) to form a custom arithmetic unit, frequently called Reconfigurable Functional Unit (RFU). It has been used for CPUs that provide fine-grained RFUs, e. g., for the simulation of the OneChip [CEC01], Chimaera [YMHB00], and XiRisc [LTC<sup>+</sup>03] architectures. Since the RFU gets its operands directly from the CPU's register-file, and since the complexity and latency of such an operation is similar to a CPU instruction, treating RFU instructions like CPU instructions is appropriate.

### Execution-Based Cosimulation

In the *execution-based cosimulation* approach the RPU is not abstracted and replaced by a functional model, but forms a simulation component on its own. A corresponding architecture cosimulator interfaces the execution-based RPU simulator to a cycle-accurate CPU simulator.

In contrast to the functional simulation approach, which simulates only one particular configuration of the RPU, the execution-based approach models the RPU itself. This RPU model includes the reconfigurable array but also the configuration and control circuitry. A particular RPU configuration is simulated by loading the configuration into the RPU model and executing the configuration.

This approach is useful for architectures that do not integrate the reconfigurable structures as tightly as an RFU, but attach the RPU to an IO or coprocessor interface. For these architectures the access to the RPU does not occur directly within the CPU's data-path with a fixed delay as for fined-grained reconfigurable CPUs with RFUs. But the delay for accessing the RPU for configuration, data-transfers and configuration sequencing varies due to the dynamic effects in the CPU.

The advantage of this approach is, that it provides a system-level cycle-accurate cosimulation. It also accounts for the various overheads involved with a reconfigurable processor architecture, for example, configuration upload, activation and sequencing, data-transfers, or latencies in accessing the IO interfaces.

## 4.2 System-Level Cycle-Accurate Co-Simulation for Zippy

In this section we will introduce our cosimulation framework for the Zippy architecture which uses the *execution-based cosimulation* approach. We integrate two cycle-accurate simulators into one co-simulation environment. This allows us to perform system-level performance evaluation and to use the appropriate simulation tool for each simulation task.

The requirements for the CPU simulator are high efficiency, cycle accuracy, and the availability of a robust code-generation framework for compiling benchmark applications. The requirements for the simulator of the RPU are cycle accuracy and the possibility for specification on different levels of abstraction, particularly on the behavioral level and the register-transfer level.

Although it is widely recognized that the raw performance of a reconfigurable coprocessor and the performance of the same coprocessor when embedded in a reconfigurable processor can differ significantly, the topic

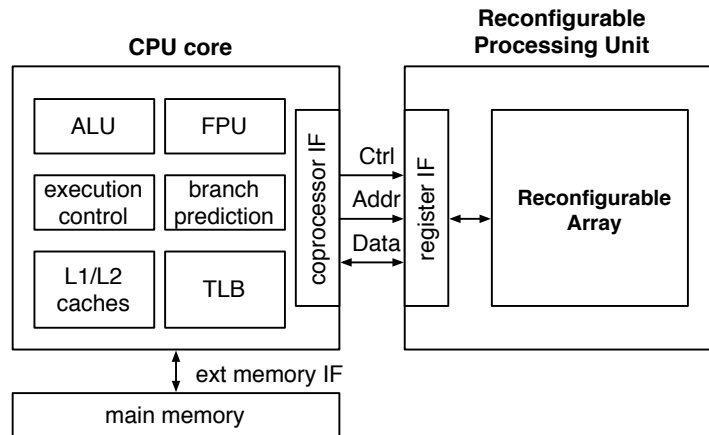


Fig. 17: System Architecture of a Generic Reconfigurable Processor

of system-level performance evaluation has not gained a lot of attention.

We have introduced our cosimulation framework first in [EPP03] and [EPP05] and to the best of our knowledge, this is the only work that specifically focuses on system-level performance evaluation for reconfigurable processors.

#### 4.2.1 Architectural assumptions

Our performance evaluation framework is rather general because it makes only a few general assumptions about the system architecture:

- the reconfigurable processor consists of a CPU core and has a reconfigurable processor attached to its coprocessor port,
- data transfers, configuration loading, and execution control (synchronization of coprocessor and CPU) is performed exclusively via the coprocessor interface,
- the CPU is simulated with the SimpleScalar CPU simulator, and
- the coprocessor is modelled with a cycle-accurate VHDL model

Figure 17 shows system architecture of a generic reconfigurable processor as it is assumed by the cosimulation framework. Actually, the cosimulation treats the RPU as a black-box and can simulate any CPU/coprocessor system, as long as a cycle-accurate VHDL model is provided.

#### 4.2.2 CPU Simulation Model

For the simulation of the CPU core and the memory architecture we use the well-established SimpleScalar CPU simulator [ALE02]. We have introduced SimpleScalar already in previous chapters of this thesis: Section 2.2.2 presents more details on the role of SimpleScalar in the recon-

figurable processor architecture, and section 3.4 presents the extensions of the software generation tool-flow to support the coprocessor interface that has been added to SimpleScalar.

SimpleScalar is an extensible, parametrized CPU simulator tool-suite that simulates a super-scalar out-of-order CPU architecture with an extended MIPS-like instruction set named PISA. It is frequently used for CPU architecture research since the simulator is available in C source code and allows for modification for academic purposes. In addition to the CPU simulator, SimpleScalar bundles also a complete set of compilation tools: a GNU C compiler based C cross-compiler and a GNU binutils based suite of assembler, linker, etc.

SimpleScalar gathers detailed, cycle-accurate execution statistics by executing the compiled application binary on a parametrized CPU model. The parameters for the super-scalar execution core include the number of fixed-point and floating-point execution units, decode, issue and commit bandwidths, and the size of the instruction fetch queue, the load-store queue, and the register update unit. The simulator supports several kinds of branch predictors and optional out-of-order execution. The memory architecture can be customized by specifying up to two levels of cache. Overall, this allows for performance evaluation of a broad spectrum of systems, ranging from small, embedded CPUs to high-end, super-scalar CPUs.

Since SimpleScalar is available in source code and can be easily extended with new instructions and even functional units, it is well suited for building a *functional simulation-based* simulator for a reconfigurable processor. For example, SimpleScalar has been used for building functional simulators for the OneChip [CEC01] and the Chimaera [YMHB00] architecture. Both architectures are reconfigurable processors with fine-grained RFUs.

For the simulation of the Zippy architecture we have chosen to use *execution-based cosimulation*. To this end, we have extended SimpleScalar with a coprocessor interface, which is modeled as a functional unit, and with instructions for accessing the coprocessor interface. For more details on this extension and the corresponding software toolflow we refer to Section 3.4.

Conceptually, the RPU coprocessor is attached to this coprocessor interface. But while the coprocessor interface is modelled within SimpleScalar, the RPU itself is modelled outside of SimpleScalar with a VHDL simulator that executes a cycle-accurate VHDL model of the RPU. The cosimulation framework takes care of bridging the coprocessor access of SimpleScalar to the external VHDL simulator and returning results from the VHDL simulation to SimpleScalar again.



### 4.2.3 RPU Simulation Model

The RPU is modelled as a cycle-accurate VHDL model and is simulated with the ModelSim simulator [Mod05]. ModelSim is a powerful, mixed-language simulator that supports VHDL and Verilog, and it was recently extended with SystemC support. Via an extension interface ModelSim provides the programmer with direct access to the simulation kernel. We use this extension interface to integrate SimpleScalar and ModelSim into a common cosimulation environment, see section 4.2.4.

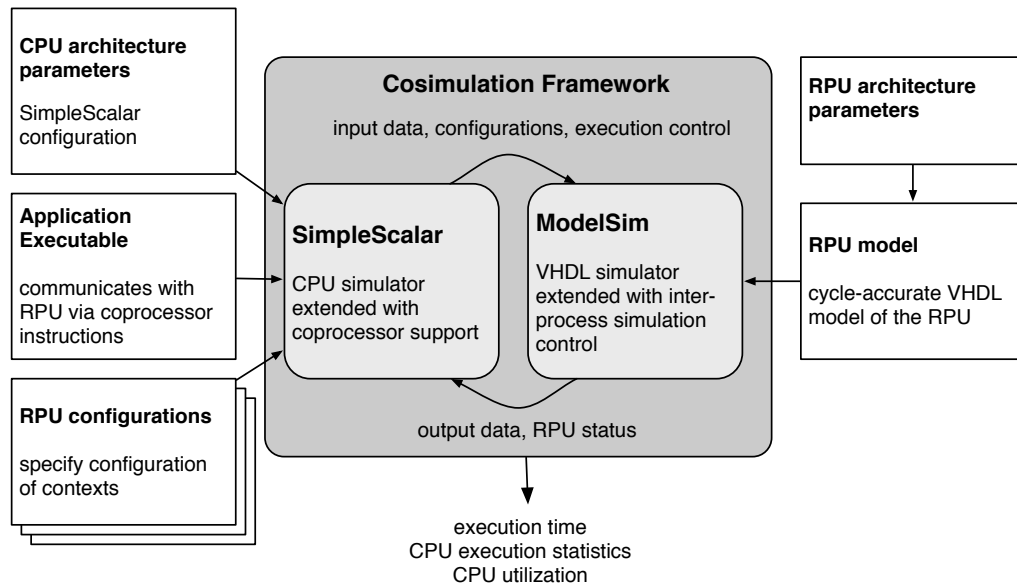
The main motivation for using VHDL for modelling the RPU is that VHDL seamlessly supports the modeling at different levels of abstraction. VHDL supports high-level behavioral modelling with non-synthesizable constructs but also modelling at the Register Transfer Level (RTL) and even at the structural level. In spite of these different levels of abstraction, VHDL allows for retaining overall cycle accuracy, while using the expressiveness of behavioral modelling for parts of the model. Using a VHDL model of the RPU enables us to use the same model for stand-alone simulation of the RPU and system-level cosimulation of the reconfigurable processor. Potentially, the VHDL description also allows for generating a prototype Very Large Scale Integration (VLSI) chip implementation if the RPU model is stepwise refined to the synthesizable VHDL subset. Hence, we can get a perfect match of the architecture simulation model and the architecture implementation.

Another strong argument for choosing VHDL is the maturity of VHDL development, simulation and synthesis tools. Sophisticated VHDL simulators ease the development of the RPU. A conventional VHDL testbench can be used for functional verification of the RPU. This process is supported by verification libraries, VHDL debuggers and wave form viewers.

### 4.2.4 Cosimulation framework

The system-level cosimulation framework for the Zippy architecture combines the cycle-accurate simulation models for the CPU core and the RPU into a common simulator.

The main result obtained from the cosimulation is the application's *execution time* measured in *cycles*. Since the cosimulation is execution-based, the execution of the application includes also the process of loading the configurations to the RPU, transferring data between CPU and RPU and switching between configurations. Additionally, SimpleScalar collects a multitude of execution statistics for the CPU core, for example, cycles per instruction, cache and branch-prediction miss-rates, and dispatch rates for the instruction fetch, load-store and register updated units. Assuming a certain CPU clock rate, the *CPU utilization (load)* can be computed as a derived metric. The CPU utilization is of interest for applications



**Fig. 18:** Co-simulation framework integrating the SimpleScalar and ModelSim simulators

that demand for fixed-rate processing (for example real-time audio signal processing) instead of throughput maximization.

In the cosimulation framework, both simulators run in parallel as communicating processes. The SimpleScalar simulator acts as the master of the simulation and has complete control over the RPU simulation. Whenever SimpleScalar encounters a coprocessor instruction, it relays the corresponding command to the ModelSim VHDL simulator. The results from the RPU simulation are sent back to SimpleScalar.

Figure 18 outlines the structure of the cosimulation framework. SimpleScalar requires three input files: the CPU architecture parameters, the compiled application executable, and the RPU configuration bitstreams. The application parts that run on the CPU are implemented in C and compiled with the GCC-based C cross-compiler provided by the SimpleScalar tool suite. The application code needs to take care of downloading the RPU configuration bitstreams and of controlling their execution. The coprocessor instructions, which we have added to SimpleScalar, are accessed using pseudo-assembler instructions. A communication and configuration library facilitates the access to these pseudo-assembler instructions from a C application. The library contains functions for downloading the configurations, switching between contexts, and for transferring data between CPU and RPU. The software tool-flow for application implementation is presented in more detail in section 3.4.

The configuration bitstream is created with the hardware tool-flow, which takes a netlist description of an application and runs an automated

placement and routing process to find a feasible implementation. The implementation is converted to a binary representation, denoted as configuration bitstream, and is downloaded to the RPU by the application at runtime. For more details on the hardware tool-flow we refer to section 3.3

### Interfacing the CPU and the RPU Simulator

For cosimulating the complete reconfigurable processor SimpleScalar must be able to control the RPU simulation. Technically, SimpleScalar and ModelSim run as separate but communicating processes. The two simulation processes communicate with commands exchanged via a shared memory area. These commands correspond to the coprocessor instructions. Communication via shared memory is an efficient inter-process communication mechanism in Unix operating systems, but requires an additional mechanism for synchronizing the memory access. We synchronize the shared memory access with a simple handshake protocol implemented with semaphores.

Since SimpleScalar is available in source code, the inter-process communication and synchronization code for accessing ModelSim is directly integrated into the simulator. For integrating the communication code in ModelSim, we extend the VHDL simulator through the *foreign language interface* [Mod04]. This extension interface allows for loading user-defined shared libraries with access to ModelSim's simulation kernel into the simulator. We use this interface to expose complete control over the RPU simulation in ModelSim to the SimpleScalar simulator.

Algorithms 5 and 6 present pseudo-code for the handshake and communication protocol that synchronizes the access to the shared memory area (SMA). Two binary semaphores (ReqPendingSem, and ReqServedSem) are used. ReqPendingSem is signaled by the SimpleScalar to inform ModelSim that a new coprocessor command is ready to be processed. After running the RPU simulation for one cycle, ModelSim signals the end of the processing with the ReqServedSem semaphore.

Keeping the simulation of SimpleScalar and ModelSim strictly synchronized suggests to communicate in every cycle. But although the shared memory data-transfers and the synchronization with semaphores is rather fast, excessive communication slows down the cosimulation unnecessarily. For performance reasons we allow the simulation time in the VHDL simulator to lag behind the CPU simulation time. Since synchronization of the CPU and the RPU is only required at the time of communication through a coprocessor instruction, we re-synchronize the simulation times of CPU and RPU only at communication events and at the end of the simulation. The pseudo-code in Alg. 5 shows this time-lag mechanism in more detail. A *lag* counter is incremented for every

---

**Algorithm 5** Inter-Process Communication and Synchronization for SimpleScalar
 

---

```

1:  $lag \leftarrow 0$ 
2: while not(application finished) do
3:   if instruction = coprocessor instruction then
4:     if  $lag > 0$  then
5:       write command to SMA (cmd=catchup(lag))
6:       signal(ReqPendingSem)
7:       wait(ReqServedSem)
8:        $lag \leftarrow 0$ 
9:     end if
10:    write command to SMA (cmd=coproc instruction)
11:    signal(ReqPendingSem)
12:    wait(ReqServedSem)
13:    read results from SMA
14:  else
15:     $lag \leftarrow lag + 1$ 
16:  end if
17: end while
18: if  $lag > 0$  then
19:   write command to SMA (cmd=catchup(lag))
20:   signal(ReqPendingSem)
21:   wait(ReqServedSem)
22:   write command terminate to SMA
23:   signal(ReqPendingSem)
24: end if

```

---



---

**Algorithm 6** Inter-Process Communication and Synchronization for ModelSim
 

---

```

1: while true do
2:   wait(ReqPendingSem)
3:   read command from SMA
4:   if command = terminate then
5:     terminate simulation
6:   else if command = catchup then
7:     run simulation for  $t_{cycle} \cdot lag$ 
8:   else
9:     run simulation for  $t_{cycle}$ 
10:    write results to SMA
11:   end if
12:   signal(ReqServedSem)
13: end while

```

---

instruction that is no coprocessor instruction. When the next coprocessor instruction occurs, the RPU simulation is synchronized first, before the new command is handed to the ModelSim simulator.

Using a VHDL simulator as the base of a cosimulation framework for a reconfigurable processor provides many advantages, mainly the reuse of a modelling language and discrete event simulation kernel for specification, stand-alone verification, (co-)simulation and potentially also implementation. Since there is increasing tool support for the SystemC modeling language [Pan01, GLMS02, Ope02] we consider SystemC modelling as an interesting alternative, once the tool support for SystemC is on par with VHDL. A pure SystemC model could be implemented by modelling the RPU in SystemC and by wrapping SimpleScalar in a SystemC model. Such a plain SystemC model could be an advantage, if a closer integration of CPU and RPU is desired, e. g., if the RPU could directly access the main memory.

### 4.3 Summary

Dynamic effects in the CPU core, variable delay for communication between CPU and RPU, and possibly data-dependent processing times in the RPU, render pure functional RPU models for the performance evaluation of a reconfigurable processor inaccurate. That is, the raw (best-case) performance of the RPU and the actual performance when integrated in a reconfigurable processor can differ significantly.

Hence, we propose system-level performance evaluation for accurately assessing the performance of a reconfigurable processor. We rely on execution-based performance evaluation based on cosimulation. The proposed framework combines two cycle-accurate simulators, the cycle-accurate SimpleScalar CPU simulator and the ModelSim simulator, which executes a cycle-accurate VHDL model of the RPU.

We argue, that using VHDL as a modeling language and thus basing the RPU simulation on a VHDL simulator is beneficial, because the VHDL simulator serves as a solid discrete event simulator and the VHDL language provides excellent support for modelling at different levels of abstraction. We have introduced a performance optimization, that minimizes the communication overhead between the CPU and the RPU simulator to the minimum.

The resulting cosimulation environment allows us to perform cycle-accurate simulation of the whole architecture and is in this respect superior to many other simulation environments for reconfigurable processors that base on functional simulation.



# 5

## Hardware Virtualization

In this chapter we introduce *hardware virtualization*. Hardware virtualization denotes a number of techniques that can be used to decouple the specification of a circuit from its execution on a reconfigurable architecture. Hardware virtualization can be an interesting implementation technique in particular for embedded systems, when hardware applications shall be executed on a reconfigurable co-processor with only modest hardware resources.

In Section 5.1 we define a classification of hardware virtualization techniques for dynamically reconfigurable architectures. We define three approaches named *Temporal Partitioning*, *Virtualized Execution*, and *Virtual Machine*. For each technique we present a survey on application and specification models, implementation architectures and runtime systems.

Section 5.2 proposes an *application specification model* for the Zippy architecture, that explicitly considers the Temporal Partitioning and Virtualized Execution hardware virtualization techniques. We review the architectural requirements for these virtualization techniques and show how the Zippy architecture supports these techniques with dedicated hardware units.

Finally, in Section 5.3 we present a *novel method and an algorithm for optimal temporal partitioning* of sequential circuits. The method treats temporal partitioning as an optimization problem and presents a problem formulation that can be solved optimally.

## 5.1 Introduction to Hardware Virtualization

Literally, hardware virtualization means that an application executes on virtualized hardware as opposed to physical hardware. “Virtual hardware” seems to be a contradictory term because hardware is supposed to have a physical existence. Furthermore, the terms “virtual hardware” and “hardware virtualization” are used for different concepts in literature.

