Dissertation

Methods to Exploit Reconfigurable Fabrics

Making Reconfigurable Systems Mature

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering and Mathematics
of the
University of Paderborn
in partial fulfillment of the requirements for the

degree of Dr. rer. nat.

Florian Dittmann

Paderborn 2007

Supervisors:
Prof. Dr. Franz J. Rammig, Heinz Nixdorf Institute, University of Paderborn
Prof. Dr. Marco Platzner, University of Paderborn
Prof. Dr.-Ing. Jürgen Teich, University of Erlangen-Nuremberg

Date of public examination: December 14, 2007

Acknowledgements

This work was carried out in the course of exploring reconfigurable fabrics during my work as a research assistant in the working group *Distributed Embedded Systems* at the Heinz Nixdorf Institute, an interdisciplinary center of research and technology of the University of Paderborn.

First of all, I wish to express my deep gratitude to my supervisor Prof. Dr. Franz J. Rammig. He gave me the possibility to work in the field of reconfigurable computing and excellently guided me through my PhD journey. I could always count on his optimism, which more than once helped me to search for the way out when I was sure to be stuck in a dead-end road.

I also like to thank my two vice-supervisors Prof. Dr. Marco Platzner and Prof. Dr.-Ing. Jürgen Teich. I enjoyed having those two experts of the field of reconfigurable computing supervising my thesis.

I like to thank all my former colleagues, who helped me in different ways to finalize my thesis, among them Achim Rettberg, Marcelo Götz, Stefan Ihmor, Klaus Danne, Jörg Stöcklein, Norma Montealegre, Simon Oberthür, and last but not least Vera Kühne.

In particular, I want to thank Dr. Achim Rettberg and Dr. Marcelo Götz for the fruitful discussions on the benefits and challenges of reconfigurable computing. In both I found two inspirational researchers for discussions that substantially contributed to the quality of this thesis.

This work was also done within the SPP 1148 *Reconfigurable Computing Systems* of the DFG, the German Research Foundation. Besides all fellows I met during colloquia, I particularly want to thank Prof. Dr. Christophe Bobda, who heavily influenced the first application for this position.

Also, I want to thank all helping students: Fabian Schulte, Alexander Warkentin, Stefan Frank, Elmar Weber, Paulin Francis Ngaleu, Simon Schütte, and Helene Schilke.

Finally, I want to thank my parents. Their open-minded style allowed me to find my own way with the knowledge of their unlimited support. Thanks also to my sister Christine and my grandmother. Both always found the right words to help me believe in myself. Last but not least my lovely thanks go to my girl-friend Carmen and her patience. She tirelessly gave me the motivation to finalize this project.

Paderborn, December 2007

Abstract

The demands for flexibility and performance presumably are the two main driving forces of modern computing systems. Both are ever increasing requirements of users, devices, machines, processes, etc. Computer scientists and engineers constantly search for methods and architectures to solve these demands. Recently, reconfigurable devices—adaptable and fairly powerful computing resources—have approached this scene.

Reconfigurable devices appear to be most appropriate for high flexibility and high performance. They process in parallel and can change their behavior by loading different configurations on their fabrics, also during run-time. However, comprehensive exploitation of the paradigms of reconfigurable computing is rarely found in modern systems. In particular, dynamic reconfiguration—the adaptation of the behavior during run-time—combined with partial reconfiguration capabilities of modern FPGAs remains unused in most modern designs.

If we search for reasons, the answers are twofold. On one hand, we have to generally argue on what benefits partial run-time reconfiguration yields, as we have to overcome challenging technical hurdles. On the other hand, also few high-level methods exist that target the partial reconfiguration capabilities of modern FPGAs and would allow a system designer to exploit partial run-time reconfiguration.

This thesis approaches the situation described along the second train of thoughts—the need for methods. We discuss new methods to exploit run-time reconfiguration, which explore given reconfiguration constraints and often target at specific application areas. In order to evaluate the methods, we integrate them into an abstracting layered view on reconfigurable computing. We also discuss the technical challenges, which have to be overcome to implement the methods. Altogether and maybe most notably, the methods themselves explain where they can be useful, thus making a case for the use of run-time reconfiguration in everyday systems—making reconfigurable systems mature.

Zusammenfassung

Flexibilität und Leistungsfähigkeit (Performance) gehören wohl zu den am stärksten nachgefragten Anforderungen an moderne Rechensysteme. So werden beide in ständig steigendem Maße von Benutzern, aber auch Maschinen, Prozessen, usw. eingefordert. Informatikern und Ingenieuren obliegt die Aufgabe stets nach neuen Methoden und Architekturen zu suchen, die dieser Herausforderung gerecht werden. In jüngster Zeit sind nun die adaptierbaren und leistungsfähigen rekonfigurierbaren Rechensysteme als weitere Architekturen hinzugekommen. Ihre Eigenschaften weisen sie als besonders geeignet für das Erreichen von Performance und Flexibilität aus. So findet eine Berechnung auf rekonfigurierbaren Systemen im Raum, d. h. parallel, statt und kann nach Bedarf während der Laufzeit durch Rekonfigurierung angepasst werden.

Trotz der offensichtlichen Vorteile sind unter den modernen Rechensystemen wenige vorzufinden, welche die Möglichkeiten der Rekonfigurierung auch tatsächlich ausnutzen. Insbesondere bleibt die dynamische Rekonfigurierung, d. h. die Anpassung des Verhaltens während der Laufzeit, kombiniert mit der Möglichkeit zur partiellen Rekonfigurierung oftmals ungenutzt. Es lasse sich zwei Begründungen hierzu ermitteln. Einerseits muss grundsätzlich geklärt sein, wo die tatsächlichen Vorteile einer partiellen Rekonfigurierung liegen, da eine solche nicht kostenneutral zu erreichen ist. Andererseits existieren aber auch nur wenige Methoden, insbesondere auf höherer Abstraktionsebene, die es erlauben die Leistungsdaten von Systemen bei Ausnutzung der partiellen Rekonfigurierung zu evaluieren und letztere dann später auch gewinnbringend einzusetzen.

In dieser Arbeit wird dem Dilemma anhand der zweiten Begründung Rechnung getragen. Es werden vier neue Methoden diskutiert, die es erlauben Vorteile der partiellen Rekonfigurierung zur Laufzeit auszuschöpfen. Um die teilweise applikationsspezifischen Methoden zu explorieren, wird ein abstrahierendes Schichtenmodell verwendet. Zudem werden die für eine Implementierung notwendigen technischen Herausforderungen näher beleuchtet. Schlussendlich sind die vorgestellten Methoden nicht nur Entwurfsmittel, sondern stellen selbst neue Möglichkeiten zur Anwendung partiell rekonfigurierbarer Rechensysteme dar. Sie zeigen auf, wo der Einsatz möglich und praxisgerecht ist.

Contents

List of Figures

1 Introduction

Along with the increasing quantity of intelligent devices such as personal digital assistants (PDAs), mobile phones, driving assistants, industrial robots, etc., surrounding us, also the demand for the quality of these devices rises. Besides mechanical issues, design aspects, usability, etc., thereby the processing performance of such devices is of particular interest. Devices should be constantly faster in terms of processing speed to provide enough performance for the latest applications, while also consuming little power and being in-field adaptable to new emerging standards. The size of the processing units of these devices thereby should be as small as possible, so that designs can be customized and mechanical issues can be solved. Moreover, sufficient processing capacity can also help to decrease response time, which often improves usability. Eventually, the overall behavior of the processing unit in large part influences the product itself and often becomes the final sales argument.

Considering these challenges from the view of an embedded system engineer, we face a tremendous market pressure, combined with today's short time-to-market, when developing such modern products. During an ever decreasing design phase, engineers have to solve the market requirements of the device under development. In the extreme example, the specifications for the processing unit change virtually before market entry or even after the device was released. To serve the market, provide performance, meet low power requirements, and be still flexible enough for in-field adaptation can easily be drudgery under these circumstances.

The manufacturers of processing resources adapt to this situation by offering a variety of processing units to the embedded system engineer. These units follow several paradigms of processing and therefore offer specific computation characteristics that must be thoroughly evaluated for each design under development. Subject to suitable methods that support the evaluation of the design under development, the appropriately chosen devices can result in an advantage over the competitor.

Classes of Computing Media

What kind of processing resources are available? Basically, they can be divided into three classes. First, the GPP (General Purpose Processor), which usually bases on the instruction cycle based von-Neumann paradigm, is found in several versions within modern systems, e. g., as PowerPC, ARM processor, x86 type, or mutated as DSP (Digital Signal Processor). Usually if the performance of a GPP is not enough, dedicated circuits—ASICs (Application Specific Interconnects)—are used. They form the second category and offer high performance yet without the possibility to be programmed. The third category—reconfigurable devices, mainly FPGAs—fill this gap, as they can execute in parallel offering high performance and still being adaptable through reconfigurability.

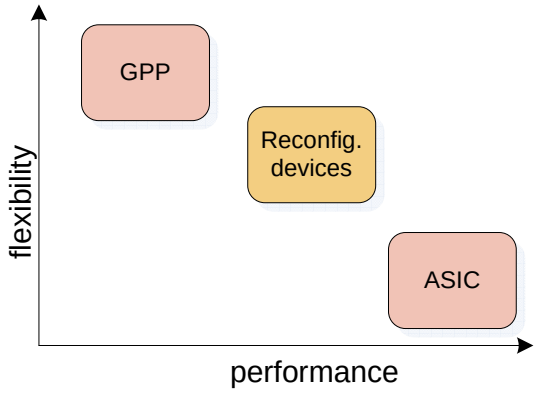

Figure 1.1: Performance vs. Flexibility

Worth mentioning, there exist multiple devices that are in between the three classes and fill in special niches, e. g., ASIPs (Application Specific Instruction Processors) or MPGAs (Mask Programmable Gate Arrays) [DW99].

When evaluating the three classes, we often encompass a graph as depicted in Fig. 1.1. The graph assigns high flexibility to GPPs, as GPPs can compute any function by executing different operations every instruction cycle. Specialized for general purpose, however, they offer low performance on average. The highest performance can be expected by ASICs. However, once fabricated the functionality of ASICs cannot be altered.

Reconfigurable devices are in between both classes. Thanks to their adaptability and reasonable performance, they combine characteristics of GPPs and ASICs. Obviously, they will never be as flexible as GPPs or well-performing as ASICs, however, reconfigurable devices in many aspects fill the gap between GPPs and ASICs, and thus serve the requirements of our modern systems in an extraordinary manner.

1.1 Reconfigurable Computing

Basically, reconfigurable devices thus combine performance and flexibility. Exploiting this mixture of the characteristics of the two standard classes of computation often is challenging. However, focusing on the important requirements of *processing speed* and *application flexibility* of modern computing systems, reconfigurable systems seem to be an appropriate solution and worth to be investigated further.

Processing in Space and Time

The fundamental difference of reconfigurable devices compared to GPPs and ASICs also is often formulated as the capability to compute in *space* and *time*. Space, as the devices offer computing area that is best explored by configurations, which make use of the parallelism of hardware. And time, as the device can be re-used—new configurations can be loaded on the same area, which was used by an even completely different application just before. To exploit reconfigurable devices, in particular FPGAs, as computing resources of modern systems, these specific attributes should be taken into account.

Run-Time Reconfiguration

Run-time reconfiguration is the most important technique for facilitating the reuse of hardware during the execution of an application. It contrasts with the so-called compile time reconfiguration, where the substrate is programmed once at the set-up time, without further alterations.

By run-time reconfiguration, we can execute applications that require more processing area than available and save physical devices, both often termed as *hardware virtualization*. Moreover, we can monitor the current requirements of a system and react accordingly by loading more suitable configurations. Thereby, in-field upgrades or bug fixes of hardware become possible. In summary, run-time reconfiguration permits time multiplexing of hardware resources and thus facilitates the processing in time and space.

Reconfiguration Overhead

Reconfiguration, however, comes with additional costs. This so-called reconfiguration overhead demands some more words, as it is the major constraint of reconfigurable computing in general and the main driving force for the methods of this thesis in particular.

Foremost, the time spent for reconfiguration, also termed the *reconfiguration latency*, cannot be neglected. Often it slows down a system significantly. To give a number, the reconfiguration phase of modern FPGAs is in the range of milliseconds.

Secondly, the reconfiguration is often done via a mutually exclusive reconfiguration port that must be accessed sequentially. As a result, the reconfiguration of more than one region at the same time (see partial reconfiguration below) is impossible in general.

Furthermore, the configuration information, generally speaking the bitstream, must be stored and therefore consumes storage area. The size of a bitstream for a modern FPGA can sum up to one megabyte or even above. Together with the storage, the whole reconfiguration process also consumes energy and thus adds to the overall demand of the system. In particular embedded systems often are power aware systems and any additional energy consumption must be seen critically.

A simple reason for the reconfiguration overhead is the fact that reconfigurable devices are primarily designed for fast processing, not for run-time reconfiguration. Thus most likely, the long reconfiguration phase of FPGAs will barely change in the future. Here, we can note two fundamental objectives that should drive any work in the area of reconfigurable systems. (1) If ever possible, reconfiguration should be avoided. (2) If we cannot avoid it, reconfiguration should be transparent and thrifty as possible.

Partial Reconfiguration

Nevertheless, some techniques exist to reduce the reconfiguration overhead. The fundamental one is partial reconfiguration. It forms a sophisticated possibility how run-time reconfiguration can take place. In contrast to a complete reconfiguration—programming the whole device—during partial reconfiguration only parts of the substrate are altered. While the former requires a complete stop of the device, the latter facilitates to continue processing as others regions are not influenced during the reconfiguration.

Partial reconfiguration basically allows us to hide the reconfiguration time. If we can pre-fetch the next reconfiguration context, we can overlap reconfiguration and execution.

Additionally, as smaller area under reconfiguration means a shorter reconfiguration time, partial reconfiguration also consumes less time. Still, partially reconfigurable systems cannot eliminate the costs associated with reconfiguration. However, they can minimize the reconfiguration overhead.

Exploiting Reconfiguration

In general, by (partial) run-time reconfiguration, we can obtain the demanded adaptability of modern systems as a means to cope with the market pressure and the decreased lifecycle of products. Moreover, the ability to change parts of the hardware on the fly during execution has proved a practical matter in many fields [GG05]. Partial reconfigurability in particular offers significant increase in flexibility for the execution of highly parallel algorithms retaining variability of hardware structural parameters.

However, to exploit the capabilities of partially reconfigurable systems and make them usable for system designers, appropriate methods are required. These methods thereby must respect constraints of reconfigurable computing, like area consumption of tasks loaded on reconfigurable fabrics, fragmentation, priority of tasks, etc. Moreover, there also should be little or even no increase of the design complexity. The paradigm reconfiguration should be well supported or even transparent for the designer. Hence, solutions are needed for the burden of reconfiguration effort, so that the concentration of the designer can be fully focused on the intended application. Thereby, appropriate methods may also help reconfigurable systems to penetrate the main-stream and completely unveil their benefits.

1.2 Contribution of the Thesis

This thesis targets on exploiting reconfigurable computing and therefore discusses four methods for partially and dynamically reconfigurable system design. We provide the methods based on a layered approach (see below) and evaluate them. Each method serves a specific scenario—a range of applications that can be implemented using the method applied. The methods are presented in increasing complexity—the first one is fairly straight forward targeting hardware virtualization. To allow for exploiting heterogeneous fabrics, the second method describes how reconfigurable fabrics may be included in a synthesis technique. The third one considers real-time constraints and beneficially uses the reconfiguration delay. Finally, the forth method introduces a completely new programming concept to reconfigurable systems on a very high level of abstraction.

All in all, the contribution of this thesis is the development and evaluation of design methods that consider the specific characteristics of reconfigurable systems. By virtue of our methods, we hope to open the field of (partial) run-time reconfiguration to a broader community, increasing the attraction and conquering new application fields, which have been so far under-represented due to a lack of appropriate methods. As auxiliary means, a tool that eases the design process and generation of partial bitstreams was developed.

In order to concentrate on the reconfigurable paradigm, we rely on abstraction. In particular, we mainly focus on the reconfigurable fabric solely. The classical task of

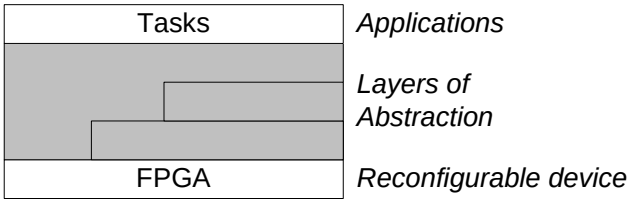

Figure 1.2: Layered Model

hardware/software partitioning, which often is an integral step during embedded systems design, may be considered as already done. Along with this, we do not consider processor (CPU) reconfigurable fabric (FPGA) interaction, the so-called coupling.

The major driving force of this thesis is to raise the level of abstraction, which is needed to tame the design complexity. In particular, such a step is inevitable if we argue—as is our opinion—that FPGAs have started to become mainstream. Extraordinary effort is needed to exhaust their potential and exploit the capability for partial run-time reconfiguration. Reconfigurable fabrics then may become more mature.

1.3 Abstracting Layered Approach

The surrounding lead for the methods shall be an abstracting layered model. It summarizes the overall motivation and structures what was done and where we rely on work of the literature. As layers allow us to separate or constrain the domains space and time, the layered approach also helps us to manage the two design parameters of reconfigurable systems, as can be seen throughout the thesis. The general layout of the model is depicted in Fig. 1.2.

1.4 Outline of the work

The thesis is organized as follows: We first summarize general characteristics of reconfigurable systems, in particular FPGAs. Then, we detail four design method or frameworks, also including application examples. A lesson learned at the end of each chapter allows us to gradually extend the characteristics considered. In detail, the chapters resemble the following:

Chapter 2 discusses reconfigurable computing in general. We start with an historical retrospection of the field. Then, we present techniques of reconfiguration and summarize design approaches of the field. Guiding principle of this chapter is the awareness of the potentials, benefits, and drawbacks of reconfigurable computing.

Chapter 3 introduces the first method, the so-called *two slot framework*. The framework is a direct approach how to hide the reconfiguration latency. It also focuses on the long reconfiguration times compared to relatively short time spent in execution.

In particular, intermediate results often must be hold back until reconfiguration has finished and processing can continue in the newly configured regions. To cope with these drawbacks, fundamental concepts are investigated under the constraint of not increasing the overall response time. The chapter also serves as a mean to detail partial reconfiguration. Moreover, some extensions to this basically known concept could be developed within this thesis.

Chapter 4 introduces partial reconfiguration to system synthesis. Particularly, the heterogeneity of modern FPGAs has been the driving force for developing the method of this chapter—a synthesis method respecting run-time reconfigurable devices. By virtue of the concepts employed in this chapter, we can also react on devices hosting multiple reconfiguration ports or systems that consist of more than one FPGA. Additionally, a whole system under development—including GPPs, ASICs, communication, etc.—can be modeled and explored. This can be achieved by the core concept of this method, which is the straight forward integration of the reconfiguration phase into a task/problem graph.

Chapter 5 introduces the new and within the course of this work developed concept of reconfiguration port scheduling. The approach allows us to target real-time applications. Aperiodic and periodic task sets may be executed—loaded dynamically—on modern FPGAs under real-time constraints using this approach. Therefore, the deadline d^* to denote the latest end of the reconfiguration process is introduced. The underlying execution environment resembles a slot-based approach.

Chapter 6 sketches another approach to exploit reconfigurable computing systems. Here, we apply concepts of algorithmic skeletons to reconfigurable system design. Algorithmic skeletons are implementation guidelines of the parallel computing domain that separate structure from the algorithm under development. By virtue of such a separation, evaluation of design in a very early phase of the development process are possible. Moreover, algorithmic skeletons enable a sophisticated concept of thread level parallelism on partial reconfigurable FPGAs without unsolvable fragmentation or communication demands.

Chapter 7 finally summarizes the thesis, draws a conclusion, and gives an outlook for future work.

2 Reconfigurable Computing

Reconfigurable computing dates back to the 1960s. Since then, several concepts have emerged—not all of them have survived. When asking for reasons, we list multiple answers. Most notably, along with the numerous benefits, we also enumerate a presumably similar number of obstacles that we have to overcome. The purpose of this chapter is to give a short and comprehensive overview of the core concepts and challenges. We therefore throw a glance at the fascination of reconfigurable computing, while not masking the problems, which are sometimes still unsolved. After a brief retrospective on the evolution in the field, we classify reconfigurable systems and the techniques proposed and in use nowadays. We discuss typical requirements for the use of reconfigurable systems as well as their application areas. Partial reconfiguration thereby is an important aspect of this chapter. Finally, such an overview would not be complete without some prominent examples of the literature, which are driven by the fascinating possibility to adapt hardware during run-time.

2.1 Introduction

Who has invented reconfigurable computing? This question may be unsolvable, yet, there exist two influencing approaches in the literature that discuss the topic a long time before the current research on (re)configurable computing has commenced.

2.1.1 Evolution

The first roots of reconfigurable systems can be traced back to Estrin's *fixed plus variable structure computer* developed at UCLA in the 1960s [EBTB63, EV62]. Estrin's machine consists of a standard processor surrounded by an array of reconfigurable hardware, with the main processor controlling the behavior of the adaptable part. The latter would be tailored to perform a specific task in hardware with the speed of hardware. Estrin's idea was well ahead of the technology at that time, so he only made a crude approximation of the idea. Nevertheless, the first conceptual approach for reconfigurable computing was born, and implemented in parts later [Est02].

While Estrin however stayed rather unspecific about the final implementation of the reconfiguration in his initial work, Franz J. Rammig—the second precursor—became very concrete on the implementation a decade after Estrin's idea. In [Ram77], Rammig describes his invention, which is a concept for editing of hardware, already presenting a similar architecture to today's FPGAs. Not only interested in the physical implementation, Rammig also discusses the requirements to obtain a hardware editor that allows

him to abstractly describe and finally generate hardware. As his reconfigurable substrate intends to host arbitrary circuits, he particularly focuses on the timing-behavior of signals. Additionally to the theoretical discussion on how to edit hardware, Rammig implemented his approach in the META-64 GOLDLAC using standard TTL-technology.

Besides these two approaches, no other works are known that discuss reconfigurable computing before the introduction of FPGAs in the 1980s. However, simpler programmable logic devices—without the ability to implement arbitrary circuitries—were already introduced under the term *programmable logic device (PLD)* in the 1970s.

Programmable Logic Devices

PLDs initially only described a class of simple configurable devices, however the term nowadays is increasingly used to summarize all devices that offer a (re)configurable substrate. The first concrete programmable devices in the context of the initial meaning have been the PLA and the PAL, both briefly discussed next.

Programmable logic arrays (PLA) consist of a plane of *AND* gates fed into a plane of *OR* gates. Their structure offers to implement any N input/M output function that match the size of the PLA. Thus, they resemble a basic and reasonably efficient approach towards reconfiguration as it is known today.

The programmable array logic (PAL) is a special case of the PLA with the *AND* array being the only programmable array; the *OR* array of the PAL is fixed. The fixed *OR* array increases the performance compared to a PLA; however, a PAL is not capable of implementing any arbitrary N input/M output function.

Multiple instances of both devices also have been combined to make up larger devices that offer more programmability. Such complex programmable devices (CPLD) often consist of macro cells that host disjunctive normal form expressions, facilitating deeper logic than simple PLDs. Furthermore, mainly implemented as non-volatile devices, they are a common choice to host and control the set-up configuration of field programmable gate arrays, among others.

Field Programmable Gate Arrays

The first arbitrarily reconfigurable devices have been the field programmable gate arrays (FPGA), which emerged in the mid 1980s. They were developed and commercialized by Xilinx, today still the largest company in the field, with the competitor Altera close to catch up. Basically, they outperform simple programmable logic like PLAs and PALs, because FPGAs can implement multilevel logic functions. The basic parts of an FPGA are a collection of programmable gates embedded in a flexible interconnect network, as well as programmable I/Os for the embedding and communication to peripherals. As being the main target architecture for the methods presented in this thesis, a detailed and comprehensive discussion is given in Sect. 2.2.3.

To illustrate the fascination of FPGAs a brief overview of the various functionalities offered by modern FPGAs shall be already given here: The original look-up-tables have emerged to become increasingly flexible by hosting carry-logic, flip-flops (to save states), etc. Additionally, dedicated hard cores such as multiply, multiply/accumulate, RAM, even complete processor cores can be found within the reconfigurable substrate

of modern devices. Moreover, some FPGAs offer partial reconfigurability, which means that regions are altered at run-time while the remainder stays untouched and active.

Splash and PAM

The capability for adaptability as given by FPGAs motivated several groups to investigate reconfigurable computing systems as new way to exploit high performance. Worth mentioning within this area are two early examples of the supercomputing domain that also can be termed the first reconfigurable computers. They were built by the IDA Supercomputing Research Center (SRC) in the USA and the DEC Paris Research Lab (PRL) in the late 1980s and early 1990s.

SRC built a systolic array called Splash [GHK+91] containing 32 Xilinx 3090 series FPGAs connected in a linear array. Some applications of Splash proved its power [Hoa93, Jai95] and thus Splash was followed by Splash II [BAK96]. Splash II offered a crossbar on top of the pure linear connection of the original Splash.

DEC PRL built a so-called *Programmable Active Memory* (PAM) named PeRLe-0 [BRV89] targeted towards image processing applications. The initial system contained the impressive number of 25 Xilinx 3020 FPGAs. Soon, DECPeRLe-1 followed as successor [VBR+96] and was effectively used in several areas, for example in the domain of neural networks [LLM95].

Both systems were driven by the goal to develop a new kind of supercomputer, composed of hardware-re-programmable components. Impressive benchmarks were shown, where the two approaches outperform software-only programmable computers of those days by one or two orders of magnitude. One of the biggest challenge for reconfigurable supercomputing however is the programmability, which still is subject of active research nowadays [BEGGK07]. For further details refer also to [GG05].

Berkeley Emulation Engine

For investigating high-end reconfigurable computing the BEE (Berkeley Emulation Engine) project—an attempt at creating a universal reconfigurable computing system—was started at the turn of the millennium. Currently, the BEE2 system [CWB05], a successor of the BEE1 [CKRB03], is in operation. BEE2 was designed to be a modular, scalable FPGA-based computing platform with a software design methodology that targets a wide range of high-performance applications such as real-time radio signal processing, simulation of large-scale, ad-hoc and traditional networks, or scientific computing.

The BEE2 system uses five large Xilinx Virtex-II Pro FPGAs as the primary and only processing elements. In addition to the reconfigurable fabrics, the BEE2 provides up to 20 GB of high-speed memory, including independent access of each FPGA to the memory using channels. Finally, the FPGAs on the BEE2 are highly connected with both high-speed, serial and parallel links.

To tackle the application at hand, users can choose the appropriate number of computational modules needed, including reconfiguration to switch to a different application. The most impressive application example that uses a set of BEE2 engines is the RAMP (Research Accelerator for Multiple Processors) project [WOK+06, KSW+07], which targets on investigating multi processor systems consisting of up to hundreds of cores.

Coarse Grain Devices

In the field, also another class of reconfigurable devices has been invented besides the dominating FPGAs. These so-called coarse grain devices derive their name from the fact that they can be programmed on a coarser level of granularity, mainly to reduce the costs of configuration logic. Examples are the PACT XPP device [PAC06], the Chimaera [HFHK97, YMHB00], or NEC's DRP (Dynamically Reconfigurable Processor) [Mot02]. For more detail, an up-to-date survey may be found in [Ama06].

Although a lot of research activity has been put into coarse grain devices, they are of little commercial success so far. Hartenstein concludes in his overview of different coarse grain devices [Har01a, Har01b] that each application must be well matched to the specific coarse grain reconfigurable device to gain reasonable benefits as we lack a universal form that is efficient for all applications. He thereby gives an agreeable argument for the bare market presence of coarse grain devices.

Systolic Arrays

Systolic arrays [KL78]—a pipe network arrangement of identical data processing units first mentioned in the late 1970s—shall be included into this section as another (arguable) influence to the emergence of reconfigurable computing. Basically, systolic arrays are two-dimensional arrangements of processors connected in a mesh-like topology, which can be found in modern reconfigurable devices as well. In both concepts, the processing elements are usually triggered by data-streams without local instruction counters.

The underlying principle of systolic arrays is to achieve massive parallelism with a minimum communication overhead. As systolic arrays are easy to implement because of their regularity, they have been discussed as a way to ease VLSI design, a novel technique at that time. Therefore, the concept of reconfiguration basically is unknown to them. However, within the realm of the XPuter research at the Technical University of Kaiserslautern, Germany [ABH+94], the KressArray [HK95, Kre96, HHHN00] was developed on the basis of systolic arrays as a reconfigurable version of systolic arrays. Furthermore, Vaidyanathan and Trahan investigate a reconfigurable mesh (R-Mesh) as a theoretical model to study reconfiguration [VT03].

Next Trend: Multicore?

Modern CPUs increasingly are multi-threading or offer a multicore design. Some of them just double the original CPU on the same die. Others offer a complete new design. Obviously, there is some relationship between multicore chips and reconfigurable devices, which are also executing in parallel and adapting the behavior.

The fundamental difference in most cases however is the program counter, which is optional in the reconfigurable computing domain and which multicore architectures do host in all of their cores. Still, the difference arguably is blurred. For example the Raw chip from MIT [TKM+02] is a two-dimensional array of programming tiles, each having a 32-Bit MIPS-like microprocessor, local instruction and data caches, and a 32-Bit pipelined floating point unit. It is a 2-D mesh network, not like a traditional bus-based multiprocessor system, and was developed under the motivation of exploiting reconfigurable computing.

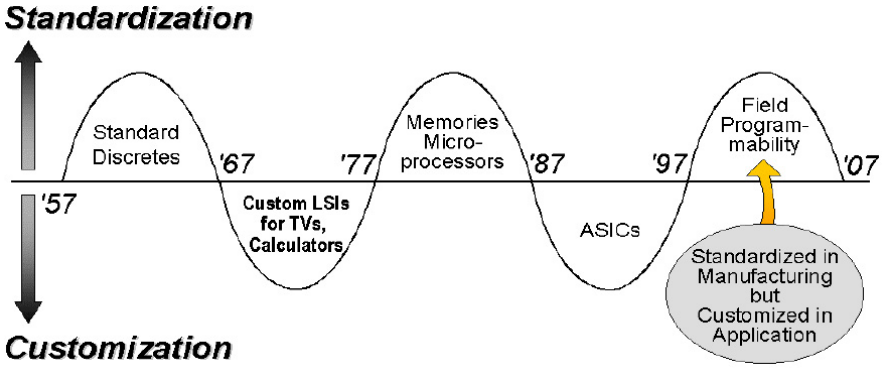

Figure 2.1: Makimoto's Wave [Mak00]

In the meantime, Tilera corporation launched a first chip—the Tilera Tile64—based on the MIT Raw project [Cla07]. The chip is called an embedded multicore device hosting 64 cores and targeting at low power applications in the embedded domain. It can run at 600 to 900 MHz with the cores arranged in a mesh style and equipped with three layer cache on each site. Furthermore, the design is claimed to scale beyond hundreds or even thousands of cores. However, the programming model for Tilera devices and all similar approaches is still in its infancies and demands for extensive research.

2.1.2 Makimoto's Wave

We also want to consider reconfigurable computing from another perspective, motivated by the question whether it is a surprise that reconfigurable computing now seems to become more and more widespread. Looking back, there have been some prognosticators, among them Dr. Tsugio Makimoto. He observed in 1986 that mainstream microchip application changes every ten years [Mak00]. This cycle—called *Makimoto's Wave*, see Fig. 2.1—describes a regularity in semiconductor trends, whose main feature lies in large-scale repetitive cycles oscillating between standardization and customization.

The background of the wave is the following: When large numbers of new technologies such as devices, architectures and software appear, the semiconductor industry as a whole moves towards standardization. Due to the need for product differentiation, added value, and the imbalance between supply and demand, the wave will reach a maximum and inverse its gradient. Next, progress in design automation and advances in technologies occur, shifting the semiconductor industry to customization. Then, reverse trends toward early market entry, cost reduction, and more efficient operation appear and the wave shows the longing for standardization.

From a macro viewpoint, the semiconductor industry can be said to repeat alternate phases of standardization and customization. Makimoto's wave thus is a concept of explaining the history of computing in general and the evolution of reconfiguration in particular. The wave predicts that the third wave will bring reconfigurable hardware into mainstream, as these devices can meet the requirement of standardizing ASICs.

Hartenstein proposed an extension of Makimoto's wave representing today's development [Har03]. In his opinion, coarse grain reconfigurable devices will be the technology for the next period of customization as they are more customized than classical FPGAs. Considering the modern trend of adding hard cores to FPGAs—making them hybrid devices—such a shift seems to be reasonable. Moreover, the trend to customize FPGAs towards application domains by offering multiple platforms that balance between dedicated circuits and freely programmable logic supports this prognostication.

Another abstraction to describe the evolution of processing resources was done by Tredennick—using the two categories *algorithm* and *resource* [Tre95]: After a period of both being fixed, we reached a time when the algorithms became variable—software could be and still is used to describe the algorithms, which are mapped onto the fixed resources (hardware) afterwards. Today, reconfigurable computing brings us to the point of both categories being variable.

Recently, Rammig formulated this combination of flexible algorithms and variable resources in the context of platform-based design [Ram07]. He particularly discussed the consequences of allowing dynamic reconfiguration at run-time—blurring the former clear distinction between application and platform. The latter may change its services offered, etc., making new modeling, analysis, and synthesis methods evident.

2.1.3 Remainder of the Chapter

As research on reconfigurable computing currently is very popular, and a complete summary would go beyond the scope of this thesis, we focus on important works with a good coverage of the ideas of the field in the remainder of the chapter. In particular, we first present a comprehensive view of the technical aspects in the next section, before we discuss application fields and design approaches. Being the main technique of this thesis, we thereby emphasize run-time reconfiguration. Finally, we conclude the chapter by a lesson learned and give a brief summary.

For further reading, we like to refer to some excellent surveying works of the reconfigurable system domain [DW99, TB01, CH02, SS05]. Also some books have been published recently [GG05, KKS04, VM05, Bob07, VS07].

2.2 Technical Aspects

After the historic-based introduction of reconfigurable devices, we below discuss remarkable techniques and details of theses processing units to understand why they claim to unite high performance and high adaptability. As the thesis is on exploiting run-time reconfiguration, we thereby emphasize techniques that facilitate this kind of reconfiguration, in particular partially reconfigurable FPGAs. An interesting question to be followed before eventually proceeding with the techniques of reconfigurable computing is the one on what *reconfigurable* actually stands for.

2.2.1 Reconfigurable versus Programmable

The ubiquitous and most successful device of reconfigurable computing—the FPGA, which stands for Field *Programmable* Gate Array—does not bear the word *reconfigurable* in it. So why speak of reconfigurable computing, and if so, where is the difference to programmable computing? It is not easy to give a satisfying enough answer. We try to approach the arguable still vague distinction by discussing answers or works from influencing people of the field.

When asked for the difference, Rolf Esser from Xilinx responded at the DATE 2007 workshop on reconfigurable computing:

> A reconfigurable architecture can implement a programmable system—hence "programmable" is a more general and higher level term.

Extending this abstract definition and looking conceptually at the difference, programmable is more linked to execution *instructions*, while reconfigurable means that the *structure* is changed. This adaptation however does not occur on the level of the physical hardware. The specific hardware fabric of a reconfigurable device is designed in such ways to be able to change the behavior—gates are re-used and their external connection is changed.

DeHon compares microprocessors and programmable logic using the notions of *instruction depth* and *datapath width* [DeH96b]. Microprocessors have an instruction depth of megabytes (on-chip cache) and a fixed datapath width of 8, 16, 32, or 64 bits, while classical reconfigurable devices like FPGAs and PLDs only store a small number of configurations (instructions), but offer a datapath width of 1, which makes them highly adaptable. Moreover, a software programmable microprocessor binds functionality at every cycle, which is not the case for FPGAs. However, considering also coarse grain reconfigurable systems, this clear distinction gets blurred. These devices comprise of datapath widths mainly in-between FPGAs and CPUs and also may rely on frequent run-time reconfiguration.

Nevertheless, the frequency of reconfiguration acts as main indicator for a distinction: While programming usually means that every cycle a new instruction is loaded or executed, respectively, reconfiguration does not occur as frequent. Both domains also support their intended reconfiguration/programming frequency by their construction.

Eventually, the distinction becomes manifested in the three categories of processing units of the introduction: ASICs are only once configurable, GPPs are programmable (reconfigurable) every cycle, while reconfigurable devices are in-between. In particular the difference of the reconfiguration frequency of GPPs, ASICs, and reconfigurable devices describes the situation along arguments that can be found all over this thesis: As *configuration* stands for a cost-sensitive process, the multiple occurrence of this step—resulting in one or more re-configurations—should be well thought off and soundly integrated in the overall design process and characteristics of reconfigurable systems.

Nevertheless, if we forbid reconfiguration, we would most likely waste capabilities of reconfigurable devices, as the adaptability drops down close to or even meets the one of ASICs. We thus have to derive a good strategy that conducts well-planned

reconfigurations based on a deepened knowledge of reconfigurable devices. We start by discussing the granularity of reconfigurable devices in the following.

2.2.2 Granularity

As already mentioned in the retrospective of reconfigurable computing, two categories of reconfigurable devices are distinguished nowadays: *fine grain* and *coarse grain* devices. The core difference between both categories is the granularity at which the functionality of the devices can be adapted. Fine grain means, we can adapt the functionality on a very low level—we may even add or remove an inverter or a gate of a circuitry. Coarse grain devices, in contrast, reconfigure on a much coarser granularity—most often their mesh of processing elements (ALUs, etc.) is adaptable in a VLIW-like (very long instruction word) style—complete instructions make up the reconfiguration information for a processing element. As coarse grain devices thus may fail to implement any hardware circuit, they are also sometimes termed pseudo reconfigurable devices [Bob07].

Granularity is also proportional to the configuration time and therefore can determine the frequency of reconfiguration. Coarse grain devices usually require a far lesser amount of reconfiguration information than fine grain devices like FPGAs, however accompanied by the drawback of a limited adaptability. Nevertheless, both exploit parallelism at multiple levels of granularity, from instruction through task level parallelism. Additionally FPGAs can make use of bit-level/boolean parallelism.