Initially, the term virtual hardware was coined to show the analogy to virtual memory. There, pages of memory are swapped in and out a computing system, allowing applications to address a much larger memory than physically existent. In the same way, a reconfigurable computing system can swap in and out portions of the hardware by a reconfiguration process, allowing applications to use more hardware than physically existent. Later on, the term hardware virtualization was used to describe mapping techniques and architectures that allow for a certain degree of independence between the mapped application and the actual capacity of the target architecture. The most radical form of hardware virtualization is to strive for complete independence of the hardware execution from the actual underlying hardware.

### 5.1.1 Hardware Virtualization Approaches

5 So far, no general accepted taxonomy for hardware virtualization approaches has been defined. We have identified *three approaches to hardware virtualization* that, although related, differ in their motivation. We define the following classification to denote these approaches: *temporal partitioning*, *virtualized execution*, and *virtual machine*.

- **Temporal Partitioning**

The motivation for this virtualization approach is to enable the mapping of an application of arbitrary size to a reconfigurable device with insufficient hardware capacity. Temporal partitioning splits the application into smaller parts, each of which fits onto the device, and runs these parts sequentially. Temporal partitioning was the first virtualization style that has been studied. It was a necessity when reconfigurable devices were too small for many interesting applications, but it is still of importance with today’s multi-million gate Field-Programmable Gate-Arrays (FPGAs)—in particular in embedded systems—for saving chip area and thus cost.

- **Virtualized Execution**

The motivation for virtualized execution is to achieve a certain level of device-independence within a device family. An application is



specified in a programming model that defines some atomic unit of computation. This unit is commonly called a hardware page. Hence, an application is specified as a collection of tasks (that fit into a hardware page) and their interactions.

The execution architecture for such an application is defined as a whole family of devices. All devices support the abstractions defined by the programming model, i. e., the hardware page and the interactions (communication channels). The members of a device family can differ in the amount of resources they provide, e. g., the number of hardware pages that are executing concurrently, or the number of tasks that can be stored on-chip. Since all implementations of the execution architecture support the same programming model, an application can run on any member of the device family without recompilation. In this respect, this approach is comparable to the device independence achieved for microprocessors by the definition of an instruction set architecture. The resulting independence of the device size allows the designer to trade off performance for cost. Furthermore, the forward compatibility lets us exploit advances in technology that result in larger and faster devices.

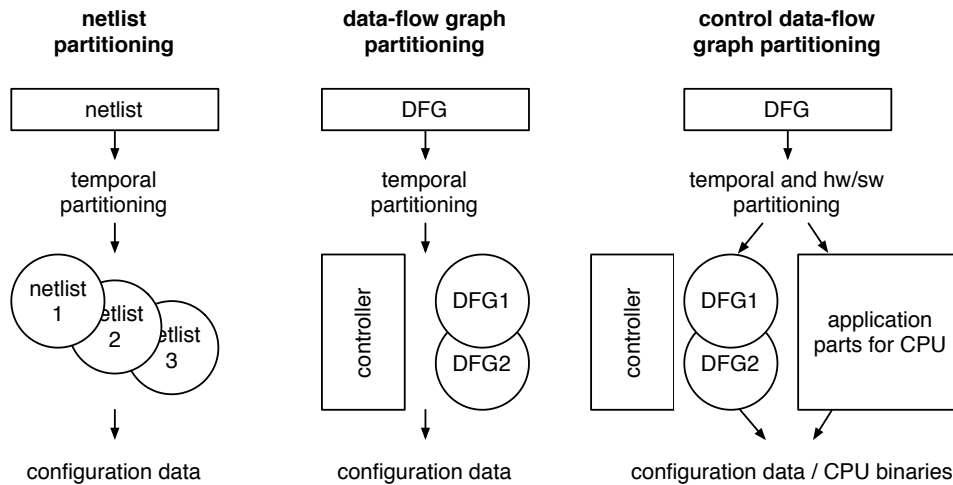
Virtualized execution requires some form of a runtime system that resolves resource conflicts at runtime and schedules the tasks appropriately.

- **Virtual Machine**

The motivation for this virtualization approach is to achieve an even higher level of device-independence. Instead of mapping an application directly to a specific architecture, the application is mapped to an abstract computing architecture. A hardware virtual machine is able to execute an application which has been mapped to such an abstract architecture.

There are two basic alternatives for constructing a hardware virtual machine: The first approach is to remap the application from the abstract representation to the native application code of a concrete architecture at the startup of the application. The virtual machine is then formally the tools for remapping the application. The other approach is to run an “interpreter” on the concrete architecture that allows for direct execution of the abstract application code.

This style of virtual hardware is analogous to the approach of platform-independent software that is used for instance in the Java virtual machine. Conceptually, a hardware virtual machine features platform-independent mobile hardware. This might be of increas-



**Fig. 19:** Approaches for temporal partitioning: netlist partitioning, data-flow graph partitioning, and CDFG partitioning

ing importance as most reconfigurable systems are connected to networks.

## 5.1.2 Temporal Partitioning

### 5.1.2.1 Application Models

The conventional way of specifying an application for a reconfigurable device is to use a hardware description language (HDL), e. g., VHDL or Verilog. This specification is synthesized to a Register Transfer Level (RTL) description and, finally, to a netlist of combinational and sequential logic elements. Design implementation tools further process this netlist and perform technology mapping, placement, and routing to generate the configuration data for the reconfigurable device.

Alternatively, an application can be specified in a high-level programming language (HLL), e. g., C/C++ or Java. A compiler builds an internal representation of the program in the form of a control data-flow graph (CDFG). From this graph, RTL descriptions and netlists are generated. The results are further processed by the same tools as in the HDL-based tool flow.

Temporal partitioning can be applied at different levels: at the level of netlists, at the level of data-flow graphs, or at the level of CDFGs (see Fig. 19).

*Temporal partitioning at the netlist level* operates on the netlists obtained by synthesis or compilation tools. Netlist partitioning does not depend on

the actual implementation language that was used for application specification. The main drawback of netlist partitioning is that one cannot make use of application-specific knowledge. In general, it is extremely difficult to regain information about the high-level structure or even dynamic behavior of the application from the netlist.

*Data-flow graph partitioning* works on operation or task graphs. Such graphs result from HDL and HLL design flows, but are also used as specification models in embedded systems design. An operation graph is a directed acyclic graph (DAG), where the nodes represent operations and the edges represent dependencies and communication.

*Temporal partitioning at the CDFG level* becomes possible if a high-level language is used for application specification. The main advantage over netlist or data-flow graph partitioning is that the CDFGs reveals information about the control-flow, e. g., loops, conditional execution, function calls, etc. The control-flow information defines all possible flows of computations and thus can be used to divide the application into partitions with only local computations (similar to basic blocks). A CDFG representation enables the compiler to perform high-level optimizations and transformations. Common optimization techniques address runtime-intensive loops and increase the parallelism by unrolling, software pipelining, or pipelined vectorization.

### 5.1.2.2 Reconfigurable Architectures

Temporal partitioning decomposes an application into smaller parts and executes them sequentially. Since the execution requires frequent reconfiguration, the efficiency of the approach is largely dependent on the ratio between a configuration's execution time and the device reconfiguration time. Conceptually, any reconfigurable architecture can be used for temporal partitioning. However, the demand for low reconfiguration overheads favors advanced architectures that allow for fast reconfiguration, e. g., multi-context FPGAs.

Temporal partitioning requires to transfer intermediate results between the partitions. While it is possible to use external memory for storing inter-configuration data, an efficient implementation of temporal partitioning demands for fast inter-configuration communication, for example, with a set of on-chip registers that can be accessed by all configurations.

All temporal partitioning approaches need a configuration controller. Usually, this control function is mapped to a processor. The resulting target architectures combine a processor with a reconfigurable device either in a single chip or as a board-level system.

### 5.1.2.3 Design Tools and Runtime System

The conventional tool-flow needs only moderate modifications to support hardware virtualization with temporal partitioning. At some abstraction level in the design tool flow the application is split up into a number of smaller parts. Additionally, a configuration controller and circuitry for inter-configuration communication is generated. The reconfiguration controller implements the runtime system which executes a static configuration schedule. The controllers for inter-configuration communication store the output data of configurations and provide subsequent configurations with input data. Each of the resulting application parts can be implemented with the same device-specific mapping, placement and routing tools as in the non-virtualized case. Whenever the resulting configuration exceeds the device capacity, the temporal partitioning step must be iterated.

When the application is specified in an HLL, the compiler can perform hardware/software partitioning in addition to temporal partitioning. Only the runtime-intensive inner loops of an application are mapped to the reconfigurable device, while the rest of the application runs on the CPU.

### 5.1.2.4 Survey of Approaches

#### Temporal Partitioning of Netlists

Temporal partitioning of netlists leads to graph partitioning problems since netlists are commonly represented with graphs models. In spite of the different motivation, these problems are similar to *structural partitioning* problems. Structural partitioning has been studied in the context of FPGA-based emulation systems [But93]. These methods partition a large netlist that exceeds the capacity of a single FPGA into a set of smaller netlists for execution on a multi-FPGA emulation system. Each partition is required to fit onto a single FPGA while respecting the limited interconnect resources between the FPGAs.

Initially, temporal partitioning and time-multiplexed execution of partitions has not been considered for logic emulation, because the long reconfiguration times for conventional FPGAs would lead to excessive overheads. With the development of multi-context FPGAs that support fast reconfiguration temporal partitioning of netlists became realistic. Inspired by early multi-context FPGAs like DPGA [TCE<sup>+</sup>95] and TMFPGA [TCJW97], first studies looked at special cases of digital logic. Trimberger [Tri98] and DeHon [DeH96b] discuss list scheduling algorithms for mapping finite state-machines in leveled logic form to multi-context architectures.

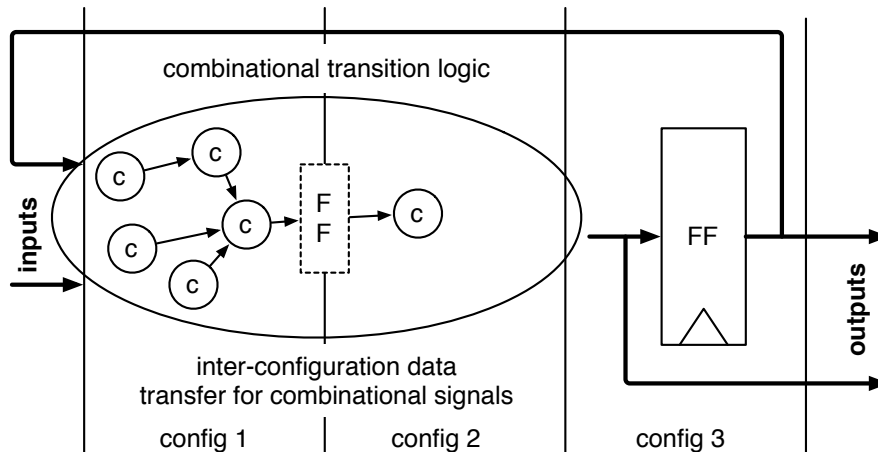


Fig. 20: Temporal partitioning of netlists (logic engine mode of TMFPGA, after [Tri98])

Figure 20 presents the mapping of a Finite State Machine (FSM) to a multi-context FPGA. The combinational state transition logic is partitioned into a number of configurations that form device contexts (configurations 1 and 2 in Fig. 20). The state of the FSM is mapped in a separate configuration (configuration 3 in Fig. 20). The combinational signals crossing the configurations are held in a set of inter-context communication registers that are shared among all contexts. At runtime, the contexts are cyclically executed.

Implementing FSMs with leveled logic can lead to a large number of intermediate signals and thus to an excessive amount of registers. Therefore, several authors proposed improved heuristics that try to reduce the size of the inter-context communication registers. A force-directed scheduling based method was studied by Chang and Marek-Sadowska [CMS97][CMS99]. Mak and Young [MY02] and Liu and Wong [LW98] used a network-flow based method. Wu et al. presented an exact ILP based algorithm [WLC01].

### Temporal Partitioning of Operation and Task-Graphs

The approaches in this group define temporal partitioning as an optimization problem over acyclic data-flow graphs. Similar to the netlist approach, the graph is partitioned into a number of configurations such that each single configuration does not exceed the device capacity. Further, the construction of a feasible configuration schedule must be possible. The primary optimization objective is the minimization of the overall execution time.

Purna and Bhatia proposed two greedy heuristics for solving this problem with constructive algorithms named the level-based and the

cluster-based partitioning algorithm [PB99] [PB98]. These methods do not directly minimize the overall execution time, but try to minimize the number of configurations, the configuration's execution times and the amount of inter-configuration data, respectively.

An exact method to solve the temporal partitioning problem was presented by Vemuri et al. [GOK<sup>+</sup>98]. The authors used a 0–1 linear program that minimizes the overall execution time. This ILP formulation was also extended to constrain the size of the memory for storing inter-configuration data and to cover other system level design steps, such as high-level synthesis [KV98] and design space exploration [SV99] [SGV01].

An approach for temporal partitioning of task graphs to partially reconfigurable devices was discussed in [FKT01] [TFS01] by Fekete et al. The resulting problem was cast as 3D packing problem and an optimal branch and bound procedure was given to solve it. Temporal partitioning was also combined with the hardware/software partitioning problem by Chatha and Vemuri [CV99]. This approach combines a greedy heuristic partitioner with a heuristic list scheduler.

### Temporal Partitioning of Control Data-Flow Graphs

The *Garp* project [CHW00] aims at creating a compiler that accelerates arbitrary applications written in C. The target architecture features a MIPS CPU core and a custom reconfigurable array co-processor [HW97]. The Garp C compiler (GarpCC) performs automatic hardware/software partitioning and maps application kernels to the reconfigurable array. The compiler pipelines loops and tries to find iterative schedules and uses similar techniques as VLIW compilers use for instruction scheduling, e. g., trace-scheduling. Garp doesn't use a dedicated runtime system but implicitly integrates the runtime system within the software part of the application.

*XPP-VC* [CW02] is a vectorizing C compiler for the PACT XPP architecture [BEM<sup>+</sup>03]. Instead of mapping only inner loops that directly fit onto the reconfigurable device, XPP-VC also uses temporal partitioning within a loop. After loop unrolling and data-dependence analysis, the compiler applies pipeline vectorization [WL01] to inner loops. Pipeline vectorization overlaps loop iterations and executes loops in a pipelined fashion. The XPP device implements a dedicated programmable configuration manager that executes the runtime system for a temporally partitioned execution. The configuration manager autonomously sequences and loads the configurations. A configuration cache is used to keep the most recently used configurations on-chip.

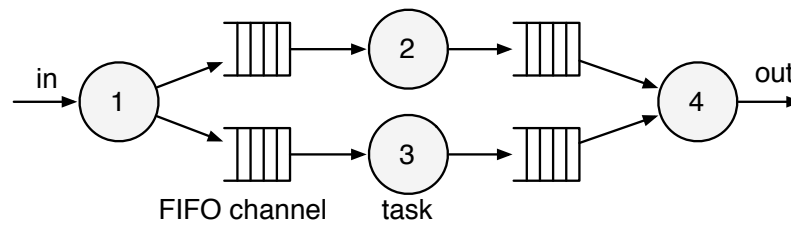


Fig. 21: Application specification with a process network

## 5.1.3 Virtualized Execution

### 5.1.3.1 Application Models

The use of *coordination languages* (or formalisms) for application modeling is widespread in embedded systems design. Coordination languages are semantically well-defined and usually more restricted than HLLs. Hence, they often allow for formal analysis of system properties such as the absence of deadlocks. Coordination languages are often tailored to specific application domains, e. g., SDF for signal-processing applications, or Petri-Nets for communication protocols.

In the majority of cases, coordination languages are used in combination with implementation languages. Large applications are decomposed into smaller execution objects, usually called tasks. The interaction of these tasks is specified using the coordination language, while the functionalities of the tasks are defined with an *implementation language*. This style of application modeling leads naturally to virtualized execution. Given a suitable device and a task scheduler, the application can be executed on the architecture. The sequence and interaction of the tasks is defined by the coordination language.

Figure 21 shows an example for an application specified in a coordination formalism named *Kahn Process Network* [Kah74]. The tasks in the process network are connected via FIFO channels of unlimited capacity. A task can run when input data is available on all of its input FIFOs. Process networks model concurrency, hence tasks 2 and 3 in Fig. 21 may execute in parallel provided that the execution architecture has sufficient resources.

Virtualized execution can be used to implement scalable and forward compatible systems, if the array uses fixed-size configurable operators. If all tasks specified in the coordination model are mapped to this fixed-size operator, the scheduler in the runtime system can map the application either spatially or temporally, depending on the number of operators that are available in a particular implementation of the reconfigurable architecture.

### 5.1.3.2 Reconfigurable Architectures

There are two basic architectural approaches to support virtualized execution. In the first approach, the reconfigurable architecture directly supports the programming model. As an application is decomposed into a number of interacting tasks, the architecture needs to implement the logic resources to execute one or several tasks concurrently and the communication channels to implement the interaction between the tasks. Such an architecture is composed out of atomic units (hardware pages) that can accommodate one task.

The second approach relies on classical reconfigurable devices and assigns the reconfigurable resources to the tasks at runtime. A task is assigned an amount of resources that matches its demand, rather than a fixed-size hardware page. While such a scheme complicates the runtime system, it leads to improved device utilization.

### 5.1.3.3 Design Tools and Runtime System

The decomposition of an application into communicating operators, according to the chosen coordination language, can either be done by the programmer or by design tools. The resulting operators are implemented by conventional synthesis and design implementation tools.

Virtualized execution requires a runtime system. For architectures with fixed-size hardware pages, the runtime system schedules the tasks according to the semantics of the coordination language and the available hardware resources. Additionally to scheduling, the runtime system has to perform allocation of resources such as communication channels, buffers and IO ports. Although computing a static schedule is possible in general, it would diminish the benefits of device-independence. Architectures with variable-sized resource assignment require a more complex runtime system. Before a task can be scheduled, a feasible placement on the reconfigurable device must be found.

### 5.1.3.4 Survey of Approaches

The *PipeRench* architecture [GSB<sup>+</sup>00] focuses on the pipelined processing of data-streams. PipeRench's programming model is to decompose the application into a sequence of pipelined operators, called stripes. Feedback is supported only within a stripe, i. e., the execution model is restricted to applications with forward pipelining of a fixed sequence of operators. PipeRench uses a C compilation toolflow that transforms the application to a data-flow language, which is the basis for the implementation tools, after all functions have been inlined and all loops have been unrolled. Hardware virtualization is achieved by allowing an application



to use an unlimited number of virtual stripes. If the number of physically available stripes is smaller than the number of virtual stripes required by an application, the configurations for the stripes are loaded at runtime. The runtime system is implemented in hardware. A hardware implementation of PipeRench with 16 physical stripes and on-chip configuration memory for 256 virtual stripes has been presented in [SWT<sup>+</sup>02].

Caspi et al. [CCH<sup>+</sup>00] developed the *SCORE model* that provides both, a specification model and a virtualized execution model for streaming applications. An application is defined as a graph of computation nodes (compute pages) that are connected by FIFOs of unbounded size. The memory used by the operators is allocated in fixed-size blocks (memory pages). The coordination of the operators is defined by the data dependencies in the execution graph. The function of the operators is specified with an RTL language with a C-like notation. So far there is no physical device that implements the SCORE application model. The runtime system is supposed to run on the CPU. It consists of the instantiation engine, that interprets the compute graph and instructs the scheduler which tasks are to be scheduled, and the scheduling engine, that manages resource allocation, placement of operators to compute pages and routing. The runtime system does also implement time-sharing of the compute-pages among operators.

The *WASMII* or DRL architecture [XA95] is a multi-context reconfigurable device that is specifically tailored to virtual hardware execution. The WASMII architecture is data-driven and allows for execution of data-flow graphs. The application's data-flow graph is decomposed into sub-graphs that fit the size of a page. A page is the basic computation unit of the device. The WASMII architecture also supports the connection of several WASMII devices to build a multichip WASMII architecture. The sequencing of pages is controlled by a static schedule that is generated at compile-time with the LS-M algorithm [LA89].

*Reconfigurable hardware operating systems* treat reconfigurable devices as dynamic resources that are managed at runtime. Similar to software operating systems, these approaches introduce tasks or threads as basic units of computation and provide various communication and synchronization mechanisms. The runtime system places and schedules tasks on the reconfigurable device in a multitasking manner and provides a minimal programming model, although being less restrictive than coordination languages.

The first description of hardware multitasking is due to Brebner [Bre96]. More recently, Wigley et al. discussed operating system functions including device partitioning, placement and routing [WK01]. Multitasking and task preemption was investigated in [SLM00] and [BD01], respectively. Scheduling and placement techniques were devised in [WP03a]

[SWPT03]. Functional prototypes that demonstrate multitasking on today's FPGA technology were also described, e. g., in [MNC<sup>+</sup>03] [WP03b].

### 5.1.4 Virtual Machine

Hardware virtualization with the *virtual machine* approach requires neither a specific application model nor a specific reconfigurable architecture. Application specification and synthesis can be performed with conventional tools, but the targeted implementation architecture is an abstract architecture. Mapping to an abstract architecture can easily be achieved by using generic synthesis and technology mapping libraries.

Defining an appropriate abstract architecture is vital to the virtual machine approach. The abstract architecture has to be generic enough to allow for an efficient remapping to different targets, but on the other hand, the abstract architecture must be close to typical reconfigurable architectures to exploit their performance potential. At loading time or at runtime, the abstract description is remapped to the actual architecture by a virtual machine. As the abstract architecture is a generalization of the actual architecture, the remapping involves running parts of the conventional tool flow, such as technology mapping, and place and route [HSE<sup>+</sup>00].

### Survey of Approaches

Issues of circuit portability in a networked environment were first addressed by Brebner [Bre98]. He coined the term *circlets* (*circuits* + *applets*) to denote mobile circuits. There is, however, no virtual hardware machine running on the target. The concept described circlets as location-independent circuits that are pre-synthesized, pre-placed and pre-routed to a specific technology. The mapping to the target FPGA is performed by a runtime system, similar to a hardware operating system.

Ha et al. [HSE<sup>+</sup>00] proposed a *virtual hardware machine* that executes *hardware bytecode*. The hardware bytecode for a circuit is essentially a technology-mapped, placed, and routed netlist for an abstract fine-grained FPGA architecture [HSE<sup>+</sup>00] with symmetrical routing. The virtual machine performs remapping of logic blocks and IO pins as well as global and detailed routing to translate the hardware bytecode to the concrete FPGA configuration. Temporal partitioning for hardware bytecode that exceeds the FPGA capacity was not considered. The proposed runtime system runs on the FPGA itself.

In [HVS<sup>+</sup>02], a framework was described that adapts the virtual hardware machine approach to a hybrid, networked target system consisting of a CPU and an FPGA. At design time, a hardware/software codesign en-

vironment partitions an application into software and hardware functions and generates the hardware and software bytecodes. A service bytecode binder combines all bytecodes into one file that is transferred to the target.

### 5.1.5 Summary

Virtualization of hardware is a rather new research area. To date, three main approaches have emerged. Each approach has its own challenges concerning design tools and runtime systems, and sometimes also device architectures.

Temporal partitioning was the first virtualization approach, applied to netlists and operation graphs. In the meantime, the main focus there has shifted to compilation from HLLs. We see the main application for temporal partitioning in embedded systems, where saving chip area and thus cost is an important optimization goal.

Virtualized execution comes in two varieties: approaches that compile to architectures with a fixed-size basic hardware element and approaches that map the atomic operator to variable-sized hardware elements at runtime. While the variable-sized approaches lead to a potentially higher device utilization, the algorithmic problems involved are hard and their solutions time-consuming.

The virtual machine approach is the newest one. Although conceptually the most powerful virtualization approach, the practical realization has yet to be shown. Remapping circuit descriptions at the target involves complex design tools and requires significant runtime. Many target systems, especially networked embedded systems, might just be not powerful enough for that.

Reconfigurable devices deliver their peak performance when a maximum on low-level parallelism and specific device features are used. A virtualization technique that sacrifices too much of this performance for the sake of device-independence will most likely not be accepted.

## 5.2 Hardware Virtualization on the Zippy Architecture

In this section, we propose an application specification model for the Zippy architecture, that considers the “virtualized execution” and “temporal partitioning” hardware virtualization techniques. Subsequently, we will briefly review the architectural requirements for these techniques and show how these techniques are supported with dedicated hardware units in the Zippy architecture.

### 5.2.1 Application Specification Model

Hardware virtualization is usually not considered in application specification models for reconfigurable processors, since most architectures do not provide dedicated support for hardware virtualization. For architectures that support virtualized execution and temporal partitioning, such as the Zippy architecture, we propose an *application specification model* which treats the application as a collection of communicating hardware and software tasks. The interaction of these tasks is formally specified with a coordination model. The hardware tasks are executed on the Reconfigurable Processing Unit (RPU) and the software tasks are executed on the CPU core.

If the hardware resource requirements for the concurrent execution of all hardware tasks exceeds the capacity of the RPU, hardware virtualization is used to generate a feasible implementation:

- a) Virtualized execution of hardware tasks is used for tasks that are sufficiently small for a direct implementation and execution.
- b) If a single hardware task is too large for a direct implementation, an attempt is made to further decompose the task into a set of smaller hardware tasks which follow the same coordination model and can be executed with virtualized execution.
- c) It can happen, that hardware tasks cannot be further decomposed or that the decomposition is very inefficient. This problem can arise for example for hardware tasks whose underlying circuits have a lot of feedback or if the circuits are densely connected. For these cases, temporal partitioning is used for further decomposing the hardware task.

The final application thus combines virtual execution with temporal partitioning. A run-time system on the CPU core runs a task scheduler that implements the coordination model that was used for the application specification.

### 5.2.2 Virtualized Execution

Virtualized execution mainly asks for two architectural features: a basic unit of computation (called operator, or hardware page), and a mechanism for communication between these operators.

In the Zippy architecture, a configuration of the reconfigurable array can be treated as the operator. The multi-context support can be used to hold several operators concurrently on-chip, without reloading them. Since each context has a dedicated set of registers, the contexts can operate without interfering with each other.

Since all configurations have access to the FIFOs on the RPU, these FIFOs can be used to implement the communication between the oper-

ators. Hence, the Zippy architecture is appropriate to implement applications that use a coordination model based on tasks communicating via FIFOs, for example, applications specified as process network (compare Sec. 5.1.3).

Virtualized execution works at the task level and sequencing between configurations is required rather infrequently. Thus, context switching and sequencing performance is less critical as in the case of temporal partitioning where a context switch is required in every cycle. Hence, the runtime system can be implemented in software on the CPU core and a dedicated context sequencer is not strictly required.

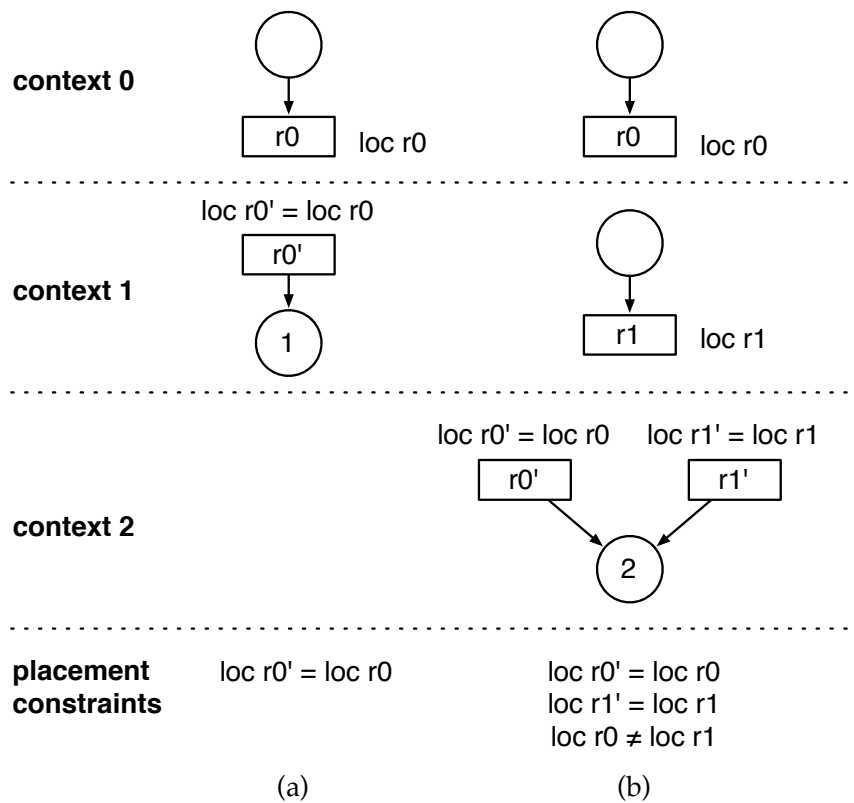
### 5.2.3 Temporal Partitioning

An efficient implementation of temporal partitioned circuits demands for a modified hardware implementation tool-flow and for additional dedicated hardware. The key requirements are: 1) fast switching between configurations, 2) fast cyclic sequencing of a pre-defined sequence of configurations, and 3) efficient and fast communication between contexts. Dedicated hardware support for all of these requirements has been incorporated in the Zippy architecture:

1. Fast switching between configurations is enabled by the design as a multi-context architecture (see Section 2.3.6). Zippy allows for storing several configurations concurrently on-chip. After the configurations have been downloaded once, they can be activated quickly, within a single clock cycle.
2. The repeated cyclic sequencing of all temporal partitions is implemented by the *temporal partitioning sequencer* (see Section 2.3.4). The temporal partitioning sequencer allows for switching between the individual configurations in every cycle without any time overhead. This sequencer is essential to leverage the capability of fast contexts switches provided by a multi-context architecture.
3. Efficient and fast communication between contexts is implemented with the input and output register files in the cells of the reconfigurable array (see Section 2.3.1). Each register file provides a dedicated register per context. This register is the target for register write operations. Read operations on an input or output register file can access also register contents written in other contexts, hence the register files can not only implement the actual delay registers (flip-flops) of the circuit, but can be also used for inter-context communication.

Using the register files for inter-context communications introduces additional placement constraints which are illustrated with Figure 22:

- Figure 22(a) shows a situation where a value is generated in context



**Fig. 22:** Inter-Context Communication via Register-Files in the Zippy Architecture: Placement Constraints

0 and is stored in register  $r0$ . Register  $r0$  is placed to an arbitrary location, denoted by  $\text{loc } r0'$ . Operator 1 in context 1 wants to read this register. To this end, we use a register  $r0'$  that acts as a proxy for  $r0$ . Consequently, register  $r0'$  must be placed to the same location as  $r0$ , and the corresponding route to operator 1 has to be established.

Hence, we need a new type of placement constraint, that links the placement of registers in context 1 to the placement in context 0. The constraint  $\text{loc } r0' = \text{loc } r0$  expresses that  $r0$  can be placed arbitrarily, but  $r0'$  must be placed to the same location.

- Figure 22(b) illustrates that an additional constraint is needed if an operator read registers from more than one context. Operator 2 in context 2 reads values that have been generated in context 0 and context 1. The placement constraints for  $r0'$  and  $r1'$  are analogous to the situation in Fig. 22(a), but an additional constraint on the placement of  $r0$  and  $r1$  is generated. Since both,  $r0'$  and  $r1'$  must be accessed in context 2, they cannot be stored in the same output register file, because only one output register can be read concurrently. Hence, we need to prevent that  $r0'$  and  $r1'$  (and consequently  $r0$  and  $r1$ ) are mapped to the same output register file. Adding an additional placement constraint  $\text{loc } r0 \neq \text{loc } r1$  avoids this situation.

These new placement constraints fundamentally differ from the existing placement constraints, since they apply not to a single context. The new constraints link the placement of the whole set of contexts, thus the placement in one context affects the placement in all other contexts.

For supporting these new placement constraints, the placement and routing process needs to be extended to work on a set of contexts instead of a single context only. A feasible implementation for all configurations can be found for example with backtracking search. After placement and routing of a randomly chosen start configuration, the implied placement constraints are propagated to the other configurations, which are subsequently implemented one by one. If no feasible implementation can be found, the search backtracks and a new search iteration is started.

We have not extended the Zippy toolflow to support these additional constraints yet. For the case-study on temporal partitioning (see Sec. 6.2) we have manually specified a fixed placement for the inter-context communication registers.

## 5.3 A Novel Method for Optimal Temporal Partitioning

This section presents a novel method for temporal partitioning of sequential circuits on reconfigurable architectures. To this end, we formally define temporal partitioning as an optimization problem. An optimal solution to the temporal partitioning problem maximizes the circuit's performance during execution while restricting the size of the partitions to respect the resource constraints of the reconfigurable architecture.

We adopt an architecture model which is compatible to the architecture models used in related work on temporal partitioning for fine-grained reconfigurable multi-context architectures. The results of this work can be directly applied not only to coarse-grained but also to fine-grained architectures.

### 5.3.1 Outline of the Method

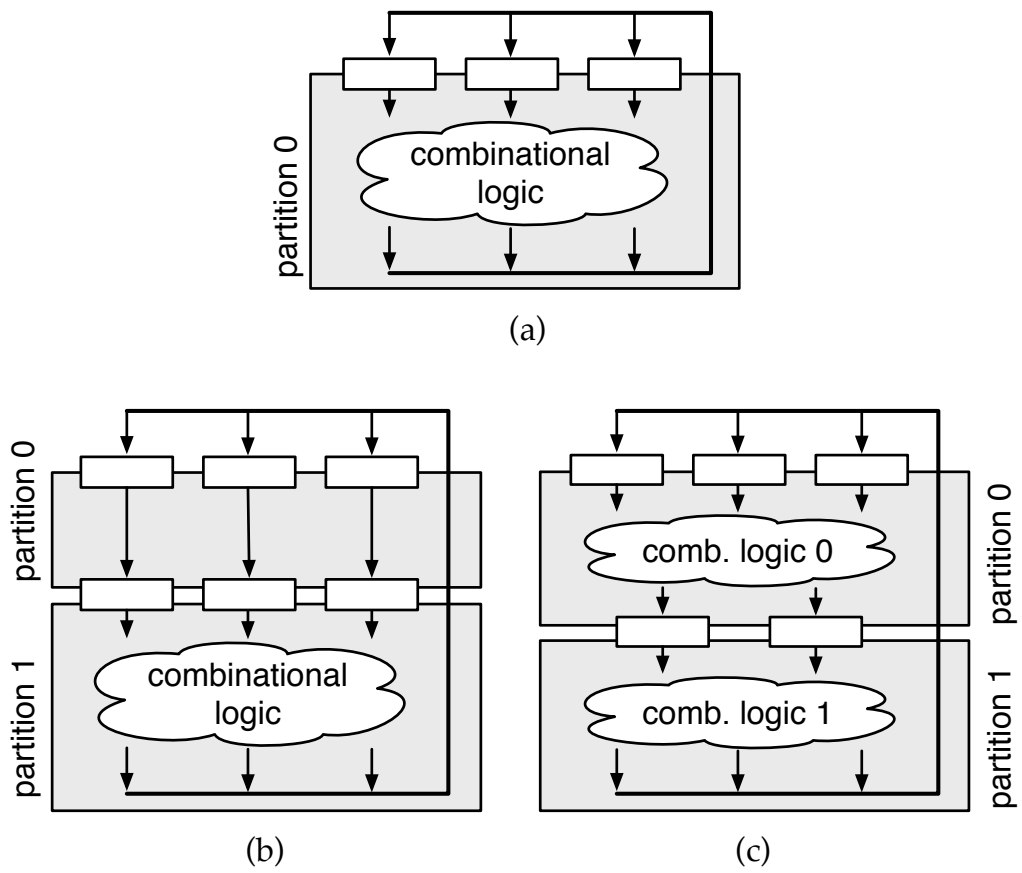
Our method uses two digital design techniques known as *slowdown* and *retiming* for generating temporal partitions. Figure 23 illustrates the basic idea of the method with an example in which a circuit is partitioned into two partitions. Fig. 23(a) illustrates the implementation of a generic synchronous circuit in a single partition. All signals originate at a register and propagate through purely combinational logic that computes the new contents of the registers. We assume that the inputs and outputs of the circuit are registered and treat reading a primary input as reading from an input register. Writing a primary output is treated as writing to the output register.

Fig. 23(b) illustrates how a 2-slow transformation of the initial circuit is obtained by replacing each register with a sequence of two registers. A 2-slow circuit performs the same operation as the initial circuit, but works at 1/2 of the data-rate, that is, each operator works only in every 2nd clock cycle. We interpret this 2-slow circuit as a special case of a partitioning into 2 partitions, where partition 0 is empty. The data-transfer between partitions is implemented with registers. Since partition 0 is empty, it simply forwards the unmodified register contents.

To balance the size of the partitions we use *retiming*. Retiming redistributes registers in a sequential circuit such that the critical path is minimized while the circuit's functionality is preserved. Fig. 23(c) illustrates how the combinational logic is distributed into two partitions. We apply additional constraints to the retiming process, to ensure that the partitions satisfy the resource constraints of the architecture.

The resulting circuit partitions are executed on a reconfigurable architecture in a time-multiplexed way and perform the same function as the





**Fig. 23:** Concept of Temporal Partitioning with Slowdown and Retiming

initial circuit (at 1/2 of the data-rate). After executing the first partition for a single cycle, the other partition is executed for a single cycle, and this schedule is continuously repeated. The frequent reconfigurations demand for a reconfigurable architecture with fast reconfiguration, for example, a multi-context architecture.

### 5.3.2 Models

In this section we introduce the architecture and circuit models this work bases on.

#### Architecture model

We use an architecture model that is an abstract variant of Trimbergers time-multiplexed FPGA (TMFPGA) architecture [Tri98]. TMFPGA is one of the first multi-context FPGA architectures and its architecture is the basis for many papers studying temporal partitioning on multi-context FPGAs (see Section 5.3.8). Although Zippy is a coarse-grained architecture, its reconfigurable array bears sufficient similarities with the TMFPGA architecture to use the same architecture model.