By referring to the granularity of reconfigurable devices as the abstraction level used to program or configure the device, we can facilitate a more detailed classification of reconfigurable devices concerning their level of granularity. During a detailed discussion of multiple academic research projects on coarse grain devices, this classification was also done in [KKS04]. The core concept thereby is to distinguish between devices reconfiguring on boolean level granularity, instruction level granularity, functional level granularity, and process level adaptability. While basic FPGAs are among the boolean level adaptable devices, several mainly theoretical research projects on (coarse grain) reconfigurable computing of the 1990s (mainly funded by the DARPA) are ranged in the three remaining categories. Noteworthy on top of that classification are hybrid reconfigurable devices, which intend to combine the benefits of the several approaches. According to [KKS04], modern FPGAs may all be classified as hybrid devices, as their embedded hard cores (DSP units, RAMs, CPUs, etc.) enable them to offer dedicated logic that is reconfigurable on a coarse level.

Concerning the market presence, coarse grain devices have little success so far. They are hardly commercially available and often still in their infants. Already mentioned in the retrospective, a beneficial matching of applications to coarse grain devices is a challenging task and must be performed mainly individually for each device. Nevertheless, the ideas of coarse grain reconfigurable devices have influenced our work. Some of our methods can also be mapped onto coarse grain devices [DB05].

Furthermore, in academia, several concepts exist to improve the acceptance of coarse grain devices. For example, a proposal of design steps for efficient usage of coarse grain reconfigurable devices enumerates *technology mapping*, *placement algorithm*, and *routing*

algorithm [Nag01]. Other approaches sort and exploit coarse grain devices according to their topologies [BGD+04, LCD03] and thus open them for a more general design space exploration using standard methods.

Nevertheless, fine grain systems are the dominating reconfigurable devices nowadays—most of all as FPGAs from the two global players Altera and Xilinx. They combine some excellent benefits: Fine grain reconfigurable FPGAs allow for efficient parallel processing, thus approximating the speed of ASICs. They can be programmed during run-time, which makes them adaptable devices. Some of them even support partial reconfiguration. To be able to soundly discuss methods for the beneficial exploitation of these modern capabilities, we put our immediate attention to FPGAs in the following.

2.2.3 Field Programmable Gate Arrays

The first commercially available FPGA—the Xilinx XC2000 [CDF+86]—already consisted of three core elements that make up the fundamentals of all modern FPGAs: Besides an array of *programmable logic cells* that can implement combinational as well as sequential logic, there are *programmable interconnects* that surround the basic logic cells, and *programmable I/O* cells that surround the two former parts.

Concerning the layout of these three core elements, multiple different concepts exist, which all go beyond the capability of only nearest neighbor communication of mesh-style systolic arrays or most coarse grain reconfigurable devices. In general, the communication infrastructure is closely related to the overall layout—the physical organization of the resources—of an FPGA. In particular, we distinguish a hierarchical, row-based, symmetrical array, or a sea of gates structure. The hierarchical arrangement is similar to a tree, where communication is routed to the parent nodes until the destination is among the children. Logic resources are mainly situated at the leafs of the tree. Row-based systems dedicate area for logic and communication in alternating stripes, therefore facilitating direct communication between logic resources only horizontally. For vertical communication, logic resources must be traversed. Eventually, the symmetrical array and the sea of gates structure comprise of equally distributed logic and communication in a two dimensional style. The difference between the latter two is that there is no space left for routing between the macro cells of the sea of gates arrangement. Interconnection of the logic resources thus takes place on top of the cells. The main benefit of this style is that arbitrary communication between the configuration cells is enabled.

The optimal arrangement has been thoroughly discussed, and the majority of the modern FPGA architectures now consist of grid style arrays of configurable logic blocks (CLBs) arranged as a sea of gates—also termed the generic island style FPGA model. Figure 2.2 depicts the typical layout of the three core elements of such an FPGA. The relatively smooth organization of the routing resources allows fast and efficient communication along the rows and columns of logic blocks. However, the flexibility of the island-style routing comes with a cost. Most current FPGAs use less than 10% of their chip area for logic, devoting the majority of the silicon real estate for routing resources.

The other communication topologies of configurable devices—hierarchical, mesh-style or row-based—can be found in coarse grain devices and some rare FPGAs. They may of-

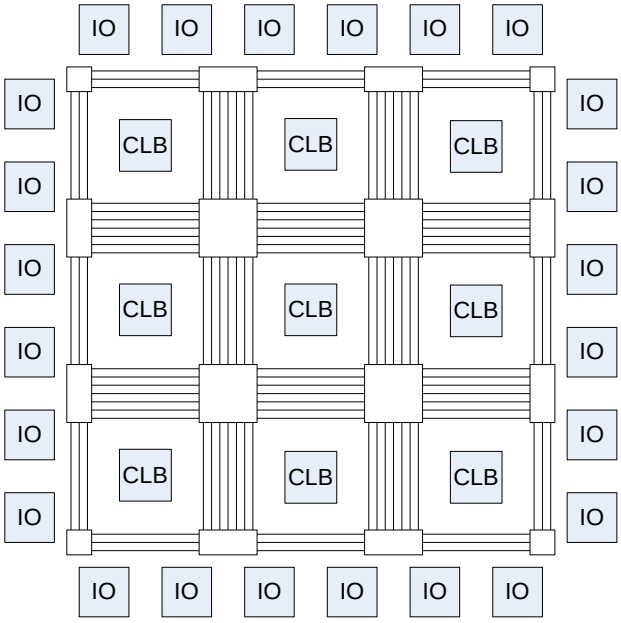

Figure 2.2: Typical layout of a modern FPGA

fer more sophisticated communication for specific applications, however, are less flexible on the average case.

Reconfigurable cells

Reconfiguration cells spread all over the three core elements—configurable logic blocks, communication infrastructure, and IOs—permit a very flexible customization and in-field adaptation. Concerning the physical implementation of the cells, we either find antifuse-based—special antifuses at every customization point—or memory-based—SRAM, Flash, EEPROM—systems. Antifuses require less area and have a lower resistance than memory-cell based systems. However, antifuse-based systems suffer from their one-time programmability and therefore are not suitable for spatial *and* temporal execution of applications, which is essential for (run-time) reconfigurable computing.

Nowadays, SRAM based configuration is the dominating technology—SRAM cells are connected to the configuration points in the FPGA, and setting the corresponding SRAM bits configures the FPGA. Their advantage of providing an indefinitely reprogrammability (configurability) eventually facilitates a reconfiguration on-the-fly during run-time. However, the chip area required for SRAM based configuration technique is relatively large. Moreover, due to the volatility of SRAM cells, the configuration must be reloaded at every new set-up.

Additionally, we have to bear in mind that referring to an SRAM-based FPGA as being completely arbitrarily reconfigurable as the term *SRAM (Static* Random *Access Memory) configurable* suggests is technically incorrect for the majority of FPGA architectures. If at all, we can only select a column or tile to be reconfigured. However, given the fact that modern memory also is accessed word-wise or even block-wise, partially reconfigurable FPGAs again are close to random reconfiguration.

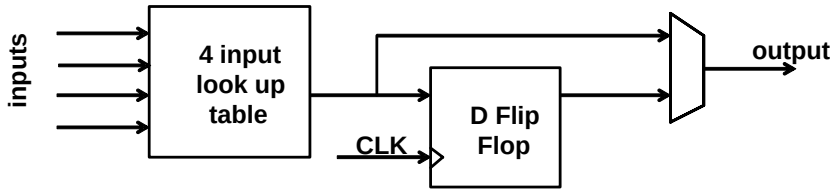

Figure 2.3: Schematic view of a configurable logic block of an FPGA

In general and independently from the configuration technology used, multiple customization points make up a larger configurable element in all modern FPGAs. These configurable logic blocks (CLBs)—sometimes also referred to as slices—consist of function generators like look-up-tables (LUT) and multiplexer (MUX), which are configured by multiple bits. For example, a k input and 1 output LUT can implement up to 2^{2^k} different functions relying on 2^k SRAM locations.

The typical FPGA has a logic block with one or more 4-input LUT(s), optional D flip-flops, and some form of fast carry logic. Figure 2.3 depicts the schematic core parts of such a configurable logic block. The LUTs enable any function to be implemented, providing generic logic. The flip-flops can be used for pipelining, registers, stateholding functions for finite state machines, or any other situation where clocking is required. Moreover, most modern cells provide fast carry logic as a special resource to speed up carry-based computations, such as addition, parity, wide *AND* operations, and other functions. A typical example of a modern configurable logic block—also termed slice by Xilinx—including carry logic and other enhancements is depicted in Fig. 2.4.

To configure the routing on an FPGA, typically a passgate structure is employed. Here, programming bits turn on a routing connection when the corresponding Bit is configured with a true value, allowing a signal to flow from one wire to another, and will disconnect these resources when the bit is set to false.

To finally discuss the third core part of an FPGA, programmable I/O basically means that we can select the I/O to be either input, output or bidirectional. This choice is usually provided by a circuitry that resembles the functionality of a tri-state.

Selected Additional Features

Partial reconfigurability can be increasingly found in modern FPGAs. Also described in more detail below, partial reconfigurability basically denotes the possibility to change parts of the current configuration of an FPGA without disturbing the other area not under reconfiguration. A helpful nature to support partial reconfigurability are the SRAM-style accessible configuration bits. Temporal processing thus becomes possible in a very flexible way, however challenging to explore, as can be seen in the remainder of this thesis.

Nowadays FPGAs not only provide a number of logical resources homogenously organized in an island style, they increasingly host multiple hard cores like multipliers, DSP (digital signal processing) units, MAC (multiply accumulate) units, high-speed serial I/O, distributed RAM, or even whole processors. Some of these hard core units are

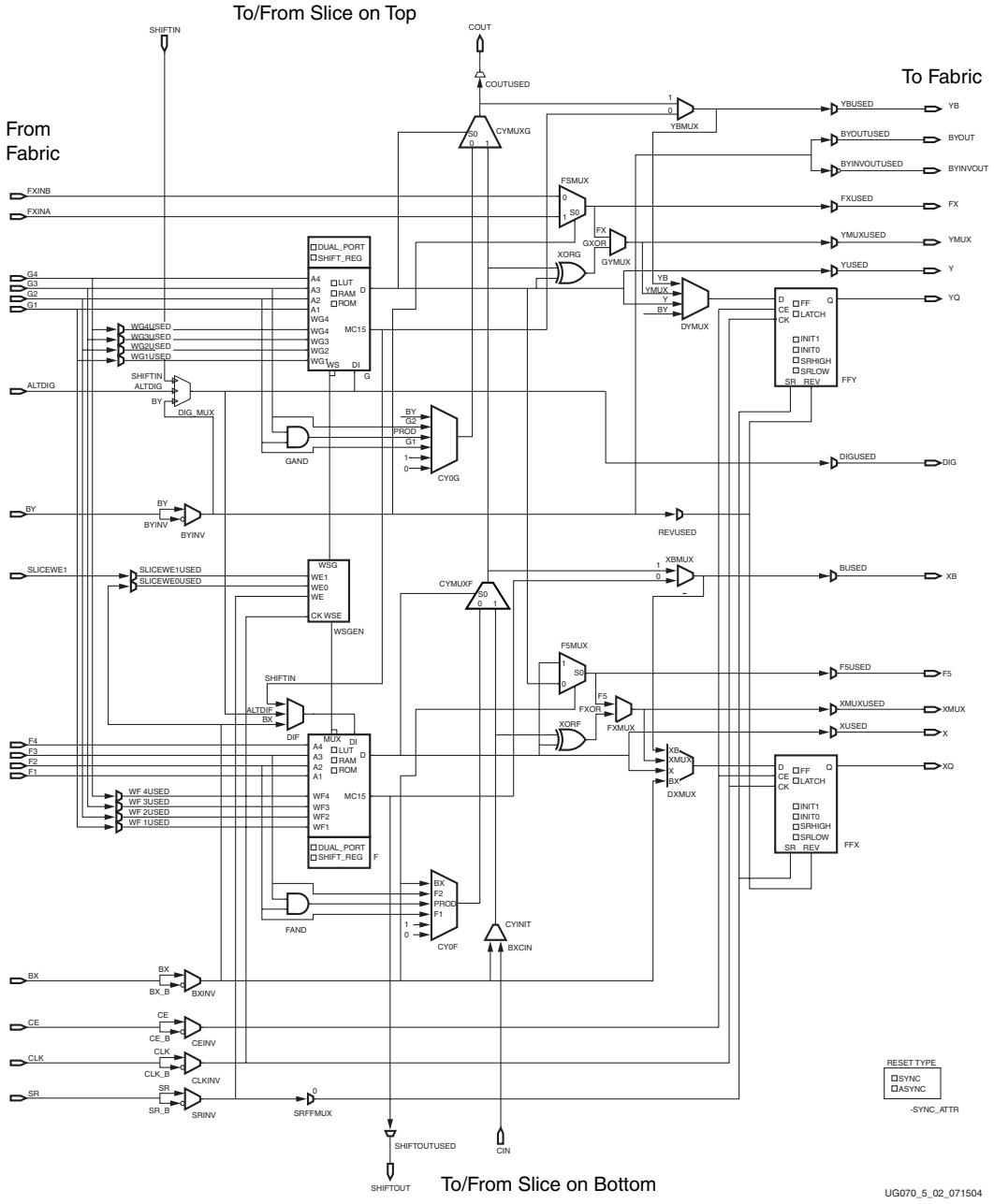

Figure 2.4: Slice of a Xilinx Virtex 4 FPGA

completely customizable, while others only facilitate a reconfiguration to a small extent. Hard cores increase the performance of FPGAs and can open them to new application fields bridging the gap between fine and coarse-grain devices.

To conclude the introduction of FPGAs, we summarize the advantages and disadvantages of FPGAs in the following. As ASICs are the natural competitors of FPGAs, the elaboration is dominated by the comparison of these two classes of computing media. An excellent resources for additional characteristics of FPGAs provides [Tri94].

Advantages of FPGAs

The main advantages of FPGAs over ASICs are the radically reduced development cost and turnaround time for implementing thousands of gates of logic. An FPGA design is done with relatively low risk, as a design iteration due to a design error neither exposes a large expense or a long delay. In particular, no expensive masks have to be manufactured, as would be the case for ASIC design.

FPGAs are also very well tested. Their high volumes combined with their regularity makes them easy exploitable for quality control. On top of that, we also can conduct efficient verification of designs aiming for execution on FPGAs—designs immediately serve as prototypes—which overall results in low testing costs. Also, as reconfiguration is a cheap and time efficient task compared to the design and manufacturing times of ASICs, iteration steps are relatively cheap.

Moreover, FPGAs are often available in the newest technology generations. For example, the current Xilinx Virtex-5 series are manufactured in 65nm technology. Basically, their homogeneous layout makes it easy to exploit the latest generation. A fact that can be similar found in the area of memory elements (RAM), which usually are among the first to change to new generations. By being in the front line of technology advances, FPGAs can exploit the benefits that come with every new generation, e. g., lower power consumption. The in-field customization also increases the life cycle advantages of FPGAs, as updates or in-field error correction becomes possible.

Comparing FPGAs to the other side of the scale—general purpose processors, DSPs, etc.—FPGAs offer the possibility to deeply pipeline and parallelize a design not given on von Neumann style alternatives. Their fine-grain customization and adaptability even allows for boolean level adaptability, which is impossible for GPPs or DSPs.

Disadvantages of FPGAs

The most severe disadvantage of FPGAs compared to an ASIC implementation of the same algorithm presumably is the size of a chip. As mentioned above, about 90 % of the area is used for routing resources in modern FPGAs. Therefore, the size of a circuitry implemented on an FPGA will never reach the area consumption of the same circuitry on an ASIC. This characteristic, which is due to the universality of an FPGA, also results in a reduced speed of circuitries when they are implemented on FPGAs.

The fine customization also yields another drawback. The large amount of reconfiguration points basically hampers low-power design, although modern FPGAs are increasingly designed towards optimizing the power consumption. Additionally, signal wiring and logic placement is relatively arbitrary, exact calculation of the signal delays is com-

plicate. FPGAs also come with a relatively large amount of pins, which hampers their use for very small embedded systems that operate on reduced bit-width or even bit-serial due to the costs of pinning or wiring.

While the heterogeneity of modern hybrid FPGAs is welcome concerning the performance, however, it must be treated correctly—the heterogeneity must be included into the design space exploration, etc. In general, the design methodology for FPGAs can be criticized. As FPGAs allow for fast testing, designers tend to implement designs more on a try-and-error basis than doing proper function testing, verification, etc.

Moreover, the volatility of FPGAs—which is true for the majority of modern devices as these are nearly all SRAM programmable—requires additional components to permanently hold the configuration, such as EEPROMS, etc. Subsequently, they also suffer a delay at start-up, waiting for the configuration to be loaded.

Nevertheless, much work has been done that proofs the benefit of fine grain granularity and high adaptability of FPGAs, e. g. in [GV00, DEM+00, HLTP02, LH02, SWP04, DBK03]. The possibility to partially reconfigure an FPGA discussed in many of these works thereby seems to be very promising. Before detailing this sophisticated technique for run-time reconfiguration, we first give an overview of programming techniques for FPGAs in the next section.

2.2.4 Programming FPGAs

Having discussed the typical technical characteristics of modern FPGAs, which allow such a device to execute arbitrary logic, we are also interested in how circuits for FPGAs are designed and how these are loaded onto the devices. Basically, the eventual configuration of an FPGA is done by a bitstream, which contains the value for each single SRAM configuration cell and is shifted bit or word-wise into the FPGA. The generation of such a bitstream however is not as easy as compiling a software programm.

The initial purpose of FPGAs was to enable hardware prototyping or to serve as (static) *glue logic*. Both domains target circuits that are used throughout the lifetime of the product—similar to ASICs—which is why bitstream generation is close to the ASIC front-end design flow. Creating such a design often requires costly optimization. As this process is usually only done once, we may accept these costs. Using FPGAs however as reconfigurable processing resources requires to frequently map different algorithms onto FPGAs. A trade-off between costs and time would be required.

Following the ASIC design flow, circuits should be described in a hardware description language (HDL) like Verilog, VHDL or by schematic entry tools. Then, by using special CAD (computer aided design) tools the bitstreams are obtained and can be downloaded to the FPGA. The time required to generate such a bitstream is in the range of minutes, hours, sometimes even days, heavily depending on the size and complexity of the circuitry and the target device. On-line generation of bitstreams thus is hardly possible.

The accepted steps involved when generating bitstreams from HDLs are *synthesis*, *mapping*, *place & route*, as well as the actual bitstream generation [Wan98]. These are mainly automated nowadays. In addition to the core HDL files that are capable of

describing the behavior and structure of the circuit, we require the user to give physical information of the system under development such as the timing characteristics of the device, mapping of specific hardware, or the connection to the platform/board the FPGA is mounted on (pinning). Such information often is collected in vendor specific files.

Due to the complexity and costs of the classical ASIC style design flow, several improvements have been proposed. For example, Xilinx released the PlanAhead suite, which is a tool to facilitate a structured design flow of FPGAs that resembles the current ideas of structured ASIC design. The focus of PlanAhead is to enable hierarchical floorplanning by streamlining the design steps between synthesis and place & route. PlanAhead offers rich functionality, however hardly raises the level of abstraction.

Concepts to completely move to a higher level of abstraction are also investigated. These are mainly influenced by current high level languages (HLLs) or reasonable extensions of them. Motivated by easing the design, the PRISM compiler was the first to investigate hardware compilation of sequential C kernels [AS93]. Nowadays most of the current approaches still rely on C-like languages like Stream-C [GSAK00], HandelC [BW06], or ImpulseC [PT05]. SystemVerilog in contrast enriches standard Verilog with features from object oriented design [SMR+04]. The major benefit of these approaches relying on HLLs is that they facilitate immediate simulation and can also include software parts. Often, the mapping onto reconfigurable devices takes part within a hardware/software codesign environment, e. g. [STB06].

However, when HLLs are used, the quality of compiler-generated circuits still lacks the equivalent manual designs, often by factor of 2–4 in area [Bob07]. Some of these languages even make an immediate synthesis to hardware impossible, requiring a complete new implementation of the final hardware part in the worst case. The low performance of these C-like languages thereby stems from their closeness to iterative programming, which originally targets the instruction code execution of von Neumann machines.

Downloading/Programming

Eventually generated, the bitstreams can be loaded via configuration access ports into the configuration cells (SRAM, Flash, antifuse, etc.) of an FPGA. In addition to the JTAG port, which originally was designed for debugging, SelectMap (8/32-Bit parallel) and Slave Serial (Single Bit Serial) exist for this purpose. The parallelism of SelectMap (8 or even 32 bits) speeds up the configuration process. Modern FPGAs like the Xilinx Virtex devices also offer self-reconfiguration through a so called ICAP (internal configuration access port). The ICAP is connected on chip to the SelectMap port, sharing the configuration resources. Finally, some of the ports also enable a readback of the configuration for debugging or re-engineering purposes.

For volatile FPGAs, the bitstreams are often stored in non-volatile devices next to an FPGA and loaded on start-up. To overcome this drawback of an additional chip and area on the board required, in the meantime, some devices exist that offer in-built non-volatile storage such as the Xilinx Spartan 3AN device that comes with the reconfigurable fabric and a flash combined into a single package.

Frequency of Reconfiguration

To basically classify the frequency of reconfiguration for further use in this work, we follow the definitions given in [Bob07]. For a non-frequent reconfiguration, where an FPGA becomes a static device and usually serves as ASIC replacement or for rapid prototyping, we use the term *compile-time reconfiguration*. Here, the loading of the bitstream is done before the FPGA is in use, holding the configuration during the operation time of the application. However, if FPGAs shall adapt their behavior multiple times during their execution—changing their configuration more frequently—we refer to their reconfiguration as *run-time reconfiguration*. In both cases *partial reconfiguration* would be technically possible. However, for a rarely occurring compile-time reconfiguration, the additional overhead to generate a partial bitstream often does not make sense.

Concluding the state-of-the-art programming of reconfigurable devices, true high-performance techniques to design and eventually download circuits for FPGAs are hardly available. This fact holds for both reconfiguration techniques—compile time and run-time. The latter more sophisticated and also more challenging reconfiguration technique, which however eventually allows us to exploit the entire capabilities of reconfigurable fabrics—processing in space and time—is discussed next.

2.2.5 Run-Time Reconfiguration

Basically, by run-time reconfiguration the configuration of a reconfigurable fabric is altered during its operating time, in the best case without halting the current processing. The immediate benefits are the reduction of the hardware resources: timesharing the available hardware and also hardware specialization. Moreover, in-field customization or power reduction by holding only those circuits that currently are required becomes possible. Section 2.3 details application fields for run-time reconfiguration.

While benefits of the possibility to reconfigure FPGAs during run-time quickly become obvious, the question of the cost for reconfiguring a device during run-time arises. Along with several design challenges discussed all over this thesis, also technical constraints exist that must be considered for achieving a benefit. Among them is the configuration overhead discussed next.

Configuration Overhead

The sheer size of the bitstream offers a challenge for run-time reconfiguration, as bitstreams are usually shifted in small units or even bit-wise into the reconfigurable device consuming a non-negligible time period. Naturally, the question how to handle this notable reconfiguration latency arises. Different concepts have been proposed.

A very promising idea is to overlap reconfiguration and processing phases. For example in a coupled system, while the processor is active, we can reconfigure the substrate in the meantime. If we can load a new configuration as soon as possible and prior to its execution, we increase the chance to finish exactly at or even before the algorithm under processing requires this configuration. Then, we have achieved a so-called *configuration prefetching* [LH02, GV00].

A comprehensive example is given in [Hau98], where the author focuses on reconfigurable co-processors. In order to load configurations in advance, special instructions are added to the code. Thereby, the maximum number of saved cycles is limited by the structure of the code. Moreover, the compiler re-arrangement of operations must be considered to avoid collision with the already inserted pre-fetch operations.

In addition, complete systems like PipeRench [CWG+98] have been designed that inherently target the reconfiguration overhead by proposing an architecture that allows reconfiguration and processing at the same time. As such pipeline reconfigurable environments require dedicated design processes and their specific architecture may even exclude some applications from practical execution on them, they are little present.

Mainly academically hosted attempts also propose multi context reconfigurable devices, e. g. [DeH96a, TCJW97]. Multi context reconfigurable devices include multiple memory bits for each programming bit location—thought off as multiple planes of configuration information. These devices can perform a switch between configurations in few clock cycles. Their inherent architecture makes them advantageous to reconfiguration time hiding, while the additional hardware required inevitably increases the costs. Most likely therefore, they are not commercially available.

Another approach is to target the number of reconfigurations required and reduce them as much as possible either by configuration caching or by avoiding reconfiguration at all based on decisions on higher abstraction levels. Some approaches also discussing the drawbacks can be found in [DST99, LCH00].

Moreover, a combination of the methods obviously increases the benefit gained. For example, the combination of configuration caching and configuration pre-fetching is proposed in [LH02]. Swapping configurations with respect to caching possibilities is discussed in [SNG01].

As the reconfiguration time can be often considered as proportional to the area under reconfiguration, there also exist concepts that target at minimizing the configuration data. These concepts often compress the bitstream, some of them already at the time of generating the bitstreams like the combitgen project [CMS06], which however is located in the realm of partial reconfiguration (see below). In the combitgen approach, we reduce the size of the bitstream by only configuring those bits that change from the current to the next configuration. Such a difference based reconfiguration particularly is of success if the amount of bits can be reduced significantly. As shown in [RM07], we can also influence the size of (partial) bitstreams at earlier stages—during the synthesis, etc. Circuits that shall share the same area therefore are generated in parallel, so that similar constructs can be mapped on the same resources, avoiding a reconfiguration of these resources.

Partial Reconfiguration

Partial reconfiguration provides another and extraordinary powerful method to ameliorate large configuration times for FPGAs. The important characteristic of partial reconfiguration summarized, it is a technique to change only parts of a reconfigurable fabric, requiring only a fraction of the complete bitstream, while also leaving other regions untouched—these regions can remain fully active.

By using partial reconfiguration, designers can dramatically increase the functionality of a single FPGA, allowing a system to be implemented with fewer and smaller devices than otherwise required. The idea of partially reconfiguring FPGAs can already be found in some works of the mid 1990s, e. g. [LD93]. One of the first comprehensive academical attempts at partial reconfiguration then was the DISC project [WH95], which offered demand-driven modification of instruction sets. During that time, only the FPGAs produced by Algotronix and Actel enabled partial reconfiguration.

Xilinx then took over Algotronix and launched the XC6200 series [Wan98]. These devices offered a RAM-style configuration interface, where the CLBs could be accessed in a matrix-like style. However, the device had little commercial success and was quickly abandoned. Nevertheless, it inspired a lot of research work, e. g. [CCKH00].

Today, the research on partial reconfiguration concentrates on the Xilinx platforms. Currently, Xilinx Spartan, Virtex-II, Virtex-II Pro, Virtex-4, and Virtex-5 FPGAs support partial reconfiguration. Additionally, Atmels FPSLIC series offers partial reconfigurability, however the capacity of these devices is limited. Therefore, Atmel fabrics are seldom the devices of choice [Atm02].

Although fascinating and increasingly supported by modern FPGAs, partial reconfiguration still is challenging to be explored and complicate to be implemented. For example, partial reconfiguration can result into fragmentation on the reconfigurable substrate. Relocation and defragmentation can therefore become necessary [CCKH00, CLC+02], and must be integrated in the above discussed methods like configuration prefetching, etc., which otherwise harmonize perfectly with partial reconfiguration [LH02].

Task reallocation can also increase the flexibility of a reconfigurable fabric by freeing area for dynamically arriving tasks [KKP05a]. Some works also consider run-time defragmentation algorithms for heterogeneous devices [KKP06]. However, the relocation of tasks on FPGAs is complex and cost intensive, in the worst case a complete readback of the current status of the logic cells of the reconfigurable substrate might be necessary. Some works therefore also consider preventative defragmentation [KKP05b] using best fit methods to allocate area for dynamically arriving tasks.

Run-time Routing

Besides the challenge to allocate dynamically arriving tasks, the communication of these tasks with the rest of the system must be considered. In an early example, we already find a discussion on how to perform such *run-time routing* of data between blocks of circuity on reconfigurable systems [BD98]. The authors present three models to evaluate trade-offs between flexibility, speed and cell count. Depending on the used fabrics, their routing supports highly parallel or intrinsically serialized communication. Similar research is still done nowadays [PAK+07].

Thereby, also bus networks or even Networks on Chip (NoC) [NTI04] are investigated as solutions for run-time routing. For example [ESS+96] introduces a reconfigurable multiple bus network, which is intended to support circuit switching as means of communication between processing elements. The model comprises n processing elements and k segmented busses. Several switches are used to set the connection at run-time—making up a modular communication infrastructure on a reconfigurable device.

Basically, NoCs heavily have influenced multiple research projects on run-time routing [BAM+05, MAV+02]. The general idea is appealing: Processing elements are connected to network elements (routers), achieving a packet-based communication. We avoid the bottleneck of a simple bus, because the routing resources are shared by all connected blocks. However, NoCs consume significant area on an FPGA.

There also exist scenarios, when buses or NoCs are not applicable, for example if inter-module communication has to be guaranteed in hard real time [GSC04] or the amount of data to be communicated is too large for bus or NoC based systems (e. g. in the area of video/audio data streaming). Direct communication must be used then [DH05].

Self Reconfiguration

Self reconfiguration entitles an approach of reconfiguration in a self-managed fashion with in-situ reconfiguration. A necessary requirement is the access of the reconfiguration port of a device from the device itself. The Xilinx ICAP (Internal Configuration Access Port) facilitates this requirement and allows us to achieve self reconfiguration for Virtex devices. As already mentioned above, concerning the physical implementation, the ICAP port connects to the SelectMap configuration port.

Self reconfiguration can become a desirable feature that can increase the performance of a reconfigurable system, see [BJRK+03] among others. Basically, self reconfiguration means the triggering of the reconfiguration process from within the device itself. Obviously, it is closely connected to partial reconfiguration, as the controlling instance should remain static and active during the reconfiguration process.

Challenges of (Partial) Reconfiguration

Nevertheless, all approaches have to cope with the constraint of a single reconfiguration port as limiting factor that will hardly change and been overcome in the future. In particular, the single reconfiguration port means that reconfigurations must take place mutually exclusive. Moreover, another constraint exists concerning the physical access to the reconfigurable substrate. In order to keep the number of pins required for the reconfiguration hardware low, the reconfiguration—submitting of the bitstream—takes place word wise, byte wise or even bit serially, consuming much time for transmission as modern bitstreams may sum up to 1 MByte or even above.

When considering the subject of reconfigurable computing from an abstract point, these challenges however are obvious and harmonize with the general fact that FPGAs are designed for fast processing and not for fast reconfiguration. Considering the history of FPGAs as ASIC replacements and the general idea of increasing the performance—mainly the computational performance—this focus of reconfigurable devices on fast processing and not on fast reconfiguration completely makes sense.

This situation of why we have and presumably will always have a reconfiguration overhead further supports the fact that reconfiguring an FPGA during run-time is not trivial on many aspects, particularly concerning the design process of reconfigurable systems themselves and the mapping of applications onto such adaptable devices in detail. All the issues mentioned—reconfiguration overhead, operation during reconfiguration, etc.—must be considered comprehensively.

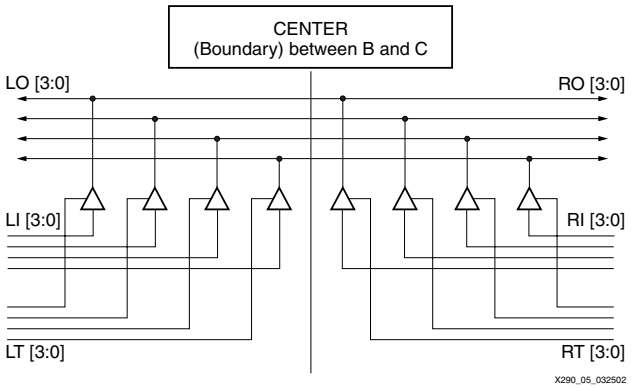

Figure 2.5: Busmacro of the Xilinx Application Note 290

Xilinx Partial Bitstream Generation

As influencing the methods of this work, we also discuss the current state-of-the-art on partial bitstream generation for Xilinx FPGAs. Such *partial bitstreams* basically are downloaded to the FPGA using the same procedure as for complete bitstreams. However, to assign the configuration information to the intended area, the configuration port (Select Map, JTAG, etc.) evaluates the header information of the bitstreams, which states beginning and length, always denoting a consecutive area. To generate such special bitstreams, an appropriate design method is required.

In general, the new configuration information of a dynamically loaded bitstream (partial or complete) is loaded glitch-free and overrides the current condition of the SRAM cells. Given this characteristic of the Xilinx FPGAs, basically two possibilities arise to dynamically alter the configuration of an executing circuitry on an FPGA. Firstly, we can adapt the whole configuration of an FPGA and merge it into a new one by calculating the difference of the two configurations and if possible only configure the different SRAM cells. This way is called the *difference based reconfiguration*. Secondly, we can determine special regions of the FPGA as being partially reconfigurable. Such a design flow is termed *module based reconfiguration*.

The Xilinx Application Note 290 [Xil04] was the first document given by Xilinx to achieve difference-based as well as modular partial reconfiguration. For the generation of partial bitstreams, it mainly follows the design flow for a modular design, where a project leader partitions the area of an FPGA into smaller regions, defines the communication between the regions and assigns each region to a specific design team. The tasks to be designed in the regions add up to the whole functionality of the system under development and therefore must be integrated afterwards.

For the communication between the regions, specific static communication resources were defined. These so-called busmacros serve as anchor for the dynamically loaded modules. They are inevitable as the non-existence of static communication and connection points would eliminate the guarantee of a proper communication. The design flow XAPP 290 therefore includes tri-state based busmacros, see Fig. 2.5. These are hard-coded macros that use tri-states and specific lines of the Xilinx Virtex-II series.

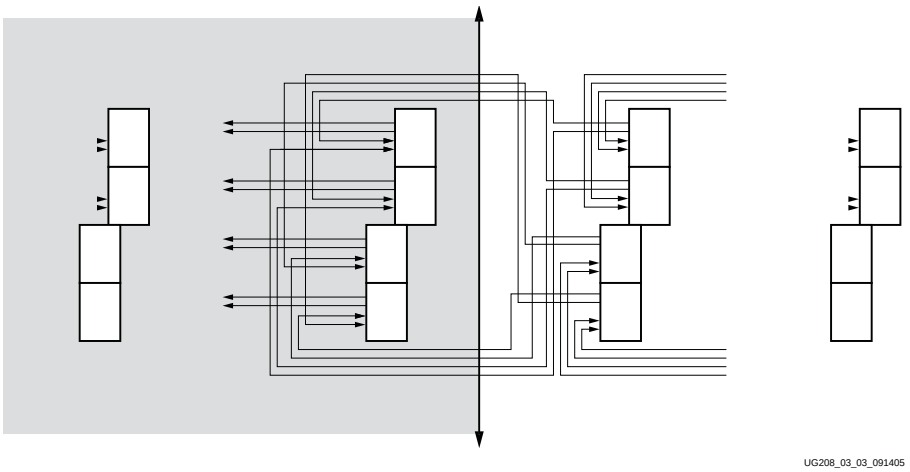

Figure 2.6: Narrow busmacro of the Xilinx early access design flow

Concerning the geometrical requirements in XAPP 290, reconfigurable regions must always span the whole height of an FPGAs, as XAPP 290 allows only column-wise reconfiguration. Several drawbacks arise, among them the difficulty to route signals crossing a reconfigurable region and the inclusion of IO pads into the partial bitstream of the column under reconfiguration. Yet, the column-wise reconfiguration is in line with the capabilities of the Xilinx FPGAs available at that time.

Along with the launching of the Virtex-4 device, Xilinx came up with a new flow to generate partial bitstreams. This so-called *early access partial design flow* is the up-to-date approach to perform the generation of partial bitstreams [LBM+06]. Basically, it follows the same idea as presented for the modular based reconfiguration of the Xilinx application note 290. However, several improvements have been added. Most important, the restriction of only column-wise partial reconfiguration has been dropped. Here, the architecture of the Virtex-4 FPGAs is of particular benefit, as tiles on top of each other make up the layout of the FPGA—similar to the standard cell layout of ASICs.

Furthermore, lookup table based busmacros have been introduced, see Fig. 2.6. This step became inevitable, as the Xilinx Virtex-4 do not offer tri-states within their reconfigurable substrate any more. Lookup table based busmacros also are more flexible and enable horizontal and vertical connections. While the hitherto presented busmacros needed to be placed on a boarder of the region rectangle, the latest ones can be placed anywhere within the region [Jac07]. These so-called single slice macros also eliminate the need to define a direction for the busmacros. However, they are currently only available for the devices of the Xilinx Virtex-5 series.

Partly in close cooperation to Xilinx, several works have been and still are carried out to improve the routing and communication issues of partially reconfigurable designs. For example, several very sophisticated concepts are discussed in the PhD thesis of Hübner [Hüb07]. The author investigates the communication capabilities of Xilinx FPGAs towards on-demand routing exploiting multiple physical characteristics of the devices. Presumably, he was also the first to propose LUT based busmacros [HBB04].

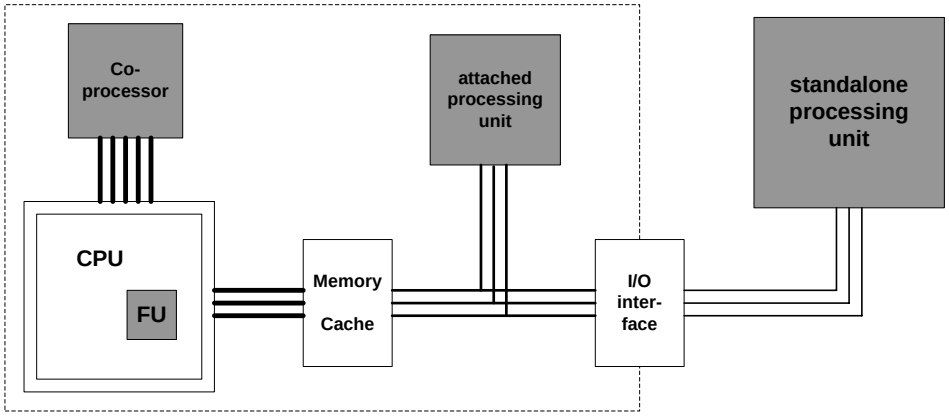

Figure 2.7: Coupling of reconfigurable devices

Both solutions presented above for the generation of partial bitstreams offer a sound theory, however their practical implementation quickly becomes cumbersome and time-consuming. A discussion on the main aspects of partial reconfiguration on Xilinx Virtex FPGAs can also be found in [BBHN04] and [BAR+05]. The authors discuss challenges like the signal integrity, global logic and inter-module communication. Also Chap. 3 discusses the challenges in some more detail.

Moreover, the PlanAhead tool of Xilinx has recently been enriched with the possibility to generate partial bitstreams [DSG05]. However, only the latest version (9.2) [Jac07] gradually enables to adapt to specific user needs, while the former version enforced to strictly follow the early access design flow as commands are invoked automatically in the background. In particular, for each different combination of reconfigurable modules, a single project had to be established.

2.2.6 Coupling

When using reconfigurable fabrics as processing devices, the question arises how they are integrated in the overall system, and in particular if necessary how a run-time reconfiguration is triggered. Most often, the substrate therefore is coupled with a controlling unit—implying that at least two devices become part of the system. However, reconfigurable systems can also comprise of stand-alone devices, for example as given by the ml310 board of Xilinx or several of the processing platforms of Celoxica Inc. In such systems, an internal control unit such as the hard core PowerPC found in Virtex-II Pro FPGAs of the ml310 board or user generated logic of the Celoxica boards is used for software parts like the operating system [Dan06], among others.

The interaction of the reconfigurable substrate with a controlling/supervising unit can range from loosely coupled to closely coupled. Figure 2.7 depicts four different coupling techniques as described in [CH02]. A similar definition can be found in [KKS04], additional referring to the levels of the Y-chart [GK83]. There, the loosest coupling comes at the system level and is characterized by slow transfer rates. At the architecture level, the reconfigurable devices act more like a coprocessor. Finally, the micro-architecture

level resembles the tightest coupling. Here, the fabric becomes a functional unit within the data path. A similar yet more detailed discussion is given in [Kal04].

The sort of coupling chosen may also affect the times of data transfer from and to the reconfigurable fabric, in particular if the data is transmitted via the supervising unit. If the reconfigurable device has immediate access to its data, however the coupling also may become negligible. As shown by the ml310 board discussed above, an FPGA may smoothly represent the sole processing device of a system.

When run-time reconfiguration is used, the coupling also may determine the appropriate frequency of reconfiguration. Loosely coupled devices may accept a longer reconfiguration phase than closely coupled devices. However, it is hardly possible to agree on the best coupling technique for run-time reconfiguration. Arguably, coupling and run-time reconfiguration may also be seen orthogonal to each other, as run-time reconfiguration can yield benefits for every coupling method.

Moreover coupling still is evolving. For example, Intel recently has opened the specification of their Front Side Bus (FSB). Xilinx took over this opportunity and implemented an Intel FSB-FPGA accelerator module that features an FSB-capable Virtex-5 FPGA module as a plug-in to an Intel Xeon CPU socket. Xilinx already demonstrated to achieve bus speeds of 800 MHz capable of supporting 6.4 Gbytes/s data transfers.

2.3 Fields of Application

After discussing the technical aspects of reconfigurable systems—in particular FPGAs—this section focuses on the past, current, and future fields of application. Basically, the majority of these fields still ignores the capabilities of partial or even run-time reconfiguration. When it thus comes to examples of reconfigurable computing, the greater number of the applications rely on compile time reconfiguration only. However, many application areas can be extended to also benefit from run-time reconfiguration. In this section, we thus do not separate the fields of application into their reconfiguration frequency. We rather sketch the general application areas of reconfigurable computing devices according to their characteristic of combining benefits from GPPs and ASICs, and detail them by considering compile time as well as run-time reconfiguration based options.

2.3.1 ASIC Design

A quiet old application field of FPGAs is the domain of ASIC prototyping, where FPGAs are used for the so-called *rapid prototyping* [KPC94]. The idea is to emulate a hardware design and test it towards its correctness. Rapid prototyping allows a device to be tested before the final production, particularly errors can be corrected before the design is manufactured. The fine granularity of FPGAs makes them the best choice for rapid prototyping, with coarse grain devices hardly applicable to the task.

As shown in [Bob03] and other works, the reconfigurability of FPGAs can also be exploited for rapid prototyping. By performing a so-called temporal partitioning of the

circuit under development, we can emulate parts of the overall design in sequence on the same FPGA. We can thereby renounce the acquisition of an otherwise costly system hosting a larger FPGA or multiple FPGAs. Rapid prototyping thus can be extended for run-time reconfiguration. It helps to save FPGA resources if the ASIC to be simulated is too large to fit on one device.

Furthermore, if (at least partially) asynchronous circuits have to be prototyped, timing becomes a first class issue. Janzen and Rammig therefore reanimated Rammig's early work and did show how the timing behavior can be prototyped using today's FPGA technology [JR97a, JR97b].

2.3.2 Replacement of Dedicated Circuits

As already pointed out in Sect. 2.2.3, reconfigurable devices—particularly FPGAs—come with multiple benefits that make them superior to ASICs in many cases. Therefore, FPGAs are used as ASIC replacements in niches since their beginnings, particularly as glue logic (see below). Nowadays increasingly and mainly because ASIC fabrication costs have grown enormously using technologies below 100nm, FPGAs have matured to become complete ASIC replacements, sometimes even if high volumes of a design are required. Obviously, there always is a trade off—such as the higher energy consumption—if reconfigurable devices are used instead of ASICs. We therefore need a comprehensive investigation of all relevant constraints for each specific application.

Domains

Using a reconfigurable device as communication bridge between two or more devices is often called *glue logic*. The reconfigurable substrate then holds the custom electronic circuitry needed to achieve compatible interfaces between the different off-the-shelf integrated circuits. Thanks to their adaptability, reconfigurable devices—FPGAs preferred—perfectly serve the requirements of such an interface [Wan98]. Depending on the characteristics of the incompatible devices, also coarse grain devices can be used. CPLDs, which have been the former dominators of the field, are little employed nowadays as they fail to offer enough adaptability for the increasingly complex protocols.

Particularly if circuits are added dynamically to a system, the (run-time) adaptability of reconfigurable devices proves to be of benefit. By virtue of reconfiguration, we can offer a customized interface for the new circuit. Such a behavior is of particular interest in the domain of IP (intellectual property) design for Systems on a Chip or suchlike. Ihmor introduces in [Ihm06] a concept for interface synthesis focusing on the communication gap of different IP cores. This gap must be particularly considered if dynamic plugging of IP cores is of interest. How partial reconfiguration capabilities help to improve the behavior of such an interface synthesis approach is shown in [ID05].

When reconfigurable devices are used as low-volume alternatives to ASICs, they usually serve markets where fast processing is essential. Among these are domains where extremely few volumes of a circuit are required, such as space applications, which easily may be unique designs. Moreover, the domain of digital signal processing, image processing, network security, cryptography or bioinformatics applications among others [GG05]

may require fast processing devices to solve the tasks in a reasonable time. FPGAs have also been proven valuable for the implementation of neural networks [EH94].

In all these domains, where reconfigurable devices replace ASICs, run-time reconfiguration allows us to perform in-field upgrades. Newer versions of algorithms may be loaded into the device, which from there on may increase the performance, reduce the power consumption, etc. Particularly correction of bugs through run-time reconfiguration can be of high interest. Furthermore, such an in-field customization may also take place remotely if access to the configuration ports is provided.

Another very interesting application area for run-time reconfiguration can be found in domains formerly dominated by antifuse-based FPGAs. Such FPGAs, whose configuration is not volatile, are usually found in radiation critical environments such as space. To overcome the drawback of one-time reconfigurability of antifuse-based FPGAs, the group of Kebschull proposes to use permanently configuration refresh techniques [Keb06], which make SRAM-based FPGAs radiation tolerant. Partial reconfiguration is thus used to ensure determined behavior of the device by constantly reloading the configurations. Concerning the radiation tolerance, this method facilitates to resemble an equivalent quality as antifuse-based FPGA, however including the benefit of in-field adaptability.

To complete the idea of replacing dedicated circuits, we also have to mention that the direct moving from applications formerly executed in GPPs to FPGAs is possible. As reconfigurable devices are in several aspects superior to GPPs—energy consumption, etc.—reconfigurable fabrics might also replace GPPs by providing the same functionality. If the algorithm then is fixed, the reconfigurability is of no further concern. At the end of the day, the FPGA again simply takes over the job of an ASIC, combining all the benefits—reduced design costs at the front.

2.3.3 Adaptive Processing

We refer to adaptive processing if the adaptability—run-time reconfigurability—of reconfigurable devices is of primary concern. Here, reconfigurable devices are an efficient alternative for GPPs, as pure software-based solutions often would require unacceptable effort (power, etc.) to provide the same performance. Adaptability of the processing device thereby is required to react on changing parameters of the environments, etc.

Adaptive processing still is in its infancy. Besides a missing of sound technical solutions how reconfigurable devices could seamlessly replace GPPs, also the purely supported non-functional requirements of portability and programmability hamper the widespread use. Nevertheless, there are some approaches both in industry and academia, which provide solutions that show first promising results.

In general, we can divide adaptive processing in three categories following the ideas of [VS07]. In *algorithmic reconfiguration*, we hold the functionality, however adapt the performance, accuracy, power, or resource requirements. Reasons are a change of the environment. *Functional reconfiguration* means the execution of different functions in hardware, however on a shared resource. The immediate focus of functional reconfiguration within the idea of adaptive processing thus is time-sharing of resources. Also orthogonal to the previous two types of reconfiguration, *architectural reconfiguration*

means the reallocation of resources to computations. The need often arises when resources become unavailable due to an algorithmic or functional reconfiguration. Other reasons are a shut down, high priority interruption, fault tolerance, etc. In general, a clear distinction of the three types is difficult and arguable mixtures exist (see below).

In general, while ASICs mainly focus data stream oriented applications, reconfigurable devices are also open for applications that require control information. By virtue of run-time reconfiguration, the data processing can be changed; using dynamic and partial reconfiguration, this adaptation can also occur online.

Algorithmic Reconfiguration

If the circuit can be reconfigured during run-time there is no need to design the circuit such that it can operate on all possible inputs. Instead, we can reconfigure the circuit by the optimum configuration based on the run-time data and gain higher speed and performance. For example in the area of cryptography, where keys must be hard-coded improving performance, the algorithmic adaptability allows for changing the key over time. Another example for employment of adaptable hardware is image processing and feature extraction in computer vision [GCL02]. Here, adaptability may be used to modify filters or to specify on shapes to be extracted, see also [Bob07].

Thereby, we basically facilitate to offer specific circuits rather than generic circuits, which may result in a performance increase by using optimum circuits. For example, we can use constant multipliers instead of generic multipliers, which improve area consumption and response time [DKR03].

The idea of an adaptable pattern matching—searching for characters being part of a larger block of data or stream—where the search pattern is dynamically provided as dedicated circuit on the reconfigurable substrate, can be of benefit for multiple application domains like bio computing, genome searching, etc. [GG05].

We also find some works in the domain of adaptive controllers, for example investigating run-time exchange of mechatronic controllers [DBK03]. Here, reconfiguration is directly used to react on changing requirements of the environment.

Another example is the domain of distributed arithmetic—a bit level rearrangement of a multiply accumulate to hide the multiplications. Distributed arithmetic allows for reducing the size of hardware. Again, run-time reconfiguration helps to target the application even more exact [DB04, Dan04].

Finally, the presumably most recent application field for algorithmic reconfiguration is the domain of software defined radio [Mit95]. Mainly for military usage and therefore hardly available in the literature, run-time reconfigurability is used to adapt to dynamic requirements of communication.

Functional Reconfiguration

Functional reconfiguration denotes the idea of hardware virtualization in the sense of sharing the same hardware for different applications (functions). The reconfigurable substrate thereby changes its behavior. We thus rely on run-time reconfiguration.

A promising example was recently proposed in the automotive domain [BHH+07]. To reduce costs, functionality formerly provided each in single ECUs (electronic control

unit), is not only merged into one ECU—as already is a current trend—but loaded dynamically into the reconfigurable processing resource of this ECU.

In System on a Chip (SoC) devices, run-time reconfiguration is also used to dynamically swap applications into a reconfigurable substrate [GRP06]. Additionally, FPGAs themselves are equipped with resources such as CPUs to provide complete SoCs. The research platform then often is classified as Reconfigurable-System-on-a-Chip, where the reconfigurable substrate plays an integrated role.

In the domain of high performance computing, reconfigurable fabrics are used as hardware accelerators for compute intensive functions. Therefore, they become additional resources in nodes of computer clusters. For example on the Cray XD1, [SH07] show application specific acceleration by FPGAs, where the reconfigurable hardware is employed for particular operations in the finite fields for Reed/Salomon coding. Furthermore, the BEE project introduced in Sect. 2.1 offers a strong platform to target high-end reconfigurable computing. Besides the already mentioned RAMP project, BEE2 for example is also used as the heart of a high-bandwidth spectrum analyzer used by the Deep Space Network [Waw07].

In general, to increase programmability and portability in the domain of high performance reconfigurable computing, the non-profit consortium OpenFPGA has emerged. With varying activity, the organization aims to foster and accelerate the incorporation of reconfigurable computing technology in high-performance and enterprise applications.

Architectural Reconfiguration

Pure architectural reconfiguration would mean that we neither change the algorithmic characteristics nor the executing function itself currently residing on a reconfigurable substrate. According to [VS07], some examples are a shut down, high priority interrupt, fault tolerance, etc. However, architectural reconfiguration can also be seen as a requirement to fulfill algorithmic or functional reconfiguration. Examples of such a mixture are discussed below after a brief focus on pure architectural reconfiguration.

Fault tolerance is a well researched application field for architectural reconfiguration. In contrast to the above mentioned idea where a device is made radiation tolerant and thereby un-vulnerable to temporary failures, permanent failures are targeted. If some of the resources show faulty behavior, we can switch to pre-implemented alternative paths [dLKNH+04, KCR06], or apply partial reconfiguration capabilities [ESSA00].

In our opinion, architectural reconfiguration also covers the idea of hardware virtualization. In contrast to functional reconfiguration, the algorithms themselves are divided into partitions that must be executed completely to derive the result of the computation [Bob03]. We can thus implement larger systems than physical hardware available.

Mixture

By increasing the level of abstraction, as also done within this work, a clear diversification into the above presented reconfiguration types becomes difficult. Often, higher-level design methods rely on two or more of the types. Some examples are presented below.

Basically, we can reduce power consumption through reconfiguration. When power optimization already is considered on a high level of abstraction [Ret07], we can define

parts of an algorithm that can be gated, re-arranged, etc. under the constraint of optimizing power. By virtue of run-time reconfiguration, we can select different and more power-optimal path variations or eliminate those parts of the design that are not needed according to the current input. We can also move tasks onto other resources that are more energy-efficient.

A recent example of high-level exploration of partial run-time reconfiguration capabilities in the automotive domain is given in [CZMS07]. The authors describe how to use partial-run-time reconfigurable hardware to accelerate video processing in driver assistance systems. They combine functional with algorithmic reconfiguration by an integrated exchange of whole pixel-level functions to adapt the behavior of the overall system, reacting on environmental conditions.

Also completely new areas are influenced by run-time reconfiguration capabilities. For example, wearable computing [PEW+03], where the use of FPGAs is motivated by the better energy-efficiency of FPGAs compared to GPPs, providing enough flexibility to react on the environment. Another example is the domain of organic computing. Several approaches rely on the reconfigurability of FPGAs, e. g. [GP06]. Moreover, the idea of self-reconfigurability is part of the self-x paradigms of organic computing. Self-reconfiguration thereby is seen independent of functional, algorithmic, or architectural reconfiguration. Often, reconfiguration shall serve as technique for emergence—a controversial way how complex systems and patterns arise out of a multiplicity of relatively simple interactions.

2.4 Design Approaches for Reconfigurable Systems

As could be seen in the previous section, reconfigurability can be of benefit in various fields. However, to explore these benefits, appropriate design methods are required, which must go beyond classical methods as reconfigurable computing means the adaptation of the hardware resources over time, something not covered by classical methods. A simple mapping of existing design approaches for computing systems to reconfigurable systems therefore may struggle to yield optimal solutions.

In the literature, we find multiple proposals how to facilitate the design of reconfigurable systems. In this section, we discuss the most promising current approaches in the scene. As the research area is quiet active, we therefore briefly overview the most important ones, also by sorting and categorizing them. Worthwhile to mention also is the funding of the research area, as it emphasizes the interest in the topic. All the well-known organizations such as the DARPA in the US, the EU, or the DFG in Germany are funding or have funded research programs in the domain of reconfigurable systems.

2.4.1 Execution Environments/Architectures

We start by considering execution environments that facilitate run-time reconfiguration. Also being design approaches themselves, execution environments are first hand fundamentally important for experimental research or abstraction. The majority of the

approaches rely on FPGAs, which are execution environments themselves and shall be included in this listing. Concerning the numerous academical approaches, we focus on those that have emerged to become a comprehensive platform.

Commercial Platforms

Already selectively mentioned in the previous sections, the Xilinx flagship Virtex series are the most widespread commercial devices that offer capabilities for research on reconfigurable computing. All of them can be reconfigured partially, with the latest improving the flexibility immensely. While the devices up to the Virtex-II Pro series require always whole columns to be reconfigured, Virtex-4 and the most recent Virtex-5 can be reconfigured tile-wise. According to Xilinx, the reconfigurable parts of the FPGA are similar to the division of the clock network. For the low-cost Xilinx Spartan FPGA, which does not fully support partial reconfiguration, some research teams still have shown working designs [PHA+07].

The devices of the second big player in the FPGA world—Altera—lack the possibility to be reconfigured partially and are therefore little present in the realm of research on reconfigurable computing. The same holds for most of the other companies, except for Atmel. However, their FPSLIC, which offers partial reconfigurability, is very small, and therefore also is not widespread in the community.

All FPGAs are basically available as single dies, however, they are preferably used embedded into evaluation platforms of companies like Avnet, AlphaData, or Xilinx itself. On these platforms, peripherals are connected to the FPGA in order to exploit different functionalities. Moreover, within the supercomputing domain, Cray released in 2004/2005 the XD1 machine that basically hosts AMD Opteron 64-Bit CPUs. Additionally, the device incorporates Xilinx Virtex-II Pro FPGAs for application acceleration, composing another platform for exploiting run-time reconfiguration.

Furthermore, platforms that already couple reconfigurable fabric and main processing unit on one die are the Tensilica and Stretch approaches. While Tensilica's Xtensa VLIW processor architecture embodies a hardware accelerator that is fixed at design time without run-time reconfigurability, Stretch Inc. claims to embed programmable logic entirely inside a software-programmable processor architecture. Stretch's S5000 and S6000 software-configurable processors are optimized specifically for high-performance video and wireless signal processing.

Coarse Grain Devices

In the area of coarse grain devices, there are some rare commercial approaches, that also initiated research on design method. For example the PACT XPP devices exploit the principles of coarse grain reconfigurable systems towards commercial applications. The XPP-III architecture expands the horizon of DSP processing by programmability. Several applications have been shown and prove the efficiency of the approach [KGS+07].

Targeting their own abstract device as well as the NEC-DRP architecture, [OSF+07] discusses run-time reconfiguration on coarse grain devices. To maximize performance, they propose the concept of processor-like reconfiguration, which allows for new configuration every clock cycle. Besides requirements for the physical layout of such an

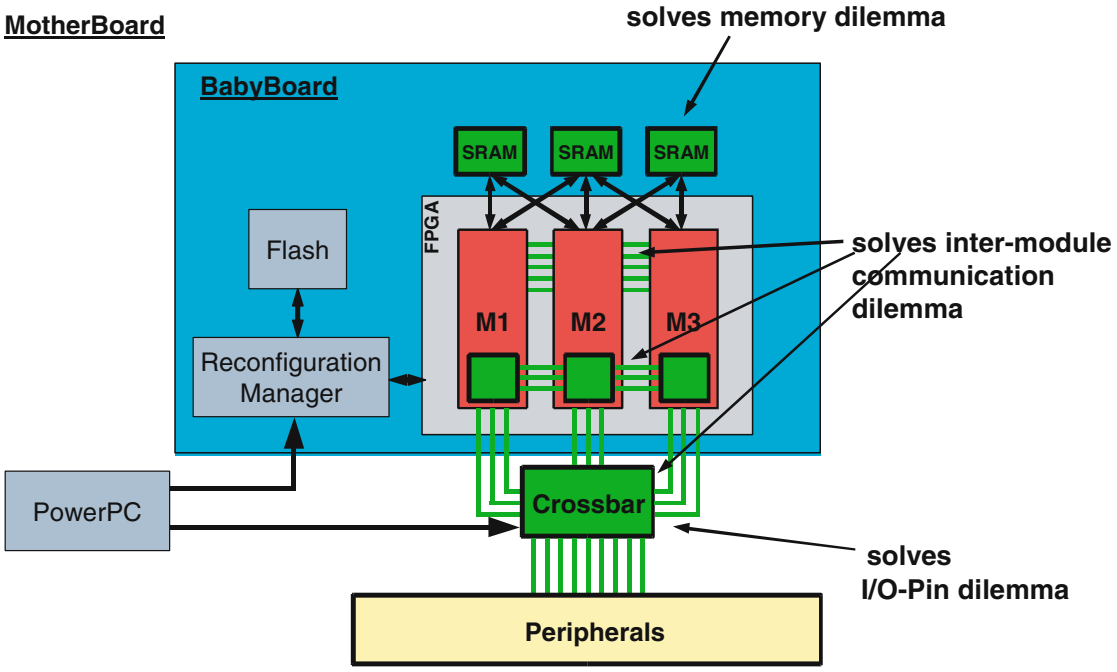

Figure 2.8: The Erlangen Slot Machine [MTAB07]

approach, they also discuss the trade-off between the number of costly contexts offered and maximum throughput possible.

Academic Approaches

Many academical works focus on the level of execution environments. Some of them use commercial FPGA platforms and map their specific experimental environments like a layer on top of them. Others provide complete environments, including the physical aspect by manufacturing their own PCB (printed circuit board). There are also some works that propose completely new FPGA technologies as a work-around of the drawbacks of the devices that are commercially available.

To raise the level of abstraction and facilitate dynamic task loading, [KPR02, WP04, FC05, UHGB04b, UHGB04a] among others have investigated the concept of a run-time execution environment implemented on top of the pure FPGA substrate. These approaches are called slot-based and either are arranged in a 1D or 2D style. Such environments dramatically help to open the field of reconfigurable computing to a broader audience, as low-level details can be abstracted.

An example of complete environments is the *Raptor* system designed at the Heinz Nixdorf Institute at the University of Paderborn. Raptor is a PCI card that can host up to six FPGA modules. Different combinations are possible and thereby lead to multiple sophisticated designs [KPR02]. Again, also the Berkeley Emulation Engine (BEE) project shall be mentioned. As introduced in Sect. 2.1, the current platform hosts five FPGAs and is used for example in the RAMP project [WOK+06], exploiting multiple levels of parallelism.

To address several drawback of commercial platforms, the codesign group at the University of Erlangen has designed the *Erlangen Slot Machine* [BMA+05, MTAB07]. As displayed in Fig. 2.8, the platform consists of an FPGA mounted on a baby board that can be solely used for the dynamic allocation of hardware tasks. Bitstreams are stored locally on this baby board and can be loaded on demand. For supervising purposes and to connect to peripherals, the baby board itself is mounted on a mother board. Particularly, the IOs of the main FPGA are connected via a memory mapped crossbar to the peripherals of the main board. We thus can establish connections very flexibly.

The capability to dynamically map IOs to peripherals on the Erlangen Slot Machine stems from the wish to overcome a major drawback of the Xilinx-II and previous series FPGAs in general and the modular design flow of the Xilinx application note 290 in particular. As partial reconfiguration on those devices requires to reload the configuration information of a complete column including the IOs of that column, the pinning of the board can easily constrain the applications. By making the link between peripheral and IO pin adaptable, this drawback can be overcome. Modern devices like the Xilinx Virtex-4 and Virtex-5 offer similar capabilities as the IOs are separated in the bitstreams as they now occupy columns on their own. Nevertheless, the Erlangen Slot Machine is a very convenient approach to exploit partial run-time reconfiguration.

In the realm of designing completely new architectures for run-time reconfiguration, we locate the Honeycomb architecture of the University of Karlsruhe [TB05, TB07]. It comprises of processing cells arranged in a hexagonal style resembling the layout of honeycombs. Furthermore, the Strategically Programmable System (SPS) architecture, which combines memory blocks and programmable blocks into a LUT-based fabric [MBKS01], or the Dynamically Programmable Gate Array [DeH96a] shall be mentioned. Moreover, proposals for alternative FPGA architectures exist. For example low power devices like LP_PGA [GR01], or the LEGO [CSR+99] device for high-speed designs.

To conclude, execution environments are an active area of research, most often including supporting design methods. Many sophisticated approaches exist, however few are present beyond their own group. Only strong commercial projects seem to have success, most likely as they offer comprehensive frameworks on top of the pure execution environments. However, expert knowledge not only limited to execution environments can be found all-round, which thus may be used to exploit so-far unused capabilities of reconfigurable devices. Some reasonable concepts can be found below.

2.4.2 Placement/Scheduling Methods

If tasks shall be executed on reconfigurable devices, there arises the question of area allocation, particularly if tasks are to be loaded dynamically onto the substrate. Often supported by a reasonable run-time execution environment, still we have to map, dispatch and schedule algorithms or applications onto (run-time) reconfigurable fabrics. Methods either take place offline or online, as separated below.

Offline Placement/Scheduling

In [FKT01], Fekete, Köhler, and Teich propose formalisms and algorithms towards module placement. They solve OPPs (Orthogonal Packing Problems) and give good insight by presenting this approach. They project boxes onto the coordinate axes, which define interval graphs. Furthermore, we can find a strong mathematical background in the paper. However, they neglect communication between the modules.

An influencing work on temporal partitioning and temporal placement again in the offline scenario was conducted by Bobda [Bob03]. By virtue of spectral placement, he introduced a new approach to derive so-called generations that are dynamically loaded on FPGAs. Moreover, he considered clustering for 1D slot-based task execution.

Online Mapping/Scheduling

For online scheduling, activation times and frequencies of the different tasks, which are only known at runtime, must be considered. [WP03] therefore introduce a technique that uses a number of queues and the two functions f_{SPLIT} and f_{SELECT}. Additionally, they consider preemption of tasks executing on FPGAs [WP04].

[SWPT03] proposes an on-the-fly partitioner towards better/less fragmentation and manages the free spaces therefore in trees. The authors also discuss heuristics for online scheduling of real-time tasks to partially reconfigurable devices [SWP03]. The two heuristics are called the horizon and the stuffing technique. Furthermore, they consider the limitations of a practical implementation. These considerations include the device homogeneity, task communication and timing, and the partial reconfigurability.

[ABI06] present another scheduling approach referring to the *strip packing* algorithm. They target column-wise reconfigurable FPGAs like the Xilinx Virtex-II series. The usage of the strip packing algorithm allows them to also consider tasks with precedence constraints and release times.

Another interest manifests in the empty space search. For example [HV04] search for maximal empty rectangles by using a staircase idea. Therefore, they reduce the problem to find a maximal staircase area. A matrix stores the status of the FPGA.

Moreover, Diessel and ElGindy show a scheduling algorithm [DH98], which comprises of two steps. In the first step, they identify a rearrangement of the tasks executing on the FPGA. This step's job is to free sufficient space for the waiting task. The second step then schedules the movements of tasks to minimize the delays to executing tasks. In order to identify the feasible rearrangements, they investigate local repacking and ordered compaction. The first part of the second step comprises arbitrary rearrangements and ordered compaction. They use ordered compaction only for the second part (moving tasks on-chip). In another work, Diessel et al. introduce also concepts for dynamic scheduling of tasks on partially reconfigurable FPGAs [DEM+00]. Therefore they propose the rearranging of a subset of the tasks executing on the FPGA.

Scheduling Tasks onto Coarse Grain Devices

In a paper concerning network topology exploration of mesh-based coarse-grain reconfigurable architectures [BGD+04], the authors explore the effects of varying the network topologies, the topology traversal strategies, and the delay models for the interconnects.

The mapping they use is trimmed to interconnection awareness. They construct the total run time by a combination of execution time and routing delay.

For exact and heuristic solutions, [BLPG01] gives a very brief overview of efficient IP based mapping techniques for coarse-grained dynamically reconfigurable array architectures. They consider genetic algorithms, hadlock's algorithm, etc.

Most of the works on placement and scheduling methods presented above are driven by the fascination to arrange tasks like 3D boxes in space and time. Multiple sophisticated heuristics have been shown. However, few works consider the reconfiguration time and the communication requirements of such a scenario. Moreover, heterogeneity hampers the idea of mapping the packing problems to reconfigurable fabrics. Nevertheless, comprehensive methods can benefit from these results and include appropriate methods.

2.4.3 Comprehensive Design Systems/Design Methods

In the literature, there exist several research proposals that target on comprehensive frameworks to design reconfigurable systems. Below, we list some important examples, roughly sorting them into increasing abstraction level.

There are some approaches that discuss compilation from classical C-style languages to reconfigurable devices. Based on the SUIF infrastructure [HAA+96], several works have investigated how to compile to reconfigurable systems [BDD+99] or how to create coarse grain parallelism [KKS04]. These approaches heavily rely on the data/control flow based intermediate graph of SUIF. Also the DEFACTO system [Bon02] combines parallelizing compiler technology with synthesis to automate the effective mapping of applications to reconfigurable computing platforms.

Starting from the Java bytecode level, [JN03] discuss a Java-based compiler for compiling software programs into reconfigurable hardware. The compiler can exploit multiple control flows and high instruction-level parallelism (ILP) degrees (beyond the basic-block level), and diminish the memory bottleneck. Other approaches are starting from Matlab with the MATCH (MATlab Compiler for distributed Heterogeneous computing systems) compiler [BSC+00] or use specific algorithms to profiles graphs towards reconfigurable computing [KKMB02].

Some approaches also rely on specific APIs that are used to describe temporal and spatial execution. For example, Koch describes in [Koc02] a concept how to generate parameterized hardware-modules based on an API and a primitives catalogue. His API allows for evaluation and creation of hardware objects targeting configurable computing machines. To also enhance the SystemC language for modeling reconfigurable computing, the OSSS+R library was developed [SON06]. It extends OSSS [GTF+02], which is a SystemC subset that can be synthesized, by specific primitives to model reconfiguration, including simulation at a high level of abstraction.

Without referring to HLL input languages, some works directly consider input graph mapping to reconfigurable systems. For example in [KV99], the authors consider integrated block-processing and design-space exploration in temporal partitioning for reconfigurable devices. They automatically partition behavioral specifications. As the recon-

figuration overhead is an issue, they propose block-processing. Moreover, in [KVGO99] an automated temporal partitioning and loop transformation approach for developing dynamically reconfigurable designs starting from behavioral level specifications is discussed.

Also works that propose complete frameworks are presented in the literature. Eisenring and Platzner [EP02] defined such a framework for run-time reconfigurable systems. The aim of their work is to provide a methodology and a design representation which allows them to plug in different design and implementation tools. They base their idea on two graphs, one for the architecture and one for the problem.

In [OGS+98], we find the integrated design system SPARCS (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems), which comprises an automatical partitioning and scheduling. However, run-time reconfiguration is not in the primary focus of SPARCS.

A comprehensive approach is SCORE [CCH+00], which allows for modeling stream-oriented computations, including the system architecture and several execution patterns. However, SCORE is a relatively heavyweight framework, in particular as it uses a proprietary modeling technique consisting of FSM (final state machine) and TM (Turing machine) nodes.

There also exist proposals for frameworks on the generation of (partial) bitstreams for run-time reconfiguration. Among others, [CCEBM04] present such a framework.

Recently, we could watch the emergence of several extensive research projects under support of the EU, which focus on integration of several design constraints. For example, the Multi-purpOse dynamically Reconfigurable Platform for intensive HEterogeneous processing (MORPHEUS project) is an integrated project which addresses solutions for embedded computing based on dynamically reconfigurable platforms and tools [TKB+07]. The AETHER project aims to tackle the issues related to the performance and technological scalability, increased complexity and programmability of future embedded computing architectures by introducing self-adaptive technologies in computing resources [PHB+07]. Similarly, the ANDRES Project targets at analysis and design of run-time reconfigurable, also including heterogeneous systems [HOH+07].

2.4.4 Miscellaneous Concepts

There are also approaches that bear reconfigurability in their structure by nature or can be easily extended to make powerful approaches for reconfigurable computing. For example the MACT architecture, invented at the University of Paderborn, Germany and already patented [RLZB04], can be extended by routers that allow for run-time exchange of parts of the data flow graph [DRW06].

2.5 Lesson Learned

Reconfigurable computing is a promising, however also quite challenging field. As could be seen throughout the whole chapter, there are many pros and cons. The core benefit

is the capability to process in space and time—we can resemble the speed of hardware as we process in parallel and still be flexible based on the reconfigurability of the devices. Reconfiguration however incurs costs, such as the reconfiguration time, the increased design complexity because of the interdependency of the two domains space and time, the need for suitable communication, etc.

For a comprehensive fruitful and beneficial exploitation of reconfigurable fabrics, we thus have to act sophisticatedly and overarching. Therefore, we have learned details about the technical characteristics, the programmability, etc., which we consider as evident if a comprehensive exploitation of reconfigurable devices shall take place. Abstraction, which is targeted by this work, can thus built a solid foundation.

Concerning the overcoming of the overhead of the reconfiguration, there always must be a balance between the possibility for fast reconfiguration and the costs for the technique to achieve these improvements. Obviously, the purpose of reconfigurable devices should not be to conduct high-performance *reconfiguration*, but to facilitate high-performance and application-oriented *processing*. Only if the reconfigurability improves the performance of the overall system, the additional costs for the reconfiguration—time overhead and infrastructure—are acceptable.

This train of thoughts must be considered at all abstraction levels. For example, an unsystematic placement of tasks onto a reconfigurable device can result in fragmentation of the substrate, and prevent further tasks to be executed as no connected area to host the tasks is available. A relocation may solve the problem, however it may also degrade the overall performance due to further costs (readback). Another example and big challenge in this context is to provide an efficient intermodule communication. Again a trade-off between heavyweight networks on a chip and lightweight but dedicated wires has to be found.

Moreover, we particulary require appropriate design methods, if reconfigurable devices shall be seen as adaptable devices. Here, an ASIC style design flow quickly becomes too cost intensive. For example, an estimation of the performance of an application that shall be executed on a reconfigurable fabric should be possible on a high level of abstraction, without the need to completely synthesize the design. Very challenging, the quality of this high level evaluation should be also very good.

The design approaches presented in the previous section discuss several sophisticated concepts and give reasonable hints for a beneficial exploitation. However, they often require improvements to yield good results, for example, many of them neglect communication. They thereby show that it is challenging to program for reconfigurable systems. On one hand, the overall complexity of computing in space and time has to be taken into account. On the other hand, the generation of circuits is complex and difficult in many aspects. Even once done (having a good design), the synthesis down to the final bitstream generation takes a long time, which makes it hardly comparable to the quickly responding compilers of software design.

Besides these design-oriented challenges, a general aspect for beneficial exploiting runtime reconfiguration still is the long reconfiguration time resulting in the reconfiguration latency. This latency must be considered for the granularity of the adaptability. Thereby, the latency is significantly reduced when partial reconfiguration can be applied, as others

parts can continue operation during a reconfiguration. Furthermore, as reconfigured area is usually proportional to reconfiguration time needed, the reconfiguration of smaller parts will result in an earlier availability of configurations.

Furthermore, we experience a lack of suitable tools that soundly support dynamic run-time reconfiguration. Partial reconfiguration—in particular the generation of partial bitstreams—still is hardly supported on the software side of the vendor products. Nevertheless, promising concepts and methods exist that are worthwhile to be investigated further. For example in the context of reconfiguration time hiding, we can summarize that configuration overlapping and caching yield good results.

In general, it is also difficult to agree on one common language (terminology) in the domain of reconfigurable computing. As already the discussion on *reconfigurable vs. programmable* showed, the definitions of the field often are vague. Moreover, the different approaches on how to design reconfigurable systems—high level languages, graph-based approaches, hardware description languages, etc.—make an agreement on one single model of computation for reconfigurable computing very challenging. Therefore, it maybe is difficult to conclude whether we already know what reconfigurable computing is and what its true challenges are.

Nevertheless, FPGAs as the precursor of reconfigurable computing have started to become mainstream. To eventually exhaust their complete potential (of which we are aware so far), comprehensive design methods are required.

In the subsequent chapters, we will built on the results of the literature and derive methods that target on applications not covered up to now. We also show where our methods extend the design approaches presented so far. Thereby, our methods gradually raise the level of abstraction concerning the design of reconfigurable systems.

2.6 Summary

In this introductory chapter, we took a glance at reconfigurable computing. Therefore, we have considered the history of this field, before detailing the technical aspects of reconfigurable devices in general. As this thesis mainly focuses on partially run-time reconfigurable FPGAs, we have spent some more lines on the capabilities and drawbacks of the fine granular FPGAs, including programming, bitstream generation, and coupling. In general, we have detailed these parts, which are important for the further work presented in this thesis. Concerning the fields of application, we have listed several domains of configurable systems including already investigated or future extensions for run-time reconfigurability. Finally, we have given a brief overview of important design approaches of the literature.