Our architecture model makes the following key assumptions:

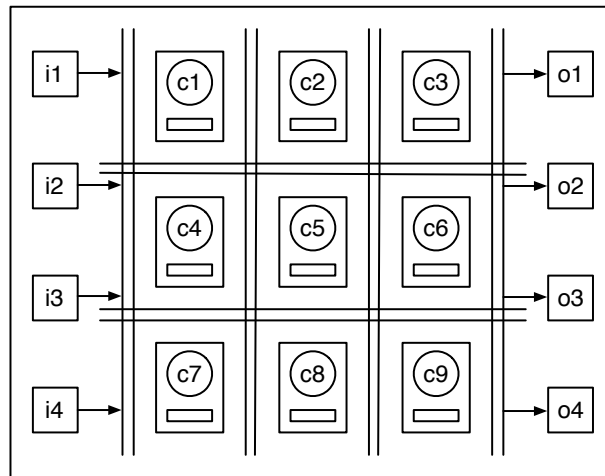
1. **Array model:** The model assumes a reconfigurable architecture with an array of uniform reconfigurable cells. The cells are connected by a programmable interconnection network which approximates full crossbar connectivity sufficiently well, such that any circuit of interest can be implemented without placement and routing congestion problems<sup>1</sup>. A schematic drawing of the architecture is presented in Fig. 24

The architecture is a multi-context architecture, i.e., several configurations can be stored on-chip concurrently. Switching from one configuration to another happens without latency (single-cycle context switch) and is controlled by a context sequencer. At runtime, the context sequencer activates each context, executes it for a single cycle, and then proceeds to the next context in the sequence. This schedule is cyclically repeated.

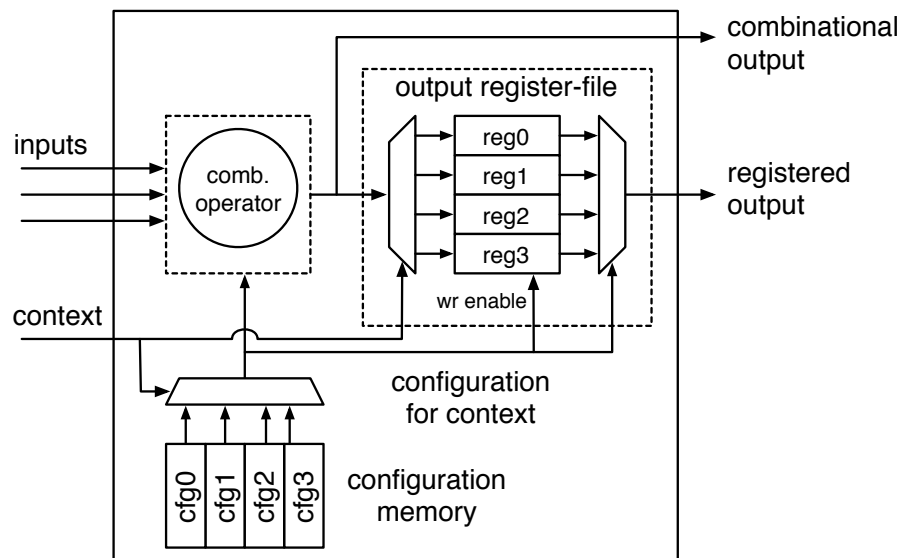
2. **Cell model:** We assume that a cell of the reconfigurable array is composed of two main components: the combinational operator, and the output register-file. A schematic view of a cell is shown in Figure 25.

---

<sup>1</sup>If the implementation on a concrete architecture with sparse routing resources leads to congestion, the maximum allowable cell utilization can be reduced from 100% to a smaller value to reduce routing congestion.



**Fig. 24:** Array Model for Temporal Partitioning. The illustration shows an architecture with  $3 \times 3$  cells, 4 input ports, and 4 output ports.



**Fig. 25:** Schematic of the cell model. The illustration shows a 3 input cell in a 4 context architecture.

The number of cell inputs is not restricted. The operator can perform an arbitrary combinational function on the inputs. The function of the cell is determined by the configuration memory. Since the architecture is a multi-context device, the relevant configuration is selected according to the currently activated context, which is an additional input signal to the cell.

The cell's storage element is implemented in the output register file. This register file is modelled after the storage element of the TMFPGA architecture [Tri98] and provides a dedicated register per context. The association of context and register is hardwired and the output of the operator block is stored in the associated register (if the register's write enable is asserted).

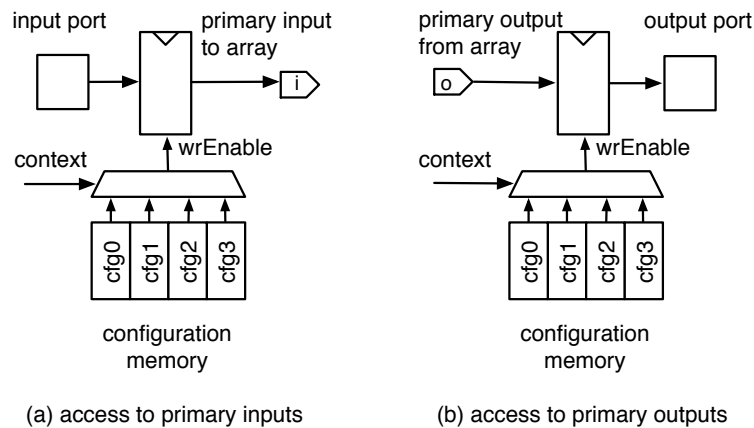
The cell has two outputs that can be read via the interconnection network. The combinational output provides direct access to output of the operator block. The registered output provides access to the content of one of the registers in the output register file. The selection of the output register is programmable and is part of the configuration.

The registered output allows for reading the register contents stored during the execution of a different context and hence enables inter-context communication. We use this communication mechanism for implementing the communication between the temporal partitions.

3. **Input/Output model:** We assume each access of the reconfigurable array to the physical inputs and outputs is registered. A configurable write enable signal determines, when the register contents is updated, see Fig. 26. Registered inputs and outputs are essential for temporal partitioned execution: The registers are used for holding the inputs and outputs of the array constant during each iteration while all partitions are executed once.

### Circuit model

The input to the temporal partitioning algorithm is a synchronous digital circuit. The circuit is specified as a netlist consisting of combinational operators, registers, primary inputs and primary outputs. We assume that the netlist is technology mapped, i. e., all operators can be directly implemented by a cell of the reconfigurable array. Since the circuit is synchronous each cycle must contain at least one register.



**Fig. 26:** Registered access to the physical input and output ports of the reconfigurable array. The illustration shows the input and output ports in a 4 context architecture.

### 5.3.3 Basic Problem Formulation

This work is founded on two digital design techniques called *retiming* and *slowdown* which have been introduced by Leiserson and Saxe in [LS91].

*Retiming* optimizes the placement of registers in a circuit to minimize the length of the critical path.

The second fundamental technique used in this work is called *slowdown*. A  $P$ -slow implementation of a circuit is obtained by replacing each register in the initial circuit with a sequence of  $P$  registers. The most interesting property for the purpose of temporal partitioning is that the  $P$ -slow transformation decomposes the circuit into  $P$  completely independent partitions. Each operators needs to run only every  $P$ -th cycle and the inputs and outputs are evaluated also only every  $P$ -th cycle. These properties are also preserved if the slowed down circuit is additionally retimed to minimize the critical path.

We use this combination of slowdown and retiming to generate optimal temporal partitions of a sequential circuit: First, we partition the initial circuit into  $P$  partitions with a  $P$ -slow transformation. In a subsequent step, we redistribute the circuit's registers with a modified retiming process that creates partitions while respecting the resource constraints of the architecture and optimizing the critical path of all partitions concurrently. Additional constraints model the increased resource demand due to temporal partitioning, e. g., inter-context communication resources.

#### Retiming

Retiming [LS91] minimizes the critical path in a digital circuit by adding and removing registers while it ensures that the functional behavior of the

circuit is preserved. Retiming uses a graph representation of the circuit's netlist. A circuit  $G = (V, E, w, d)$  consists of a set of vertices  $V$  and a set of edges  $E$ . The vertices  $v \in V$  model the operators in the circuit. Each operator performs a combinational function and is annotated with its propagation delay  $d(v)$ . Directed edges model the connections between operators. We use the notation  $u \xrightarrow{e} v$  to denote an edge  $e$  that connects operators  $u$  and  $v$ . Each edge is weighted with its register count  $w(e)$  which represents the number of registers along this path in the circuit. The graph is a multi-graph, i. e., two vertices can be connected by several edges with different weights. Primary inputs and outputs are modeled as ordinary vertices with a propagation delay of 0.

A simple path  $p$  in graph  $G$  is defined as a sequence of vertices connected by edges, such that no vertex is contained twice:

$$p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} v_k$$

The path weight  $w(p)$  is defined as the total number of registers along a path  $p$ :

$$w(p) = \sum_{i=0}^{k-1} w(e_i) \quad (5.1)$$

Similarly, the path delay  $d(p)$  is defined as the sum of the propagation delays of all operators along a path  $p$ :

$$d(p) = \sum_{i=0}^k d(v_i) \quad (5.2)$$

To restrict the set of graphs to graphs which have a physical implementation as synchronous digital circuits, we require that the propagation delay of all nodes is non-negative ( $d(v) \geq 0$ ) and that the edge-weight is also non-negative ( $w(e) \geq 0$ ). Further, combinational feedback loops are prohibited, i. e., the path weight of any cycle in  $G$  is strictly positive.

The minimum feasible clock period is a metric for the performance of a circuit and is bounded by the longest purely combinational path in the circuit, which is denoted as *critical path*  $\Phi(G)$ :

$$\Phi(G) = \max \{d(p) \mid w(p) = 0\} \quad (5.3)$$

Retiming computes a vertex labeling that assigns each vertex  $v_i$  a retiming value  $r(v_i)$ . The retimed circuit has the same structure as the original circuit, but the edges  $u \xrightarrow{e} v$  have a different weight  $w_r(e)$  which is defined as:

$$w_r(e) = w(e) + r(u) - r(v) \quad (5.4)$$

Let  $c_{\max}$  be a positive real number that represents an upper bound on the length of the critical path of the retimed circuit. For example,  $c_{\max}$  can be set to the length of the critical path of the initial circuit. Leiserson and Saxe show [LS91], that there exists a retiming  $G_r$  of  $G$  such that the critical path  $\Phi(G_r) \leq c$  if and only if the Mixed-Integer Linear Program (MILP) given in Equations 5.5–5.8 has a solution. The variables  $s(v)$  are auxiliary variables. For all operators  $v \in V$  and edges  $(u \xrightarrow{e} v) \in E$ , minimize  $c$  such that:

$$r(u) - r(v) \leq w(e) \cdot P \quad (5.5)$$

$$s(v) \geq d(v) \quad (5.6)$$

$$s(v) \leq c \quad (5.7)$$

$$r(u) - r(v) + \frac{s(u) - s(v)}{c_{\max}} \leq P \cdot w(e) - \frac{d(v)}{c_{\max}} \quad (5.8)$$

where:

$$s(v) \in \mathbf{R}, 0 < s(v) < c_{\max}, r(v) \in \{0, \dots, P - 1\}$$

We treat  $c$  as the optimization objective that is minimized. The solution to the optimization problem defined by Equations 5.5–5.8 is a circuit implementation that maximizes performance, i. e., the critical path length is minimal:  $\Phi(G_r) = c$ .

### Slowdown

In a  $P$ -slow circuit each register is replaced by a sequence of  $P$  registers. Hence a  $P$ -slow transformation of  $G(V, E, w, d)$  is obtained by multiplying the edge weights by  $P$ , i. e.,  $G_s = G(V, E, P \cdot w, d)$ . The most interesting property for the purpose of temporal partitioning is that the  $P$ -slow transformation decomposes the circuit into  $P$  completely independent partitions. Each operator needs to run only every  $P$ -th cycle and the inputs and outputs are evaluated also only every  $P$ -th cycle. These properties are also preserved if the slowed-down circuit is additionally retimed to minimize the critical path.

We have extended the Mixed Integer Linear Program (MILP) formulation of retiming to support combined slowdown and retiming by multiplying the edge weight with the slowdown factor  $P$  in Equations 5.5 and 5.8, respectively.

### Partitioned Execution and Performance

The temporal partitioning process defines the mapping of operators to individual partitions. The partition to which a node  $v$  is mapped directly

corresponds to the node's retiming value  $r(v)$ , as determined by the retiming process. While the unconstrained retiming process (Equations 5.5–5.8) maximizes performance, it does not consider resource constraints. We apply further constraints to the  $r(v)$  values to limit the number of operators that can be mapped to a particular partition and to model resource requirements imposed by inter-partition communication. These additional constraints are discussed in Sec. 5.3.4.

All operators in the same partition define a sub-netlist which is implemented in one configuration of the reconfigurable architecture. At runtime, these contexts are executed in the order  $c_0, c_1, \dots, c_{P-1}$ , each for a single cycle.

Since the temporal partitioned circuit is  $P$ -slow, all  $P$  contexts must be executed once to perform the same computation as the initial circuit computes in a single cycle. Consequently, the performance of the temporal partitioned circuit is lower than the performance of the initial circuit. Practical considerations demand that each context is executed for the same cycle time, that is bounded by the critical path  $\Phi(G_r) = c$  of the retimed circuit. The performance of the temporally partitioned execution relative to the execution of the initial circuit is thus given as:

$$\text{rel. performance} = \frac{\Phi(G)}{\Phi(G_r) \cdot P} \quad (5.9)$$

Since  $\Phi(G)$  can be considered a constant property of the initial circuit, optimal temporal partitioning should thus minimize the nominator of Equation 5.9.

The objective function  $\Phi(G_r) \cdot P$  is non-linear in  $P$  and hence cannot be directly optimized by an MILP solver. But since  $P$  is bounded by the number of physical contexts in a concrete reconfigurable architecture, an optimal value for  $P$  can be determined by exhaustive or binary search, i. e., solving the MILP for all possible values of  $P$ .

### 5.3.4 Resource Constraints

This section introduces the resources constraints of the execution architecture and their mathematical modelling.

The solution to the MILP optimization problem defined by Eq. 5.5–5.8 determines the optimal partitioning of a circuit, but doesn't consider any resource constraint. For finding an implementation that satisfies the resource constraints of the execution architecture while concurrently maximizing the performance, we need to model the resource demand.

We introduce a binary variable  $x_{i,p}$  per operator and partition to for-



ulate mapping, capacity and communication constraints:

$$x_{i,p} = \begin{cases} 1 & \text{if node } v_i \text{ is mapped to partition } p \\ 0 & \text{otherwise} \end{cases}$$

Eq. 5.10 defines that each operator must be mapped to exactly one partition, and the set of partitions to which an operator can be mapped to is restricted to  $r_i \in \{0, \dots, P-1\}$  in Eq. 5.11. The value  $r_i$  denotes, to which partition an operator is mapped.

$$\sum_{p=0}^{P-1} x_{i,p} = 1, \quad \forall v_i \in V \quad (5.10)$$

$$r_i = \sum_{p=0}^{P-1} p \cdot x_{i,p} = 1, \quad \forall v_i \in V \quad (5.11)$$

$$x_{i,p} \in \{0, 1\}, \quad r_i \in \{0, \dots, P-1\}$$

The implementation of a partition in a context of the architecture requires resources for implementing the operator nodes and additional resources for inter-context communication. We account for these costs by defining two *cost* variables: *nodecost* and *comcost*.

While we treat combinational operators the same as primary inputs and outputs in the retiming process, we need to differentiate between them for computing the resource demand. We divide the set of operators into three subsets that represent the combinational operators ( $V_{op}$ ), the primary inputs ( $V_{pi}$ ), and the primary outputs ( $V_{po}$ ).

### Operator Resource Demand

We define  $nodecost_{i,p}$  as the resource demand caused in partition  $p$  by the operator  $v_i$ . For this work, we assume unit cost, i. e., the implementation of each combination operator causes a resource demand of  $resop_i = 1$ :

$$nodecost_{i,p} = resop_i \cdot x_{i,p} = x_{i,p} \quad \forall v \in V_{op} \quad (5.12)$$

We assume, that there are sufficient primary input and output nodes. Since these dedicated nodes are distinct from the cells and thus do not reduce the capacity for implementing combinational operators they do not cause any cost:

$$nodecost_{i,p} = 0 \quad \forall v \in \{V_{pi} \cup V_{po}\} \quad (5.13)$$

We assume, that the reconfigurable architecture provides  $K$  cells and thus can accommodate  $K$  combinational operators. We limit the number

of combinational operators mapped to each partition with the following constraint:

$$\forall p \in \{0, \dots, P-1\} : \sum_{v_i \in V} \text{nodecost}_{i,p} \leq K \quad (5.14)$$

### Inter-Context Communication Resource Demand

The cell's output register-files are used for inter-context communication. Since each cell has a dedicated register-file, the combinational output of a cell can be stored in the associated register without affecting the resource supply of the array.

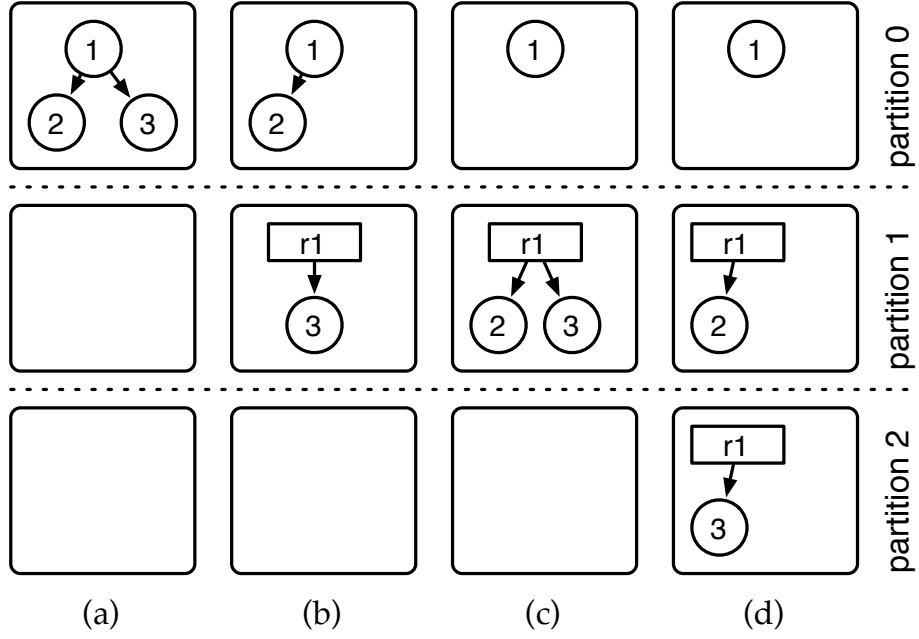
Assume operator  $v_1$  is mapped to partition  $p_1$  and its output defines the value of net  $n_1$ . If operator  $v_2$  wants to read net  $n_1$  from a different partition, it needs to access the output register file of  $v_1$  where the intermediate value is stored. Since only one register of the register-file can be read at once, the number of intermediate signals that can be read in a context is bounded by the number of cells. This forms a resource constraint. Additionally, the data transfers via the register-files also introduce cell placement constraints which have been discussed in Section 5.2.3.