3 Two-Slot Framework

In this chapter, we introduce a two slot framework as our primary approach for exploiting partial run-time reconfiguration. The core part of the framework is a platform for reconfigurable computing that comprises of exactly two regions in which user tasks can be executed. The two regions are closely connected and compose a unity. Tasks or partitions of a task are loaded alternatingly into these regions allowing for hardware virtualization and configuration time hiding. The two slot architecture therefore most likely is the most direct approach to exploit partial run-time reconfiguration capabilities. Despite its simplicity, the architecture explores the major obstacles of partial run-time reconfiguration, e. g., static communication, application mapping, or partial bit-stream generation. We thus consider the architecture as ideal to introduce reconfigurable computing conceptually as well as from an implementation point of view.

3.1 Introduction

The previous chapter on reconfigurable systems already has shown that we have to take care of multiple often low-level issues such as the inter-module communication or the task allocation for a beneficial exploitation of reconfigurable systems. In particular, the execution of tasks on the substrate including both the temporal and spatial dimension is challenging. Only a comprehensive consideration of both dimensions will result in valuable benefits and therewith justify the usage of a reconfigurable substrate.

One of the main challenges undoubtedly is the reconfiguration overhead—the duration of the reconfiguration itself. As the configurations are usually shifted byte- or even bit-wise into FPGAs, and multiple thousand bits are needed to configure the logic, even relatively fast reconfiguration ports like the SelectMAP port with up to 66 MHz and up to 32 Bit width will require milliseconds to reconfigure the several million SRAM cells of modern devices. Such a delay cannot be accepted by any application, in particular if the whole reconfigurable substrate comes to a stop during the reconfiguration process.

Here partial reconfiguration comes in and improves the issue on two sides. Firstly, partially reconfigurable systems allow us to reconfigure only parts of an FPGA, while the other parts remain untouched and can keep on processing. We thus can interweave reconfiguration and execution, and hide the reconfiguration time. Secondly, a smaller area to be reconfigured directly results in a linearly shorter reconfiguration time.

However, by employing partial reconfiguration, the design complexity increases. When placing tasks to be executed on the FPGA at arbitrary positions, the FPGA will become fragmented, especially if multiple tasks share the same reconfigurable fabric. Thus, *fragmentation* must either be maintained by a reconfiguration controller or prevented

by fixed regions—so-called slots. We apply the latter in this chapter, which also eases another challenge of partial reconfiguration, the intermodule *communication*:

Tasks require access to input variables, might have to store intermediate results, and have to write back the final result; even communication between multiple tasks may occur. Moreover, for arbitrary task placement, dynamic routing would be necessary. If the location of the tasks is reduced to a fixed set of regions (slots), we can implement a dedicated and static communication infrastructure or a bus topology for communication. In such a case, we can also provide access to additional communication resources (e. g. fast rocket IOs) by routing them to some specific slots.

Another important aspect of partial reconfiguration is the configuration scheduling. As the substrate is time-shared among several tasks, a control unit—also termed *reconfiguration manager*—should supervise and trigger the loading of new (partial) bitstreams. Foremost, such a manager must consider the difference in duration of the reconfiguration times and the times spent in execution. Intermediate results thereby often must be hold back until the reconfiguration has finished and processing can continue.

Two Slot Architecture

To consider all the constraints mentioned above, a lightweight approach like the two slot framework is a perfect start. Here, two deeply connected slots constrain the substrate and offer a direct access to partial reconfigurability. Despite only two regions, the core characteristics of partial reconfiguration are present in such a framework. Thereby, the architecture of the framework is of major concern as it takes care of the low-level reconfiguration issues and enables the reconfiguration time hiding. By providing two fixed-size slots, the simple yet effective two slot run-time environment brings reconfiguration to its basics. In addition to an in-depth discussion on the challenges during the design, we particulary show how we can integrate alternating execution of the slots into the overall framework. The two-slot framework thereby facilitates multiple application scenarios.

Intended Application: Hardware Virtualization

Basically, the two slot framework focuses on hardware virtualization. Hardware virtualization has been discussed in several works (e. g. [Bob03]) and can be characterized by reconfigurable fabrics being used to make up a larger application than would fit on the device. This feature is of particular interest in the area of embedded systems or for the prototyping of very large VLSI circuits where often only a limited amount of resources is available. These resources are then time-shared by the same application.

To gain the benefits of such a hardware reuse, the applications themselves also have to serve the execution scheme and the architectural constraints like maximum area and communication of the reconfigurable regions. Therefore, we can either restrict the input to fittingly partitioned tasks or may have to partition the input algorithms appropriately.

Despite its simplicity, the two slot approach can be extended to open for larger application fields. Notably is the possibility to focus on low power scenarios. Therefore, we investigate the usage of the multiple clock domains of modern FPGAs as means to adapt the execution times. Another natural extension of the two slot environment is to include the architecture as an IP (intellectual property) core into a larger system.

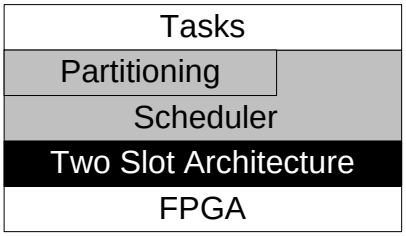

Figure 3.1: Layered approach for the two-slot framework

3.1.1 Layered Approach

Our two slot framework solves the demand for abstraction to handle the complexity of hardware resources in reconfigurable devices in a very simple way. Nevertheless, we raise the level of abstraction and thereby slightly open the field of reconfigurable fabrics—particularly FPGAs—to the world of computer scientists.

Figure 3.1 shows the layered approach for the two slot framework. On top of the FPGA we employ the two slot architecture layer. Tasks for execution on the FPGA must be constrained in order to be executed by this layer. We therefore might have to partition the tasks, which we add as a dedicated layer. Furthermore, a scheduling or dispatching layer must exist that takes care of the reconfiguration process. Such a service can be used by an operating system, among others.

3.1.2 Organization of the Chapter

The rest of this chapter is organized as follows. First, we abstract the problem and define a solution strategy. Thereby, we introduce the concept of reconfiguration phase RT and execution phase EX. This fundamental concept accompanies us in the remainder of the thesis. Therefore, we spent some more lines on the concept.

Then, we follow the layered approach. We describe the architecture and its implementation, including the intermodule communication between the two slots. We solve the problem of simple yet flexible data flow between two slots. Thereafter, we introduce our two partitioning methodologies and evaluate them. Then, we consider the scheduling for the two slot framework. In the experiment section, we show an application example and evaluate its behavior. We also discuss extensions of the two slot framework. We show how to maintain the overall response time, while reducing power consumption, and discuss the two slot architecture as a powerful IP core for use in (embedded) system design. Very important is the lesson learned of this chapter as it drives the further work of this thesis. Before we summarize, we evaluate related work.

3.2 Concept

The concept of the two slot environment aims at simplicity. Nevertheless, it is embedded in a comprehensive abstraction of partial run-time reconfiguration, which is basically

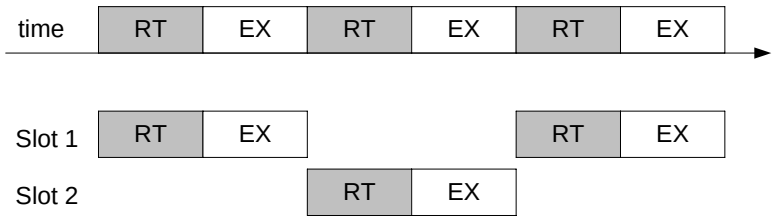

Figure 3.2: Reconfiguration (RT) and execution (EX) phase, simple example.

motivated by resource reuse and reduction of costs. The hereby necessary underlying abstracting model of execution on partially reconfigurable FPGAs is presented first.

3.2.1 Problem Abstraction

Our model of execution can be described abstractly as displayed in Fig. 3.2. The core idea is to make the two relevant phases of tasks executing on reconfigurable systems explicit: reconfiguration and execution phase. The reconfiguration phase (RT) represents the configuration of the hardware itself. It has to occur before the second phase, which is the execution phase (EX). In the lower part of Fig. 3.2, we display the available slots vertically and their occupation over time horizontally. The slots are active alternatingly. RT means that a slot is in reconfiguration, while EX denotes the execution of a task.

As FPGAs facilitate to execute more than one task on their substrate at the same time, multiple EX phases may reside on the FPGA in parallel. Thanks to partial reconfiguration, EX and RT phases of different tasks can also be conducted in parallel. We thus can introduce configuration pre-fetching within the FPGA, see Fig. 3.3. However, the commonly given constraint of only one reconfiguration port leads to the exclusiveness of RT phases. In Fig. 3.3, we also display the general motivation for partial reconfiguration capabilities of such a system. If we interweave EX and RT phases of different tasks, we can hide the reconfiguration time. Partial reconfiguration thus results in an improved overall response time of the task set.

It is worth to mention that both phases are inescapable parts for the processing in space and time. Yet, during the RT phase no immediate benefit for the processing is gained. Moreover, as up-to-date FPGAs comprise one reconfiguration port only, we cannot reconfigure multiple areas at the same instance of time, a fact that must be considered in all approaches of partial run-time reconfiguration.

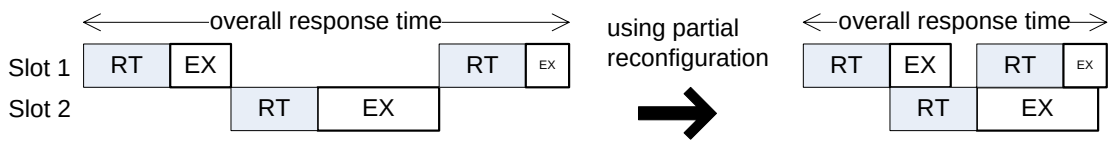

Figure 3.3: Scheduling of reconfiguration (RT) and execution phase (EX) showing the benefit of partial reconfiguration.

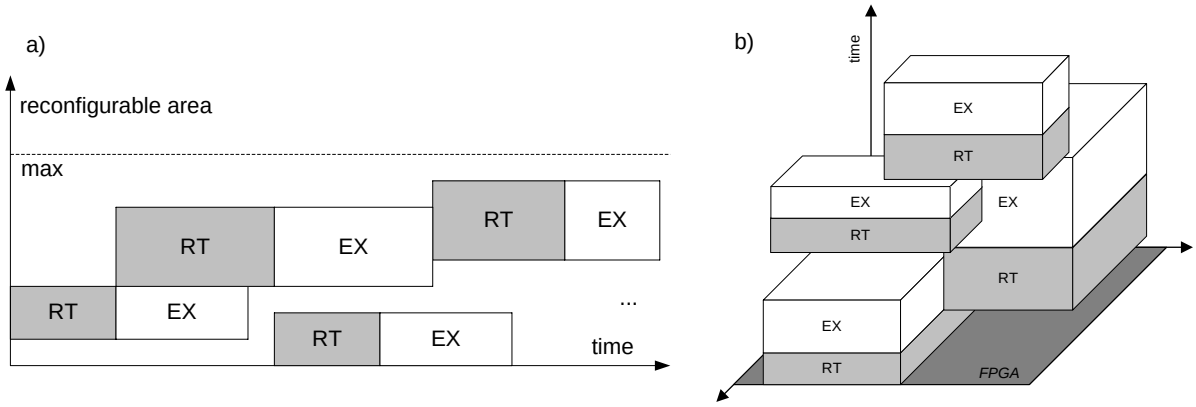

Figure 3.4: Scheduling example of *EX* and *RT* phases on a one-dimensional (a) and two-dimensional (b) reconfigurable fabric.

Excursion

Basically, the execution model of *RT* and *EX* phase facilitates different combinations of these two phases, however constrained by *RT* and *EX* of one single task always forming a unity. If we consider the reconfigurable substrate as a freely programmable region, a schedule as depicted in Fig. 3.4 a) could arise. There, four pairs of *RT* and *EX* phase are dispatched to the area (y-axis) over time (x-axis). As there exist no constraints to the area, the tasks are distributed uncontrolled on the fabric. We thus waste space and suffer fragmentation. Finding an optimal schedule becomes challenging.

Furthermore, if the area of the reconfigurable fabric can also be configured in a two-dimensional style as given by the Xilinx Virtex-4 and Virtex-5 FPGAs, the complexity increases once more. Such an example is displayed in Fig. 3.4 b). Here, tasks compete for connected area to serve their spatial requirement. Poorly placed tasks may force successors to wait until a region large enough is available, despite the accumulated free space already would be enough.

The optimal schedule heavily depends on the area under reconfiguration, the device which hosts the tasks, the task and its execution time itself, the dependencies between tasks, and so on. As we can map the problem on a multiprocessor scheduling problem with task placement constraints, possibly also targeting heterogeneous processors, including strong dependencies between the *EX* and *RT* phase of tasks, etc., deriving the solution of the optimal schedule becomes NP-hard [GJ79, BBD05]. Additional requirements like resource conflicts during task execution (e. g. sharing of the same bus) have to be considered also. Nevertheless, the phase model is open for all variations.

We formalize the general problem as follows:

Problem 1 *For a problem P given as a set of tasks Γ to be executed on a reconfigurable substrate S, we want to derive a placement β and a schedule γ with respect to one or multiple optimizing goals.*

This multi-objective optimization problem desires special care as several optimization criteria are possible: minimization of the overall execution time, resource usage, power consumption, etc. Particularly, if a top-level designer exploits all degrees of freedom, the design process quickly becomes tedious and may require numerous iterations. It is therefore advisable to constrain the design space and guide the designer through the process, for example, by reducing the solutions to a Pareto-front from which the designer can choose. By holding some parameters fixed, e. g., relying on good results as is done in platform-based design, the design process becomes manageable and more abstract.

An Example

Considering the problem of optimizing the scheduling of the *RT* and *EX* phases as a classical pipeline problem, we would argue for both phases to be as homogeneous as possible in order to avoid idling times and maximize the performance. Such an optimization problem can be found in the design of pipelines for the classical von Neumann machine, where the stages should be packed as densely as possible to yield the maximum performance. However, this reduction to a known problem must not be optimal if we consider execution in space and time.

In reconfigurable systems, *RT* phases, which are part of such a pipeline, should occur as rarely as possible. Moreover, if they occur, they should be as short as possible. Simply speaking, the *RT* phase is of no direct benefit for the processing. It only can improve the overall behavior of our system—and thereby secondly being beneficial for the processing—if we can accept the additional costs. Therefore, a balanced pipeline may be unwished in the case of reconfigurable computing.

The methods how to derive a good scheduling thus must be selected carefully. Moreover, the reconfiguration overhead very often is high, which means that the *RT* phases are very long and often dominate the *EX* phases. In addition, relying on more than two slots for the execution of tasks increases the complexity of e. g. the inter-module communication between the slots, etc. Nevertheless, there exist good reasons for variations. We give answers in every chapter of this thesis. Here, we focus on the maybe simplest shape of this pipeline, where we have two slots that are alternatingly in reconfiguration and execution phase, offering to hide the reconfiguration overhead.

3.2.2 Problem Solution Strategy

The problem to solve thus is to design a two-slot architecture and integrate it in an overall framework for comprehensive design. A first draft of the architecture is depicted in Fig. 3.5. During the processing in one area, another region can be reconfigured, using the concept of configuration pre-fetching. Besides the core capability to hide reconfiguration time, we can sequentially host parts of an algorithm otherwise too large for complete implementation on the device available. Assuming the termination of the reconfiguration in time, seamless processing is possible, while intermediate results are buffered in the control area. The temporal existence of parts of the application on the device is transparent to the user.

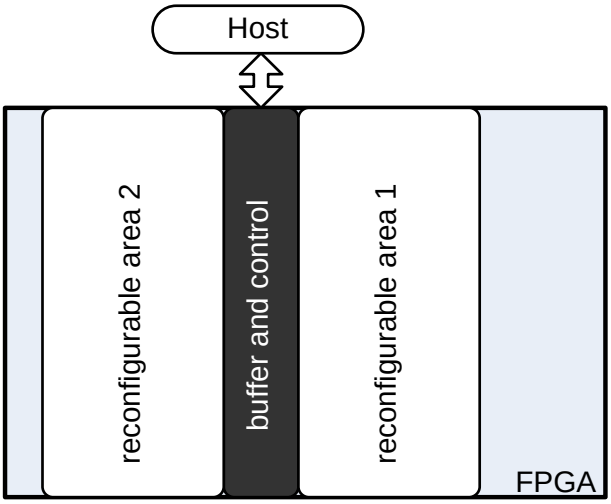

Figure 3.5: Draft of the two slot architecture.

Several other constraints are met by this architecture. For example it is in line with modern FPGAs, which comprise one reconfiguration port only. Moreover, static inter-module communication can be achieved on basis of having fixed reconfigurable regions.

For the sequential execution of partitions of tasks on the two slot architecture, we still need an appropriate scheduling method as part of the overall framework. Such a method performs best, if a sound input in the form of suitably partitioned applications can be provided. Therefore, we require partitioning methods, which derive partitions comprising similar area requirements. The generation of such partitions may also focus on equal execution times, which often conflicts with precedence constraints and the aim to optimize communication.

To summarize the framework, tasks—once loaded—can execute with the speed of hardware. The reconfigurability adds the adaptability to the system or enables the execution of algorithms too large for the device, using the concept of virtual hardware. The reconfiguration is an integrated part of the whole execution, however, it should not become the bottleneck of the system. The framework must consider that during reconfiguration the area under reconfiguration is blocked, as well as time is consumed.

3.3 Run-Time Architecture

We discuss the implementation of the two-slot architecture below, including the challenges that had to be overcome. Basically, similar approaches of reserving some chip area for the controlling overhead and dedicating some other to user logic can be found in the literature [BMA+05, KPR02, WP04, FC05]. These approaches—in general termed as slot-based—served as valuable inspiration. As few of the examples of the literature comprise two slots only, we still had to design multiple issues from scratch. We start by discussing the behavioral, structural and physical constraints of our approach.

3.3.1 Fundamental Design Constraints

The behavior, which was already defined above, briefly summarized should provide the following: Execution of partitions of one or more applications on two run-time reconfigurable regions that are occupied alternatingly and thus provide reconfiguration time hiding. The regions can be reconfigured independently, but only one after the other, as the reconfiguration is mutually exclusive. The data processing is done inside these reconfigurable slots. The alternating reconfiguration facilitates complex functions to be calculated by means of successive computations, while the intermediate results are kept.

The intermediate results thus must be stored within the architecture. Moreover, a specific connection point for the external communication must be provided. As these communication requirements remain fixed during the execution of applications, we implement them in a static module, see below for more details. The static module may also serve as control unit of the run-time reconfiguration. Therefore, it supervises the modules and communicates with an external host to initialize the reconfiguration. Using a self-reconfiguration port like Xilinx ICAP would make the architecture a stand-alone system without requiring a host processor.

The structure of the architecture thus becomes obvious and comprises of the two reconfigurable and one static region, as well as the communication structure in between them. The reconfigurable modules are termed slot A and slot B. There is no direct connection between the two reconfigurable modules as the information exchange between them is always routed through the static part. Figure 3.5 depicts the three elements of the architecture, also including the communication to an external system.

The physical arrangement of the modules depends on the constraints of the reconfigurable substrate used. Multiple alternatives are possible. When implementing all three modules on one fabric, however, only a few layouts become reasonable. As we have two slots only, the arrangement always is in 1D style, meaning that the two run-time reconfigurable slots can be lined up in one direction. However, only devices which allow a two dimensional reconfiguration—such as the Xilinx Virtex-4 and 5 series—can host the slots on top of each other. Other devices require the slots to be in their own vertical column each—facilitating a horizontal arrangement. To keep the signal delay small, the slots should be put close to each other. As the static area provides the intermodule communication between the sections, we should reserve a static area next to or between the slots. This spacial arrangement of the modules allows the controller to directly communicate to both of the slots. Any data flow that is to be passed from one slot to the other inevitably crosses the controller.

The actual physical arrangement of the controlling section again depends on the characteristics and capabilities of the devices used. On Virtex 4 and 5 devices, it may be located above, below or in between the reconfigurable slots. In contrast, older devices constrain the location due to their architectural characteristics despite of modern design flows as the Xilinx Early Access design flow described in Sect. 2.2.5. On such devices, the actual reconfiguration still takes place in a column-wise fashion—even if only a part of a column is declared to be partially reconfigurable, the whole column will be reconfigured. In such columns, the identical configuration information is written to the SRAM cells

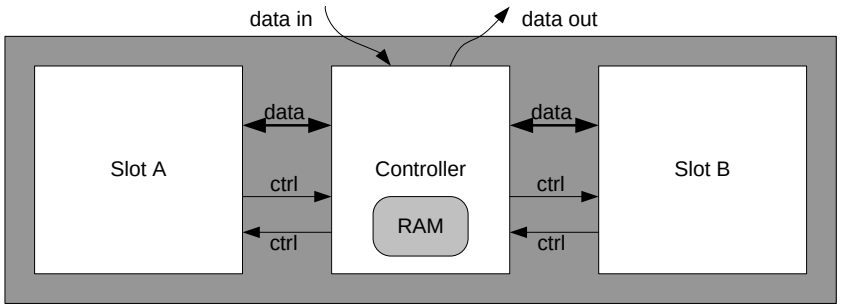

Figure 3.6: Floorplan of the architecture with the memory-based communication. Each slot is connected to the controlling unit by a data bus and control signals in both directions. The controller comprises RAM for holding the intermediate results.

which are beyond the reconfigurable region. As we may suffer unwished re-initialization of constant values particularly concerning the flip-flops of the configurable logic blocks, sensible logic should be avoided in columns that undergo run-time reconfiguration (in fact it is unofficially claimed to be suppressed by the Xilinx design tools).

3.3.2 Intermodule Communication

A proper design of the intermodule communication is a necessary part of a sound implementation of the architecture. We have implemented several alternatives, also to be able to evaluate benefits and drawbacks. Basically, the communication facilitates the data exchange between the two reconfigurable regions. In general, to establish a proper connection between a reconfigurable module and a static unit, dedicated signals have to cross the borderline. The points of crossings thereby are assumed to preserve between the reconfigurations, as given by using Xilinx busmacros.

The first connection strategy presented below is a lightweight buffer-based approach, followed by a proprietary bus-based solution that focus on providing simple yet effective communication protocols. After that, we also consider the example of using standard bus connections and sketch an approach based on the message passing paradigm.

Buffer-based Solution

As the major goal of our first approach, we focus on a lightweight approach reducing communication and area overhead. Therefore, the intermediate area hosts buffers that are directly connected to the static inter-module communication resources. Each buffer is accessed by one of these static communication resources of one slot, holds one single bit only, and transmits this bit to the next slot. The benefit of this solution is a very small area footprint. In the best case, we only have to provide the same amount of buffers as the communication bit width between reconfigurable slot and control unit. The same buffers then are shared for the data flow in both directions—from slot A to slot B via the control unit and vice versa. However, as we so far cannot overcome the constraint of single-directed busmacros, we currently have to set up twice the amount.

The drawback of this lightweight approach obviously is the necessity to either establish the worst case amount of busmacros (all applications must be known at design time of the architecture), or to limit the data flow between subsequent partitions of an application to a previously set maximum number of busmacros. Both solutions constrain the maximum bandwidth and thus may hamper the flexibility of the overall two slot framework.

Another approach would be to adapt the wiring during run-time. However, it is very challenging to dynamically establish communication lines. In fact, using the standard tools, the run-time reconfiguration of communication structures between reconfigurable and static parts so far is not possible, as it causes problems to the signal integrity (possible open signals during reconfiguration). Furthermore, a maximum amount of busmacros may be given by the technical characteristics of the FPGA.

Proprietary Shared Memory Solution

In this solution, a memory can store large intermediate results, while the access to the memory is resource optimized by a specific bit-width, see Fig. 3.6. The solution thus resembles the idea of shared memory, as both regions communicate via a single memory space. Access protocols therefore have been designed that harmonize with alternatively execution of tasks in slots, having the other slot reconfigured [War06], [WD06].

For the protocols, we again focused on a lightweight approach reducing communication and area overhead. To keep the communication between a slot and the controlling unit flexible, we offer two data transfer modes: sequential mode and random access mode. We implemented the communication protocols by means of finite state machines.

The underlying protocols define the way the data is transferred over time, specifically which component performs a write or read operation on the communication bus. The user has the responsibility of building the functional module instances placed inside the slots, and therefore is responsible for supporting these protocols.

In the *sequential transfer mode* protocol, the transmitting of the data comprises three parts: load all data to the slot, process the data inside a slot, and store the result to the controller again. We do not transfer addresses, as the data is transmitted word-wise sequentially over the bus. The data wires or the bus can thus be used exclusively for the data. Another advantage of this approach is that it grants bit-parallel access to the entire data set after it has been transferred from the control unit into the slot. If an algorithm needs the whole data at once, it is advisable to use this particular mode.

For the *random access transfer mode*, we derive more flexibility of loading and storing data, however, having higher costs. Here, a slot can request data as and when the input data is required for the computation during the entire execution time. This is done by transmitting an address of this particular data, thereby wasting a precious transfer cycle when a combined address and data bus is used. In contrast, having a dedicated address bus, the number of busmacros required increases.

A block-wise data transfer can also be implemented in the random access transfer mode, thus reducing the overhead of transmitting addresses for each single data word. Moreover, as a load or store operation can be performed in parallel to data manipulation, we can shorten the overall execution time in the random access mode by parallelizing computation and read/write operations.

Standard Bus-based Shared Memory Solution

Besides setting up a protocol on our own, we can also achieve the intermodule communication by a standard and usually heavyweight bus-based solution. Selecting a widespread bus would ease the acceptance and portability of the architecture, as designers do not have to implement the dedicated protocols of the solution above.

In a master thesis supervised by us [Sch08], therefore the Wishbone bus was used. Wishbone is an open source hardware bus that operates synchronously to a single clock. It serves as an interface to hardware cores written in different languages and thus offers a sound possibility to integrate portable cores. A short glance at the technical details allows us to rate the bus for our implementation purpose. The data width can be selected to be 8, 16, 32, and 64-bit. Thereby, address bus and data bus are separated. The bus facilitates a master slave system, with an arbiter to regulate bus access.

As main advantage, the user logic can rely on the well-documented wishbone characteristics and must not follow a proprietary format. This solution thus is more user-friendly when porting the design to another FPGA. The disadvantage of the bus is that it comprises of a high amount of wires and connections, which must travel through busmacros when crossing module boundaries.

Message Passing Solution

The two previous solutions target at the concept of shared memory, as the data transfer between the slots is done solely by referring to the same memory block. On top of this model, we could also install a message passing system. As a complete implementation of a standard message passing system, like the MPI (message passing interface) would require much resources, a lightweight approach as proposed and implemented in [SC06, SNRC06] would be appropriate. Moreover, the idea of message passing can be even more effectively applied to environments comprising of more than two slots.

3.4 Partitioning

The main application of the two-slot architecture is hardware virtualization—executing tasks that require more area than available. If not given a priori, we therefore have to derive partitions of the initial algorithm that can be executed sequentially. Such a method is called *temporal partitioning*. It differs from spatial partitioning in that the sequences of partitions must strictly preserve the precedence order.

In principle, there exist multiple partitioning methodologies. In this work, we first show a simple concept, before we extend a more sophisticated approach that was developed by Christophe Bobda at the Heinz Nixdorf Institute. As a prerequisite for our specific problem, each partition can maximally consume the area of a slot.

3.4.1 Simple Temporal Partitioning

We rely on data flow graphs comprising vertices and directed edges. The vertices represent basic arithmetic operations such as $+$, $\cdot$, $<$, etc. The derived partitions from this

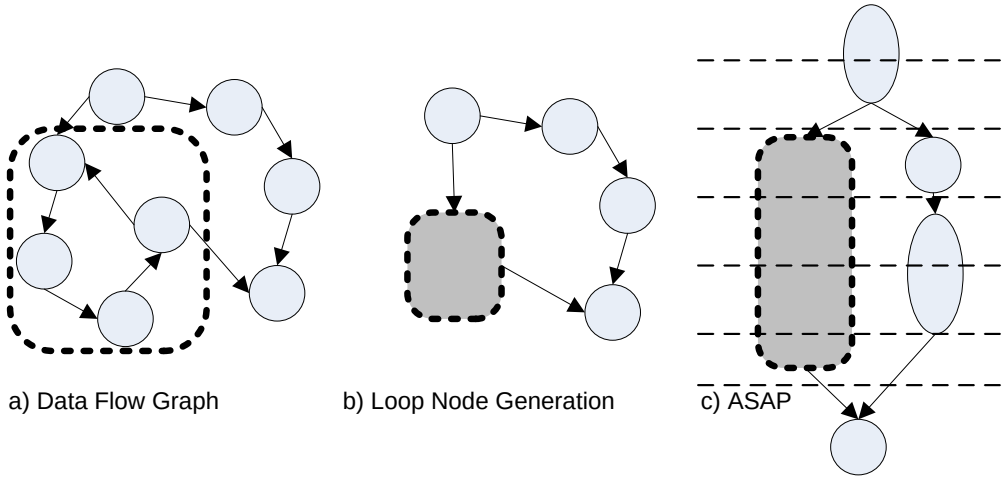

Figure 3.7: Simple partitioning, encapsulation of cycles

graph should each represent a block of closely connected processing operations, thereby also reducing communication requirements between blocks.

As we have to ensure a sequential chain of partitions, we group loops first and compose them into extended vertices, see Fig. 3.7. These vertices prevent having data/control dependencies of loops exiting the partition boundaries, cf. *basic blocks* of compiler design. Obviously, if such an extended node exceeds the available area of a slot, the partitioning is not possible and a redesign on a higher level of abstraction must be done.

Next, we order the data flow graph temporally, using the scheduling techniques ASAP (as soon as possible) or ALAP (as late as possible). Basically, these methods preserve the temporal precedence constraints. Additionally, ASAP/ALAP give each node a start and an end point on a global time-scale. In order to obtain the partitions finally, we enqueue the vertices in a list following the given order of the ASAP/ALAP schedule. Similar to list scheduling, we accumulate vertices to a partition until we meet the area constraint (refer to Fig. 3.8).

If we face communication between two vertices, which are not adjacent to each other, we include a NOP-vertex (no-operation-vertex) to all partitions that the communication spans. Figure 3.9 shows an example. Thus, we ease inter-module communication, as communication requirements can be traced.

As the result, we gain partitions having similar area requirements, all under the threshold of the slot. However, the execution times of the modules may be different, see Fig. 3.8.

3.4.2 Spectral Based Partitioning

Bobda [Bob03] came up with a new idea of temporal partitioning, particularly focusing on the communication. In reconfigurable devices, the communication between the partitions is of high costs, and we often have only a limited amount of communication resources passing the partition boundaries. Additionally, fewer communication results in a beneficial reduction of intermediate data storage.

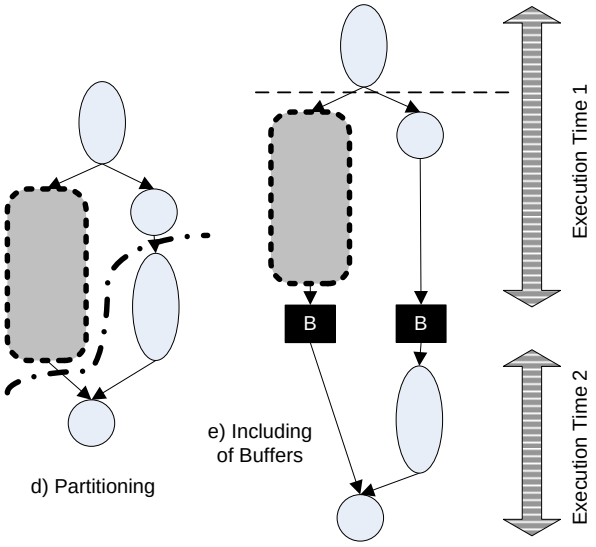

Figure 3.8: Partitioning continued

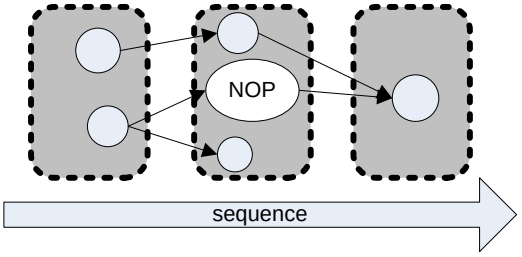

Figure 3.9: Including a NOP-node

The so-called spectral method—based on the spectral analysis of the input graphs as proposed in [Hal70] for VLSI design—re-arranges graphs in space respecting the communication by a quadratic objective function of the distances between nodes (wire length model). Its output is a communication optimized placement suggestion.

For spectral placement, the input graph must be given as the Laplacian matrix B, derived from the connection matrix C and the degree matrix D ($B = D - C$). The goal is to minimize the sum of squared distances between the nodes. Therefore, we apply the Lagrange multiplier method with the k Lagrange multipliers $\lambda_1, \lambda_2, \ldots, \lambda_k$. The solution are B's Eigenvectors associated to the k smallest non zero Eigenvalues. These Eigenvectors place the vertices in space, whose locations now refers to communication requirements optimized concerning the wire length.

In detail, Bobda uses the first two non-zero Eigenvectors to derive the x and y coordinates of a geometrical arrangement of the nodes. Then, he relies on the third Eigenvector as reference for the temporal partitioning. As the spectral method ignores the direction of edges, the immediate partitioning along the third Eigenvector in most cases does not respect the precedence constraints of the tasks. Therefore, Bobda iteratively derives bisections of the graph based on the ordering given by this Eigenvector. The data flow between each newly generated pair of subgraphs is homogenized by means of applying a modified version of the Kernighan Lin algorithm, which prefers to move tasks that reduce the amount of wrongly directed edges. Finally, two adjacent partitions will have edges in one direction only and a schedule can be derived.

The weakness of Bobda's approach is that during the Kernighan Lin bi-partitioning, valuable information of the proximity of nodes as initially given by the third Eigenvector can be lost over time. Depending on the size and geometry of the graph, nodes that initially are in complete different locations can end up sharing the same partition and

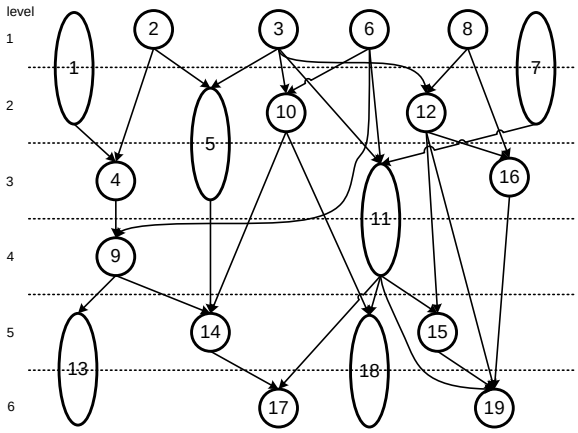

Figure 3.10: ASAP schedule of a data flow graph

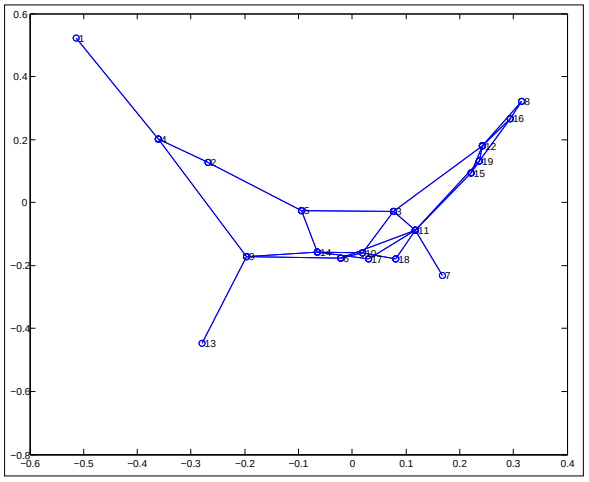

Figure 3.11: 2D spectral placement of the data flow graph of Fig. 3.10

thereby distorting the result of the spectral analysis. Moreover, the inherent parallelism given by the data flow graph is blurred in the spectral placement. The temporal ordering, which allows for sorting of vertices into levels that denote parallelism among nodes, is hardly respected during Bobda's method.

Therefore, we use a combination of the spectral method and temporal scheduling in order to derive mutual exclusive configurations that can be loaded in sequence onto FPGAs. As published by us in [DB05] for the domain of coarse grain reconfigurable systems, we combine the results of the ASAP scheduling (see Fig. 3.10) and the spectral placement (see Fig. 3.11). We derive an efficient placement for processing elements (PEs) of a coarse grained reconfigurable device. As displayed in Fig. 3.12, we therefore select the vertices and their coordinates of level 1 of the ASAP scheduling in the spectral placement of the data flow graph. As the distances of the vertices respect their closeness in terms of input for later nodes of the graph, we allocate the vertices to the PEs by a consecutive assignment. We start with the node comprising the smallest x and largest y value (top-left corner), which becomes the node for the PE in the top-left corner of the reconfigurable device. We continue until the node comprising the largest x and smallest y value is reached.

When scheduling the next level, we have to take care of vertices that have already started with their execution in the previous level, as these nodes should stay at their occupied PE. Additionally, their location acts as an indicator for placing close nodes of the ASAP scheduling in proximity, as far distributed results would foil the intended reduction of communication achieved by the spectral method.

We thus extract the corresponding vertices of the next (micro) level from the communication optimized spectral placement of the data flow graph and lock the already placed vertices. Then, we fill the gaps (speaking in terms of free PEs between locked PEs) by assigning the remaining vertices of this level. We thereby achieve a PE assignment that

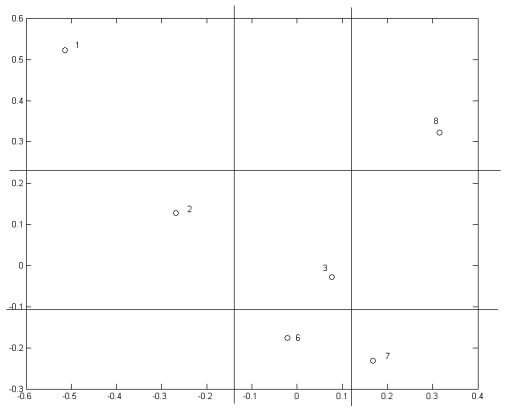

Figure 3.12: Nodes of ASAP level 1 are assigned to processing elements

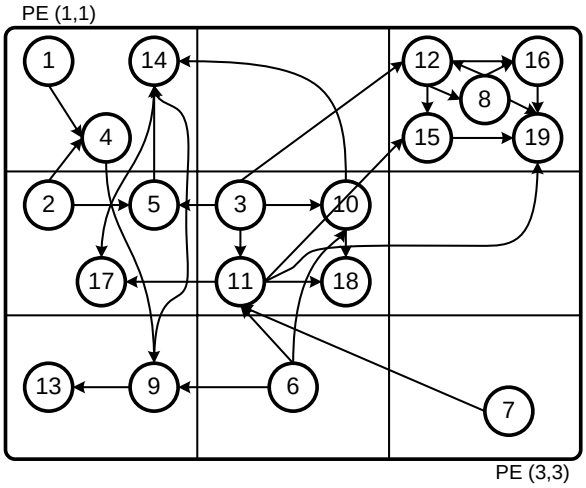

Figure 3.13: Result of the ASAP and spectral based node assignment

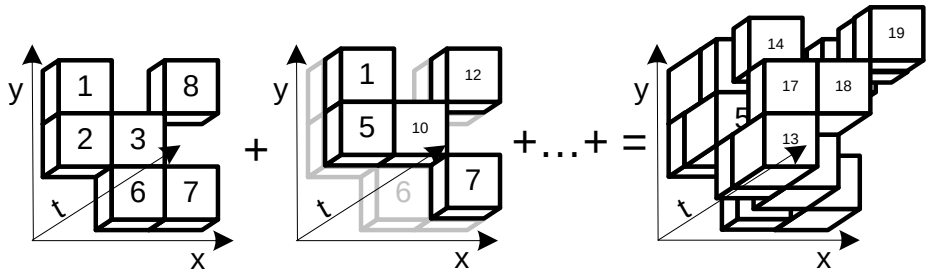

Figure 3.14: Complete schedule of the graph of Fig 3.10 for a coarse grain device comprising nine processing elements

respects the assignment of the previous level and serves as solid basis for the next level due to the PE assignment referring to the spectral placement of the whole graph.

In the end, we obtain direct neighbor connections or connections to the same processing element in the next micro level in majority. For the exemplary data flow graph, Fig. 3.13 shows the result, where we have mapped all levels into one figure—all vertices allocated to the same PE are displayed in the corresponding square box. The complete schedule including the temporal axis is displayed in Fig. 3.14, where the micro levels, which comprise a valid placement each, are combined. Thereby, different execution times of the nodes result in different heights of the boxes (nodes) in the final schedule.

Application for the Two Slot Framework

Our approach for the coarse grain architecture does not seamlessly scale to the requirements of the temporal partitioning for the two slot framework. For example the location information gained from the spectral arrangement and used to bind nodes to processing elements is only of secondary interest for the two slot framework. We might use this

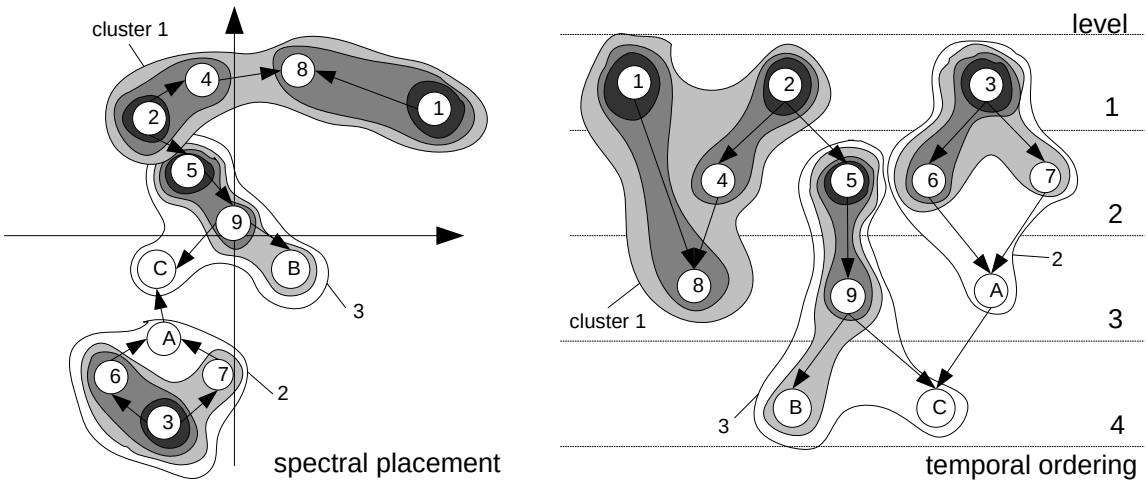

Figure 3.15: Clustering of a task graph: the clustering steps are denoted by the shaded areas, going from dark shaded to light shaded ones

information for easing the placement of registers, etc. within the reconfigurable regions.

Nevertheless, from our approach targeting coarse grain devices, we inherit the idea of combining the two objectives *parallelism* and *communication* that basically are contrary against each other. To recall, if we strictly follow the ASAP schedule, we can achieve a very high parallelism, as all nodes of one level become member of the same partition and thus can be executed in parallel. However, cuts of the initial graph do not respect communication constraints. In contrast, by referring to the spectral based clustering (partitioning), we derive clusters according to communication costs, however omitting precedence constraints and potential parallelism of nodes in the graph. We have designed an approach that combines both, the temporal schedule and the spectral placement.

The core idea is to obtain a clustering that derives partitions according to the spectral distance between nodes, refer to the example depicted in Fig. 3.15. To derive closely connected clusters, we gradually find the cluster members only by following the edges between the vertices. The nodes to start with are the nodes that are assigned to the first level according to the temporal ordering of the task set. Similar to depth first search, we start from these nodes and add the particular closest nodes first, referring to the spectral placement. If the closest node belongs to a cluster and the combined area requirements of both clusters do not extend the area available for a reconfigurable region, we combine these two clusters building a new one. Eventually, we close a cluster the latest before the maximum area of a reconfigurable region is met.

For finding new starting nodes for subsequent clustering, we refer to the temporal ordering. In detail, among those nodes who are not clustered, we select the nodes whose predecessors are clustered to become new starting nodes. Moreover, we prefer those nodes who are assigned to a lower level according to the temporal ordering. Thus, we gradually proceed along the time line (level) of the graph, and thus achieve to respect the task level parallelism inherent of the temporal ordering.

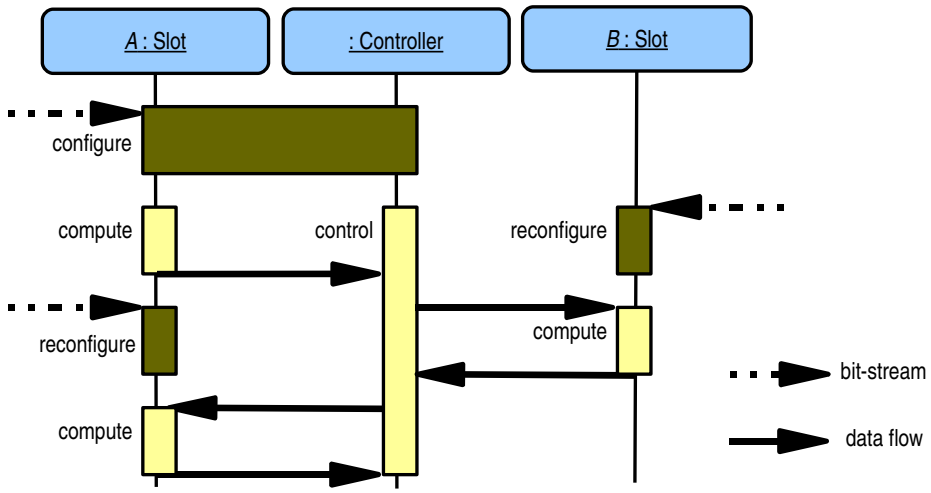

Figure 3.16: Sequence diagram of the slots and the controller entities. The data flow is indicated by fat arrows. [War06]

In the end, we derive clusters that respect the precedence constraints, bear task level parallelism of their members, and exhibit low external communication requirements thanks to the cluster building according to the spectral arrangement.

Two improvements have been developed. First, our concept improves its results, if the finding of the clusters can be done in parallel. If so, all clusters are filled in parallel, improving the probability to find the most closest node among all nodes. Conflicts then are resolved randomly. Second, if initial clusters only consume a fraction of the reconfigurable area and are combined to final clusters even if no direct communication exists, we can improve the overall response time. In detail, we then increase the amount of tasks executing in parallel, however most often suffering higher communication costs. To find an optimal solution that meets the architectural characteristics of the two slot architecture, usually multiple iterations become necessary.

To summarize, our procedure is a mixture of a clustering heuristic on the basis of the information given by the spectral graph and the precedence constraints of the temporal ordering of the nodes. To derive optimal results, several iterations may become necessary.

3.5 Scheduler

Having now a well-formed set of tasks or partitions, we can dispatch these partitions on the two slot architecture. Therefore, we first look at the scenario when hardware virtualization is applied, before we also discuss other scheduling techniques that may be used for independent task sets.

Scheduling for Hardware Virtualization

Including set-up time, the overall behavior in case of hardware virtualization on the basis of a chained set of partitions is as follows (refer to Fig. 3.16):

After the FPGA is powered up it has to be configured with the bitstream containing the initial top design. This top design includes the implementation of the controlling unit, and the first (left) slot filled with the first bitstream. Once the initial reconfiguration is finished the device can be fed with input data, which is stored in the controller.

After the input data transmission has been accomplished, the schedule can be started. The controller therefore activates the first (left) slot (A) by setting a control signal. When finished, the second (right) slot (B) is activated and simultaneously the reconfiguration of slot A is triggered. This behavior then is repeated with swapping A and B for each new configuration until the last partition of the algorithm could be processed. Afterwards, the control unit provides the final result to the requesting host. Thus, the computation in slot A and the reconfiguration of slot B take place simultaneously and vice versa, having the intermediate result stored in the controlling unit.

Note that due to the technical constraints of Xilinx FPGAs, we can also load the second (right) slot with the initial bitstream. Such an initial bitstream must always configure the whole FPGA. Subsequently, the configuration time stays the same as otherwise this area would be configured with blank bits.

To summarize, the input undergoes several steps of transformation, for each of which the corresponding task is loaded into the left or the right slot. Which slot is chosen depends on the tasks' serial number in the sequence to be even or odd. The intermediate results are always transferred to the controlling unit and are saved there temporarily.

Scheduling Independent Tasks

If the two-slot architecture shall be used for executing n independent tasks in the two slots, the question of minimize makespan—the overall response time or schedule length—arises. For a comprehensive consideration, the RT phases must be taken into account, which occur mutually exclusive. Basically, the duration of all RT phases is equal, while EX phases differ in execution time. Therefore, we have to consider the EX phases for optimizing scheduling.

Obviously, if all execution phases are shorter than the reconfiguration time ($EX_i \leq RT \ \forall \ 1 \leq i \leq n$), we simple schedule the tasks in an arbitrary order having the task with the shortest EX time sequenced last. Thus, makespan can be minimized.

Based on ideas of *parallel machine scheduling problems with a single server* [AWG06], we generally have to schedule a set of independent jobs having different execution times onto two identical machines, see also Chap. 5. In general, the problem is shown to be NP-hard [HPS00], however some heuristics for special cases could be derived. Some interesting ones for our problem are presented below.

For the special case of having tasks with equal set-up (RT) times, [KW97] describes the solution to first map the set of tasks to the problem of scheduling two parallel executing machines without set-up time. Thus, the set-up time gets included into the execution time of a task. Then, they transform the solution of this problem to our problem of having two machines (slots) and a single mutually exclusive server (reconfiguration port).

If we release the assumption of always having the same reconfiguration time, some more scheduling strategies are possible. Different reconfiguration times for the same area can be achieved, if we reconfigure only those SRAM cells that are changed between the configurations. Some approaches for this situation exist in the literature [CMS06, RM07] (ref. also to Sect. 2.2.5). However, these approaches are still in their infancy and therefore hardly applicable. Nevertheless, for future application, we still discuss such scenarios.

Again, if all EX phases are shorter than the RT phases of the other tasks ($RT_i \leq EX_j \ \forall \ 1 \leq i \neq j \leq n$), [AW02] proves that any alternating sequence with the task with shortest processing time sequenced last is optimal with respect to minimizing makespan.

In the same work, they also consider the case of all tasks having the same execution length: $EX_j = EX \ \forall \ j$. Here, minimal makespan can be achieved, if we first sort the tasks according to the duration of their RT phases. Then, if the number of tasks is even, we schedule the tasks according to the sequence: $RT_{2k-1} \leq RT_{2k-3} \leq \ldots \leq RT_5 \leq RT_3 \leq RT_1 \leq RT_2 \leq RT_4 \leq \ldots \leq RT_{2k}$, where $n = 2k$ for some k. Similar, for an odd number of tasks, we sequence the tasks according to $RT_{2k} \leq RT_{2k-2} \leq \ldots \leq RT_2 \leq RT_1 \leq RT_3 \leq \ldots \leq RT_{2k+1}$, where $n = 2k + 1$. Finally, the authors give two heuristics for the general case, where there are no constraints to EX and RT.

Note that also a subset of the partitions of a hardware virtualization may be independent and be applied to these scheduling algorithms.

3.6 Experiment

For evaluating the concept, we have implemented several prototyping environments on multiple platforms hosting different FPGAs. As the vendor tools only support the design of such partially reconfigurable systems to a limited amount, we had to overcome multiple challenges and accept compromises for the final implementations. Particularly the design of partial bitstreams is a daunting task, which eventually resulted in the design of our own tool *Part-E*, described in Appendix A. In the following, we discuss the general proof of concept implementation on a Xilinx Virtex-II Pro FPGA, as well as an application example that exploits the capabilities of a Xilinx Virtex-4 device.

3.6.1 Proof of Concept Implementation

We have implemented a prototype of a design with the two slots and a controlling unit. Both of the reconfiguration slots can be loaded with different tasks as intended by the behavior of the two slot framework. We also provide the user a way to define the tasks by filling out prepared entities with functional description. A notable amount of this work was done within a bachelor thesis [War06] and has been published in [WD06].

In detail, we have implemented this prototype on a Xilinx Virtex-II Pro FPGA embedded in the Avnet Virtex-II Pro Evaluation Board. The FPGA holds the controller and the two partially reconfigurable slots. For external communication, we use the periphery of the board. In detail, we use dip-switches to provide input data and a seven segment display to display the output values.

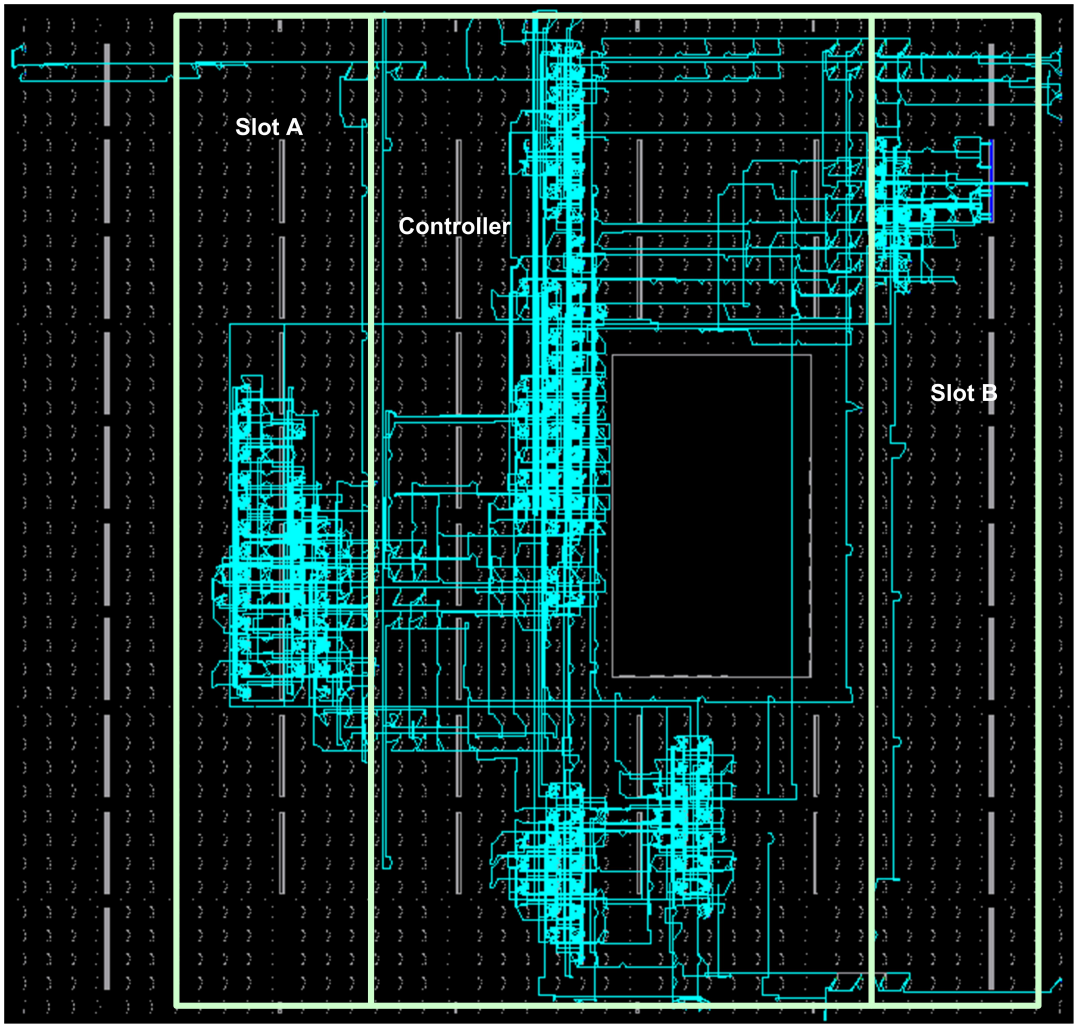

Figure 3.17: Layout of the two slot architecture on a Virtex-II Pro FPGA [War06].

Due to the architectural constraints of the Xilinx Virtex-II Pro FPGA, each run-time reconfiguration section spans a number of whole columns of the FPGA. For the inter-module communication, we use the tristate based busmacros. Apart from the I/O area defined by the development board, the Virtex-II Pro FPGA served our needs sufficiently for this proof-of-concept architecture. In particular, as the IO pinning for the peripherals is distributed all around the FPGA area, we required connection wires crossing the whole fabric including the crossing of reconfigurable regions (see upper part of Fig. 3.17).

Figure 3.17 shows the layout of the FPGA. We use a bit-width of 8 Bit for transferring the data or address, respectively. The controller including the communication infrastructure is very small and consumes less than 300 4 input LUTs (look-up-table). Although, the bit-width of the data and address transfer can be parameterized, it can become the bottleneck of the design, as the maximum amount of bus macros using the tristate version can be limited due to architectural constraints of the FPGA.

In general, the throughput of our data transfer protocols depends on the frequency of the whole system. Furthermore, if data transfer from and to the slots must take place in blocks before and after executing the task, the data transmission time has to be added to the task's execution time. However, if some results are earlier available than other ones, we can start data transmission already during the execution of the task, decreasing therefore the processing time of the task.

Portability

We have designed the sources of the two-slot referring to generic language constructs to facilitate portability of the design. For example, we can change the bit-width and other parameters by generic information. Furthermore, inferred structures, such as Block RAM and finite state machine, were preferred as device primitive instantiations. Moreover, we can change the number of bus macros used for the data bus, or define the size of the memory in the controller.

Nevertheless, porting the design to another FPGA generation proved to be complex, partly as new functional properties and design constraints arose, and partly as the physical assignment of the reconfigurable regions and static communication resources proved to be complex. For example, tri-state based busmacros could not be used any more (see below). Moreover, tool or documentation support for suitable area assignment on the reconfigurable substrate is hardly given and therefore makes porting to different FPGAs a daunting task. Efficient physical layout proved to require expert knowledge.

3.6.2 Cryptography Example

For a more complex experiment, we have used a cryptography scenario, relying on the symmetric cipher *triple DES*. The algorithm serves the requirements of the two slot framework very well as it is naturally split up into three equally sized single DES cores that are invoked one after another. We use a fixed key, which allows us to reduce the area requirement of each single DES compared to one generic DES core. Run-time reconfiguration still facilitates a regular key exchange for maintaining security.

The sequence on the architecture looks as follows: We load DES 1 in slot A and start processing in this section. In parallel, we reconfigure the other region (slot B) by loading DES 2. Once DES 2 has started computation on basis of the results produced by DES 1 and stored in the control section, we do not need DES 1 anymore. Thus, we replace DES 1 by DES 3. When DES 3 finally has deciphered the message, the result is stored in the control unit region and is marked as ready for further usage.

The input data for triple DES sums up to a blocksize of 64 Bits, which is transformed in the different DES stages without altering the length. We thus have to transmit 64 Bits between the stages of the DES algorithm. Considering the requirement of single directional busmacros, this sums up to 128 wires between each slot and the controlling unit. To reduce this high amount of wires, we implemented the shared memory solution.

We selected the Virtex-4 FPGA for the final design. As it represents a reasonably changed generation of Xilinx FPGAs, we could not simply merge the Virtex-II Pro implementation to this device. Most of all, Virtex-4 FPGAs lack tri-states and therefore

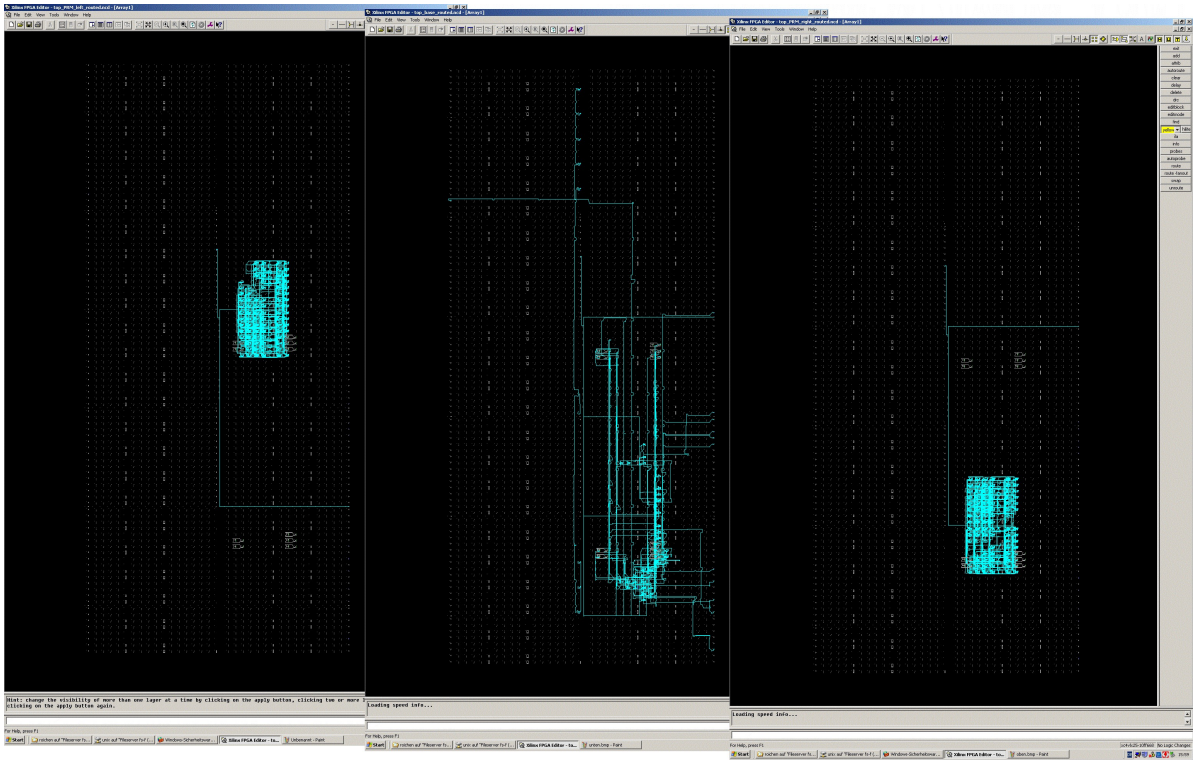

Figure 3.18: Layout of the two slot environment on a Virtex-4 FPGA. Left: the upper slot, right: the lower slot, in the middle: the controller logic.

require lookup table based busmacros. However, the Virtex-4 series allows for horizontal and vertical arrangement of reconfigurable regions, which eases the layout. Figure 3.18 shows the Xilinx FPGA editor display of the layout.

Moreover, we implemented a blockram of 8 Bit word-width and an address depth of 2^8 as shared memory. As input signals, this implementation then has the 8 signals for the 8 Bit data transfer, as well as some additional signals for control logic, particularly reset logic. The output signals sum up to 8 signals for the data out, 8 Bit for the address selection of the blockram, as well as again some control signals like a done signal. As each busmacro combines 8 signals, we thus require two busmacros for the input signals and three busmacros for the output signals, summing up to five busmacros for each slot.

For the hardware implementation, we used code provided by `opencores.org`. The reference implementation basically is generic—it may serve as decryption and encryption unit. Such a single DES core consumes about 300 slices on a Virtex-4 FPGA. If we use a fixed key, we can reduce this size by one third now accounting for only 200 slices.

Moreover, the additional logic in each of the slots required for data transfer sums up to about 100 slices. In detail, we require a state machine for gathering the data in 8 Bit words from the shared memory of the controller, as well as for sending the result back to the control area. As the DES accepts and provides the data as one block,

we have additionally implemented local latches for the input and output of the DES cipher. Furthermore, the reset logic to start the execution must be user-implemented for partially reconfigurable regions.

In summary, we thus require more than 300 slices for one module. To derive the best size of a slot that can host this module, we have to take the physical characteristics of the FPGA into account. Foremost, partially reconfigurable regions on a Virtex-4 FPGA always span a multiple of 16 CLBs (1 CLB equals 4 slices) in height. Moreover, the width of a reconfigurable region must also equal complete CLBs, having 1 CLB as a theoretical minimum. When using exactly 16 CLBs as the height of our slot, we thus require a width of 6 CLBs to host the approx. 75 CLBs of our module. This closest ceiling size of a reasonable partial bitstream then sums up to 384 slices.

The size of a partial bitstream on our Virtex-4 FPGA for reconfiguring these 384 slices sums up to about 24 kByte. If we use the Select MAP reconfiguration port of the FPGA, we can transmit the partial bitstream using a bit-width of 8 Bit and a reconfiguration speed of approx. 60 MHz. These parameters result in a duration of the partial reconfiguration of approx. 70 μs.

If we set the execution frequency of the circuitry to about 50 MHz, which is a reasonable number for the Virtex-4 FPGA, however, the reconfiguration time contrast with the duration of the execution phase of a few μs. In detail, each single DES requires 17 clock cycles for computation. For the data transfer, we have to add another 20–50 clock cycles. Therefore, we will result in less than 100 clock cycles for each single DES.

Moreover, the implementation of the two slot example proved to be challenging, as much expert knowledge is required to derive the placement of the reconfigurable areas, the busmacro placement, and the generation of the partial bitstreams. Even if a prototypical implementation is available, the porting to a specific application quickly can become complex. Nevertheless, to summarize the cryptography example, besides the problems of portability, which we could ease by virtue of our Part-E tool (see Appendix A), the remarkable difference of execution and reconfiguration time must be treated appropriately. A possible solution therefore is shown in the next section.

3.7 Extensions

The two slot framework can be extended in several ways to make up a more powerful concept. To show two meaningful examples, we first introduce concepts of low power design to the framework in the following. Thereafter, we describe how to use the framework as an IP core within a larger system.

3.7.1 Low Power Considerations

In most cases, the duration of a *RT* and an *EX* phase started at the same time will not be of equal length. Particularly the *EX* phases differ in their execution times, while the time required for a *RT* phase stays the same among all modules, as we always reconfigure

the same area. Most often the RT phases will be longer than the EX phases on average. In any case, the slower of the two phases has to wait for the other one to finish.

We employ this inevitable idling time due to different durations to introduce power saving. Therefore, we scale down the processing/reconfiguration frequency of those phases that would finish before the processing or reconfiguration in the other slot has finished. Frequency reduction is a well-known concept in low power design to reduce the energy consumption, thereby increasing the power efficiency.

The overall response time stays the same, if only idle times are avoided and no additional delay is added. Basically, both slots either performing a reconfiguration or an execution always start at the same time. To not harm the overall response time and achieve a maximum reduction of power consumption, we have to ensure that both phases finish contemporaneously not exceeding the initial maximum duration of the two phases. By selecting a suitable frequency, we can ensure this constraint. Then it holds that the pair EX_i and RT_{i-1} shows the identical maximum finishing time $\max(t_{EX,i}, t_{RT,i-1})$.

Background on Low Power

In general, the formula for the power consumption P of a design, which has V_{dd} as working voltage and f_{clk} as operation frequency is $P \approx \frac{1}{2} C_L \cdot V_{dd}^2 \cdot f_{clk}$. In order to compare designs, the result then must be multiplied by the time d an operation needs on this device, and is termed as the Power Delay: $PowerDelay = P \cdot d$. In the literature, we find two basic possibilities to save power:

clock gating Once a calculation is finished, the results are stored and the circuit is switched off by clock gating. In absence of power leakage, no further power is consumed after switching off the clock.

frequency reduction Higher frequencies consume more power. Thus, the clock is reduced and less energy is needed. In such systems, the working voltage can be reduced additionally.

Both can produce reasonable results. Strictly following these formulas, there would be no difference between *clock gating* and *frequency reduction*. Yet, the activation and de-activation of clocks consumes energy and additional clock-cycles. Thus, the reduced clock frequencies approach is often preferred due to this advantage. Furthermore, a smaller frequency often also allows for reduction of the working voltage. As this voltage is a squared value in the power calculation formula, results can be very effective then.

Still, frequency adaptation can consume time and therefore power as well. Sophisticated methods refer to a number of fixed clock frequencies only.

Frequency Adaptation in the Two Slot Environment

As the RT phase most often will be longer than the EX phase, we have investigated concepts of delaying the execution times of intermediate parts of algorithms. Basically, we reduce the clock frequency of the EX phase until the duration of the two phases are equal. To reduce the number of frequencies required, we cluster tasks into a set of clock domains. We thus introduce a multi-clock approach known from low-power design to partially reconfigurable systems.

Basically, modern FPGAs support different clock frequencies within the same design. Multiple clock signals are therefore fed into the clock network, which is often divided in multiple domains, also sharing the same regions. The improvement on power consumption by using different frequencies can hardly be measured on modern FPGAs. However, the approach can be of use in the near future. For instance, Xilinx's Virtex-4 FPGA was also designed with respect to power efficiency [Tel05].

The physical implementation of a clock scaling on modern FPGAs bases on the fact that Digital Clock Managers (DCMs) on such FPGAs can be used to not only stabilize the high and low phases of clocks but also to provide different frequencies. This technique often is more accurate than generating clock frequencies by virtue of user circuit. Alternatively, we could also use different external frequencies.

Concerning the two slot architecture, the intermediate control unit becomes responsible to generate, encapsulate and bridge the different clock frequencies. Therefore, it needs to provide different frequencies by either using the DCMs, external frequencies, or multiplying the input clock. These frequencies are fed into the reconfigurable regions using the different clock domains given by most modern FPGAs. Modules loaded to the slots then connect to their required frequency, leaving the others untouched.

For the data transmission to and from the slots via the buffer or memory in the control unit, we can rely on dedicated cores that offer a sophisticated bridge between different clock regions. Particularly dual-ported block RAMs are capable of operating on two different clock frequencies for each port. Alternatively, if the clocks are all a fraction of a maximum master clock, this maximum frequency becomes the reference frequency for the RAM. Data transmission from slots operating on a slower frequency will then consume a multiple of the time, however, not harming the behavior of the RAM.

For the solution of using a common bus for the data transmission between slots and control unit, we require the bus to accept clients operating on different frequencies. Otherwise, the user would have to adapt the interface of each single module to the bus depending on the operation frequency. We thereby would increase the design complexity and presumably contradict the idea of using a standard bus for convenient reasons.

To finally conduct frequency scaling with respect to power consumption, the open question of the assessment of the frequencies remains.

Clustering of Tasks with Different Execution Times

We want to constrain our design to a low number of clock frequencies, avoiding a dedicated frequency for every task. Therefore, we group those tasks to partitions that have similar execution times—*EX* phases—if executed by a common frequency f. Hence, the number of partitions equals the number of different frequencies required. We assign a dedicated frequency to each of these partitions, so that the execution times of the tasks approximately become the same.

We assign the maximum frequency of the system $f_{\max}$ to the group of tasks that have the longest *EX* phases. Moreover, we declare $f_{\max}$ to be the reference frequency f_{ref}. All other frequencies depend on f_{ref} and thus change relatively if f_{ref} changes. In particular if the duration of the *RT* phase (t_{RT}) is longer than the maximum of all *EX* phases ($t_{EX,\max}$), we reduce f_{ref} until the two durations meet: $t_{EX,\max} \approx t_{RT}$. All other

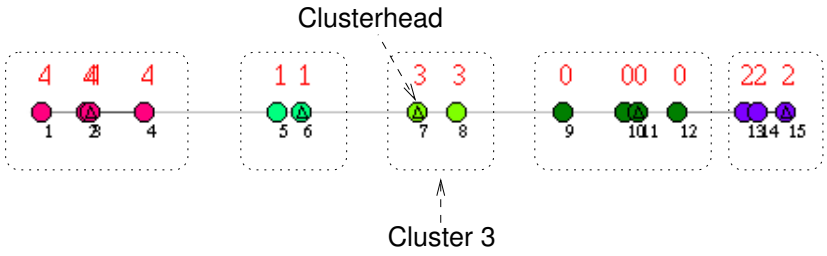

Figure 3.19: Example of produced cluster formation

frequencies then change also and hence the idle time due to waiting for a reconfiguration to be finished is reduced to a minimum.

To cluster the tasks into partitions according to the duration of their *EX* phases, we have developed a heuristic that has a very low computational complexity [DH06]. The algorithm is based on division of labor principle encountered in social insects [Hei08]. As a result of our clustering algorithm, each task will be assigned to a cluster ϕ.

First, we map the tasks on a line. The position of a task on the line corresponds to the duration of its *EX* phase. Closely placed tasks should become members of the same cluster, as they show similar response times if executed by the same frequency. To derive the clusters, we use the model of division of labor of ant to delegating the role of *clusterhead* to the tasks with high fitness [BDT99]. A task has a response threshold for every other task based on the distance of the tasks on the line. Tasks engage in becoming clusterhead when the level of the task-associated stimuli exceeds their thresholds. In our heuristic, the probability $T_\Theta(s)$ of a task to become clusterhead given a stimulus s is:

$$T_\Theta(s) = \frac{s^n}{s^n + \theta^n}, \tag{3.1}$$

where $n > 1$ determines the steepness of the threshold Θ (we use $n = 2$). The threshold models the fitness of the task to the role—a small threshold means that the task is a good candidate to be the clusterhead. The fitness of a task is calculated by

$$\theta = \frac{aver_{neig\_dist}}{neigh_{count}}, \tag{3.2}$$

where $aver_{neig\_dist}$ gives the average distance to the neighbors on the line and $neigh_{count}$ gives the number of neighbors (that can be 0 (no neighbors if we have only one task), 1 (task with shortest and longest *EX* phases), or 2 (all other tasks)). Tasks with more neighbors and small distances are preferred to be the clusterhead. Moreover, the stimulus s is proportional to the round r of the algorithm ($s = k \cdot r$), with k const $0 \leq k \leq 1$.

When a clusterhead arises, it starts to select its members. The clusterhead (and each member) helps to select the other members. The selection is done based on the distance to the cluster. We include the (not clustered) tasks having minimum distance to the cluster. The algorithm shows the best behavior if all clusters are built in parallel, which we can approximate by repeatedly iterating over the set of tasks.

In general, the process repeats until the number of clusters matches the architectural constraints of the FPGA—the number of clock domains available. Each cluster finally

denotes a set of tasks sharing the same clock frequency. Figure 3.19 shows an example output after execution of the clustering algorithm. In three rounds, it was possible to cluster the tasks of the system. As the tasks of cluster ϕ_2 have the longest EX phases (rightmost positions on the line), we assign to them the reference frequency f_{ref}, as no other cluster needs to be executed by a higher frequency to obtain the same execution time. We finally fix f_{ref} by referring to the duration of the RT phase. The frequencies for the other clusters then become a fraction of f_{ref}.

3.7.2 The Two Slot Framework as an IP Core

Besides using the two slot architecture as stand-alone execution environment, it can be integrated as an IP core within a System-on-a-Chip (SoC). The two slot architecture thereby offers a transparent and sound possibility to establish hardware acceleration. In particular, the size of the hardware circuits need not to be limited to the actual slot size, if the circuitry can be divided into partitions that can be loaded subsequently. Hardware virtualization thus is offered to SoCs.

The controller of the two slot environment acts as link to the remaining parts of the SoC. The overall behavior then is as follows: An application requests the two slot machine by denoting the algorithm to be executed on it, as well as providing the data input for the processing. Then a control thread takes care of the proper dispatching. Therefore, the appropriate bitstreams are loaded to the slots and the data is fetched, processed and finally written back to a specific place. Several implementation alternatives are possible, particularly concerning the location of the control thread as well as the grade of transparency of such a system. In the course of a diploma thesis under our supervision, we investigated the practicability of the two slot environment as an IP core within a larger system, also discussing the alternatives [Sch08].

Two gross architectural alternatives were developed. In the first one, the slot is integrated in the bus structure of a processor system. Here, the processor itself takes over the responsibility for supervision. In particular, the processor also triggers the reconfiguration. Therefore, the internal configuration access port (ICAP) is included as a core attached to the bus into the system. By virtue of the ICAP, the partial bitstreams can be reconfigured internally from the system. Therefore, the partial bitstreams are hold in a database, accessed by the controller and forwarded like standard data transfers to the ICAP. The ICAP core then triggers the partial run-time reconfiguration.