As soon as an operator in a context reads an output register, the register content becomes local to the partition and no additional resources are required if other operators in the same partition read the same register too. Hence, at most one communication register is inserted per net and partition.

Figure 27 illustrates the use of inter-context communication resources for a simple circuit of 3 nodes. No communication resources are required, if all operators are mapped to the same partition, see Fig. ??(a). In Fig. 27(b) a communication resource  $r1$  is used in partition 1 to allow operator 3 reading the net defined by node 1 in partition 0. Fig. 27(c) shows that several operators can read the same net. A communication resource is needed in each partition that reads a net defined in a different partition, hence in Fig. 27(d) a communication resource is required in partition 1 and in partition 2.

We denote the communication resource cost that is caused in partition  $p$  due to accessing net  $i$  (defined by operator  $v_i$ ) as  $\text{comcost}_{i,p} \in \mathbf{N}_0$ .

For modelling  $\text{comcost}_{i,p}$  we introduce two binary decision variables *defined* and *used*. Each net  $n_i$  is defined by its source  $v_i$ . Thus,  $\text{defined}_{i,p}$  is



**Fig. 27:** Communication registers for inter-context communication

identical with  $x_{i,p}$ :

$$\begin{aligned} \text{defined}_{i,p} = x_{i,p} &= \begin{cases} 1 & \text{if net } n_i \text{ is defined in partition } p \\ 0 & \text{otherwise} \end{cases} \\ \text{used}_{i,p} &= \begin{cases} 1 & \text{if net } n_i \text{ is used in partition } p \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

For determining the variable  $\text{used}$ , we define an auxiliary variable  $\text{netuses}_{i,p}$  (Eq. 5.15) that sums up how many times net  $n_i$  is used in partition  $p$ . The index of the summation runs over all sinks of net  $n_i$ .

$$\text{netuses}_{i,p} = \sum_{j \in \text{sinks}(v_i)} x_{j,p} \quad (5.15)$$

Now we can define  $\text{used}$  with the following non-linear comparison:

$$\text{used}_{i,p} = \begin{cases} 1 & \text{if } \text{netuses}_{i,p} > 0 \\ 0 & \text{otherwise} \end{cases} \quad (5.16)$$

Using standard linearization techniques, the expression of Eq. 5.16 is translated to the linear constraints shown in Eq. 5.17–5.18.

$$\text{netuses}_{i,p} - \text{used}_{i,p} \cdot G_{\max} \leq 0 \quad (5.17)$$

$$-\text{netuses}_{i,p} - (1 - \text{used}_{i,p}) \cdot G_{\max} < 0 \quad (5.18)$$

$$netuses_{i,p} \in \{0, \dots, G_{max}\}, used_{i,p} \in \{0, 1\}$$

Eq. 5.17–5.18 need an upper bound for  $netuses$  for which we use the maximum degree of the circuit's graph  $G_{max}$ :

$$G_{max} = \max_{\forall v_i \in V} |sinks(v_i)| \quad (5.19)$$

Using the  $used$  and  $defined$  variables we can now define a lower bound on  $comcost_{i,p}$  as follows:

$$comcost_{i,p} \geq used_{i,p} - defined_{i,p} \quad \forall v_i \in V_{op} \quad (5.20)$$

$$comcost_{i,p} = 0 \quad \forall v_i \in \{V_{pi} \cup V_{po}\} \quad (5.21)$$

Since the reconfigurable architecture provides  $K$  cells, a partition can read at most  $K$  output register-files. We formulate the resource constraint on the communication resources as:

$$\forall p \in \{0, \dots, P-1\} : \sum_{\forall v_i \in V} comcost_{i,p} \leq K \quad (5.22)$$

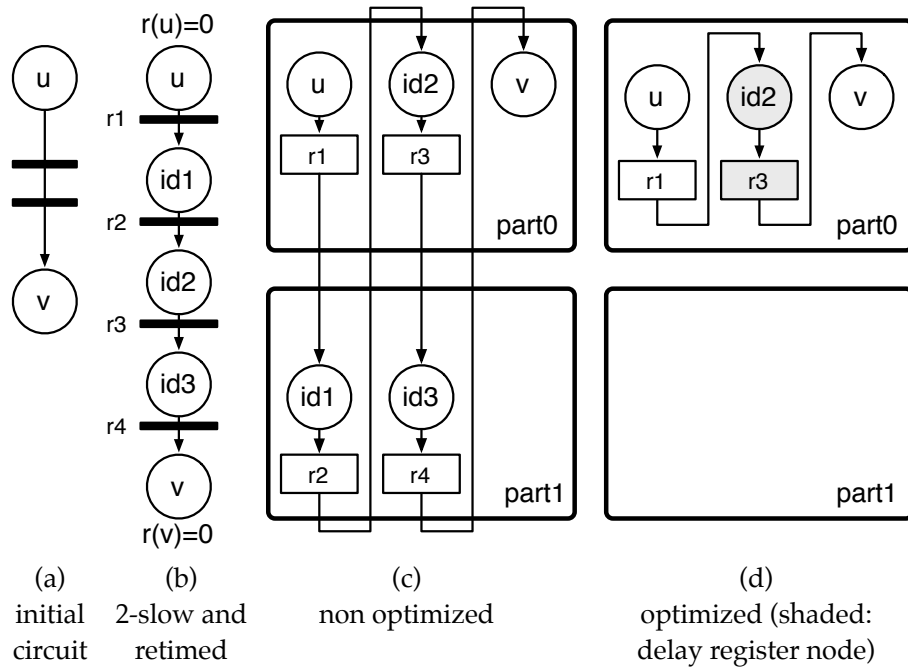
Note, that  $comcost_{i,p}$  as defined by Eq. 5.20 is a lower bound on the communication resource cost, rather than the actual cost. We prefer this approach, since formulating the variable  $comcost_{i,p}$  as lower bound is easier than computing the actual cost, but serves the MILP problem formulation equally well.

Consider the example from Fig. 27(b):

- Eq. 5.20 defines  $comcost_{1,0} \geq -1$ , but since  $comcost$  is non-negative, the MILP solver will use the solution  $comcost_{1,0} = 0$ .
- Actually, only one communication resource is required for accessing net 1 in partition 1. But Eq. 5.20 defines:  $comcost_{1,1} \geq 1$ , thus  $comcost$  is not bounded. However, due to the resource constraint in Eq. 5.22 an excessive usage of communication resources will prohibit a successful implementation of the circuit. Hence, the optimization process will push  $comcost$  towards the lower bound and the MILP solver will use the solution  $comcost_{1,1} = 1$  if resources are becoming scarce in the optimization process.

### Delay Register Nodes

Slowdown and retiming can generate edges with more than  $P$  registers. Applying a straight-forward assignment of operators to partitions can lead to inefficient mappings for such cases. As an optimization step, we introduce *delay registers* which, in turn, need to be included in the cost model. We illustrate the need for delay registers with the example of Fig. 28.



**Fig. 28:** Delay Register Nodes

Suppose the circuit in Fig. 28(a) is part of a larger circuit, that is partitioned into two partitions. The circuit comprises two operators connected by two registers. Applying 2-slowdown and retiming transformations to the overall circuit maps nodes  $u$  and  $v$  to the same partition ( $r(u) = r(v) = 0$ ) and increases the number of registers between  $u$  and  $v$  to 4. Fig. 28(b) explicitly shows identity operators (id1, id2, id3) between the registers to illustrate, that the registers are always coupled with an operator in our cell model. According to our execution model, each register marks the boundary of a partition. Fig. 28(c) illustrates the non-optimized implementation on a two context architecture. The identity operator nodes pass the unmodified values to the output registers, which are then used for inter-context communication. This implementation is suboptimal, since the cells in partition 1 do not implement any function. Note, that registers  $r2$  and  $r4$  do not introduce additional delay, but only route the outputs of  $r1/r3$  to partition 1 and back again.

An optimized implementation is presented in Fig. 28(d). The cells in partition 1 have been removed and the output of register  $r1$  is directly fed to the cell that implements (id2/ $r3$ ) and finally to operator  $v$ . We denote the cell that implements (id2/ $r3$ ) as *delay register node*. This optimization is applicable whenever the circuit has edges with more than  $P$  registers (after slowdown and retiming).

We can handle delay registers in the MILP formulation in two ways:

Either the use of delay register nodes is avoided by adding further constraints to the MILP, or the demand for delay registers is explicitly modelled and the resource constraints are extended to incorporate the delay register resource demand.

Delay register nodes can be avoided by restricting the number of registers on any edge to a maximum of  $P$  registers. The number of registers on an edge  $e = (u \xrightarrow{e} v)$  in the partitioned circuit is computed as  $\text{reg}(e) = P \cdot w(e) + r(v) - r(u)$ . Hence, delay registers are avoided by adding the following constraint to the MILP:

$$\forall e = (u \xrightarrow{e} v) \in E : P \cdot w(e) + r(v) - r(u) \leq P \quad (5.23)$$

To ensure, that solutions without delay register exists, we replace any edge  $e = (u \xrightarrow{e} v)$  in the initial circuit that has more then 1 register ( $w(e) > 1$ ) with a sequence of  $w(e)$  identity operators, each followed by a single register. This transformation ensures at least one solution without delay registers exists, namely, mapping all operators to a single partition.

The drawback of using this additional constraint is that it constrains the solution space for the temporal partitioning problem, which could result in sub-optimal performance.

If we allow the use of delay registers, the number of delay registers due to an edge  $u \xrightarrow{e} v$ ,  $r(u) = r(v) = p$  in partition  $p$  is given as :

$$\text{delayreg} = \left\lfloor \frac{P \cdot w(e) + r(v) - r(u)}{P} \right\rfloor \quad (5.24)$$

Hence, we can define the delay register node cost in partition  $p$  due to accessing net  $n_i$  (defined by operator  $v_i$ ) as:

$$\forall \text{ edges } u \xrightarrow{e} v, \forall p = 0, \dots, P - 1 : \\ dregcost_{i,p} \geq \underbrace{\left( \frac{P \cdot w(e) + r(v) - r(u)}{P} - 1 \right)}_{\text{delayreg}} - (1 - x_{j,p}) \cdot D_{\max}$$

We convert the non-linear truncating operation in Eq. 5.24 to a linear constraint by using a lower bound on delayreg. The second term ensures, that  $dregcost_{i,p}$  is only attributed to partition  $p$  if the sink  $v_j$  of net  $n_i$  is mapped to partition  $p$ . Otherwise, a large constant  $D_{\max}$  is subtracted and makes the constraint non-binding.

To account for the delay register resource demand, Eq. 5.14 must be extended to include  $dregcost_{i,p}$ . Since the delay registers may also require communication resources, the communication resource constraint (Eq. 5.22) must be adapted accordingly.

### 5.3.5 Solving the Temporal Partitioning MILP

We have implemented a tool flow that generates the MILP formulation for the temporal partitioning problem from the circuit's netlist and the architecture parameters (number of partitions, context capacity). The resulting MILP problem is solved with the CPLEX [CPL] optimizer.

The resulting temporal partitions are implemented on the reconfigurable architecture using the architecture specific placement and routing tools. As shown in Sec. 5.2.3 inter-context communication implies placement constraints for the circuits in the individual context. While we have automated the generation of the temporal partitions, we have not automated the placement constraint generation and implementation process yet.

### 5.3.6 Extension to Functional Pipelining

Constraints Eq. 5.10 and Eq. 5.11 restrict the partition to which an operator can be mapped to the interval  $r(v) \in \{0, \dots, P-1\}$ . This restriction ensures, that retiming does not change the delay of the circuit's outputs relative to the inputs. If we allow for introducing additional delay, the same retiming formalism can be also used to compute a circuit implementation that uses functional pipelining with a potentially even shorter critical path.

To this end, we can extend the MILP for optimal temporal partitioning analogously to the formulation used by De Micheli for scheduling cyclic sequencing graphs with functional pipelining (cf. Sec. 5.6.2 in [DM94]). With this extension, the operator's retiming values  $r(v)$  are not restricted to the number of partitions anymore, but are bounded by a delay bound  $L$ , i. e.,  $r(v) \in \{0, \dots, L\}$ . The mapping constraints Eq. 5.10 and Eq. 5.11 need to be adapted accordingly.

The partition  $p(v)$  to which an operator  $v$  is mapped is defined as  $p(v) = r(v) \bmod P$ . Partition  $p(v)$  thus implements all operators with retiming values  $r(v) = i + k \cdot P \mid 0 \leq i + k \cdot P \leq L, k \in \mathbf{N}_0$ , and the capacity constraints Eq. 5.14 and Eq. 5.22 must be extended accordingly.

### 5.3.7 Example

We illustrate our optimal temporal partitioning technique with the example of a 2nd-order Infinite Impulse Response (IIR) filter. IIR filters are basic building blocks in many digital signal-processing applications. Fig. 29(a) shows the schematics of the IIR2 circuit after a 2-slow transformation. The circuit is composed of 10 combinational operators (op1, ..., op10) and  $5 \times 2$  registers. The arithmetic functions of the operators are irrelevant for the temporal partitioning process and are thus not further detailed in the figure. We preprocess the circuit's netlist by replacing edges with

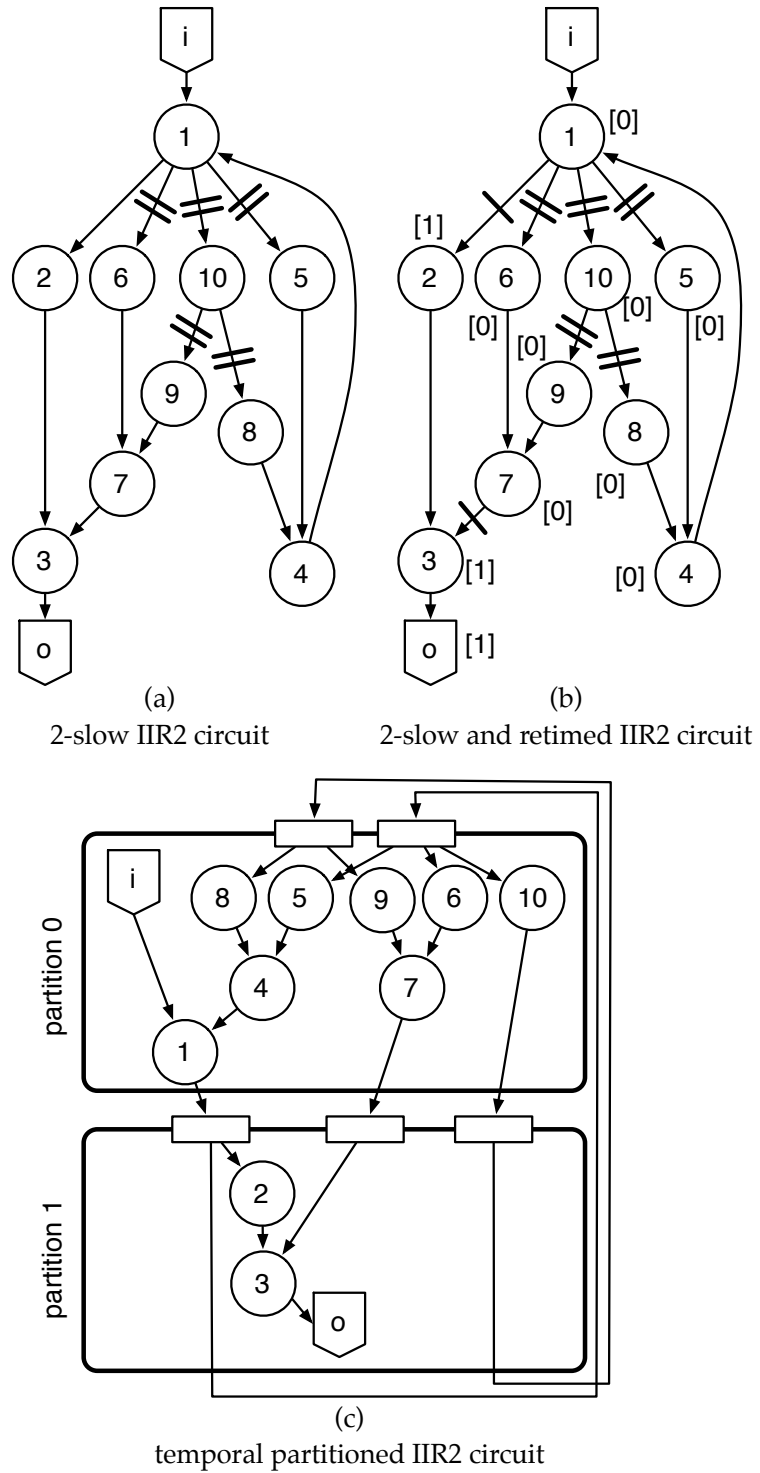


Fig. 29: Temporal Partitioning: Case Study IIR2 Filter



multiple registers by a sequence pass-through operators followed by single registers and we avoid delay registers by employing the constraints defined in Eq. 5.23.

We solve the MILP defined in Sec. 5.3.3 and Sec. 5.3.4 with the following parameters: context capacity  $K = 8$ , number of partitions  $P = 2$ , and upper bound for the clock period  $c_{\max} = 10$ . Solving the MILP with CPLEX takes less than 1ms on a 900 MHz SunFire280R machine. The retiming values for the nodes are annotated in brackets in Fig. ??(b). The registers in this figure already reflect the modified register placement due to retiming. The temporally partitioned implementation on a two context architecture is shown in Fig. 29(c).

### 5.3.8 Related Work and Discussion

In this section we will discuss related work in temporal partition of sequential circuits and we will compare these approaches to our approach.

#### Related Work

The development of multi-context FPGAs, such as [TCJW97, DeH94] enabled rapid dynamic reconfiguration which is a requirement for efficient temporal partitioning. Initial work on temporal partitioning for sequential circuits on time-multiplexed FPGAs was published Chang and Marek-Sadowska and Trimberger.

Chang and Marek-Sadowska present extensive work on temporal partitioning of sequential circuits on time-multiplexed FPGAs. The algorithm in [CMS97] bases on critical path scheduling, a variant of list-scheduling. After list-scheduling, the schedule is iteratively optimized by local optimizations. In later work [CMS98, CMS99] the authors presents a generic model for time-multiplexed communicating logic and introduce a scheduling algorithm based on force-directed scheduling. All three papers strongly focus on minimizing the buffers needed for inter-context communication. The need for minimizing the buffers is motivated by targeting the Dharma and DPGA architectures, where inter-context communication is expensive.

Trimberger proposes a list-scheduling based method [Tri98] for temporal partitioning of circuits for the execution on his TMFPGA architecture [TCJW97]. Unlike to the work of Chang and Marek-Sadowska, the reduction of inter-context communication resources is not that important in the TMFPGA architecture, because each reconfigurable cell contains a dedicated inter-context communication register-file. After publication of Trimberger's papers, most authors started to adopt the TMFPGA architecture as the standard architecture model for further work on temporal

partitioning.

Trimberger's list-scheduling algorithm computes the ASAP and ALAP scheduling times for all operators. The algorithm schedules the nodes such that their ASAP/ALAP constraints and the precedence constraints are satisfied. If several operators can be scheduled in one step, a set of heuristics is used to select an operator. The method assumes, that each partition contains only one level of logic, which implicitly maximizes the performance. If the length of the critical path exceeds the number of configurations, more than one logic level is packed into a context. After an initial schedule is obtained, the schedule is optimized by pairwise exchange of operators.