In a more heavyweight architecture, the emphasis was on transparency. Therefore the two slot architecture was enriched by an own controlling system consisting of controller (MicroBlaze CPU), memory, timer, interrupt controller, and the ICAP. These devices are attached to their own bus (on chip peripheral bus), which communicates by a bridge with the main processor. The benefit of this implementation is that the control thread is part of the two slot machine and does not load the main CPU with the task of reconfiguring the slots as well as with the data transfer from and to the slot. Moreover, the additional memory, which is part of the two slot subsystem, can be used for intermediate data storage. Thus, the memory in the controller of the two slot machine can be avoided and larger data can be transmitted.

In both examples, the communication between the two slots and the controlling unit was implemented using the wishbone bus system. Moreover, for the implementation, the Xilinx EDK (Embedded Development Kit) was used, which allows for designing SoCs comprising of one or more CPUs and their access to peripherals via IP cores attached to buses. Nevertheless, the capability for run-time reconfigurability is not supported by the EDK tool and had to be implemented manually.

3.8 Lesson Learned

The two slot framework served as introductory investigation of the task of designing reconfigurable systems. Despite its simplicity, multiple challenges had to be solved, which foremost allows us to draw the conclusion that the design of beneficial reconfigurable systems should be guided appropriately.

Most of all, if runtime reconfiguration shall be exploited, the reconfiguration overhead must he handled appropriately. The clear distinction of reconfiguration and execution phase—as introduced in this chapter—as well as their strict consideration overall within the design method helps to guide the design process and enables reasonable results despite the overhead. Furthermore, the benefits of hiding the reconfiguration time could be demonstrated in a systematic manner and is worth to be investigated further.

Moreover, we have shown that a clearly structured architecture eases the execution of tasks. The simplicity of the two slot architecture brings this demand to its basics. Still, it enables a sound hardware virtualization, as tasks are loaded into the two regions of the architecture on demand and are executed following the ordering given. Thanks to the alternating execution in the regions, we can interweave reconfiguration and execution, thus hiding the reconfiguration latency. As intended by hardware virtualization, tasks may be larger than the area available.

We have also shown that the physical implementation of an architecture for partial run-time reconfiguration proved to be a challenge in itself. Multiple hurdles like intermodule communication, area assignment, fragmentation prevention, etc. have to be considered. While the aforementioned capabilities enable sophisticated designs, the development process itself is characterized by a mainly too high level of freedom. The two slot architecture shows how numerous of these degrees of freedom can be constrained while still maintaining a reasonable performance. We thereby abstract the hardly controllable flexibility of an FPGA and offer guided access to the fabric through an architectural layer.

In general, we have shown that the envisioned layered approach is reasonable. In the two slot framework example, the architecture layer gathers all the implementation details, while the scheduling/dispatching and partitioning layer focus on the applications to be mapped onto the reconfigurable system. The clear distinction between the architecture, the scheduling/dispatching and the partitioning layer allows for separated optimization on the basis of well-designed interfaces between the layers. Even by virtue of the simple two slot framework, we could see that abstraction is extremely helpful to capture the challenges of reconfigurable computing systems design.

Furthermore, the clear distinction of the responsibilities as given by the layered approach allowed us to extend the framework quite easily. We have shown two examples, the investigation of the low power possibilities and using the two slot architecture as an IP core. Both proved to be applicable in a straight forward manner. Concerning the dispatching of tasks onto the two slots, we have shown the similarity to the single server scheduling concept, whose results are valuable for several task sets.

The two slot framework definitely is part of the general idea of reducing the complexity for the application developer, enabling her/him to concentrate on the designing of the actual computational structure. The framework hides the problems concerning runtime reconfiguration—communication and data transmission, reconfiguration scheduling, frequency adaptation, etc. Our contributions to the approach—the scheduling by virtue of concepts of the single server domain, the intermodule communication for transparent data transmission, partitioning concepts to map applications to the requirements of two sequentially executing slots—as well as the extensions—low power, IP core—expand the abilities and the fields of application for the simple two slot architecture.

To conclude, the two slot architecture approach serves a basic factum of reconfigurable computing—it hides the reconfiguration latency. Moreover, a high amount of extensions, while still leaving the core architecture intact, can be thought of. On this basis, we are motivate to further explore reconfigurable fabrics in the subsequent chapters. In particular, we are motivated to overcome the major drawback of the two slot architecture—the strict limitation to two slots. Although powerful, reconfigurable computing can mean more than simply alternating execution in two slots.

3.9 Related Work

In the literature, there are only a few contributions that consider architectures comprising precisely two reconfigurable slots. However, there are several approaches that focus on hiding the reconfiguration latency by interweaving computation and reconfiguration and thereby come across architectural concepts similar to ours. We present the architectural works first, before considering also the other aspects of this chapter like intermodule communication and low power processing on FPGAs.

Architectures

In [GV00] a conceptual approach for a lightweight reconfiguration platform is presented that minimizes the overall time of an application execution by means of using only two slots that are alternately reconfigured, performing configuration prefetching. The model used is similar to ours, and consists of two partitions on one reconfigurable device. The authors present a partitioning methodology improving the design latency. Moreover, they assume the reconfiguration time is comparable to the execution time, achieving a maximal overlap. Therefore, the authors discuss so-called block processing to amortize the reconfiguration overhead. Block processing, which is based on loop restructuring also found in [KVGO99], executes gradually on multiple sets of input data. Yet, block processing is applicable only for applications that process large streams of input data.

PipeRench [GSB+00, SWT+02]—also mentioned in Sect. 2.2.5—is a well-researched project of pipelined reconfigurable computing systems that investigates the problem of interweaving *RT* and *EX* phases from an abstract position. The authors call PipeRench a programmable datapath and target virtual hardware design through self-managed dynamic reconfiguration. Stripes are the basis of the system. PipeRench proves the benefits of using pipeline concepts for the execution of algorithms on reconfigurable systems. However, it is close to coarse grain systems and therefore only conceptually related to our work.

The same conceptual relationship holds for the works on configuration prefetching, which already were discussed in Sect. 2.2.5. We want to emphasize the early work [Hau98]. The author basically investigates configuration prefetching on a reconfigurable system model. He also sketches the requirements for optimal prefetch, which must consider the whole architectural constraints as is done in our work.

Communication

Several reconfigurable concepts, environments, or fabrics found in the literature address the problem of intermodule communication—the data transfer between dynamically reconfigured parts—in various depth. Few works consider the physical implementation problems when an algorithm is split into several partitions that are executed sequentially as it is done in our two slot framework. However, the problem of data transfer between temporally loaded modules whose communication requirements are also temporally, can be found in multiple reconfigurable execution environments. All the approaches that address fine grain reconfigurable systems like FPGAs share the general challenge of dynamically establishing communication between modules.

Xilinx itself proposes to use so-called bus-macros as fundamental concept for communication with and between reconfigurable modules [Xil04]. Bus-macros being tristate or look-up-table-based [HBB04, Hüb07] define a static communication structure that serves most communication requirements. Thus, bus-macros or similar concepts can be found in the majority of reconfigurable execution environments. However, the conceptional strategies for the communication itself are different (see below). Some of them have additionally been designed to solve on-line connection strategies.

Several authors (e. g. [WP04, UHGB04b]) use a bus system to enable data exchange between modules in their run time environments. The modules, which are placed on the reconfigurable area, must plug into this bus. Therefore, each component implements the bus transaction and an arbiter manages the bus-assignment. The bus enables deterministic behavior concerning different modules at the same location. Similar to our bus-based solution, there is always some overhead for the transaction: Bus access has to be granted by an arbiter and possible collisions lead to delay in data communication.

Recently, multiple improvements for the communication concepts of reconfigurable designs as offered by Xilinx have been introduced. In [Hüb07], a communication module called *bus com module* is discussed, which offers generic connection of modules in a slot-based system. For the hardware implementation, the author thereby introduces several sophisticated extension to the basic Xilinx busmacros, which allow him to overcome the problem of varying location of wires when signals cross reconfigurable regions. For free

2D placement of hardware modules at run-time, [HKKP07] introduces a communication macro for Xilinx FPGAs. The macro comprises a homogeneous communication infrastructure for any FPGA of the Xilinx Virtex family. Moreover, the Erlangen Slot Machine [BMA+05] comprises of sophisticated communication concepts including bus or shared memory based solutions.

Low Power

To approach the problem of power reduction on reconfigurable fabrics, several works present concepts to estimate the power consumption on FPGAs [SKB02, WOK+00]. Their techniques mainly are architecture dependent, however, they often can isolate the main power consumers. In general, the SRAM-based reconfiguration technique itself emerges as the main challenge.

Moreover, power consumption of reconfigurable devices has been addressed in several research efforts [MFK+00, SJ02] and most recently in [Tel05]. While the former works describe techniques how to achieve power reduction on existing architectures, the latter one also mentions that FPGA vendors increasingly focus on the problem of power consumption. Furthermore, new architectural concepts for the design of power-optimized reconfigurable fabrics are investigated in the literature [GZR99].

In [NB04], the authors describe dynamic scheduling for SoC platforms and investigate scheduling algorithms regarding power-performance trade-offs. Therefore, they use clock gating and frequency scaling to minimize the power consumption. The authors rely on two different clock frequencies, describing the potential benefits only rudimentarily. Our approach extends their concept by detailing the problem of frequency variation.

Finally, an interesting discussion on power saving by using a fine-grain real-time reconfigurable pipeline gives [KZP03]. Pipeline stages are disabled and bypassed whenever data-rates are low. Basically, we inherit from this work the potential of power saving by adapting the resources to the computational requirements.

3.10 Summary

In this chapter, we have introduced reconfigurable computing by virtue of a simple yet flexible example, the two slot framework. The framework bases on the introduced concept of execution (EX) and reconfiguration (RT) phase, both composing a unity for tasks to be computed on reconfigurable fabrics. Having two slots that can be reconfigured independently, we can apply the concept of reconfiguration time hiding by starting the EX and RT phases of two different jobs in parallel. We have discussed the architectural requirements to support such an alternating execution of task in two slots, including the important aspect of data transmission between reconfigurable regions. Moreover, we have implemented a reference architecture, which operates with two reconfigurable regions and one intermediate communication and control part.

As the framework targets on hardware virtualization, we have investigated partitioning strategies that preserve the precedence constraints for proper gradual execution. The partitions, each describing a sub-problem of the overall algorithm, then can be executed

in sequence. We have also discussed scheduling algorithms for a set of independent tasks that make use of the two slot architecture. We could thereby show the proximity of our problem to the domain of single server scheduling.

Moreover, several extensions to the architecture have been discussed, most notably the low power considerations. Here, we have presented an example how algorithms can be executed on reconfigurable devices using the concept of frequency scaling. Furthermore, we have discussed the described architecture as a reconfigurable fabric IP that can be parameterized and used in different designs such as Systems on a Chip.

4 Specification Graph Approach for Reconfigurable Fabrics

The previous chapter has shown that a sound design of reconfigurable systems often is challenging in many aspects. In particular the implementation of an execution environment that facilitates partial runtime reconfiguration demands tremendous effort and can hardly be argued to be done anew for every single design. This chapter therefore investigates a method how to map applications onto already proven runtime environments—as typical for the so-called platform-based design. Moreover, we include characteristics of modern FPGAs like heterogeneity and extend the design space towards multiple FPGAs. We thereby introduce a modeling and synthesis methodology that respects the specific requirements of run-time reconfiguration. The method is based on profound concepts, and expands known notations and model techniques.

4.1 Introduction

The main motivation of the method of this chapter are the requirements of modern embedded systems. These systems increasingly are multi-functional: In addition to a fixed core functionality, they must offer application dependent behavior. For example, mobile phones basically serve as communication devices. Therefore they can connect to multiple networks (GSM, WLAN, etc.), which require different algorithms for transmission/reception that furthermore depend on the currently active application. Additionally, mobile phones offer organizer functionality, entertainment, navigation, etc.

Basically, the computational hardware of such systems consists of a variety of processing units, including CPUs, DSPs, ASICs, and increasingly reconfigurable fabrics (most often FPGAs). Depending on the quality of service required, applications are executed on one of these devices. Reconfigurable devices thereby are the number one choice, if high performance and adaptability has to be granted. Furthermore, modern embedded systems often require new updates to be loaded into the system, which can again be offered by reconfigurable devices through a post-fabric reconfiguration.

To allow for beneficial exploitation of reconfigurable devices within such modern embedded systems, appropriate modeling and design methods are required. The so-called platform-based design concept thereby helps to consolidate reconfigurable logic as core parts within such systems. Subsequently, comprehensive synthesis and design space exploration methods are required. Such methods should respect the whole range of functionality of the reconfigurable fabrics, in particular the heterogeneity and the reconfiguration delay of modern FPGAs.

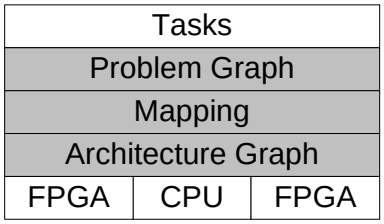

Figure 4.1: Layered model

As comprehensive methods to exploit these characteristics within the integrated design of embedded systems are not known, we introduce a synthesis methodology for reconfigurable systems that respects the specific requirements of run-time reconfiguration in this chapter. In particular, we shape and improve approaches and methods of embedded system design for the design of partially and run-time reconfigurable fabrics that themselves can be part of a complex system.

4.1.1 Layered Approach

Again, the method of this chapter shall be described using the layered approach. Figure 4.1 depicts how the method can be partitioned into three layers that lay in-between the tasks and the processing devices (one or more FPGAs, optional CPUs, etc.). Thereby, the tasks are expressed using a problem graph, which is a description of the structure and behavior of an application including its need for reconfiguration. An architecture graph abstracts the run-time environment consisting of FPGAs, CPUs, etc. The mapping in-between both layers connects tasks to processing resources. A sound description of problem, architecture and mapping given, we can derive Pareto-optimal solutions for the design under development.

4.1.2 Organization of the Chapter

In detail, this chapter is organized as follows: In the next section, we introduce the problem conceptually, including the characteristics of execution environments and the application areas. We also focus on an approach from the literature that has influenced our work. Then, we follow the layered model, where we first start from top, describing the problem graph layer. Next, we discuss the architecture graph before we meet in the middle. The three layers together also form the so-called specification graph. Furthermore, we discuss synthesizing strategies for this specification. We apply the concept on an example, take a look at the lesson learned, and review related work, before we finally close the chapter with a summary.

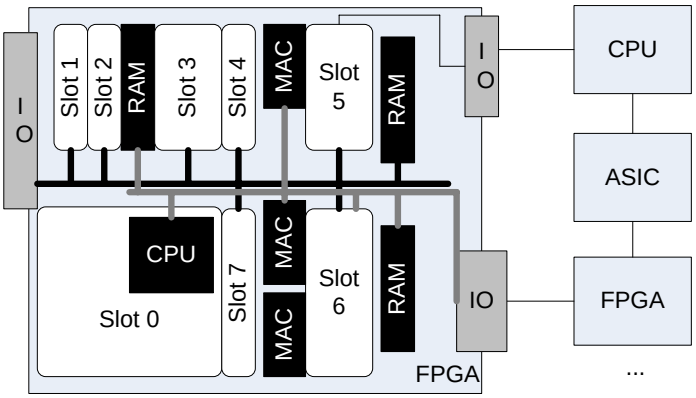

Figure 4.2: Schematic draft of an exemplary system focusing on the execution environment based on an FPGA.

4.2 Concept

As could be seen in the previous chapters, the reconfiguration and the diversity of FPGAs must be considered comprehensively to avoid them becoming the bottleneck of the system. Basically, run-time reconfiguration on one hand adapts a system, on the other hand consumes time. Moreover, module placement for heterogeneous devices as focused here demands special care. To include partial run-time reconfiguration and derive comprehensive methods for the design flow, we firstly state the problem abstractly, assign it to a subject matter of the literature, and sketch the solution procedure.

4.2.1 Problem Abstraction

The problem to be mapped on our system must be given as universal directed task dependence graph $G(V, E)$. The execution of such a graph is similar to an homogeneous dataflow graph, where processes are only activated for execution if all their predecessors have finished their execution.

The system to execute this graph is a combination of computation resources such as processors, ASICs, and reconfigurable devices (ref. to Fig. 4.2). We focus on the latter relying on an execution environment that helps to abstract the challenging details of dynamic resource allocation on FPGAs. Several groups have proposed such runtime environments for task execution on FPGAs, e. g. [BMA+05, FC05, KPR02, WP04].

In addition to these approaches, our environment may include all the modern FPGA capabilities such as embedded hard cores, etc. Thus, we will have a number of heterogeneous slots arranged on the FPGA. Sometimes, such a structure is also referred to as tiled. However, in our target environment the tile size and functionality must not be homogeneous. Regardless of the layout, the slots/tiles can be reconfigured individually, however using the single and mutually exclusive reconfiguration port. For message and data exchange, a static intermodule communication of the slots is defined by the environment and can range from direct wiring to a common bus or NoC for all slots.

Due to the heterogeneity, *application mapping* becomes complex, since tasks to be executed on slots may require different resources and their execution time may change depending on the resources available. Nevertheless, dedicated resources such as multipliers or DSP units are welcome concerning performance as their hard core implementation may notably speed up the processing.

Moreover, the reconfiguration overhead—duration of the reconfiguration phase—must be respected. Despite the fact that heterogeneous FPGAs might decrease the reconfiguration time, as more functionality is provided by fewer reconfiguration bits, the duration usually still cannot be neglected. Moreover, the single reconfiguration port of each reconfigurable device constrains the reconfigurability and requires a sequential ordering of the reconfiguration phases. If however our system hosts more than one FPGA, we could also reconfigure two regions at the same time if those regions are not located on the same FPGA, using, therefore, two different reconfiguration ports.

To summarize, for our scenario, the reconfigurable devices are abstracted by well-designed execution environments that respect the specific characteristics of the reconfigurable devices including runtime reconfiguration constraints such as suitable slot arrangement and communication structure. Such execution environments can be designed by FPGA experts that explore optimal solutions concerning placement, routing, etc. The method discussed in this chapter allows for reuse of these execution environments.

4.2.2 Background

As the method spans multiple research areas, the relevant background on each of them shall be briefly given next.

Platform-Based Design

A known methodology of the literature that argues for pre-defined and well-designed runtime environments is the so-called *platform-based design* approach. Platform-based design [KMN+00, SVM01] focuses on optimizing the system towards a specific application, while also offering freedom for extensions. A platform may be best defined as a library of components together with their composition rules, however not only containing computational blocks but also communication components [SVCBS04]. Using a platform avoids a design from the scratch, as it is predefined for a specific type of application. Thereby, a platform may also be extended by adding and integrating some more devices, growing over time.

Using a platform limits choices, thereby providing faster time-to-market through extensive design reuse. This reuse is moved beyond the individual IP block level to a reuse of architectures of hardware and software blocks. However, the limitation of choices also reduces flexibility and performance compared with a traditional ASIC or full-custom design methodology [Goe02]. For the design of complex products, we can easily accept this drawback, as we dramatically reduce development risks, costs and time-to-market.

Platform-based design is also termed *meeting-in-the-middle* process, where successive refinements of specifications meet with abstractions of potential implementations [SVCBS04]. Moreover, it harmonizes with the ideas of modern embedded systems, where

the behavior is composed of some core functionality (e. g., determined by the application environment) and additional dynamically selected functionalities (e. g., triggered by user interaction). Dynamically arriving applications occupy the left over, added, or temporarily unused resources. On CPUs, the free processor utilization can be used. On reconfigurable devices, we would load new configurations into unused areas. Our method presented in this chapter shows how platform-based design of runtime environments means an open-minded focus, in order to allow new tasks to enter the reconfigurable fabric, or to allow costly designed and reliable existing run-time environments to be exploited for additional applications. We thereby comprehensively show a way how to introduce partial run-time reconfiguration to the design of embedded systems.

In particular, we map our tasks onto runtime environments on FPGAs implemented by specialists. Such execution environments then include the heterogeneity of modern FPGAs and their tremendous design effort can pay off. For example, such platforms allow for providing sophisticated communication between the partitions within the fabric and to off-chip devices. In particular for runtime reconfiguration, execution environments are a very convenient way as they also hide design challenges.

Specification Graph Method

To avoid re-inventing the wheel, we are interested in a proven design concept as a modeling and synthesis approach that accepts our problem (graph G) as input and maps it on a runtime environment. Known approaches hardly consider partial run-time reconfigurability. We thus have to add the characteristics of reconfigurable systems like the reconfiguration time, dedicated computation resources, heterogeneity, communication requirements, I/Os, etc. We also target on transparently adding the reconfiguration latency to the scheduling. Furthermore, we want to allow multiple reconfiguration fabrics—thus having more than one reconfiguration port in the overall system.

Our approach bases on the methodology of Blickle et al. [BTT98]. They propose system design based on a specification graph, which consists of a problem graph and an architecture graph. Then, the combination of all phases of the system synthesis is an important means: the specification graph G_S, the allocation α, the binding β, and the scheduling γ. Having all these four elements in a valid combination results in a valid overall system. The fundamental characteristic of the specification graph method thereby is the inclusion of communication into the design as discrete nodes.

Haubelt improves this concept significantly in [Hau05, HOGT05, HTRE02]. He introduces hierarchy for raising the level of abstraction and for exploring reconfigurable hardware. The aim is architecture design for multiple purposes. Furthermore, he proposes the concept of hierarchical mapping edges, which significantly improves the specification graph model. We rely on several improvements of Haubelt's work. Still, he does not consider partial runtime reconfiguration in his approach.

4.2.3 Problem Solution

We model our design using two graphs: the problem graph and the architecture graph. The former describes the tasks and their precedence constraints and the latter the pro-

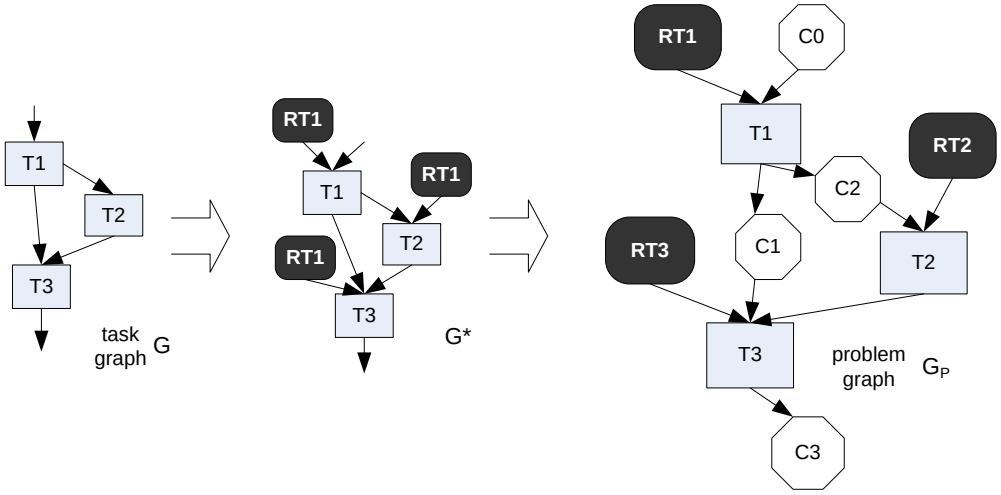

Figure 4.3: Task dependence graph G, extended by RT nodes to G^*, and corresponding problem graph G_P with communication nodes and reconfiguration phases.

cessing elements of the execution platform. Additionally, we rely on mapping edges between the problem graph and the architecture graph, which describe feasible bindings of tasks to resources. These edges are annotated with weights such as the time or power needed. We thus get the specification of the system as introduced by [BTT98]. Specification graphs and, in particular, their mapping edges are the underlying model for a design space exploration. As a consequence, proper mapping edges are crucial for reasonable results.

The final aim is to derive a binding and scheduling that is optimal in a multi-dimensional design space. Such Pareto-optimal points are computed by a design space exploration, based on the specification graph. For example, with our approach we are able to improve the response time and to reduce the power consumption. The benefit is that we can rely on optimally designed environments and thus focus on the mapping of problems and applications onto such platforms.

4.3 Problem Graph

The two constraints reconfiguration duration and mutually exclusive reconfiguration port demand for suitable integration into the modeling so that a comprehensive exploration of the reconfigurability is possible. In order to be able to respect these specific needs of (partially) reconfigurable devices, we extend our initial task dependence graph $G(V, E)$ via several steps into a problem graph $G_P(V_P, E_P)$, ref. to Fig. 4.3.

In detail, the initial graph $G(V, E)$ only consists of task vertices $V = V_T$ and communication edges $E = E_T$. As core step extending the method of Blickle et al., we add the reconfiguration phases V_{RT} to derive the intermediate graph G^*. To connect the new vertices V_{RT} with the task vertices V_T, every V_T gets an additional input from its reconfiguration phase V_{RT}, depicted by additional directed edges E_{RT}. The vertices V_T

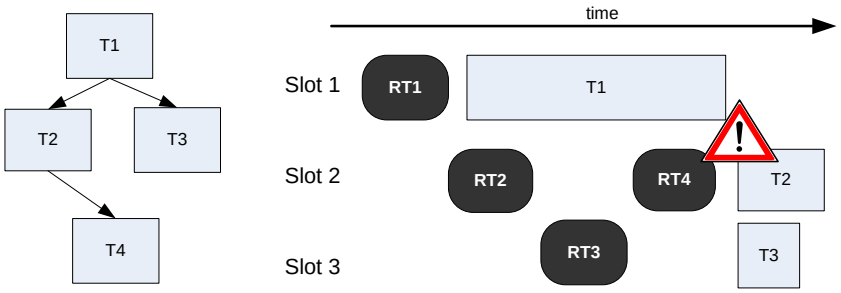

Figure 4.4: *Left* the task dependence graph, *right* a Gantt chart of a possible schedule: Conflict as Task 4 was reconfigured before Task 2 was executed.

and V_{RT}, as well as the edges E_T and E_{RT} then make up the important intermediate graph $G^*(V^*, E^*)$ with $V^* = V_T \cup V_{RT}$ and $E^* = E_T \cup E_{RT}$.

By adding the reconfiguration phases to become integrated vertices of the task graph, we basically do not extend the modeling set of the synthesis and can rely on already existing methods. Furthermore, the specific RT nodes V_{RT} for each task node V_T implicitly introduce reconfiguration prefetching to the model, as the RT nodes can be scheduled independently. However, as this fundamental step is new to the domain of system synthesis, it requires special care in the mapping phase, see below.

Next, as proposed by Blickle et al. [BTT98], we add the communication vertices V_C and derive the problem graph $G_P(V_P, E_P)$ from G^*. Therefore, we break all edges E_T between task vertices V_T and add communication vertices V_C in between. The edges E_T therefore also are split into edges from tasks V_T: E_{T0} and to tasks E_{T1}, both denoting the precedence constraints. By adding these new edges between task and communication nodes that respect the previous data flow, the model is completed. Figure 4.3 displays the result. To summarize, the problem graph $G_P(V_P, E_P)$ finally consist of the vertices $V_P = V_T \cup V_{RT} \cup V_C$ and the edges $E_P = E_{T0} \cup E_{T1} \cup E_{RT}$. Note that we do not have to add communication vertices between RT phases and task nodes, as there is no data transmitted between these two vertices.

In addition to the reconfiguration phase vertices V_{RT}, the integrated explicit nodes for communication form the basis for comprehensive synthesis of partially reconfigurable hardware, as communication requirements are now part of the system specification. Thus, we introduce dedicated communication needs of reconfigurable devices to the modeling phase of the design process.

Nevertheless, if we directly schedule this problem graph, we might get into a conflict. As reconfiguration phase and execution phase are independent nodes of the graph G_P, subsequent tasks could be loaded—reconfigured—on the same area of an FPGA, without intermediate execution phase. Figure 4.4 depicts an example. In particular, scheduling algorithms dispatching tasks in the order of ASAP (as soon as possible) would schedule the reconfiguration phases subsequently without respecting whether the corresponding tasks have been executed. This scenario must be prevented.

We therefore define intervals during which the area of a task must be preserved on an FPGA—the slot that has just been reconfigured must stay in this configuration until the

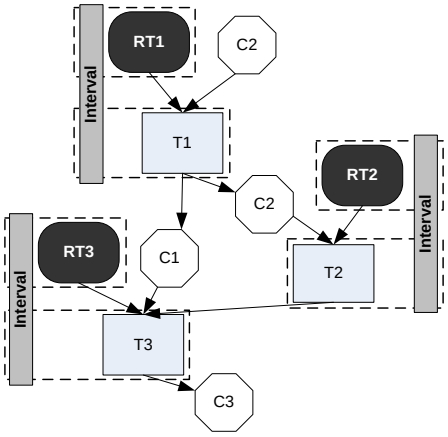

Figure 4.5: Problem graph with intervals/life cycles.

execution of the corresponding task has been taken place. We apply so-called life periods as displayed in Fig. 4.5. Life periods are often used in the domain of register allocation and initially help to constrain the amount of registers needed. Such intervals can be easily considered during the scheduling phase and are often solved using graph coloring, e. g., by the left-edge algorithm or similar approaches [KP87, PKG86, VKKR02].

Note that by attaching life periods, there is no fundamental change in the formal model or a reduction to a specific subset necessary. Thus, we can still apply known concepts from system synthesis. The information of the life periods can be easily used by the scheduling algorithm to prevent the conflict described above.

4.4 Architecture Graph

In order to achieve separation of behavior and structure, as proposed in the Y-chart approach by Gajski and Kuhn [GK83], we add a separate graph for the architecture. This architecture graph $G_A(V_A, E_A)$ becomes the second part of our formal model. Similar to Blickle et al. [BTT98], G_A basically abstracts the architecture and seamlessly includes the architecture into the design method. Moreover, compared to platform-based design, G_A also represents the platform.

Basically, the platform/architecture of our system is an execution environment that offers slots/tiles, which also can be combined, and an appropriate communication infrastructure. The graph G_A therefore basically consist of the slots and the communication. As the reconfigurability shall become an integrated part of the execution of tasks, we also add the reconfiguration port to the architecture graph. Figure 4.6 shows an example. We note the following about the characteristics of the new node: While providing no additional processing resources, the reconfiguration port is mandatory for the reconfiguration phase. Moreover, the reconfiguration port usually does not have connections to the other resources of the FPGA (ICAP and special architectures being the exception).

All in all, the architecture graph allows for abstracting an execution environment. Referring to the layered approach, G_A employs a layer of abstraction for the efficient

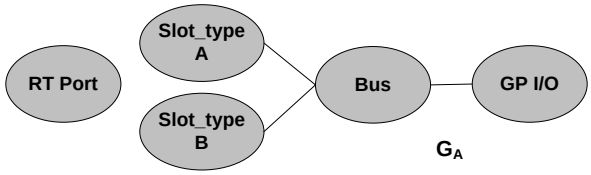

Figure 4.6: Exemplary architecture graph with two different slots, reconfiguration port, bus, and I/O.

and otherwise often time consuming usage of reconfigurable devices. The architecture graph thereby is capable to describe the heterogeneity of modern FPGAs, as nodes must not be homogeneous. The environments also may be the result of previous design space explorations, which have been carried out by a design method that resides below the one we target in this chapter.

By using such a detailed model of a run-time environment, we can take into consideration scenarios, where heterogeneous resources compose the execution platform of modern embedded systems. Moreover, we thereby are not forced to limit ourselves to one single FPGA only. Additionally, other resources like CPUs, etc. may become part of the architecture graph. By virtue of this flexibility, we thus resemble a comprehensive platform-based design, as we can consider heterogeneous platforms for a set of behaviors.

4.5 Mapping

To finally combine the two graphs, we derive a combination of the problem graph G_P and the architecture graph G_A. We therefore map vertices of the problem graph to resources of the architecture graph and derive the specification graph $G_S = G_S(V_S, E_S)$, cf. [BTT98]. In the end, the vertices of the specification graph consist of the vertices of the problem and the architecture graph $V_S = V_P \cup V_A$, while the edges are the edges of the problem and the architecture graph, as well as mapping edges E_M: $E_S = E_P \cup E_A \cup E_M$.

4.5.1 Basic Mapping Edges

To derive the specification graph, we have to map nodes of G_P to appropriate nodes of G_A by mapping edges E_M. The three elements of the problem graph require different consideration and therefore must be investigated in more detail.

The *computational vertices* V_T are assigned to slots, on which they can be executed. The tasks might fit into different slots, resulting in multiple mapping edges. All these possibilities are drawn. We attach the mapping edges with information about executing this task in the selected slot. In particular, we annotate the execution time, which is essential to derive a reasonable schedule. Furthermore, additional weights like power consumption are possible and can be used to improve the results of the binding phase.

Similar, the *communication nodes* are mapped to communication resources. Again, we attach specific characteristics to the edges, like duration of the communication or the load of a bus. This information again constrains the scheduling and binding.

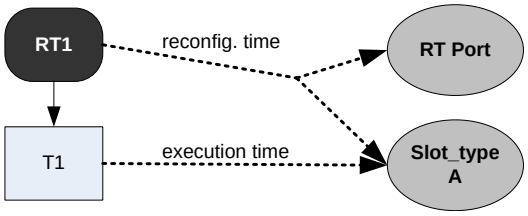

Figure 4.7: Specification graph with mapping edges of task and reconfiguration phase to slot and slot + reconfiguration port, respectively.

Finally, the *reconfiguration vertices* V_{RT} must be added to the specification graph. Thereby the following has to be considered: A reconfiguration phase can only take place if two constraints are fulfilled. First, the reconfiguration port must be free; second, the slot to be reconfigured must be available. A valid mapping edge must respect these interdependent requirements. Therefore, we rely on hierarchical mapping edges as proposed by Haubelt [Hau05]. These edges originally were introduced to increase the modeling power of the specification graph approach. In our case, they perfectly serve the requirements without the need to introduce a new modeling technique. By virtue of the hierarchical edges, we can denote the need for both reconfiguration port and slot. Figure 4.7 shows the concept where such an edge originates from the reconfiguration phase as source and splits to the two targets reconfiguration port and slot.

In general, a mapping edge for a reconfiguration phase is an edge E_M that has one vertex of V_P as source and at least two vertices of V_A as sink. One of the sink vertices always is the reconfiguration port. We map the reconfiguration phases on the basis of the already placed mapping edges—considering allocated slots for the associated tasks only. Thereby, we attach the specific reconfiguration time to the edges. Depending on the slot we chose, we derive the reconfiguration time based on the area consumed by this slot. This time usually stays the same, as always the whole slot has to be reconfigured.

4.5.2 Extensions

In some cases, a task might require more than one slot for execution. We can model this requirement by applying the concept of hierarchical mapping edges. Figure 4.8 displays an example. Obviously, the mapping edge of the reconfiguration phase must be adapted as well, including the addition of the reconfiguration times of the two slots.

Moreover, our target system is not limited to one single reconfigurable device. In such platforms with multiple reconfigurable fabrics—especially in the case of FPGAs—each fabric usually provides an own reconfiguration port, resulting in the possibility to perform reconfiguration on different devices in parallel. Such an improvement of the flexibility of the architecture can result in an increased overall performance of the system. However, architecture graphs for two or more FPGAs can quickly become complex or even unmanageable. Blickle et al. [BTT98] propose a concept of hierarchy concerning the architecture graph. We can apply this concept in order to abstract the architecture and introduce simplicity to the specification graph.

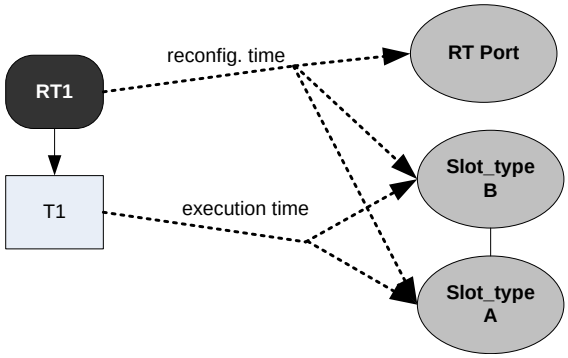

Figure 4.8: Part of a specification graph with task that requires two slots for execution.

Figure 4.9 shows a fictional scenario with two FPGAs. The first architecture graph G_{A1} (in the middle of the specification graph of Fig. 4.9) shows the architecture fine-granular, while the second one G_{A2} abstracts the architecture using a coarser granularity on a higher level only consisting of the two FPGAs. Mapping edges between G_{A1} and G_{A2} build the membership of the slots and reconfiguration ports to the specific FPGAs.

In the particular example of Fig. 4.9, we have displayed a scenario where two reconfiguration ports—thanks to two FPGAs—can be of benefit. If we select the emphasized mapping edges of Fig. 4.9, we can increase the overall response time, as the parallel executable task T_2 and T_3 are placed on different FPGAs and their reconfiguration phase can take place simultaneously. Figure 4.10 shows the corresponding Gantt chart.

To summarize the mapping, we take into account the heterogeneity of modern FPGAs, the reconfiguration overhead (phase and time), and the mutual exclusiveness of the reconfiguration port. Thus, we approach a realistic model for a partially reconfigurable run-time execution environment within modern embedded systems.

4.6 Design Space Exploration

The implementation—deriving a valid combination of allocation α, binding β, and scheduling γ—finally is done on the basis of the annotations to the mapping edges. As this synthesis task usually is NP-complete, we need suitable heuristic methods to gain solutions. Most often, we also have multiple conflicting design goals, such as power efficiency and the overall response time. To solve such a multi-objective problem, a design space exploration is recommended. In general, we focus on the Pareto-optimal points of the design space. The synthesis to derive these points can be done by applying evolutionary or genetic algorithms. During this synthesis, we have to select (β) a set of valid mapping edges from the specification graph G_S and schedule (γ) the corresponding problem vertices onto the resources referred to by the mapping edges. Blickle et al. propose a similar solution strategy based on genetic algorithms.