Liu and Wong [LW98] reformulated the precedence constrained scheduling problem as a network-flow problem. The authors present a technique for modeling nets in combinational and sequential circuits as network flows. The objective of the proposed algorithm is to find a temporal partitioning, such that the inter-context communication demand is minimized while the size of the partitions is well balanced. The problem is solved by iteratively computing minimal cuts using the max-flow min-cut theorem.

Wu et al. present an Integer Linear Program (ILP) formulation for the multistage precedence-constrained partitioning problem [WLC01]. This method outperforms other methods since it allows for a globally optimal solution of the problem, while most alternative approaches perform only local optimization and use heuristics. The problem formulation is exactly the same as in [LW98], but is exactly solved with an ILP. The solution minimizes the amount of inter-context communication resources while balancing the size of the partitions.

## Discussion

Since we use a TMFPGA-like architecture model, our work can be directly compared to related work on temporal partitioning of sequential circuits for time-multiplexed FPGAs.

A common characteristic of all related approaches is, that no method treats temporal partitioning as optimization of a defined performance metric. In contrary, the approaches try to implicitly optimize multiple, potentially contradicting, optimization goals concurrently, such as balancing the size of the partitions and minimizing the demand for inter-context communication resources. Our method conceptually differs from the other approaches, because it defines a performance metric and uses this metric as the only optimization objective. All implementation related constraints are explicitly formulated as constraints.

Another commonality of the related approaches is the treatment of temporal partitioning as a precedence constrained scheduling problem

for an acyclic data-dependency graph. This data-dependency graph is defined by the circuit's netlist. The proposed methods treat the data-dependency graph as statically given, disregarding that it originates from a digital circuit, which can be modified with function preserving transformations. Excluding transformations of the circuit's data-dependency graph generally leads to suboptimal performance of the temporal partitioned implementation. In our approach, we allow for structural circuit modifications with retiming to maximize the circuit's performance in the optimization process.

Summarizing, our method has the following advantages over related work:

1. *Optimizing for Performance* The performance of the temporal partitioned circuit implementation is determined by the longest critical path in all partitions. Our problem formulation optimizes *exclusively* for performance. The resource constraints of the implementation architecture are explicitly formulated as constraints, but not optimized as secondary goals.
2. *Structural Modifications* Retiming changes the structure of the circuit by moving the circuit's registers to optimize the performance. These structural modifications increase the solution space, and, in general, lead to better solutions.
3. *Exact method* The method is based on a strict mathematical problem formulation that can be solved exactly. In contrast to the majority of related approaches that use heuristic methods, we can guarantee that the result is a globally optimal solution (within the scope of the model).
4. *Extensibility* The MILP formulation can be easily extended with variable operator delay, variable operator cost, variable context capacity, etc. The problem can still be solved optimally under these extensions.
5. *Automated Pipelining* Retiming can be also used to find optimally pipelined circuit implementations. Hence, the problem formulation for temporal partitioning can be extended to allow for automated pipelining. Pipelining can further increase the performance of the temporal partitioned circuit at the expense of additional delay (see Sec. 5.3.6).

Using an MILP problem formulation for solving the temporal partitioning problem has two main drawbacks:

1. The MILP formalism requires that any extension to the problem is also a linear constraint. Some non-linear constraints can be converted to linear constraints, but, in general, including non-linear constraints in heuristic partitioning algorithms may be easier.
2. In the worst case, the time for solving a MILP grows exponentially

with the problem size. While in practice many large MILP problems can be efficiently solved, there may exist large instances of the temporal partitioning problem that cannot be solved in a reasonable amount of time. Further experimentation with large benchmarks will be required to see, whether this theoretical limitation is relevant for the application to a rather small, coarse-grained reconfigurable CPU architecture.

## 5.4 Summary

In this chapter we defined a classification of hardware virtualization techniques for dynamically reconfigurable architectures. We denote the approaches as temporal partitioning, virtualized execution, and virtual machine. For each technique, we have presented a survey on application models, implementation architectures and algorithms.

We have introduced an application specification model for the Zippy architecture that explicitly includes hardware virtualization and we have shown, how temporal partitioning and virtualized execution are supported by the Zippy architecture with dedicated hardware resources. With two case-studies we have demonstrated the application of virtualized execution and temporal partitioning on the Zippy architecture.

Finally, we have presented a novel method for temporal partitioning of sequential circuits. The method treats temporal partitioning as an application of slowdown and retiming and formulates optimal temporal partitioning as an optimization problem. We present an MILP formulation of the problem that can be solved optimally.

# 6

## Experimental Results

In this chapter, we present experimental results for two case studies that have been performed. The case studies illustrate the application of hardware virtualization techniques on the Zippy architecture. The case studies are implemented with the implementation tool-flow, that has been introduced in Chapter 3, and are simulated with the co-simulation framework, that has been introduced in Chapter 4. The case studies hence exercise the complete hardware and software implementation tool-flow and the co-simulation framework and demonstrate that the tool-flow and co-simulation can handle real-world examples.

The case study in Section 6.1 presents an application of the *virtualized execution* virtualization technique, that has been introduced in Section 5.1.3. We investigate the trade-offs in performance and utilization for three different implementations of the same application: a pure CPU implementation, an implementation on a single-context Reconfigurable Processing Unit (RPU), and an implementation on an 8-context RPU.

In the case study in Section 6.2 we present an application of *temporal partitioning*, which has been introduced in Section 5.1.2. The example compares the execution of the application on a large instance of the Zippy architecture to the temporal partitioned execution of the same application on a small Zippy instance.

### 6.1 Virtualized Execution of a Digital Filter

In this case we study an application of *hardware virtualization by virtualized execution* on the Zippy architecture. Virtualized execution is suited

for data streaming applications that map well to (macro-)pipelines where each pipeline stage is implemented by one configuration of the reconfigurable array. Hence, the basic unit of computation (hardware page) in this example is a  $4 \times 4$  instance of the Zippy reconfigurable array. Macro-pipelining is a special case of virtualized execution and restricts the communication between the subcircuits to forward pipelining, hence feedback exists only within a subcircuit.

As an example, we present the partitioning and mapping of Finite Impulse Response (FIR) filters [OS99] which are important building blocks in many digital signal-processing applications. We show how FIR filters of arbitrary order—that are too large to fit entirely into an RPU configuration—can be implemented with virtualized execution. We consider the implementation on three architectural variants of the Zippy architecture: an implementation that uses only the CPU core, an implementation on a single-context RPU, and an implementation on a multi-context RPU with 8 contexts.

### 6.1.1 FIR Filter Partitioning and Mapping

Macro-pipelined virtualized execution requires that the application is split into a sequence of operators with forward-pipelining between operators only. For FIR filters it's possible to split a large filter into a sequence of communicating subfilters with algebraic manipulations of the transfer function. The answer  $Y(z)$  of an FIR filter, given by its transfer function  $H(z)$ , to an input signal  $X(z)$  can be computed as  $Y(z) = H(z) \cdot X(z)$ .  $H(z)$  is defined as a polynomial in  $z^{-1}$ :

$$H(z) = h_0 + h_1z^{-1} + \dots + h_mz^{-m} = \sum_{i=0}^m h_i \cdot z^{-i},$$

and can be factorized into smaller polynomials  $H_i(z)$ , i. e.,  $H(z) = H_1(z) \cdot H_2(z) \cdot \dots \cdot H_l(z)$ . Each of these polynomials represents an FIR filter of smaller order. Hence, this algebraic manipulation splits up the initial FIR filter into a cascade of FIR filters (subfilters), where each subfilter implements a part of the factorized transfer function. If the input data-stream is filtered through a cascade of these subfilters, the initial FIR filter function is computed.

In this case-study we implement a 56th-order FIR filter as a cascade of eight 7th-order filter stages. Each stage is implemented in the so-called 'transposed direct form'. Figure 30 shows a simplified schematic of the mapping of one of these stages onto the reconfigurable array<sup>1</sup>. The filter coefficients are part of the configurations and are computed by factorizing

<sup>1</sup>In fact, this case study uses a predecessor [Enz04] of the Zippy RPU architecture.

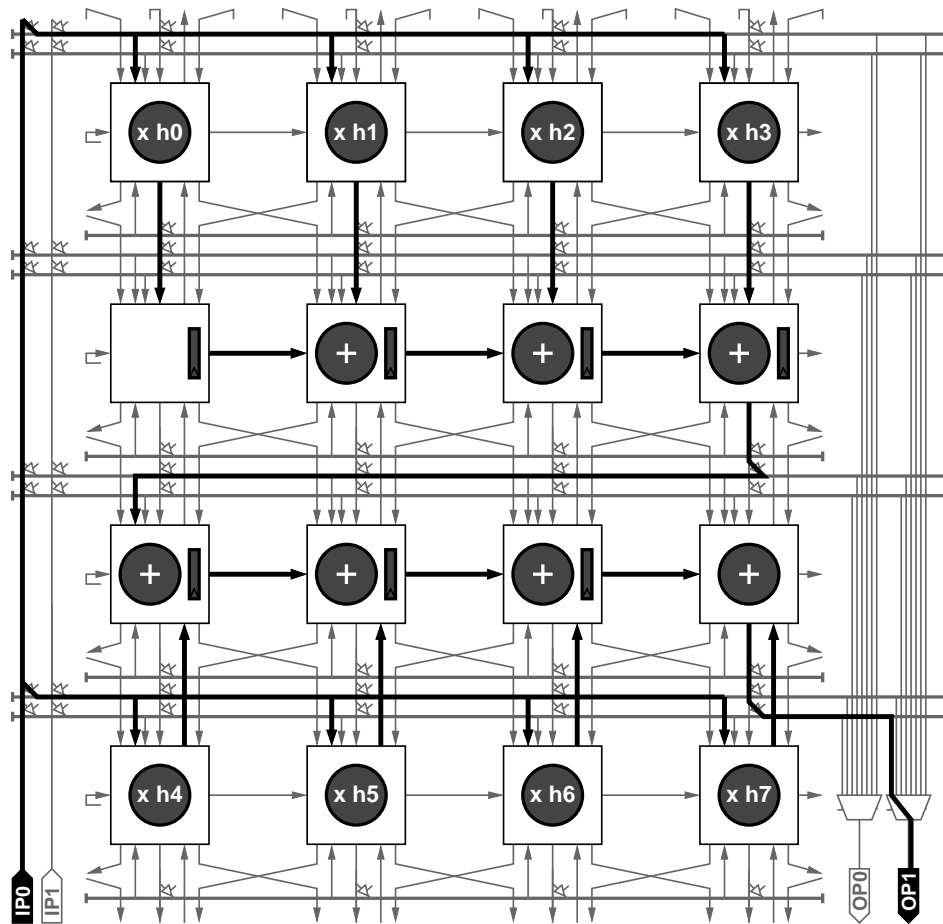


Fig. 30: One 8-tap FIR filter stage mapped to the reconfigurable array

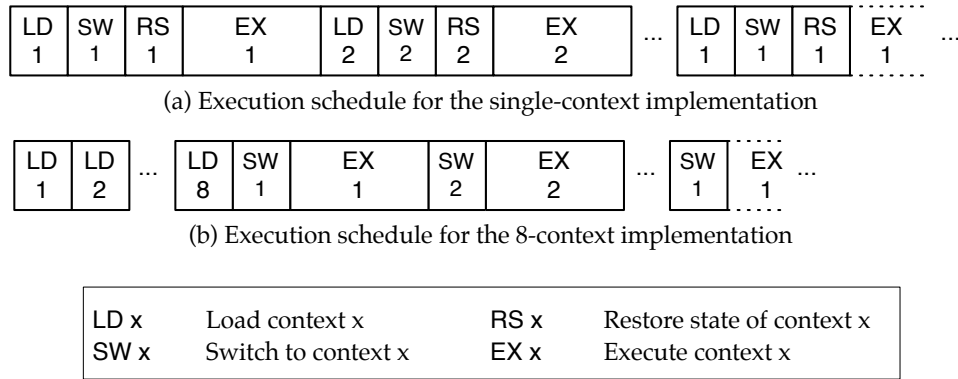
the initial transfer function of the filter. Each FIR subfilter comprises delay registers (see Fig. 30), which form the *state* of the RPU context. At the next invocation of the context, the subfilter requires that the state, i. e., the register contents, are restored before the context is executed again. Figure 31 illustrates the execution schedule for the application on the single-context and the 8-context implementation.

In the single-context implementation, each execution of a subfilter is preceded by a configuration load, switch (activation), and restore phase. The execution phase can run for an arbitrary number of cycles. The output of a subfilter is intermediately stored in a FIFO of the RPU and constitutes the input data for the subsequent subfilter. The length of an execution phase is bounded by the size of the FIFOs.

The restore phase restores the delay registers to their values at the end

---

But since this predecessor architecture implements a true subset of the functionality of the latest Zippy architecture (cf. Chapter 2.3) the architectural differences do not matter for this case-study.



**Fig. 31:** Execution schedule for the virtualized FIR Filter implementation

of the last execution. The restore phase is needed, since the single-context architecture provides only a single set of registers whose contents are overwritten if a different configuration is executed. We implement the restore operation by overlapping subsequent data blocks [OS99], which results in an execution overhead. In the case of the 8-context implementation the state is preserved automatically and no overlapping of data blocks is required since each context has a dedicated set of registers.

## 6.1.2 Experimental Setup

In total, we process 64k samples organized in data blocks in each simulation run. The size of the data blocks is a simulation parameter and corresponds to the depth of the FIFO buffers available on the RPU. We vary the depth of the FIFOs between 128 and 4k words. A data block is written to the RPU, processed sequentially by the eight FIR subfilter stages, and finally the result is read back from the RPU.

A control task running on the CPU core controls the loading and activation of contexts according to Fig. 31. In the case of the 8-context implementation, each context is loaded only once at initialization. In the single-context implementation the configurations must be loaded at each invocation of a context. Further, the delay registers of the context are restored by overlapping the data-blocks by 57 samples (filter taps of the non-virtualized FIR filter).

The RPU architecture in this example features a  $4 \times 4$  instance of the Zippy architecture with 2 horizontal north buses, 1 horizontal south bus and no vertical east buses. The parameters of the CPU core correspond to our “embedded CPU configuration” (see Section 2.2.2) and are summarized in Tab. 6.



Architecture Parameter	Embedded CPU
Integer units	1 ALU, 1 mult.
Floating point units	1 ALU, 1 mult.
L1 I-cache	32-way 16k
L1 D-cache	32-way 16k
L2 cache	none
Memory bus width / ports	32 bit / 1
IFQ / RUU / LSQ sizes	1 / 4 / 4 instr.
Decode / issue / commit bandwidth	1 / 2 / 2 instr.
Execution order	in-order
Branch prediction	static

**Tab. 6:** Configurations of the embedded CPU model

### 6.1.3 Results and Discussion

The simulation results, that depend on the implementation architecture and of the FIFO buffer size, are presented in Table 7 and Figure 32.

Table 7 shows the execution time in cycles for the various architectures. For comparing the performance of the implementation, the table also shows the execution time normalized to  $\text{cycles}/(\text{tap} \cdot \text{sample})$ . The execution time of the filter for the pure CPU implementation varies only slightly with the block size and amounts to 110.65 million cycles on average. Figure 32(a) shows the speedups relative to this execution time.

Fig. 32(b) presents the CPU load on the CPU core for the different architectures. For computing the CPU load, we assume a real-time system that filters blocks of data samples at a given rate. When the filter computation is moved from the CPU to the reconfigurable array, the CPU is relieved from these operations and can use this capacity for running other tasks. However, the CPU still has to transfer data to and from the FIFOs, load the configurations to the RPU on demand, and control the context switches. The load given in Fig. 32(b) determines the spent CPU cycles normalized to the CPU only system.

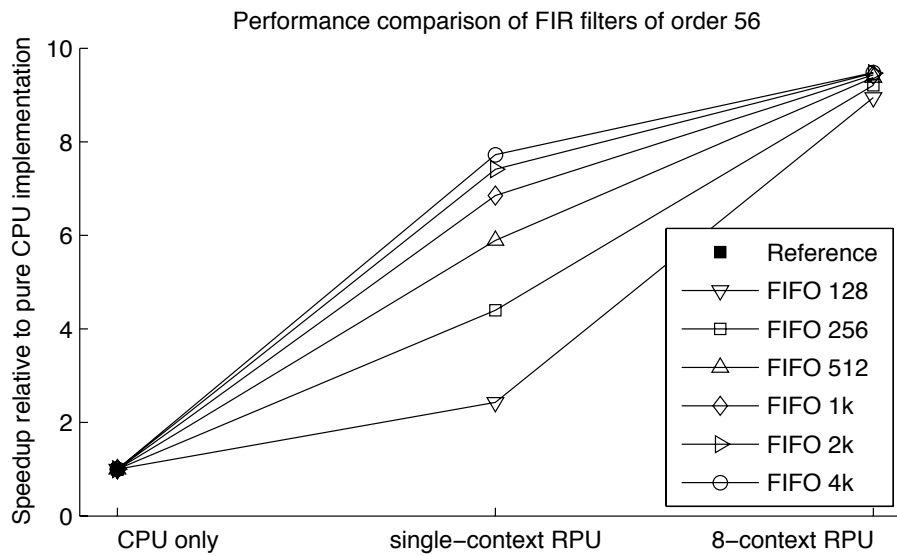
This case study shows, that we can not only implement the large FIR filter, but also achieve two main benefits over a pure CPU implementation: First, the computation is accelerated, and secondly, the CPU is relieved from some operations and can devote the free capacity to other functions.

We point out the following observations:

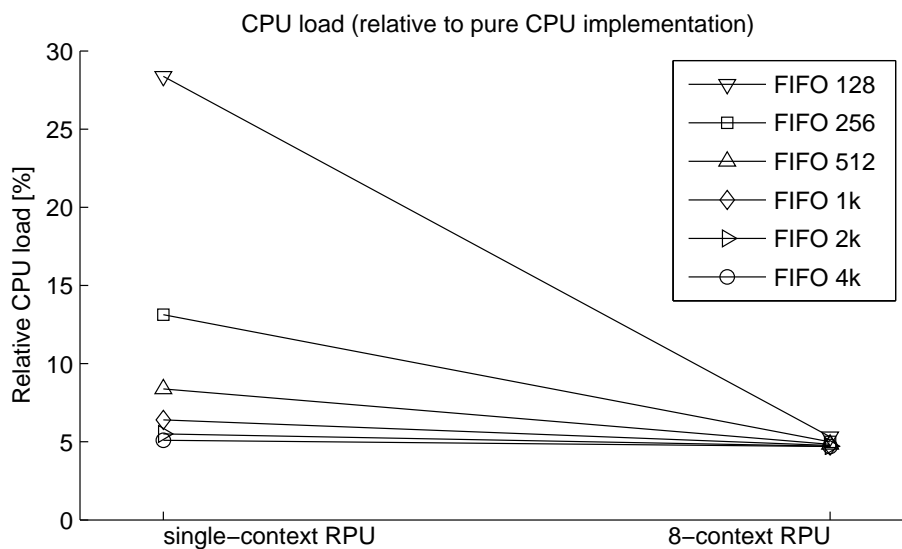
- Using an RPU we achieve significant speedups, ranging from a factor of 2.4 for a 128 word FIFO single-context device up to a factor of 9.5 for an 8-context RPU with a 4096 word buffer.