If we use a genetic algorithm to explore the solutions, the overall design flow is as follows: On the basis of an initial population, where we arbitrarily select a number of

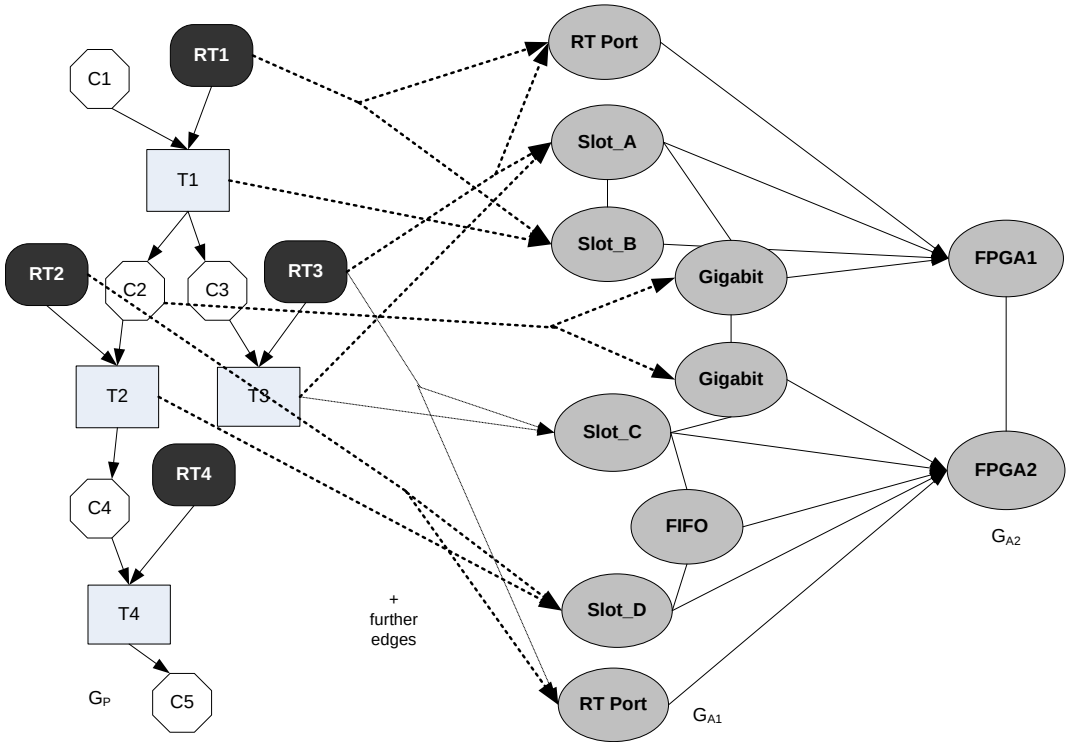

Figure 4.9: Specification graph with three graphs: one problem graph G_P and two architecture graphs. The system consist of two FPGAs with two individual reconfiguration ports.

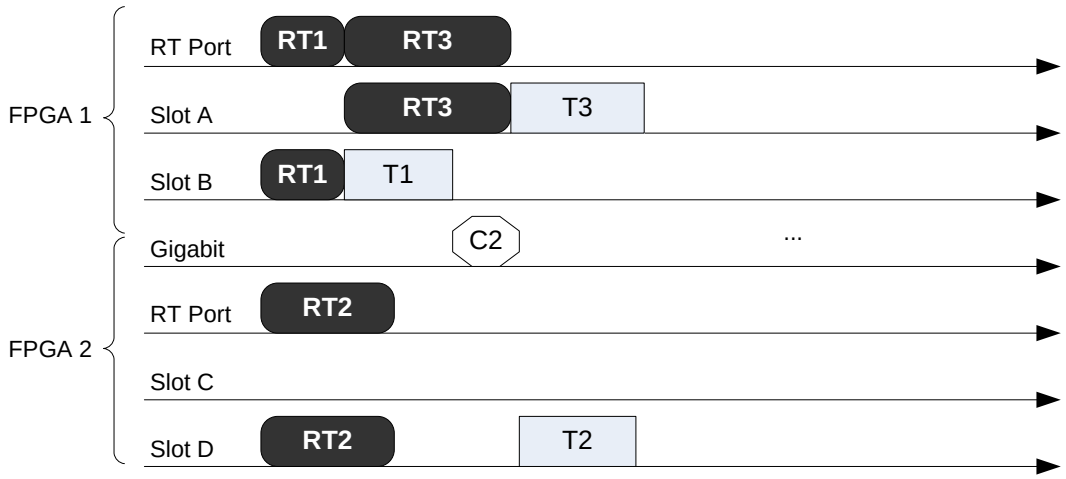

Figure 4.10: Gantt chart of a schedule of Fig. 4.9 where two reconfiguration ports are of benefit (communication resources are displayed shortened)

mapping edges for each individual of the population, we gradually improve this population in terms of evolutionary selection (survival of the fittest). Therefore, we require a fitness function to rate the individuals. The fitness can be derived from the characteristics of the selected mapping edges, based on their annotations. We also include the scheduling into the fitness function, similar to [BTT98].

Next, crossover and mutation will be applied to a number of good individuals, which have been selected according to their fitness value. We thus get the next population and can re-start the evolution. After a pre-defined number of iterations or if no further reasonable improvements occur, we stop the algorithm. Due to the randomness of the generation of the individuals, we also gain individuals, whose selected mapping edges do not denote a valid solution. By degrading their fitness value, we can temporarily accept such individuals, see below. Among the fittest individuals of each population, we always select those which form the Pareto front, gradually improving this front.

An individual thus is a selection of mapping edges, referred to as chromosomes. The width of a chromosome is the same as the number k of mapping edges E_M of the specification graph G_S. The binary entries are set to true, if this mapping edges is selected and set to false otherwise. Comparing the genetic algorithm to its natural counterpart, we refer to the entries of the individuals also as genomes.

Thereby, we will also generate individuals that cannot be synthesized feasibly due to several conditions. (1) There might be vertices V_P of the problem graph which have no binding to any resource, as no mapping edge leaving them is selected. (2) Some vertices V_P might have more than one mapping edge linking V_P to several different resources, resulting in multiple bindings. (A sophisticated scheduler might still be able to schedule such a situation by selecting a suitable binding itself. However, we should degrade the fitness of such an individual.) (3) The selected mapping edges might also harm the necessary precedence relations of RT phase and task execution concerning the resources. They might link to different slots and therefore a valid configuration cannot occur. The task will suffer starvation. (4) A similar problem occurs for the communication between slots. If for adjacent vertices of the problem graph the selected mapping edges result in slot and communication resource selections that have no links in the architecture graph, data transmission is impossible.

We decided to cover all these problems by the scheduler, which becomes a core part of the fitness function. Therefore, it either rejects a schedule completely (resulting in an infinite overall scheduling time) or degrades the fitness function significantly.

Additionally to the scheduling, the fitness function must rate the solutions in terms of other parameters attached to the mapping edges. Most often, the power consumption will be given. We can add these values from all selected mapping edges within the fitness function. Thus, the fitness function rates the binding and the scheduling. It returns as fitness value a vector of the parameters of this individual. For our example, we get the overall response time and the accumulated power consumption. We can arrange these results in space and select the fittest among them for deriving the next population. The selection is finally done randomly, however preferring those with a high fitness value.

For the two important steps crossover and mutation, which follow the evaluation of the fitness, we base on the standard procedure of genetic algorithms [Hol92].

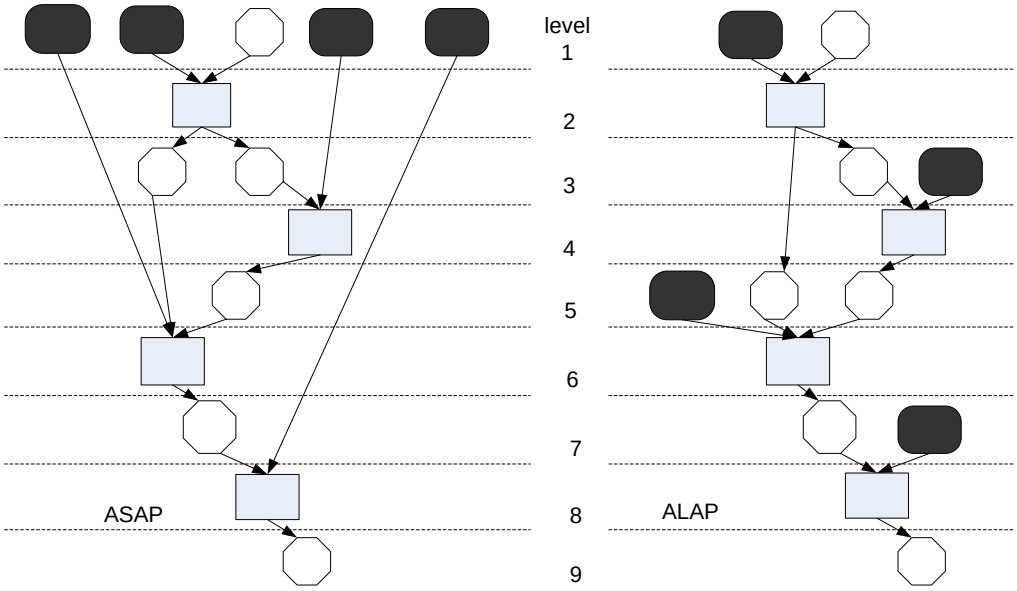

Figure 4.11: Task graph with communication and reconfiguration phases ordered according to ASAP (left) and ALAP (right).

Scheduling

The scheduling, which is a fundamental part of the fitness function, demands special care. In particular, we have to investigate how the added reconfiguration phases constrain the scheduling. In the following, we thus give an overview of the scheduling, including a prototype example of an algorithm. Basically, there are good heuristic scheduling algorithms for a given allocation and binding available, so it is not necessary to load the evolutionary algorithm with this task.

The input for the scheduling comprises the following: First and foremost, we base on the selected mapping edges E_M of the specification graph G_S as generated by the generation of individuals (binding). Moreover, we rely on the life periods denoted within the problem graph G_P and the adjacency information of nodes given by the architecture graph G_A and the problem graph G_P.

We use a list scheduling—the nodes are sorted into a list, which basically denotes the scheduling sequence. The list is derived by referring to the possibility to arrange the nodes over time. The simplest way is to order the nodes according to ASAP (as soon as possible) and first add all nodes of level 1 to the list, then the nodes of level 2, and so on. However, considering an ASAP ordering, we would insert all RT phases at the beginning of the list, as they are all in level 1, see Fig. 4.11.

In contrast, ordering the problem graph according to ALAP (as late as possible), we insert only those RT phases into the first level, whose corresponding tasks are assigned to level 2. This ordering matches the requirement of a chronological order of the RT phases according to their corresponding task much better than ASAP. Therefore, we prefer ALAP to fill our list for the scheduler.

Algorithm 1 Scheduling Algorithm

```
1: L ⇐ G_P
2: while L ≠ ∅ do
3:     v_P ⇐ next(L)
4:     if v_P has only scheduled predecessors then
5:         if v_P ∈ V_T then
6:             if ∀v_A ∈ V_A selected by v_P: activated by suitable interval then
7:                 schedule v_P
8:                 L ⇐ L\v_P
9:             else
10:                degrade the fitness and exit scheduling
11:            end if
12:        else if v_P ∈ V_RT then
13:            if ∀v_A ∈ V_A selected by v_P: no intervals pending then
14:                schedule v_P
15:                L ⇐ L\v_P
16:            end if
17:        else if v_P ∈ V_C then
18:            if ∀v_A ∈ V_A selected by v_P: ∀ṽ_P ∈ (predecessors and successors of v_P): ∃
   adjacent ṽ_A ∈ V_A selected by ṽ_P then
19:                schedule v_P
20:                L ⇐ L\v_P
21:            else
22:                degrade the fitness and exit scheduling
23:            end if
24:        end if
25:    end if
26: end while
```

As the fitness value bases on the overall response time of a schedule, we have to schedule the task set according to the selected mapping edges. In detail, to compute the overall response time, we deploy a vector that holds the resource occupation, including an additional entry for the life periods. The width of the vector resembles the amount of resources V_A of the architecture graph G_A. Each basic entry denotes the added computation time of this resource (including unavoidable idling times). In the additional entry for the intervals, we can set and unset the interval's number to denote whether or not an interval is active on that resource. We then always refer to the interval entry prior to scheduling a reconfiguration phase. By gradually filling and updating this vector, we eventually derive the schedule. In the end, the maximum time over all vector entries denotes the overall response time of the schedule.

In particular, each time a vertex v_P can be scheduled (see below for the detailed conditions to permit scheduling), we add the time (reconfiguration time or execution time) of the mapping edge to the selected resources of v_P. If more than one resource is

occupied by v_P, we first equal the times of the selected resources in the vector to become the maximum time of the selected resources, before adding the time given by the mapping edge. Particularly RT phases (V_{RT}) require both, the reconfiguration port and the slots to be free as they start the reconfiguration in all selected resources simultaneously. We thereby unavoidably add idling times to some resources.

Concerning the idea of prefetching reconfigurations, we always scan the whole set of unscheduled tasks for nodes that are ready for scheduling. A vertex is ready for scheduling if all its predecessors have been scheduled. We thus can schedule RT phases out-of-order, as these vertices are always ready for scheduling, because they do not have predecessors. However, the ALAP order guarantees that we still prefer those RT phases, whose corresponding tasks have an early position in the list.

Algorithm 1 shows the pseudo-code of the algorithm implemented. The algorithm operates on a list L that is initially filled with all nodes v_P of the problem graph G_P. The scheduler takes vertex v_P out of the list L, if v_P's predecessors have been scheduled. We decide based on the type of v_P what to do next.

If v_P is an ordinary task (EX phase) V_T, we check whether all slots required have been reconfigured. This verification can be done by considering the interval (life period) that are assigned to this task. Only if all resource nodes (slots) are blocked by the correct interval, a valid scheduling can follow. The resource vector is updated and the node removed from the list L. If the interval constraints cannot be fulfilled, we degrade the quality of this individual and exit scheduling.

In case v_P is a RT phase (V_{RT}), we first check whether the selected slots to be reconfigured are unoccupied—no intervals have been activated for these slots. If they are occupied, we skip this v_P and schedule it later by not removing this node from the list L. Otherwise, we occupy the reconfiguration port as well as the necessary slots and schedule the node. We also set the interval entry of the resources to the interval number assigned to this RT phase. As explained above, the finishing time of this reconfiguration procedure equals the reconfiguration time added to the maximum time of the corresponding entries in the resource vector.

Please note that intervals could also be substituted by a search based on the current condition of the nodes of the problem graph as well as the nodes of the architecture graph. However, only in the simplest case the reconfiguration that already resides in the region to be reconfigured (waiting for a execution that could not be scheduled yet, refer to Fig. 4.4) belongs to a task that is adjacent to the task, whose reconfiguration shall be performed, making a complete traversal of the problem graph necessary in the worst case. Moreover, if a reconfiguration phase shall take place on several regions, we have to start the search for each resource the reconfiguration phase maps to.

Finally, for communication vertices, we verify whether data flow between predecessor and successor nodes can take place according to the selected resources. At least one of the requested resources must offer a connection to the resources occupied by the previous or succeeding node. Again, we exit the scheduling and degrade the fitness, if communication constraints cannot be met.

Extensions

To also integrate ideas of reducing the reconfiguration times by considering the differences between configurations (refer to Sect. 2.2.5 and 3.5), the scheduling could be extended. Prior to adding a new task to the reconfiguration port, the scheduler therefore would have to derive the actual reconfiguration time of this task considering the already loaded bitstreams in the slots. However, the difference between configurations is hardly to calculate on the high level of abstraction our method targets at.

Using the same strategy, the scheduler may also take care of loops—traveling along the same vertices of the problem graph more than once. In the case of loops, caching of reconfigurations can be of benefit. If slots already hold the correct configuration, we get a configuration time of 0, and the reconfiguration phase can be dropped—termed as caching configurations. We still have to set the life period to the corresponding slots for a proper mapping of the task that belongs to this reconfiguration phase. Furthermore, a life line as presented by us in [ID05] may help to properly implement the loops. Such a line takes care of the causality of the tasks of a loop over the iterations. In detail, each new iteration stretches a new line over the initial node(s) of this loop. After mapping one node, the line gets pushed to the successors of this node. As no two life lines may overtake each other, we will always start a vertex $V_{P,n}$ of iteration n after $V_{P,n-1}$ of iteration $n-1$ has finished its execution.

To summarize the scheduling, each type of vertex is treated slightly differently. However, as the RT phases are included as ordinary nodes, the scheduler can intrinsically include the reconfiguration into the overall schedule, enabling reconfiguration time hiding. Moreover, we directly rely on the information of the life periods that was specified during the problem graph generation. The scheduler thereby might delay the RT phase of a task ready to schedule and prefer another RT phases whose slots are not occupied by a life period.

4.7 Experiment

To demonstrate the practicability of the method, we sketch an example application in the following. We therefore assume an execution environment implemented on a mobile device onto which we map a scenario of the image processing domain. In the scenario, we consider resource constraints of mobile devices and show a possibility to maintain usability. Based on the specification, we derive solutions and explore them.

Image Processing on Mobile Devices

Most of modern portable devices comprise of an attached or built-in camera that is capable to take pictures. Immediately after taking a picture, users often want to manipulate their pictures by applying filters, etc. As such algorithms often require relatively high computation power, they may be executed in hardware. To offer a variety of filters, reconfigurable fabrics can help, as they can temporarily host circuitries of the filters.

Moreover, typical mobile devices often comprise of relatively large storage but low CPU computing performance. Image filtering applications often are an add-on to the

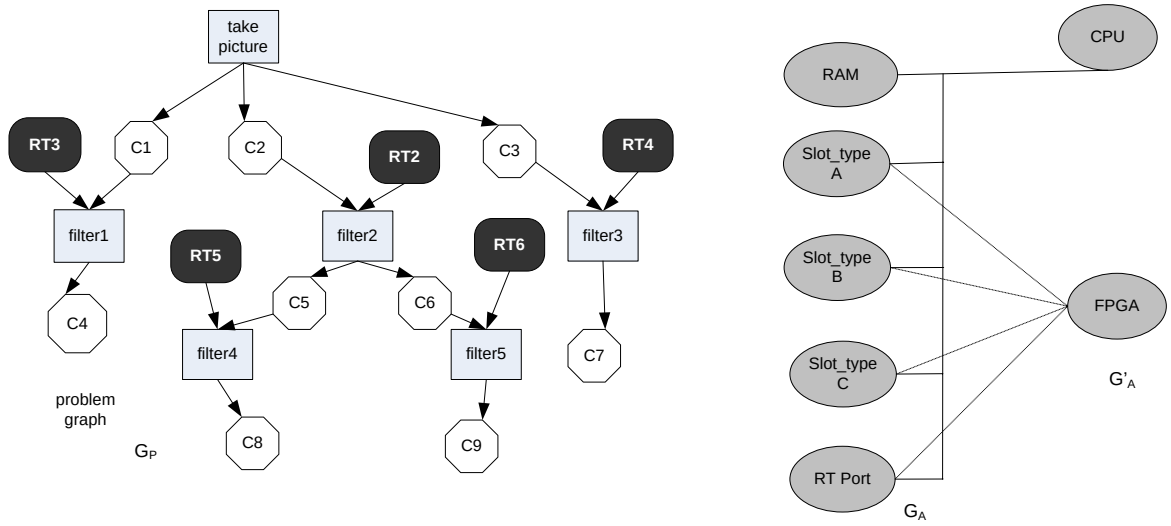

Figure 4.12: Specification graph of image filtering using limited resources, for the mapping edges please refer to Table 4.1.

basic behavior and therefore must be matched to existing processing resources installed in the device. Our method targets on exploiting such a given platform.

In detail, the application examples is as follows: The picture taken by a camera is displayed and additionally stored in the RAM. The user can now select among a set of filters, which are applied to the picture taken. To increase the response time—providing the filtered images as fast as possible—we pre-compute the most likely set of filters and store the image results in the RAM for fast access.

In Fig. 4.12, we depict a specification graph of such a scenario. The problem graph on the left side consists of the input node for taking the picture, as well as five filter nodes that partly also built on each other. The original problem graph is enriched by communication as well as reconfiguration nodes. Furthermore, we assign intervals (life periods) to the problem graph, which are not depicted due to complexity reasons.

For the architecture graph, we assume a heterogeneous environment consisting of a small CPU, a memory, and three reconfigurable regions on one FPGA. The slots are heterogeneous—they are of varying sizes and host different dedicated resources. As the method mainly focuses on reconfigurable fabrics, we leave the CPU abstract and do not detail it further. Basically, we require the CPU for accessing the input image. Data transfer between the slots and between slots and CPU takes place via the memory.

The mapping is simply done by first estimating (e. g., using algorithmic synthesis) the resource requirements of the tasks and communication nodes. Depending on the requirements, we can map the computational tasks to one or more slots. During drawing the mapping edges, we also attach the execution time estimation. Then, the reconfiguration nodes can be mapped to the nodes of the architecture graph, referring to the previously drawn mapping edges of the computational nodes and including the reconfiguration port as additional resource for each mapping. We thereby annotate the reconfiguration times to the edges. Finally, the communication nodes are assigned to resources. In

	A	B	C	A + B	B + C	A + C	A + B + C
filter 1	200	250	300	150	100	250	100
filter 2	175	100	125	n/a	100	125	75
filter 3	100	250	n/a	150	125	n/a	100
filter 4	200	150	150	150	150	100	50
filter 5	75	150	n/a	75	100	n/a	75

Table 4.1: Execution times of the filters in one or more slots given in μs

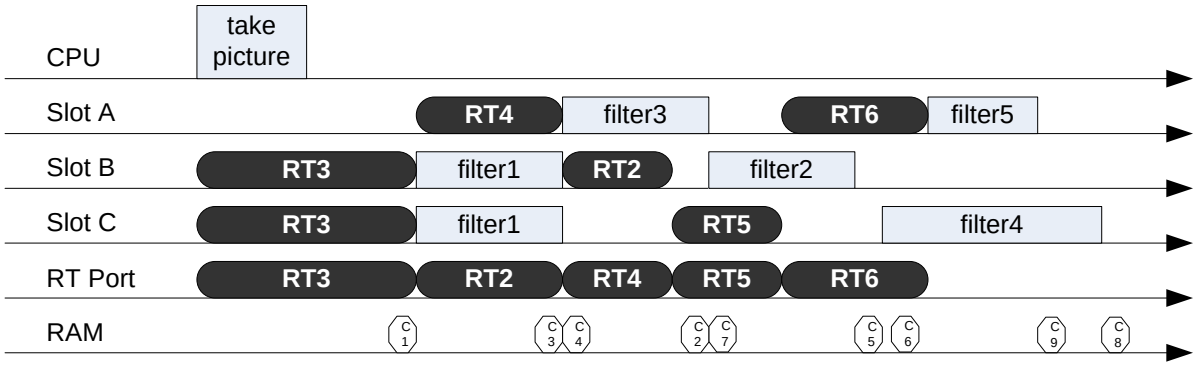

Figure 4.13: Example Gantt chart of the image filtering example derived from a Pareto-optimal solution

the scenario, the only architectural node that can handle communication is the memory node. Moreover, we quantitatively add assumptions about the power consumption to the mapping edges. For each reconfigurable slot, which is used or reconfigured, the power consumption thus is assumed to be 1 unit. Two slots require 2 units, and so on. Figure 4.12 displays the specification graph. The mapping edges area omitted because of complexity reasons an can be easily derived by referring to Table 4.1.

Next, we have to derive a Pareto-optimal binding and scheduling out of this specification graph based on the mapping and the annotated weights of the edges. This step is done by using the evolutionary algorithm as described in the previous section.

Results

We assume a comparable duration of reconfiguration phase and execution time of the filters. Referring to the cryptography example of the previous chapter (Sect. 3.6), the reconfiguration phase of a reasonable sized slot sums up to approx. 100 μs. We therefore have used the reconfiguration times for slot A: 100 μs, slot B: 75 μs, and slot C: 75 μs. Furthermore, for each communication vertex V_C, we assume a duration of 15 μs to transfer a complete image.

The number of mapping edges, which equals the width of a chromosome, sums up to 70. In detail, we have one mapping edge for the *take picture* task, which maps to the CPU, nine mapping edges for the nine communication vertices V_C, which map to the RAM, 30 mapping edges (refer to Table 4.1) from the filters to one, two, or three slots,

and the same number (30) mapping edges from the reconfiguration vertices V_RT to the reconfiguration port and the slots, which are to be reconfigured.

We used 30 individuals for each population. As the example thus is of low complexity, we could derive valid results in less than 200 rounds of the genetic algorithm. Moreover, the binding of the *take picture* task as well as of the communication nodes already is determined by the mapping edges, as there are no alternatives for these vertices. We thus have set these entries of the chromosome to valid (1) as the default value.

The Gantt chart of one of the Pareto-optimal solutions is displayed in Fig. 4.13. In the example, several filters are executed in parallel. Moreover, the concept of reconfiguration phase prefetching can be seen in the figure. For example filter 4 can start immediately after the data transfer C 5 is done, as its configuration could be loaded during the execution phase of filter 2.

4.8 Lesson Learned

To conclude, influenced by the platform-based design concept, we have defined a model and methodology to explore heterogeneous and partially reconfigurable systems. By virtue of the model given as a specification graph, we can synthesize applications onto run-time execution environments that have been designed by experts. Moreover, we do not rely on vendor tools and their adaption to modern FPGAs, but integrate FPGAs into standard design techniques, focusing on the ability to partially reconfigure FPGAs during run-time. The general result of this chapter thus is an approach for synthesizing reconfigurable systems based on the paradigm of platform-based design.

The method contributes to open reconfigurable systems to a broader number of engineers or system designers. A main reason is the abstraction from the difficulties of reconfiguration by providing an integrated means how to explore the resources of reconfigurable devices. These resources are treated as ordinary computational nodes onto which tasks can be mapped. The additional reconfiguration phases then introduce the reconfigurability to the system, for which the reconfiguration port also has been added.

The method is in line with the lesson learned of the two slot framework, where we could see that the design of environments that allow for partial runtime reconfiguration is challenging. The synthesis approach of this chapter enables to exploit runtime environments implemented on modern FPGAs, including their heterogeneity. We can rely on a given basic environment and map our architecture on this environment. Moreover, we improved the concept of equally sized slots towards heterogeneity.

The focus on execution environments eases the final (partial) bit-stream generation for the FPGAs. Moreover, having specific areas, there is no external fragmentation. However, the predefined execution environment in general and the fixed sizes of the reconfigurable regions in particular can also become a drawback of the method. If algorithms do not match the execution environment given, we have to initiate a complete new design, under the focus of generating a different task dependence graph.

Due to this constraint of having a fixed execution environment, the approach may fail to yield optimal results in terms of user requirements. On one hand, the capability of

reconfigurable devices to not only offer partial reconfigurable areas but also to adapt the whole execution environment is sacrificed towards an effective mapping method. On the other hand, the mapping as presented hardly can cover all information about an application. For example, priorities of tasks so far are not part of the approach. Moreover, the genetic algorithm may take a long time to find a solution, which in the end is obvious if the original intention of the algorithm would have been considered. For example, the computation of the bottleneck inevitably should be done on the most well-performing architecture node.

Nevertheless, the method of this chapter brings us towards true and realistic modeling of reconfigurable FPGAs, respecting all constraints—the communication (routing), placement, heterogeneity (hard core multipliers, etc.), reconfiguration times, reconfiguration constraints, I/O restrictions. Deriving a suitable environment on an FPGA is a difficult task, as all these challenging problems have to be considered. Relying on a fixed environment makes the system also better concerning fault tolerance, reliability, etc. We can thereby leave this tasks to experts of the FPGA domain.

Moreover, the concept of our method can also be used for large solution spaces, as the specification graph can be used as solid basis for heuristic algorithms. In particular the modeling and the derivable synthesis presented in this work will deploy their entire capabilities in larger systems. However, already the small example of an image filtering problem has shown the value of the method.

Finally, the method of this chapter basically is an extension of the specification graph approach to cover also partial reconfiguration. No new method had to be introduced, only improvements to the scheduling algorithm are necessary, which are transparent to an application designer. Thus, for designers used to the specification graph approach, the extensions of this chapter allow her/him to also consider partial run-time reconfigurable devices during the design space exploration.

4.9 Related Work

Besides the extension of the work of Blickle et al. [BTT98], which we have done in this chapter, we find also some other works related to synthesis for reconfigurable devices.

An interesting approach targeting reconfigurable devices with partial dynamic reconfiguration capability shows a physically aware hardware-software partitioning scheme for minimizing application execution time [BBD05]. The authors consider the exclusiveness of the reconfiguration port and the need for adjacent free columns to place tasks. Similar to the Blickle/Haubelt approach, they start with task dependence graph. On basis of the Kernighan-Lin/Fiduccia-Matheyes (KLFM) algorithm they have developed a heuristic for synthesizing such task dependence graphs on partially reconfigurable architectures. Results of this approach support the method of this chapter. Additionally, we consider communication as an integral part and therefore integrate it into the synthesis and design space exploration. We also improve the concept of heterogeneity and allow differently shaped slots. Furthermore, our method supports environments composed of multiple reconfigurable fabrics.

Eisenring and Platzner [EP02] use within their framework for design and implementation of reconfigurable systems also the approach of problem graph, architecture graph and mapping. As they do not add reconfiguration phases and reconfiguration port to the graphs, they provide a different concept how reconfiguration can be achieved, not primarily focusing on the reconfiguration time prefetching.

Despite little consideration on nowadays heterogeneous FPGAs, [HMM04] presents a sophisticated concept of modeling and optimizing run-time reconfiguration using evolutionary computation. The approach focuses on optimally reducing the run-time reconfiguration overhead during the HW-SW partitioning stage, which is done by detecting functional commonality.

In [MSV00a, MSV00b], the authors describe a hardware-software partitioning and scheduling approach for dynamically reconfigurable systems. Their method targets at multi-rate, real-time, periodic systems, based on genetic and list scheduling algorithms.

In [JYLC00] a HW/SW co-synthesis for run-time reconfigurable systems is presented, relying on an exact algorithm (ILP) and a KLFM-based approach. The ILP algorithm considers the single reconfiguration controller bottleneck and reconfiguration time hiding. However, while scheduling, the authors of the algorithm presented do not consider physical task placement constraints.

4.10 Summary

In this chapter, we have presented the integration of partially reconfigurable resources (FPGAs) into the specification graph synthesis, respecting the specific requirements of the reconfigurable devices, such as reconfiguration port, intermodule communication, etc. Therefore, we combined and expanded the specification graph approach, so that we can easily include partial reconfiguration and the corresponding reconfiguration times to our system. In general, our approach targets on the mapping of problems onto runtime execution environments (platform-based design). Relying on already proven concept of system modeling, known solutions for system implementation or exploration can be applied. We extend the known methods in order to handle specific requirements of reconfigurable systems, in particular heterogeneity and reconfiguration time. Furthermore, our work can handle multiple FPGAs having multiple reconfiguration ports. Such systems bear more parallelism. The final implementation is derived by a genetic algorithm, including a sophisticated scheduler for the fitness function of such an algorithm.

5 Reconfiguration Port Scheduling

The last chapter showed that fixed execution environments gain benefits for comprehensive exploitation of reconfigurable computing in several application areas. One of these application areas is the domain of reactive systems, as shall be shown in this chapter. Such systems constantly interact with their environment and therefore often demand for real-time behavior. They thus comprise of tasks having hard deadlines, for which appropriate scheduling algorithms must be offered. If these tasks are loaded dynamically onto reconfigurable fabrics, the scheduling algorithms must respect the characteristics of reconfigurable computing—reconfiguration overhead, mutually exclusive reconfiguration port, etc. In this chapter, we introduce a new approach for executing tasks in time-shared slots of a pre-defined execution environment under real-time constraints. The core concept of the approach bases on the loading of the bitstreams through the single reconfiguration port and thereby enables the application of well-known mono-processor scheduling algorithms.

5.1 Introduction

When partially run-time reconfigurable FPGAs are to be used as resources in hard real-time systems, the two dimensions area and time have to be considered in the focus of availability and deadlines. In particular, area requirements must be guaranteed for the duration of the tasks, so that the tasks can meet their deadlines. Thereby, reconfigurable devices can execute several tasks in parallel, which requires additional management.

To conduct proper task allocation—preventing fragmentation, offering communication, etc.—execution environments that abstract the space demand by dedicated pre-defined reconfigurable regions are a great help. Several of such execution environments have been proposed and implemented by different research groups. The environments thereby present a promising platform for real-time task execution.

For real-time scheduling tasks onto such platforms, however few approaches exist in the literature. Moreover, these approaches mainly neglect a fundamental bottleneck: the *reconfiguration port*. As all resource requests are served by this mutually exclusive device, profound concepts for scheduling the port access are vital requirements for FPGA real-time scheduling. The exclusive port is occupied for some reasonable time during reconfiguration, which cannot be neglected.

Thereby, a specific characteristic of the reconfiguration port, which looks restrictively in the first place, helps us to reduce the problem to a well-researched area. As the port must be accessed *sequentially*—the reconfiguration requests are served mutually exclusive—we can inherit and apply mono-processor scheduling concepts for the schedul-

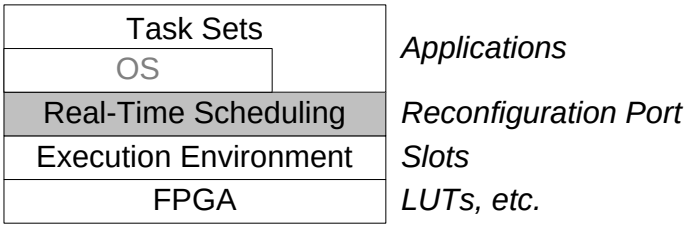

Figure 5.1: Layered Approach

ing of the reconfiguration phases of tasks to be executed on the execution platform. Mono-processor scheduling algorithms have been thoroughly discussed in several works.

Both scenarios—mono-processor and reconfiguration port—must derive a sequential schedule for an exclusive resource. However, a sequentially scheduled reconfiguration phase only prepares the actual processing of a task, which still shall be executed in parallel to other tasks residing in neighboring slots. Reducing mono-processor scheduling algorithms to a slotted architecture that allows for parallel execution of tasks therefore seems limiting at the first view. However, we show in this chapter that we still can meet the overall requirement of parallel executing tasks meeting their deadlines, in some cases even superior to other mapping or scheduling approaches, for example, as the concepts and algorithms of mono-processor scheduling are of low complexity.

Thus, we investigate the application of single processor scheduling algorithms to task reconfiguration on reconfigurable systems in this chapter. We determine necessary adaptations and propose methods to evaluate the scheduling algorithms for this novel concept. In particular, the reconfiguration process is implicitly integrated in a framework for real time execution of tasks on FPGAs, where the sequential access of the reconfiguration port is beneficially used for meeting deadlines.

In detail, we investigate several scheduling strategies known from single processor real-time systems, where tasks can arrive at the same or arbitrary times to the system. We investigate independent task sets and propose a novel approach where a task may be preempted in its reconfiguration phase, in order to achieve a feasible schedule. In general, the interesting problem arise, if we have less slots than tasks available. We thereby target on resource sharing of tasks.

5.1.1 Layered Approach

As displayed in Fig. 5.1, the real-time reconfiguration port scheduling again is based on an abstracting layered approach. The input given to the scheduling layer are task sets, which are executed by a real-time scheduling algorithm residing in the layer. A vital characteristic of the reconfiguration port scheduling approach is the reliance on a well-designed abstracting execution environment. In particular, we use a homogeneous slot-based execution environment to derive well-designed reconfigurable regions and to abstract from the low-level reconfiguration issues of FPGAs. On this abstraction, we apply the reconfiguration port scheduling.

As an extension, the scheduling algorithm can be activated through an operating system (OS). Then, the operating system dispatches the tasks by referring to the real-time scheduling layer. Basically, an operating system is optional but still recommended. The focus of this chapter thereby is on the scheduling layer.

5.1.2 Remainder of the Chapter

The rest of the chapter is organized as follows. First, we discuss fundamental considerations of the reconfiguration port scheduling approach. Then, we explain the reconfiguration port scheduling in more detail relying on aperiodic task sets. In Sect. 5.4, we discuss periodic task scheduling and how to guarantee feasibility for this most common scenario of real-time scheduling. Then, we investigate caching techniques for reconfiguration port scheduling. Before a discussion of the lesson learned, we report on experiments conducted in the context of this work. Finally, we list related work and summarize the chapter.

5.2 Concept

Our real-time scheduling layer accepts task sets and dispatches them on an execution environment. As this environment is an essential part, we first discuss the required architectural characteristics, before we define the problem of real-time task scheduling and finally introduce the reconfiguration port scheduling.

5.2.1 Execution Environment

A sound execution environment allows us to concentrate completely on the specific problem of reconfiguration port scheduling. Basically, the execution environment therefore must comprise a number m of slots to accept tasks for arbitrary execution in one of the slots. Hence, several constraints for the execution environment arise.

Foremost, to allow for arbitrary slot allocation, we require homogeneous slots—slots having similar characteristics, particularly concerning the area. Thus, each task that meets the area constraints of one slot can be executed in any of the slots. We thereby avoid external fragmentation by accepting internal fragmentation. Some architectural concepts additionally allow for dynamic width assignments of the tasks, while others hold the size fixed. For our method, the latter one primarily serves our needs.

Furthermore, the environments should abstract communication by offering a suitable communication structure, e. g. such as in the Erlangen Slot Machine approach [BMA+05]. Here, the communication to the peripherals is established through a run-time reconfigurable switch, allowing for any peripheral to be connected to any IO pin of the reconfigurable fabric. Thus, distinct communication channels are made up for each executing task. Alternatively, the communication with peripherals can also be achieved via a shared bus large enough to not impose the bottleneck of the system. Note that the tasks do not require inter-task communication, as they are independent.

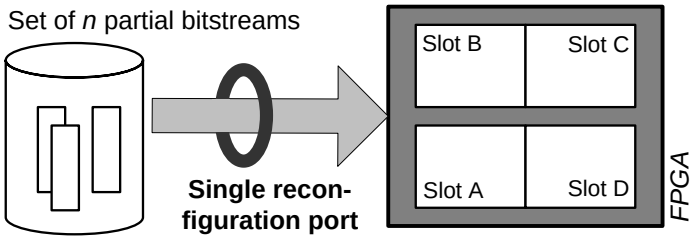

Figure 5.2: Execution environment having homogeneous slots

Moreover, we require a reconfiguration controller for the dispatching of the arriving tasks. This controller, which can also be located externally, provides the access of our real-time scheduling requests to the execution slots. Often, a CPU is included to the system as hard- or soft-core. This CPU then handles the reconfiguration.