Architecture	Buffer [words]	Execution time [cycles]	Norm. Performance [ $cyc/(tap \cdot sample)$ ]
CPU only	4096	110672354	29.63
	2048	110654459	29.61
	1024	110646831	29.62
	512	110643416	29.62
	256	110643924	29.62
	128	110650685	29.62
	single-context	4096	14325169
2048		14923289	3.99
1024		16147674	4.32
512		18778091	5.02
256		25163909	6.74
128		45520990	12.12
8-context		4096	11669279
	2048	11687130	3.13
	1024	11730940	3.14
	512	11801405	3.16
	256	12014197	3.22
	128	12368245	3.31

**Tab. 7:** Execution time and efficiency for virtualized execution case-study



(a) Relative speedup of reconfigurable processor implementation



(b) Relative CPU load of reconfigurable processor implementation

**Fig. 32:** Results of virtualized execution case-study

- The system performance in terms of speedup and CPU load depends on the length of the FIFO buffers. Enlarging the FIFOs increases the performance but also the filter delay. For instance, a single-context RPU using a FIFO with 1k words instead of 128 words improves the speedup from 2.43x to 6.85x (factor 2.82) and reduces the CPU load from 28.4% to 6.4% (factor 4.43). But at the same time, the latency is increased by a factor of 8. Practical applications, e. g., real-time signal processing, could limit these potential gains by imposing delay constraints.
- Figure 32 shows, that also a single-context implementation can provide decent speedups and CPU load reduction, if the application is not delay sensitive and large FIFO buffers are used. But if an 8-context implementation is used, the context loading and switching overheads are largely reduced and the performance becomes almost independent of the FIFO size. Hence, an 8-context implementation allows for low-latency processing.
- With increasing FIFO size, speedup and CPU load reach an asymptotic value of about 9.5x for the speedup, and 4.7% for the CPU load. For these asymptotic cases, the execution time is dominated by the data-transfers from the CPU to the RPU's FIFO and vice versa.

This case study shows, that *virtualized execution* is a useful technique for implementing a macro-pipelined application on the Zippy architecture. We have shown, that a large FIR filter can be partitioned and executed on the Zippy architecture. One configuration for the RPU corresponds to the hardware page required for virtual execution. The hardware pages communicate via the on-chip FIFOs that are accessible from all contexts. The runtime system for sequencing the contexts is implemented by the CPU core.

The results of the case study emphasize the importance of the system-level cycle-accurate simulation for architectural evaluation and optimization. Only a system-level evaluation approach allows us to accurately quantify the design trade-offs.

## 6.2 Temporal Partitioning of an ADPCM Decoder

This case study illustrates *hardware virtualization by temporal partitioning* on the Zippy architecture. We compare the execution of an ADPCM application on a large instance of the Zippy architecture with the execution

on a smaller instance of the same architecture. The large instance requires more area but allows to map the complete application into one configuration. The smaller instance requires temporal partitioning to run the application. For reference, we compare both implementations to a pure software implementation of the application running on the same CPU core.

### 6.2.1 Application

Adaptive Differential Pulse Code Modulation (ADPCM), also known as ITU-T G.721, is a well established speech coding algorithm. ADPCM compresses the data rate by a factor of 4 while providing acceptable quality for voice signals. The decoder uses a simple predictor that predicts the next 16bit output value as the sum of the current output value and an increment. The increment is adapted based on a 4bit input signal using a non-linear function, which is defined by two look-up tables. Listing 6.1 presents the C code for the ITU's reference implementation of the ADPCM decoder.

Based on the reference implementation, we have designed a hardware implementation of the ADPCM decoder which is shown in Figure 33. ADPCM uses 31 combinational operators (that can be directly implemented by a cell), 3 dedicated registers, 1 input, and 1 output port. The dedicated registers can be implemented within the input register files of the Zippy cells, cf. Fig. 6 on page 23. Thus the hardware implementation requires an execution architecture with at least 31 cells.

### 6.2.2 Experiments

We perform three experiments:

- A) For the *non-virtualized* implementation, we have chosen a reconfigurable array of size  $7 \times 7$ . Although a  $6 \times 6$  array would provide a sufficient number of cells, the dense interconnect structure of the ADPCM netlist leads easily to congestion and makes placement and routing on a  $6 \times 6$  array rather difficult. Using a  $7 \times 7$  array relaxes the implementation constraints and allows the tools to quickly find a routable implementation.
- B) For the *virtualized implementation* the full netlist has been manually partitioned into three smaller sub-netlists, such that each of them fits onto an array of size  $4 \times 4$ . Figure 33 presents the division of the initial netlist into three contexts. As explained in Section 5.2.3, we use the output register files of the cells for inter-context communication. To simplify the hardware implementation process for this case

```
1 static int indexTable[16] = {
2     -1, -1, -1, -1, 2, 4, 6, 8, ... };
3 static int stepsizeTable[89] = {
4     7, 8, 9, 10, 11, 12, 13, 14, 16, 17, ... };
5
6 void
7 adpcm_decoder(char indata[], short outdata[], int len,
8               struct adpcm_state *state){
9     signed char *inp; short *outp;
10    int sign, delta, step, valpred, vpdiff, index;
11    int inputbuffer, bufferstep;
12
13    outp = outdata; inp = (signed char *)indata;
14    valpred = state->valprev; index = state->index;
15    step = stepsizeTable[index]; bufferstep = 0;
16
17    for ( ; len > 0 ; len-- ) {
18        if ( bufferstep ) {
19            delta = inputbuffer & 0xf;
20        } else {
21            inputbuffer = *inp++;
22            delta = (inputbuffer >> 4) & 0xf;
23        }
24        bufferstep = !bufferstep;
25        index += indexTable[delta];
26        if ( index < 0 ) index = 0;
27        if ( index > 88 ) index = 88;
28        sign = delta & 8;
29        delta = delta & 7;
30        vpdiff = step >> 3;
31        if ( delta & 4 ) vpdiff += step;
32        if ( delta & 2 ) vpdiff += step >> 1;
33        if ( delta & 1 ) vpdiff += step >> 2;
34        if ( sign )
35            valpred -= vpdiff;
36        else
37            valpred += vpdiff;
38        if ( valpred > 32767 )
39            valpred = 32767;
40        else if ( valpred < -32768 )
41            valpred = -32768;
42        step = stepsizeTable[index];
43        *outp++ = valpred;
44    }
45    state->valprev = valpred; state->index = index;
46 }
```

---

Lst. 6.1: C-Code for the ADPCM Decoder (Excerpt from the Reference Implementation)

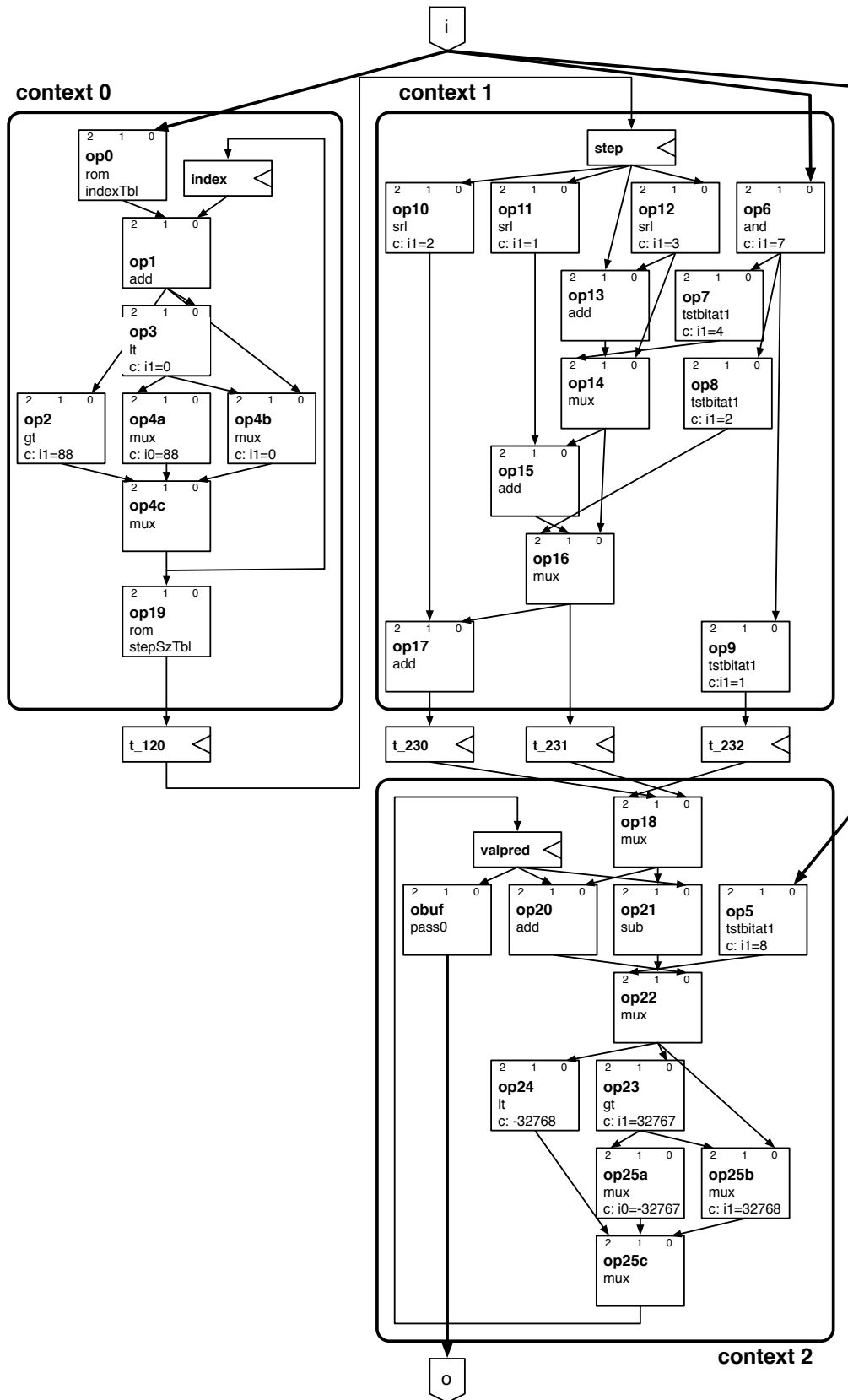


Fig. 33: ADPCM: application netlist

Architecture	No RPU, CPU only	Large RPU without TP	Small RPU with TP
Implementation results			
total cycles [k cycles]	39260	23941	24896
cycles/ sample [cycles]	157.0	95.8	99.6
rel. speedup	1	1.64	1.58
array size [cells]	0	49	16
Co-simulation results			
simulation time [s]	177	3767	15210
instruction rate [inst/s]	121543	3687	942
rel. simulation time	1	21.3	85.9
rel. instr. rate	1	0.0303	0.00775

**Tab. 8:** Simulation Results for the ADPCM Decoder Case-Study

study, we used dedicated cells with fixed placement as virtualization registers (denoted by  $t_{x,y}$ ). For each sub-netlist a configuration is generated using the Zippy hardware implementation tool-flow. The constraints for the fixed placement of the virtualization registers have been manually added to the sub-circuit's netlists.

Figure 33 shows that for this temporal partitioned implementation of the ADPCM algorithm all feedback paths of the circuit stay within a single context. It must be emphasized that this is not a requirement for hardware virtualization by temporal partitioning. Thanks to the virtualization registers, arbitrary feedback cycles between contexts are possible.

- C) The *pure software implementation* uses the C source code of the ADPCM reference implementation. The code has been compiled with SimpleScalar's GNU C compiler.

### 6.2.3 Experimental Setup and Results

We have evaluated the performance of these three implementations using our system-level co-simulation framework (see Chapter 4). We use the *embedded CPU configuration*, see Tab. 1 on page 18. For the RPU configuration we use the parameters shown in Tab. 3 on page 32, the size of the array is set to  $4 \times 4$  and  $7 \times 7$ , respectively.

We have used the software toolflow (see Section 3.4) for creating the applications binaries. Since using memory-mapped IO for accessing the



RPU requires to strictly retain the order of instructions, we have turned off compiler optimization for all three experiments to prevent any disarranging of instructions.

For performance evaluation we determine the execution time for decoding 250'000 ADPCM samples, processed as 250 blocks of 1000 samples each. By averaging over 250 iterations we try to approximate the sustained application performance and to reduce the effect of the application setup phase.

The execution time are cycle-accurate on the system-level. That is, the execution time includes all overheads, such as reading the input data from memory, transferring data between CPU and RPU and vice versa, downloading the configurations, etc.

Table 8 summarizes the results of the case study and demonstrates the trade-off involved in hardware virtualization with temporal partitioning. We point out the following observations:

- Using a reconfigurable co-processor yields a speedup over the pure CPU implementation of 1.64 when using the large reconfigurable array without temporal partitioning, and a slightly reduced speedup of 1.58 for the temporal partitioned implementation. The rather small difference of 6% in the speedups for the temporal partitioned and the non-partitioned case suggests, that the zero-overhead temporal partitioning sequencer (see Section 2.3.4) handles the context sequencing efficiently.
- Comparing only the raw performance of the hardware accelerated kernel can be seriously misleading. The non-partitioned ADPCM decoder decodes 1 sample per cycle, while the temporal partitioned implementation is 3 times slower (3 cycles per sample). While this large difference of raw-speedups (factor of 3) could lead to the expectation that the implementation without temporal partitioning performs significantly better, system-level simulation reveals, that the actual performance gain is merely 6%.

This observation strongly supports our claim that system-level performance analysis is a necessity for an accurate performance assessment of a reconfigurable processor application.

- The example also shows the trade-off between chip size and performance. In his dissertation [Enz04] Enzler estimates that adding a 4 context Zippy RPU with a  $4 \times 4$  reconfigurable array to an embedded CPU core increases the chip area by about 25%. Hence with 25% increased chip area, the performance is increase by a factor of 1.58 when temporal partitioning is used. Without temporal partition-

ing, the hardware size for the RPU would be significantly larger (49 vs. 16 cells) while the performance gain is only 6%.

- Table 8 presents also data about the performance of the simulation environment. The total simulation time increases a consequence of the additional RPU simulation by a factor of 21 for the experiment without temporal partitioning, and a factor of 86 for the temporal partitioned implementation. The temporal partitioning case is slower, since changing the active configuration of the RPU in every cycle leads to more signal transitions and the computational effort of discrete event simulation correlates with the number of signal changes.

With this case study we have demonstrated that hardware virtualization with temporal partitioning is feasible. We have studied the trade-offs between execution time and hardware requirements. We conclude that temporal partitioning offers a sensible approach to reduce the hardware requirements while still making use of the high performance of application-specific accelerator circuits. The reduction in hardware requirements is in particular attractive for embedded systems, where chip area is scarce due to cost constraints.

### 6.3 Summary

In this chapter we have demonstrated with two case studies that hardware virtualization can be applied to the Zippy architecture. We have shown that the parametrized software and hardware implementation tool-flows are capable to implement real world examples.

We have used the system-level performance evaluation framework to obtain cycle-accurate performance data. The parametrized architecture model enables us to compare the performance of a variety of different instances of the Zippy architecture.

The case studies have shown, that hardware virtualization allows us to successfully trade-off RPU chip area for performance. This trade-off is important for applying reconfigurable processors in cost sensitive embedded systems.

# 7

## Conclusions

In this chapter we will summarize the contributions of this work, draw conclusions and outline directions for future work.

### 7.1 Contributions

In this work, we have made the following contributions to the state of the art in coarse-grained reconfigurable processor architectures for embedded systems:

- **Reconfigurable processor architecture**

We have designed a new coarse-grained reconfigurable processor architecture named Zippy. Zippy is not a single concrete architecture, but a widely parametrized architecture and simulation model, which defines a whole family of architectures. The architecture has been co-designed with a corresponding design tool-flow, a simulation environment and an application specification model. Zippy is tailored to digital-signal processing of data streams in embedded systems and provides dedicated hardware units to efficiently support hardware virtualization techniques.

- **Hardware Virtualization**

We propose to use hardware virtualization in embedded systems when executing applications on reconfigurable processors. We have defined a classification of hardware virtualization techniques for dynamically reconfigurable architectures and have discussed their application to the Zippy architecture. Hardware virtualization enables

us to execute applications that exceed the capacity of the Reconfigurable Processing Unit (RPU) and allows us to trade-off area for performance. With two case-studies we have demonstrated that hardware virtualization is feasible on the Zippy architecture and that the area-performance trade-off due to hardware virtualization can be exploited for finding cost-effective implementations in embedded systems.

- **Application specification model**

We have described an application specification model that is suited to specify streaming data processing applications that are executed on a reconfigurable processor. The specification model decomposes the application into a set of communicating tasks whose interactions are specified with a coordination model. This kind of application specification is suitable to be mapped to a reconfigurable execution architecture that uses hardware virtualization.

- **Design Tools and Performance Evaluation**

We have developed a hardware and software implementation tool-flow that generates an implementation from the application's specification. To cope with the parametrized nature of the Zippy architecture, the design tools are also parametrized to support all variants of the Zippy architecture.

We advocate system-level cosimulation as the appropriate performance evaluation method for a reconfigurable processor. We have presented a corresponding co-simulation environment, which combines a cycle-accurate CPU simulator with a cycle-accurate RPU model into system-level, bit-exact and cycle-true performance evaluation framework. This allows us to perform design-space exploration for the whole family of Zippy architectures.

- **Novel method for optimal temporal partitioning**

We have developed a novel method for optimal temporal partitioning of sequential circuits. The method bases on a Mixed Integer Linear Program (MILP) problem formulation and solves the problem optimally, while most related approaches use heuristic approximations. In contrast to related work, our approach directly optimizes the performance of the partitioned circuit and allows for function-preserving structural modifications of the circuit.

## 7.2 Conclusions

In this work we have studied dynamic reconfiguration of a reconfigurable processor in the context of embedded systems. Related work proposes to use dynamic reconfiguration to increase the utilization of the reconfigurable resource and to implement “multi-tasking” of circuits. In contrast, we have focused on using dynamic reconfiguration to implement hardware virtualization techniques in order to mitigate the effect of limited hardware resources. With two case studies we have shown that hardware virtualization is feasible in practice, and that it also creates interesting area vs. performance trade-offs. Thus, hardware virtualization on reconfigurable processors could have interesting applications in embedded systems with constrained reconfigurable resources.

We have built our architecture simulation environment on existing work. To this end, we have extended the SimpleScalar CPU simulator and integrated it with the ModelSim VHDL simulator into a cycle-accurate co-simulation environment. Reusing these existing simulators allowed us to reuse the solid simulator and the compilation tool-chain for the CPU and to model the reconfigurable array in VHDL. Using VHDL as modelling language proved very effective, since it allowed us to model the architecture at different levels of abstraction while retaining overall cycle-accuracy.

We have argued, that neglecting communication, configuration and control overheads can lead to distorted performance estimations for the reconfigurable processor. Hence, we have advocated to use cycle-accurate, system-level co-simulation for performance evaluation. Our case studies confirm, that a system-level performance evaluation is effectively of great importance. For example, in the FIR case study the use of a the reconfigurable co-processor yields a significant speedup of up to a factor of 10. In contrast, the speedups obtained in the ADPCM case studies are rather modest (approximately 60%). This result may seem surprising, given that the ADPCM decoder core has a high raw-speedup (i. e., if communication and control-overheads are neglected) of about 160x. In this specific application, the communication overheads are significant and limit the obtainable performance. We conclude, that evaluating the effective performance of an application running on a reconfigurable processor requires a detailed analysis of the communication and computation patterns, which is enabled by system-level co-simulation.

We use a netlist-based design entry for application specification. Since the architecture is based on coarse-grained computing elements, a netlist is essentially equivalent to a signal-flow diagram. Our experience with implementing stream-processing applications from the digital signal-processing domain shows, that specifying an application as a netlist of

coarse-grained operators is feasible and convenient. An advantage of the netlist-based design entry is that the netlists can be used directly in the subsequent placement and routing process.

For placement and routing of circuits on the Zippy architecture, we have adapted algorithms from the FPGA domain. We noticed, that the placement and routing can take excessively long, if we are solely using a simple placement algorithm, that iteratively improves an initial, random placement using stochastic optimization. We attribute this fact to the sparseness of routing resources in the coarse-grained Zippy architecture, which causes the placer to generate many placements with infeasible routing. While stochastic optimization is still valuable for optimizing a placement, the sparse interconnect in coarse-grained architectures demands for novel methods to generate initial placements, which account for the structure of the interconnect. We have developed a placement heuristic that favors local interconnect for connecting neighboring cells and uses buses for high fan-out nets.

For the case studies, we have performed the required application-partitioning step by hand. This task is tedious and error-prone and is hence only feasible for small applications. Further, in general, manual partitioning leads to implementations with sub-optimal performance. This has motivated us, to find a way for automated application partitioning. Our novel temporal partitioning algorithm can serve as a mathematical foundation of performance-optimal application partitioning. Once the method is fully integrated into the hardware tool-flow, it will allow the designer to specify and to implement applications of arbitrary size, without considering the resource limitations of the execution architecture.

### 7.3 Future Directions

In the following, we will briefly outline three possible directions for further research:

- A main obstacle for research in reconfigurable architectures is the lack of design implementation and evaluation tools for newly defined architectures. This work has shown that parametrized architecture models and parametrized design-tools are feasible. While the VHDL model of the RPU and the routing architecture graph (used by the placer and router) represent largely the same information, the corresponding models have been created separately by hand. An interesting generalization of the Zippy design and simulation tools would be to further separate the architecture-independent parts of the tools from the architecture-dependent parts. This would

allow us to build a framework that automatically generates design and simulation tools from a common, formal architecture description. Such a framework could significantly speed up the design and evaluation process for new reconfigurable processor architectures.

- Another direction for future work is to elaborate the design-flow for hardware virtualization. While we have proposed a specification model for streaming data-processing applications that are executed with hardware virtualization, the corresponding implementation tool-flow is not fully automated. Creating an automated end-to-end tool-flow would be a big step towards the acceptance of hardware virtualization in embedded applications. Designing such a tool-flow bears a number of challenges, such as, dividing the application specification graph into partitions, performing automated temporal partitioning for tasks that exceed the resource supply of the RPU, or finding deadlock-free context schedules.
- Finally, our work on the novel method for optimal temporal partitioning of sequential circuits could be extended. We have already mentioned, that adding functional pipelining could further improve performance of the partitioned circuits. Another direction is to apply the method to a concrete architecture, i. e., the Zippy architecture. It will be interesting to study, to what extent the idealized architectural assumptions of the method impact the applicability to a concrete architecture and whether the runtime of the MILP solver is acceptable for practical applications.





# A

## Acronyms

**FPGA** Field-Programmable Gate-Array

**ASIC** Application-Specific Integrated Circuit

**PDA** Personal Digital Assistant

**RPU** Reconfigurable Processing Unit

**LUT** look-up table

**ZNF** Zippy Netlist Format

**CAD** Computer-Aided Design

**VLIW** Very Long Instruction Word

**RFU** Reconfigurable Functional Unit

**RTL** Register Transfer Level

**VLSI** Very Large Scale Integration

**ADPCM** Adaptive Differential Pulse Code Modulation

**CDFG** control data-flow graph

**HLL** high-level programming language

**HDL** hardware description language

**DAG** directed acyclic graph

**FIR** Finite Impulse Response

**ILP** Integer Linear Program

**MILP** Mixed Integer Linear Program

**IIR** Infinite Impulse Response

**FSM** Finite State Machine

# Bibliography

- [ALE02] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.
- [Alt02] Altera. *Altera Excalibur Devices: Hardware Reference Manual*, v3.1 edition, November 2002.
- [ASI<sup>+</sup>98] 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)*, volume 1388 of *Lecture Notes in Computer Science*, pages 55–60. Springer-Verlag, 1998.
- [Atm05] Atmel. *Atmel FPSLIC: AT94K05/10/40AL Datasheet*, 1138h-fpslic/05 edition, June 2005.
- [BBS<sup>+</sup>00] 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, November/December 2000.
- [BD01] Gordon Brebner and Oliver Diessel. Chip-based reconfigurable task management. In *Proc. 11th Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 182–191. Springer-Verlag, 2001.
- [BEM<sup>+</sup>03] V. Baumgarte, G. Ehlers, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt. PACT XPP – a self-reconfigurable data processing architecture. *Journal of Supercomputing*, 26(2):167–184, September 2003.
- [BMN<sup>+</sup>01] V. Baumgarte, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt. PACT XPP – a self-reconfigurable data processing architecture. In *Proc. 1st Int. Conf. on Engineering of Reconfigurable*

- Systems and Algorithms (ERSA)*, pages 64–70. CSREA Press, 2001.
- [Bre96] Gordon Brebner. A virtual hardware operating system for the Xilinx XC6200. In *Proc. 6th Int. Workshop on Field Programmable Logic and Applications (FPL)*, pages 327–336. Springer-Verlag, 1996.
- [Bre98] G. Brebner. Circlets: Circuits as applets. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 300–301. IEEE Computer Society, 1998.
- [BRM99] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.
- [BTM00] 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)*, pages 83–94, 2000.
- [But93] Mike Butts. Tutorial: FPGAs in logic emulation. In *Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*. ACM, November 1993.
- [CCH<sup>+</sup>00] 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)*, volume 1896 of *Lecture Notes in Computer Science*, pages 605–614. Springer-Verlag, 2000.
- [CEC01] 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)*, pages 141–150, 2001.
- [CH02] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.
- [Cha00] Chameleon Systems. Wireless base station design using reconfigurable communications processors. White Paper, V1.0, 2000.
- [CHW00] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, April 2000.

- [CMS97] Douglas Chang and Malgorzata Marek-Sadowska. Buffer minimization and time-multiplexed I/O on dynamically reconfigurable FPGAs. In *Proc. 5th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 142–148. ACM, 1997.
- [CMS98] Douglas Chang and Malgorzata Marek-Sadowska. Partitioning sequential circuits on dynamically reconfigurable FPGAs. In *Proc. 6th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 161–167. ACM, 1998.
- [CMS99] Douglas Chang and Malgorzata Marek-Sadowska. Partitioning sequential circuits on dynamically reconfigurable FPGAs. *IEEE Trans. on Computers*, 48(6):565–578, June 1999.
- [CPL] CPLEX optimizer. <http://www.cplex.com/>.
- [CV99] K.S. Chatka and R. Vemuri. Hardware-software codesign for dynamically reconfigurable architectures. In *Proc. 9th Int. Workshop on Field Programmable Logic and Applications (FPL)*, pages 175–184. Springer-Verlag, 1999.
- [CW02] Joao M. P. Cardoso and Markus Weinhardt. XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. In M. Renovell M. Glesner, P. Zipf, editor, *Proc. 12th Int. Conf. on Field Programmable Logic and Applications (FPL)*, number 2438 in Lecture Notes in Computer Science, pages 864–874. Springer-Verlag, August 2002.
- [DeH94] A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proc. 2nd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pages 31–39, 1994.
- [DeH96a] A. DeHon. DPGA utilization and application. In *Proc. 4th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 115–121, 1996.
- [DeH96b] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, 1996.
- [DM94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
- [DPP02] Matthias Dyer, Christian Plessl, and Marco Platzner. Partially reconfigurable cores for xilinx virtex. In *Proc. 12th Int. Conf. on Field Programmable Logic and Applications (FPL)*, volume 2438

- of *Lecture Notes in Computer Science*, pages 292–301. Springer-Verlag, 2002.
- [EMHB95] Carl Ebeling, Larry McMurchie, Scott Hauck, and Steven Burns. Placement and routing tools for the Triptych FPGA. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 3(4):473–482, December 1995.
- [Enz04] Rolf Enzler. *Architectural Trade-offs in Dynamically Reconfigurable Processors*. PhD thesis, Diss. ETH No. 15423, Swiss Federal Institute of Technology (ETH) Zurich, 2004.
- [EPP<sup>+</sup>01] 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*, volume 4525 of *Proceedings of SPIE*, pages 135–146, 2001.
- [EPP03] Rolf Enzler, Christian Plessl, and Marco Platzner. Co-simulation of a hybrid multi-context architecture. In *Proc. 3rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 174–180. CSREA Press, 2003.
- [EPP05] R. Enzler, C. Plessl, and M. Platzner. System-level performance evaluation of reconfigurable processors. *Microprocessors and Microsystems*, 29(issues 2–3):63–73, April 2005.
- [FFM<sup>+</sup>99] 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*, pages 364–365, 1999.
- [FKT01] S.P. Fekete, E. Köhler, and J. Teich. Optimal fpga module placement with temporal precedence constraints. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 658–665. IEEE Computer Society, 2001.
- [GG97] M. Gokhale and D. Gemersall. High level compilation for fine grained FPGAs. In *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 165–173. IEEE Computer Society, 1997.
- [GLMS02] Thorsten Grötzer, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, May 2002.

- [GOK<sup>+</sup>98] S. Govindarajan, I. Ouais, M. Kaul, V. Srinivasan, and R. Vemuri. An effective design system for dynamically reconfigurable architectures. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 312–313. IEEE Computer Society, 1998.
- [GSB<sup>+</sup>00] 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, April 2000.
- [Hau97] J. R. Hauser. The Garp architecture. Technical report, UC Berkeley, CA, USA, October 1997.
- [HMSS01] James Hwang, Brent Milne, Nabeel Shirazi, and Jeffrey D. Stroomer. System level tools for DSP in FPGAs. In *Proc. 11th Int. Conf. on Field Programmable Logic and Applications (FPL)*, volume 2147 of *Lecture Notes in Computer Science*, pages 534–543. Springer-Verlag, August 2001.
- [HSE<sup>+</sup>00] Yajun Ha, Patrick Schaumont, Marc Engels, Serge Vernalde, Freddy Potargent, Luc Rijnders, and Hugo De Man. A hardware virtual machine for the networked reconfiguration. In *Proc. 11th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pages 194–199, 2000.
- [HVS<sup>+</sup>02] Y. Ha, S. Vernalde, P. Schaumont, M. Engels, R. Lauwereins, and H. De Man. Building a virtual framework for networked reconfigurable hardware and software objects. *Journal of Supercomputing*, 21(2):131–144, February 2002.
- [HW97] 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)*, pages 12–21, 1997.
- [Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing*, pages 471–475. Ed. North-Holland Publishing Co., August 1974.
- [KV98] M. Kaul and R. Vemuri. Optimal temporal partitioning and synthesis for reconfigurable architectures. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 389–396. IEEE Computer Society, 1998.

- [LA89] X.-P. Ling and H. Amano. A static scheduling system for a parallel machine (SM)<sup>2</sup>-II. In *Proc. 2nd Parallel Architectures and Languages, Europe*, LNCS 365, pages 118–135. Springer-Verlag, June 1989.
- [Lee01] Edward A. Lee. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M01/11, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, March 2001.
- [LKD03] Jong-eun Lee, Choi Kiyoungh, and Nikil D. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design & Test of Computers*, 20(1):26–33, January–February 2003. 10.1109/MDT.2003.1173050.
- [LS91] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.
- [LTC+03] 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, November 2003.
- [LW98] Huiqun Liu and D.F. Wong. Network flow based circuit partitioning for time-multiplexed FPGAs. In *Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*, pages 497–504. ACM, November 1998.
- [MMF98] O. Mencer, M. Morf, and M. J. Flynn. PAM-Blox: High performance FPGA design for adaptive computing. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 167–174, 1998.
- [MNC+03] J-Y. Mignolet, P. Nollet, P. Coene, D. Verkest, S. Vernalde, and L. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 989–991. IEEE Computer Society, March 2003.
- [MO99] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. *IEICE Trans. on Information and Systems*, E82-D(2):389–397, February 1999.
- [Mod04] Model Technology Inc. *ModelSim SE Foreign Language Interface*, version 6.0c edition, November 2004. <http://www.model.com>.



- [Mod05] Model Technology Inc. *ModelSim SE User Manual*, version 6.0c edition, January 2005. <http://www.model.com>.
- [MY02] Wai-Kei Mak and Evangeline F.Y. Young. Temporal logic replication for dynamically reconfigurable FPGA partitioning. In *Proc. ACM Int. Symp. on Physical Design (ISPD)*, pages 190–195. ACM, April 2002.
- [Ope02] Open SystemC Initiative (OSCI). *SystemC 2.0 User's Guide*, 2002. <http://www.systemc.org/>.
- [OS99] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Prentice Hall, 2nd edition, 1999.
- [Pan01] P. R. Panda. SystemC: A modeling platform supporting multiple design abstractions. In *Proc. 14th Int. Symp. on Systems Synthesis (ISSS)*, pages 75–80, 2001.
- [PB98] K. M. G. Purna and D. Bhatia. Temporal partitioning and scheduling for reconfigurable computing. In *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 329–330. IEEE Computer Society, 1998.
- [PB99] K. M. G. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Trans. on Computers*, 48(6):579–590, June 1999.
- [PEW<sup>+</sup>02] Christian Plessl, Rolf Enzler, Herbert Walder, Jan Beutel, Marco Platzner, and Lothar Thiele. Reconfigurable hardware in wearable computing nodes. In *Proc. 6th Int. Symp. on Wearable Computers (ISWC)*, pages 215–222, 2002.
- [PEW<sup>+</sup>03] Christian Plessl, Rolf Enzler, Herbert Walder, Jan Beutel, Marco Platzner, Lothar Thiele, and Gerhard Tröster. The case for reconfigurable hardware in wearable computing. *Personal and Ubiquitous Computing*, 7(5):299–308, October 2003.
- [PL91] Ian Page and Wayne Luk. Compiling OCCAM into FPGAs. In *Proc. 11th Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 271–283. Abingdon EE&CS Books, 1991.
- [Ple01] Christian Plessl. Reconfigurable accelerators for minimum covering. Master's thesis, ETH Zurich, Computer Engineering and Networks Lab, March 2001.

- [PP01] Christian Pleschl and Marco Platzner. Instance-specific accelerators for minimum covering. In *Proc. 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 85–91, Las Vegas, Nevada, USA, June 2001. CSREA Press.
- [PP02] Christian Pleschl and Marco Platzner. Custom computing machines for the set covering problem. In *Proc. 10th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 163–172, Napa, USA, April 2002. IEEE Computer Society.
- [PP03a] Christian Pleschl and Marco Platzner. Instance-specific accelerators for minimum covering. *Journal of Supercomputing*, 26(2):109–129, September 2003.
- [PP03b] Christian Pleschl and Marco Platzner. TKDM - a reconfigurable co-processor in a pc's memory slot. In *Proc. 2nd Int. Conf. on Field Programmable Technology (FPT)*., pages 252–259, Tokyo, Japan, December 2003. IEEE Computer Society.
- [SC01] B. Salefski and L. Caglar. Re-configurable computing in wireless. In *Proc. 38th Design Automation Conf. (DAC)*, pages 178–183, 2001.
- [SGV01] V. Srinivasan, S. Govindarajan, and R. Vemuri. Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for Multi-FPGA architectures. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(1):140–158, February 2001.
- [Sim06] The MathWorks, Inc., Natick MA, USA. *Simulink Simulation and Model-Based Design*, version 6 edition, 2006.
- [SLL<sup>+</sup>00] 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.
- [SLM00] H. Simmler, L. Levinson, and R. Männer. Multitasking on FPGA coprocessors. In *Proc. 10th Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 121–130. Springer-Verlag, 2000.
- [SUA<sup>+</sup>00] Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii, and M. Motomura. A virtual hardware system on a dynamically reconfigurable logic device. In *Proc. 8th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 295–296, 2000.

- [SV98] 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)*, pages 78–85, 1998.
- [SV99] V. Srinivasan and R. Vemuri. Task-level partitioning and RTL design space exploration for Multi-FPGA architectures. In *Proc. 7th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 272–273. IEEE Computer Society, 1999.
- [SWPT03] Christoph Steiger, Herbert Walder, Marco Platzner, and Lothar Thiele. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In *Proceedings of the 24th International Real-Time Systems Symposium (RTSS)*, pages 224–235. IEEE Computer Society, December 2003.
- [SWT<sup>+</sup>02] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor. PipeRench: A virtualized programmable datapath in 0.18 micron technology. In *Proc. 24th IEEE Custom Integrated Circuits Conf. (CICC)*, pages 63–66, 2002.
- [TCE<sup>+</sup>95] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon. A first generation DPGA implementation. In *Canadian Workshop on Field-Programmable Devices (FPD)*, pages 138–143, 1995.
- [TCJW97] 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)*, pages 22–28, 1997.
- [TFS01] J. Teich, S.P. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *Journal of Supercomputing*, 19(1):57–75, May 2001.
- [Tri98] Steve Trimberger. Scheduling designs into a time-multiplexed FPGA. In *Proc. 6th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 153–159. ACM, 1998.
- [VNK<sup>+</sup>03] 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, November 2003.
- [WC96] 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)*, pages 126–135, 1996.

- [WK01] Grant Wigley and David Kearney. The development of an operating system for reconfigurable computing. In *Proc. 9th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, April 2001.
- [WL01] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):234–248, February 2001.
- [WLC01] Guang-Ming Wu, Jai-Ming Lin, and Yao-Wen Chang. Generic ILP-based approaches for time-multiplexed FPGA partitioning. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1266–1274, October 2001.
- [WP03a] Herbert Walder and Marco Platzner. Online scheduling for block-partitioned reconfigurable devices. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 290–295. IEEE Computer Society, March 2003.
- [WP03b] Herbert Walder and Marco Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *Proc. 3rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 284–287. CSREA Press, June 2003.
- [XA95] Xiaoping Ling and H. Amano. WASMII: An MPLD with data-driven control on a virtual hardware. *Journal of Supercomputing*, 9(3):253–276, 1995.
- [Xi105] Xilinx. *Xilinx Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, v4.5 edition, October 2005.
- [YMHB00] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHI-MAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. 27th Int. Symp. on Computer Architecture (ISCA)*, pages 225–235, 2000.