A suitable run-time execution environment may look like the prototype displayed in Fig. 5.2. Concerning the communication, we assume one bus that runs on the highest possible frequency and allows each slot to demand the bus only a fraction of the highest speed (time sharing). As tasks on FPGAs generally are clocked relatively low, this bus is assumed to not become the bottleneck of the system. Partial reconfiguration capabilities enable a single slot to be reconfigured keeping remaining ones in execution.

Recently, several authors have proposed appropriate architectural concepts for fine-grained run-time reconfigurable systems [UHGB04b, WP04]. The architectures usually base on Xilinx Virtex FPGAs and comprise a specific number of slots, in which tasks are dynamically allocated and executed. The majority of these environments additionally implements a bus for task communication and a CPU for supervision.

Efficient executing of tasks on such environments still is not a trivial problem. Apart from area assignment, de-fragmentation and communication problems, which are extensively studied and abstracted on the above mentioned platforms, the reconfiguration itself demands further investigation. The duration of the RT phases and the sequential access of the reconfiguration port require special care.

5.2.2 Problem Abstraction

In order to formulate our aim to guarantee real-time constraints, we first abstract the dispatching of tasks onto the above mentioned execution environments. If we have n tasks to be executed, each in one of the m slots, and $m < n$—the number of slots is smaller than the number of tasks to be executed—we have to reuse the same slot for multiple tasks. Moreover, all tasks are loaded (by means of slot reconfiguration) through one single port. In order to handle this limitation of resources, we need a suitable mechanism to schedule the tasks.

Again, we model every task of our system with two different phases. The reconfiguration phase (RT) represents the configuration of the hardware itself. The RT phase needs to occur before the second phase, which is the execution phase (EX). Figure 5.3

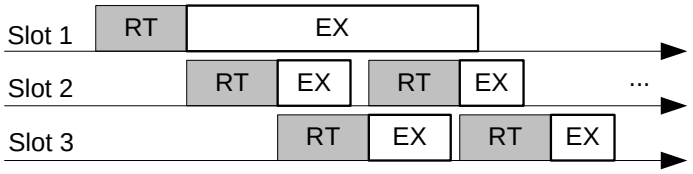

Figure 5.3: Example occupation of three slots.

shows these two phases of tasks scheduled onto three slots. Horizontally, we display the available slots and their occupation over time. As all tasks have the same size, the *RT* phases are of the same duration.

The task sets accepted by our real-time scheduling layer must follow standard requirements of real-time scheduling theory [But04]. We schedule a set Γ of n independent tasks τ_i. The tasks τ_i have different execution times $t_{EX,i}$ and can have periods T_i. All tasks have relative deadlines D_i. This deadline of the execution time is equal to the period, and can be computed for the kth instance by $d_{i,k} = T_i \cdot k$, or $d_{i,k} = T_i \cdot k + \phi_i$, if an offsetting phase must be considered. The meeting of this deadlines is essential for a correct behavior of the system. Furthermore, every task has a reconfiguration time $t_{RT,i}$, which is in our case constant for all tasks due to the execution environment that harmonizes area requirements.

As mentioned in the previous chapters, reconfiguration times can be reduced by applying a difference based reconfiguration and only changing those cells that alter between two subsequent configurations. However, in the worst case still the whole configuration information of a slot must be reconfigured. As we want to give hard real-time guarantee, we have to consider this worst case and therefore have to use the reconfiguration time given for a complete reconfiguration of one slot.

5.2.3 Reconfiguration Port Scheduling

The two characteristics of the reconfiguration process—sequential reconfiguration port and duration of the reconfiguration—enable the appliance of methods of the single processor domain: We assign reconfiguration phases sequentially to the exclusive reconfiguration port. If the reconfiguration phase for each instance of a task finishes early enough, the task can meet its deadline.

We thus schedule the reconfiguration phases on the reconfiguration port and therefore are interested in when the *RT* phase must have finished. Therefore, we introduce the relative (absolute) deadline D^* (d_i^*). This deadline is computed by

$$D^* = D_i - t_{EX,i} \qquad (d_i^* = d_i - t_{EX,i}), \tag{5.1}$$

denoting the latest relative (absolute) finishing time for an *RT* phase. If the reconfiguration for a task can be completed before D^* (d_i^*), the task will meet its overall deadline. We thus schedule the reconfiguration phases $t_{RT,i}$ with regard to their deadline D^* (d_i^*).

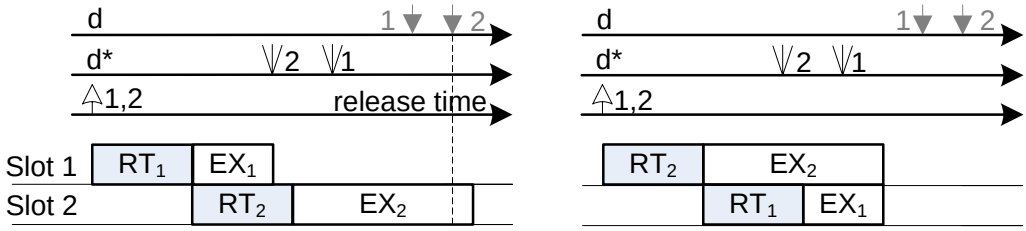

Figure 5.4: Scheduling according to d (left) and d^* (right), synchronous arrival times.

If we can guarantee the finishing of RT_i before its absolute deadline d_i^*, we also guarantee the completion of EX_i before d_i.

Figure 5.4 shows an example of the reconfiguration port scheduling concept. In the figure, we display a scenario of an execution environment comprising of two slots and two tasks to be executed on the slots. If we use the deadlines d of the tasks to make up the scheduling order—τ_1 is scheduled prior to τ_2—τ_2 will miss its deadline. However, using d^* as deadline for scheduling, τ_2 gets preferred and is scheduled first due to its earlier d_2^*. Subsequently, both tasks can meet their deadlines.

Moreover, for scheduling arbitrary task sets on mono-processors, also preemption is required. Preemption is a fundamental concept of many real-time scheduling algorithms, as it allows us to serve arriving tasks with higher priorities immediately. Thereby, preemption often increases the schedulability. More details on the characteristics of such task sets are discussed in the subsequent sections.

Concerning the technical implementation, we do not allow the preemption of EX phases due to several reasons: Basically, a preemption of EX results in the need to reconfigure the whole area twice. As the currently executing configuration of a slot is replaced by a new configuration, we have to reconfigure the slot once more to resume execution after the preempting higher priority task has finished execution.

Moreover, the question of saving states arises. Some concepts exist [Göt07], however they require additional implementation effort. Alternatively, we can perform a readback of the configuration of the task under execution. However, as a readback basically means to read in the state of the whole configuration bits of a slot using the same hardware resource as the reconfiguration port, a readback consumes approximately the same time, also occupying the port for the duration of the readback. We thus would double the additional time—time for the readback plus time for the subsequent reconfiguration after the preemption—in the worst case.

We therefore introduce the concept of preempting the RT phase—tasks are preempted during their reconfiguration. Although technically complex for the implementation, the preemption of RT phases does not substantially increase the whole reconfiguration time, as preempted reconfigurations are directly resumed in the future. The preemption thus is achieved by stopping the reconfiguration and resuming it at a later point in time. Such a behavior allows us to broaden the reconfiguration port scheduling by meeting the requirement of offering preemption.

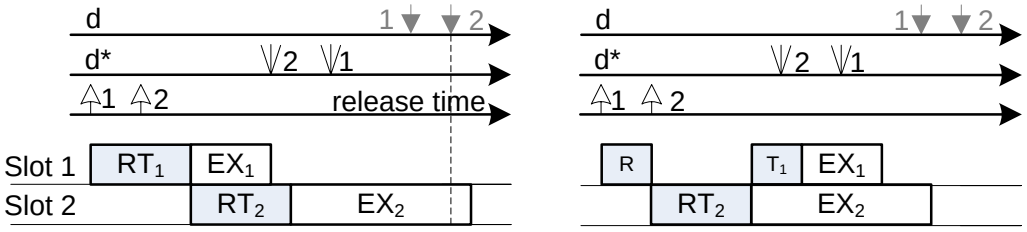

Figure 5.5: Scheduling two tasks according to d^* without preemption (left) and using preemption (right).

In Fig. 5.5, we display an example where such a preemption is of benefit. Comparing this scenario with the one depicted in Fig. 5.4, the tasks now have different arrival times. Without preemption—as displayed in the left part of the figure—τ_2 would again miss its deadline, as it cannot preempt τ_1 despite having a shorter deadline d^*. If preemption of the reconfiguration phase is allowed, τ_2 having a higher priority interrupts the reconfiguration of τ_1 and forces the reconfiguration manager to start its RT phase resulting in an earlier start of its EX phase. The reconfiguration phase of τ_1 is resumed immediately after the preemption. Eventually, both tasks can meet their deadlines.

Moreover, as the partial reconfiguration techniques of current FPGAs do not offer to interrupt the reconfiguration phase at any instance of time, but only after frames, we have to consider this minimum reconfiguration unit $RT_{min} = \Delta$ within our calculations below. This time interval Δ then also becomes the smallest time-step for tasks arriving to be scheduled by the scheduling layer.

In order to derive a schedule for task reconfiguration, we can now completely specify the parameters of the task τ_i in Table 5.1. The definitions are close to the ones in [But97]. In general, a schedule should satisfy an optimization criteria like the minimization of the overall response time or the maximum lateness, etc. We want to emphasize the maximum lateness, which is a known metric for the performance evaluation $L_{max} = \max_i(f_i - d_i)$.

In the following, we detail the reconfiguration port scheduling by applying d^* on a set of aperiodic tasks.

Table 5.1: Definitions

s_i	start time
r_i	release time
f_i	finishing time
d_i	execution deadline
$t_{EX,i}$	computation or execution time
t_{RT}	reconfiguration time
L_{max}	maximum lateness
d_i^*	reconfiguration deadline
Δ	minimum reconfiguration unit

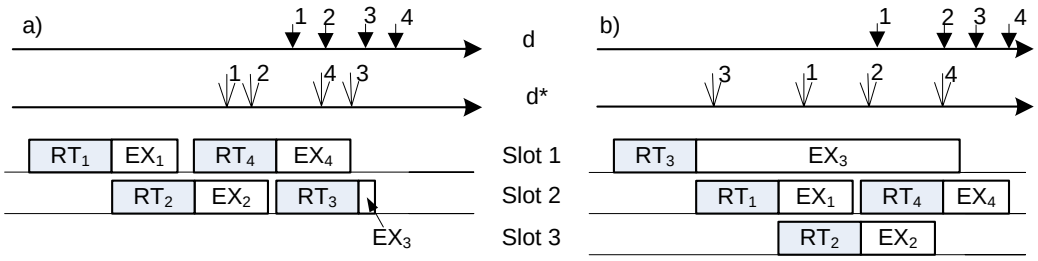

Figure 5.6: Scheduling task sets according to EDD. a) two slots, b) three slots

5.2.4 Parallel Machine Problems with a Single Server

We can compare the theory underlying our approach to the *parallel machine problems with a single server* [BDFK+02], see also Sect. 3.5. In these problems, a set of identical parallel machines is used for executing independent jobs of a task set. Prior to the execution of tasks, a set-up time of the job occurs, which is performed by a single server that dispatches the jobs to the machines. The parallel machines of this approach are comparable to the reconfigurable regions of our execution environment, while the server resembles the reconfiguration port.

However, in contrast, we allow preempting the reconfiguration phase, which is not allowed in the single server case. A direct mapping of our problems to the single server domain therefore is hardly applicable. We thus focus on the results of the works of the mono-processor scheduling domain.

5.3 Aperiodic Task Scheduling

In the case of aperiodic tasks that have no dependencies, we distinguish the two cases of synchronous and asynchronous task arrival. Basically, the former allows us to perform a schedulability analysis before executing the tasks, while the latter requires dynamic adaptation of the schedule.

We investigate the two known aperiodic task scheduling strategies from single processor design to solve these two cases: EDD and EDF. Motivated by the similarity of the single processor scheduling and the behavior of the reconfiguration port of the introduced execution platforms, we show how we can use these algorithms from the single machine environment in the domain of reconfigurable task scheduling [DF07b].

5.3.1 Synchronous Arrival

The scenario is as follows: A set of n aperiodic tasks has to be loaded into m slots ($m < n$), using the mutual exclusive reconfiguration port. The tasks have synchronous arrival time, but can have different execution times $t_{EX,i}$ and deadlines d_i. Note that schedules for this scenario do not need preemption as no new tasks will enter the system at run-time and all tasks arrive at the same time.

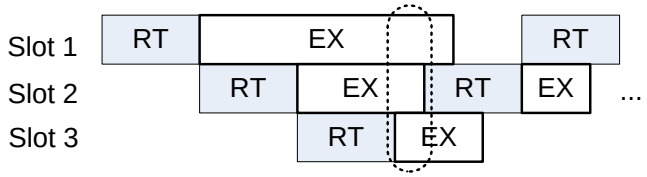

Figure 5.7: The *full load of slots* condition if three slots and n tasks having different execution times are scheduled.

The similar sequential scheduling problem is solved in the single processor environment with respect to minimizing the maximum lateness using JACKSON's algorithm, also called *earliest due date (EDD)*. The algorithm executes the tasks in order of non-decreasing deadlines. We apply EDD to our scenario and schedule the RT phases, using d_i^* as deadlines. Figure 5.6 shows some examples.

However, as each RT phase is followed by an EX phase that occupies the slots for the duration $t_{EX,i}$ of the task, the occupation of the slots also has an impact on the start of the next reconfiguration phase. In particular, both the port and at least one slot have to be available. Otherwise, when a reconfiguration request is pending and all slots are in EX phase, we suffer a waiting time and have to block the reconfiguration request, as displayed in Fig. 5.7. Due to the wanted avoidance of EX phase preemption, we suffer a specific condition, which we call from here on *full load of slots: fls*. We denote the duration of such a *fls* as δ_i. Only those tasks τ_i after the first m tasks to be scheduled can suffer this condition, with m as the number of slots.

Such a scenario, where the seamless scheduling of RT phases cannot be guaranteed is not covered by the original EDD algorithm. We thus extend the EDD algorithm as displayed in Algorithm 2. In the algorithm, we use a vector $\mathbf{z}$ to denote the occupancy of the slots. Below, we also show that in this case, the optimality of EDD cannot be guaranteed. However, every slot is executing—the FPGA is fully utilized and does not waste free space.

Algorithm 2 Earliest Due Date for Reconfigurable Slot Architectures

```
1: z ⇐ occupancy of the slots
2: if reconfiguration port is inactive (i. e., no RT phase is active) then
3:     Find slot where no EX is active
4:     if all slots are in EX phase then
5:         Wait until at least one slot is available
6:     end if
7:     Reconfigure slot r, (r = ind (min {z_1, z_2, ..., z_m}))
8:     Update (z)
9: end if
```

The scheduling for EDD can be improved by noting that the potential *fls* can be reduced in one single case. If for two subsequent tasks τ_i and τ_{i+1}: $t_{EX,i} > t_{EX,i+1}$, $t_{EX,i} \geq t_{RT}$, and $t_{EX,i} < t_{RT} + t_{EX,i+1}$, we can swap τ_i and τ_{i+1}. Thus, the starting times of the next two RT phases will be improved. This holds for $m \geq 2$.

Moreover, if we can guarantee at least one free slot at the beginning of each *RT* phase, all results of EDD of the single machine environment hold and EDD is optimal in our scenario with respect to minimizing the maximum lateness. A sufficient but not necessary condition to guarantee at least one free slot of the m slots available is

$$\forall i : t_{EX,i} < t_{RT} \cdot (m-1), \quad \text{for} \quad m \geq 2. \tag{5.2}$$

If Eq. 5.2 holds, every *EX* phase started in a slot m_k finishes before $(m-1)$ *RT* phases in the other slots m_l, $l \neq k$ could take place. Thus, it is guaranteed that slot m_k becomes free the latest before it is required for hosting the next *RT* phase in turn. This holds for all m slots ($1 < k < m$). A seamless reconfiguration thus can be guaranteed thanks to offering a free slot for any new reconfiguration request. However, the condition (Eq. 5.2) is only sufficient, not necessary. For example, Fig. 5.6 b) depicts a scenario where Eq. 5.2 does not hold ($t_{EX,3} > t_{RT} \cdot 2$, $m = 3$) and we still have at least one free slot available for any new reconfiguration request.

Furthermore, by virtue of Eq. 5.2 we can derive the maximum number of slots required to guarantee seamless reconfiguration. We therefore use the longest *EX* phase of all tasks $t_{EX,\max}$ and the reconfiguration time of a slot t_{RT} to derive m:

$$m = \left\lceil \frac{t_{EX,\max}}{t_{RT}} \right\rceil + 1. \tag{5.3}$$

The estimation only provides an upper bound, as less slots may also result in a seamless schedule, still minimizing the maximum lateness. Referring again to Fig. 5.6 b), m would equal 5 using Eq. 5.3, as $t_{EX,\max} = t_{EX,3} \lessapprox 4 \cdot t_{RT}$. However, the task set can be scheduled optimal (minimizing the maximum lateness) if only three slots are available. Using Eq. 5.3, we thus can only estimate the number of slots needed by an upper bound.

Guarantee

For the schedulability analysis, we construct the schedule referring to a vector ($\mathbf{z}$) that displays the current slot occupancy. Despite the a priori knowledge of the tasks, we have to dynamically react on the *fls* phases. These phases can be compared to dynamically arriving jobs, which however are ordered according to Jackson's rule and do not cause preemption, only additional *delay*.

When we want to perform a guarantee test—to guarantee that a set of tasks can be feasibly scheduled—we need to show that, in the worst case, all tasks can complete before their deadlines. The guarantee test for EDD in the single processor case is $\forall i = 1, \ldots, n : \sum_{k=1}^{i} t_{EX,k} \leq d_i$. In our scenario, due to the possible delays when all slots are occupied and the next *RT* phase is postponed (*fls*), we must extend every scheduled task by a possible additional δ_i. Thus, it must hold

$$\forall i = 1, \ldots, n : \quad \sum_{k=1}^{i} (t_{RT,k} + \delta_k) + t_{EX,i} \leq d_i. \tag{5.4}$$

The δ_k depend on the current occupation of all slots of the system and are difficult to compute. Therefore, our guarantee test avoids the explicit calculation of the δ_k by

computing the slot occupancies iteratively. We use the vector $\mathbf{z}$ that holds the current status of each slot. In the vector fields, we sum up the individual occupancy with respect to the global availability of the reconfiguration port. Therefore, we sequentially run through the schedule produced by EDD of Algorithm 2 filling this vector $\mathbf{z}$, whose entries z_l represent the slots of the reconfigurable fabric. The vector is updated each time a new RT phase starts. After the update, the vector's entries display when (time) their corresponding slots can be reconfigured next, also concerning the global condition—the occupancy of the reconfiguration port. Thus, by extracting the field with the smallest value, we can determine the next slot r for reconfiguration of the next task t_j. We apply

$$r = \text{ind}\left(\min\left\{z_1, z_2, \ldots, z_m\right\}\right), \tag{5.5}$$

while *ind* is the index function (see also Line 7 of Algorithm 2). If multiple z_l are minimal, the selection is done randomly.

In detail, the entries of the vector are updated ($z_{r,\text{old}} \Rightarrow z_{r,\text{new}}$) as follows: We add t_{RT} and $t_{EX,j}$ to the field of the selected slot (z_r): $z_{r,\text{new}} = z_{r,\text{old}} + t_{RT} + t_{EX,j}$. In order to update all other fields $z_l, l \neq r$, the following equation holds:

$$z_{l,\text{new}} = \max\left\{z_{l,\text{old}}, (z_{r,\text{old}} + t_{RT})\right\}. \tag{5.6}$$

Thus, if the finishing time of the RT phase of slot r is larger than $z_{l,\text{old}}$, slot l may be reconfigured, when the currently started reconfiguration has finished ($z_{l,\text{new}} = z_{r,\text{old}} + t_{RT}$). Otherwise, if slot l will still be in EX phase when slot r has finished reconfiguration, we must not select slot l for reconfiguration. Therefore, z_l keeps its value ($z_{l,\text{new}} = z_{l,\text{old}}$), which is larger than $z_{r,\text{new}}$ indicating its next availability for reconfiguration.

Hence, we can answer the question of feasibility of a task t_j—whether the deadline d_j of task t_j can be met. After each update of the vector due to the dispatching of a task t_j, it must hold $z_{r,\text{new}} \leq d_j$. After scheduling all tasks, we can calculate the overall finishing time $t_{overall}$ as

$$t_{overall} = \max\left\{z_1, z_2, \ldots, z_m\right\}. \tag{5.7}$$

Example

An exemplary sequence of the vector during a guarantee test for the scenario of Fig. 5.7 will look like the following. We start with an empty vector and schedule the first two tasks:

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} : \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} t_{RT,1} + t_{EX,1} \\ t_{RT,1} \\ t_{RT,1} \end{pmatrix} \rightarrow \begin{pmatrix} t_{RT,1} + t_{EX,1} \\ t_{RT,1} + t_{RT,2} + t_{EX,2} \\ t_{RT,1} + t_{RT,2} \end{pmatrix} \rightarrow$$

The update of the vector after dispatching the third task will result in keeping of the value of z_2, according to Equation 5.6.

$$\begin{pmatrix} t_{RT,1} + t_{EX,1} \\ t_{RT,1} + t_{RT,2} + t_{EX,2} \\ t_{RT,1} + t_{RT,2} + t_{RT,3} + t_{EX,3} \end{pmatrix} \rightarrow \begin{pmatrix} t_{RT,1} + t_{RT,2} + t_{EX,2} + t_{RT,4} \\ t_{RT,1} + t_{RT,2} + t_{EX,2} + t_{RT,4} + t_{EX,4} \\ t_{RT,1} + t_{RT,2} + t_{EX,2} + t_{RT,4} \end{pmatrix} \rightarrow \ldots$$

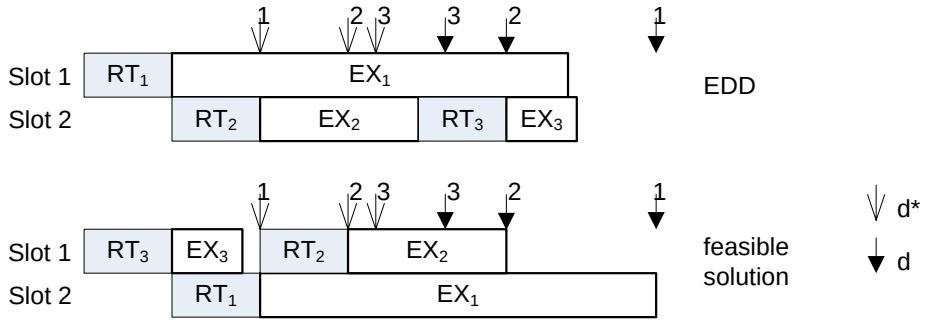

Figure 5.8: EDD can fail to produce a feasible schedule.

After each update of the vector, we can prove the feasibility ($z_{r,\text{new}} \leq d_j$) and determine the next slot ($r = \text{ind}\left(\min\left\{z_1, z_2, \ldots, z_m\right\}\right)$) and the time instance for reconfiguration ($\min\left\{z_1, z_2, \ldots, z_m\right\}$).

Limitations

As stated above, when using EDD for the reconfigurable slot scheduling of reconfigurable architectures, we cannot guarantee optimality. In fact, EDD can miss to produce a feasible schedule. Figure 5.8 shows the problem, which is due to the possible additional δ_i of each task. We can also see that we have to dissociate from the statement that EDD also reduces the maximum lateness in our reconfigurable environment.

To summarize, using EDD, we can guarantee the minimization of the maximum lateness only if no reconfiguration phase is delayed.

5.3.2 Asynchronous Arrival

We now release the restriction of synchronous arrival of all tasks—tasks may dynamically enter the system. If we have such arbitrary arrival times, preemption becomes an important factor. In the literature, we find that when preemption is not allowed, the problem of minimizing the maximum lateness and the problem of finding a feasible schedule become NP-hard [LRKB77, LRK77, KIM78]. If preemption is allowed, Horn [Hor74] found an algorithm, called *Earliest Deadline First* (EDF), that minimizes the maximum lateness. The algorithm dispatches at any instance the task with the earliest absolute deadline. In particular if a newly arriving task has a shorter deadline than the currently executing task, the new task must be preferred and the currently executing must be interrupted. Thus, preemption is required to schedule asynchronously arriving jobs.

As stated above, we preempt tasks during their RT phase, when the calculation has not started and no context saving, etc. is necessary. Figure 5.9 depicts an example of three tasks executing in three slots. In order to implement the preemption, we divide the area reconfigured during a RT phase into columns c_j. These columns are of equal size and comprise the equal reconfiguration time Δ, which sums up to the reconfiguration time of the whole slot: $\sum \Delta = t_{RT}$. The reconfiguration process then looks as follows:

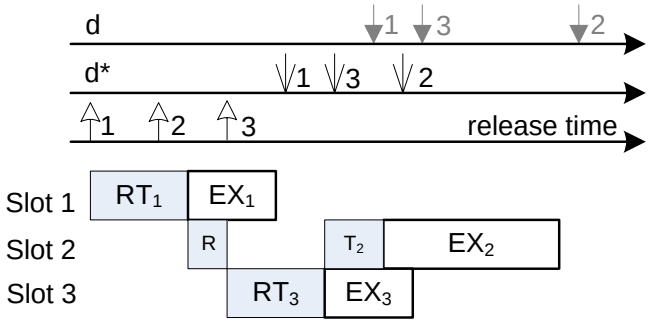

Figure 5.9: EDF schedule on a three slot machine using preemption.

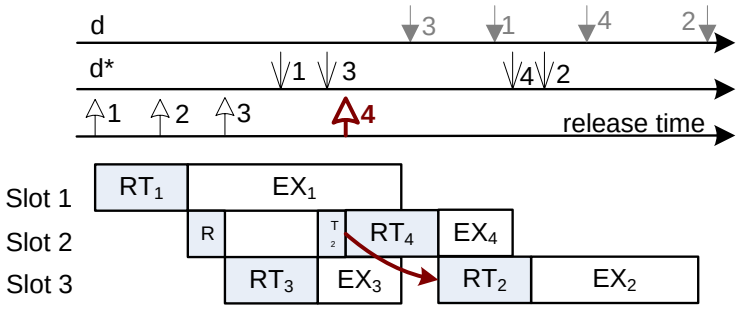

Figure 5.10: Full reconfiguration capacity of an EDF schedule

gradually all the c_j of task τ_v are loaded in slot s_v of the reconfigurable fabric. If a new task τ_w enters the system and has an earlier deadline d^*_w, we preempt task τ_v—task τ_v frees the reconfiguration port and task τ_w starts to reconfigure.

Depending on the current occupation of the fabric, different scenarios for the slot assignment of task τ_w are possible. (1) If we have another free slot available ($s_{\text{free}} \neq s_v$), we use this slot. After the RT phase of τ_w we can resume the RT phase of τ_v at the interrupted point. (2) If no free slot is available, we can use slot s_v of the interrupted task. s_v becomes the slot for τ_w ($s_w \Leftarrow s_v$) and RT_w overwrites all already configured parts of τ_v. Thus, after finishing the reconfiguration of τ_w, we cannot resume the reconfiguration phase (RT_v) of the preempted task. Instead, we have to restart RT_v completely, as already loaded parts of the bitstream are lost.

When we thus apply EDF to our scenario relying on d^*, in addition to the *fls* condition, which occurs here as well, we experience the *full reconfiguration capacity* (*frc*) condition (ref. to Fig. 5.10). Here, at least one slot is in RT phase, while all slots are occupied (by either RT or EX). As explained above, we have two possibilities when a new high priority job arrives and the *frc* scenario holds. We either can delay the job similar to the *fls* condition, or we can force a preemption of one of the slots currently in RT mode. Such a preemption will result in the killing of the preempted job, as all already reconfigured parts of this task are abandoned in favor of the high priority task. The mono-processor EDF does not include such a situation.

Algorithm 3 Earliest Deadline First for Reconfigurable Slot Architectures

```
1: if d*_new < d*_current then
2:     if all slots are in EX then
3:         wait for next free slot
4:     else if all other slots s_i ≠ slot(τ_current) are in EX then
5:         add τ_current completely to Q
6:         reconfigure now free slot
7:     else
8:         add rest of τ_current to Q
9:         Reconfigure next free slot
10:    end if
11: else
12:    Insert τ_new in queue Q
13: end if
```

Implementation of EDF

The implementation of EDF (refer to Algorithm 3) for our scenario bases on a queue Q, which orders all tasks according to their deadlines d^*. As mentioned above, if a new task dynamically arrives to the system and its deadline is smaller than the task currently in RT phase ($\tau_{current}$), we start the preemption process. Note that we put $\tau_{current}$ at the head of the queue. Depending on the slot we use to reconfigure, we either mark $\tau_{current}$ as partly loaded and assign the rest of its reconfiguration time to the queue, or, in the case of $s_w \Leftarrow s_v$, we put $\tau_{current}$ and its complete t_{RT} to the head of the queue.

Limitations

EDF in the mono-processor domain minimizes the maximum lateness. Similar to EDD, applying EDF for the reconfigurable port scheduling of reconfigurable environments, we cannot guarantee this minimization. Again, if all slots are in EX phase, EDF cannot load a dynamically arriving task as executing tasks are assumed to be non-preemptive. As stated above, we deal with an NP-hard scenario in such a case.

Furthermore, Fig. 5.10 displays the major limitation where a RT phase might have to be restarted completely. This killing of a RT phase most often increases the overall response time and enforces a complex scheduling test to be done online after each new task has entered the system.

Often, for an optimal schedule, we would require the reconfiguration port to stay idle despite a task pending. Obviously, no scheduling algorithm can predict whether to schedule this task or to delay its reconfiguration phase. However, for the case of non-idle scheduling algorithms (the algorithm does not permit the mono-processor to become idle if there are active jobs), [JSM93] proves that EDF is still optimal (minimizing the maximal lateness) in a non-preemptive task model. If we thus schedule a task set that only suffers the *frc* conditions, and do not allow killing of tasks (hence not preempting the task), we have similar conditions to those of [JSM93].

5.3.3 Experimental Results

We have conducted several test sets in order to rate the performance of EDD and EDF in our scenario. We have set up a reconfiguration port simulator that allows us to randomly generate task sets, schedule them according to various algorithms, respecting different priorities, and display the schedules for visual control. We have randomly generated 1000 task sets with varying parameters. For the duration of the execution and reconfiguration phases, we have used values depending on the ratio l, with $\frac{t_{EX}}{t_{RT}} \approx l$, and $l = .25, .5, 1, 2, 3, 4$. The number of slots has been in the range of 3 to 5, while the number of tasks always was significantly higher, so that tasks have to share slots. We have scheduled both EDD and EDF with d^* and d as deadlines.

Concerning the schedulability of EDD, we find a schedulability of approx. 90 % for d^* and 70 % for d among the schedulable task sets. However, the schedulability concerning d^* significantly increases with $l \ll 1$, and decreases with $l \gg 1$, while using d as ordering deadline of EDD behaves oppositional. Obviously, this is due to a dominance of the EX phase. The schedulable task sets not found are due to the *fls* condition. When EDF is applied, both conditions (*fls* and *frc*) can occur. Thus, the performance is slightly worse, again with d^* outperforming d. Finally, for both EDD and EDF the approach performs best if $t_{EX} \leq t_{RT} \cdot (m-1)$, refer to Eq. 5.4.

5.4 Fixed Priority Periodic Task Scheduling

Periodic activities (sensory data acquisition, control loops, etc.) often represent the major computational demand of embedded systems. In real-time scheduling theory priorities are principally applied to such jobs. Contention for resources is resolved in favor of the job with the higher priority that is ready to run.

Our tasks have the relative deadlines D_i equal to their periods T_i. However, as we schedule the RT phases, we derive the priorities of our jobs by referring to the relative deadline D_i^*. The task with the shortest D_i^* gets the highest priority. We thus have to schedule a set of periodic tasks with deadlines less than periods. A similar mono-processor scheduling algorithm is denoted deadline monotonic (DM) [LW82]. It is an extension of the more common rate monotonic scheduling scheme. According to the DM algorithm, each task is assigned a priority inversely proportional to its relative deadline. Thus, at any instant, the task with the shortest relative deadline is executed. Figure 5.11 shows an example.

5.4.1 Schedulability Analysis

The sufficient and necessary schedulability test of a DM algorithm can be done by the response time analysis [ABR+93, ABRW91], with the longest response time R_i computed at the critical instance as the sum of its computation time and the interference I_i due

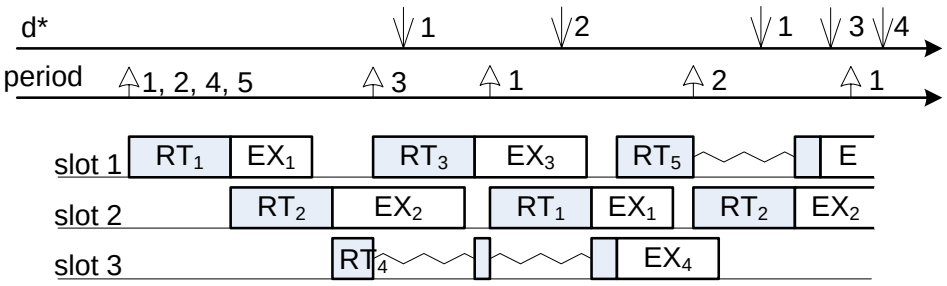

Figure 5.11: Fixed priority example.

to preemption by higher-priority tasks:

$$R_i = t_{RT,i} + I_i, \qquad \text{where} \quad I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil t_{RT,j}. \tag{5.8}$$

If $R_i < D_i^*$ for all tasks, the set is schedulable. We can derive step-wise solutions for this problem. The critical instance occurs when all tasks are released simultaneously.

When scheduling the reconfiguration port, however, we face several circumstances where the response time analysis would not produce the correct result. Both conditions presented above, the *fls* and the *frc* scenarios demand special care, as they both can impose additional delays to the scheduling of the jobs—the increase of I_i. Both, *fls* and *frc* can hardly be calculated in advance. Thus, we propose alternative strategies to guarantee the schedulability.

One possibility is to schedule the hyper-period of a task set. The hyper-period equals the least common multiplier of all periods. Schedulability of one hyper-period guarantees that subsequent hyper-periods can also be scheduled—the whole task set is schedulable. Thereby, we can solve the question of schedulability, as the *fls* and *frc* conditions will occur equally in all hyper-periods. However, there might exist cases, were a hyper period does only exist asymptotically and therefore is too large to be constructed in advance.

As second solution, we handle the *fls* and *frc* conditions using known and appropriate techniques of real-time scheduling that harmonize with the standard DM response time analysis. We derive the two suitable techniques in the following.

5.4.2 A Server for Full Load of Slots Sections

During the *fls* condition, all slots are in *EX* phase while a new instance of a task τ_i should be scheduled. τ_i, independent of its priority, demands for the reconfiguration port. Despite the availability of the port, but due to the occupancy of the slots, we cannot schedule this task. It is delayed.

In order to respect such a delay, we rely on the notions of aperiodic job scheduling. Aperiodic jobs are scheduled by virtue of servers in the mono-processor scheduling domain. In our scenario, we assume an aperiodic job τ_a arriving at the same time as the

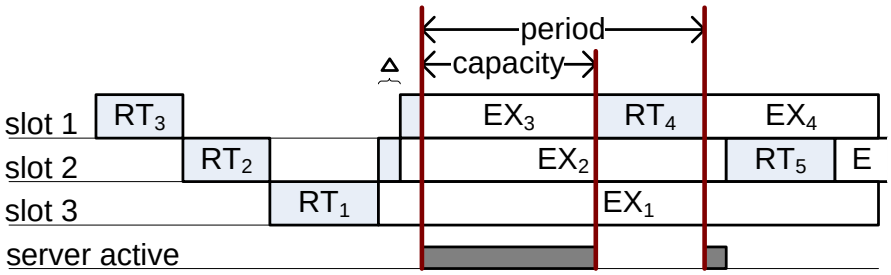

Figure 5.12: Server for *fls*: Worst Case.

new instance of job τ_i arrives. τ_a stands for the delay of τ_i and has the corresponding computation demand (t_a = delay of τ_i). For scheduling τ_a, we use a server, which must have both, enough capacity to serve τ_a completely and a higher priority than τ_i to become active immediately. If we find such a server, and can integrate it into the schedulability analysis of our scenario, we can guarantee the handling of *fls* conditions. Please note that the tasks to be scheduled by the server do not become physically implemented parts of the system. In contrast, the tasks are only virtually executed by the server. The server therefore may also be considered to execute virtual, however inevitable for proper scheduling of the whole task set.

To derive a suitable server, which guarantees the immediate scheduling of τ_a, we have to consider several characteristics discussed in the following. Basically, the computation demand of τ_a depends on the current allocation of the slots, which is difficult to be computed in advance, as we would have to investigate all possible *fls* conditions for a task set. For our scenario, it is however not important to know the particular computation demand. It is enough, if we know the maximum computation demand, which then becomes the capacity of the server (see below). Every time the server becomes active, we only have to guarantee that there is enough capacity to activate the server for the possible longest delay of τ_i, so that the starting time of τ_i can be bridged.

As the server must always execute in favor of any other job (forced by *fls*), we assign the highest priority to the server. The minimal relative deadline of the server therefore can be calculated by

$$D_{S,\min} = \min(D_i^*) - \Delta. \tag{5.9}$$

Thus, its deadline will be shorter (by the smallest possible value Δ) than all other deadlines D_i^* of the tasks τ_i. Hence, the priority of the server is maximized. As the server does not have an *EX* phase, its deadline is equal to its period and $D_S^* = D_S$. In order to calculate the exact deadline of the server, we have to know the capacity.

In Fig. 5.12, we display the worst case *fls* scenario. It occurs when all slots have started with the *EX* phases as close as possible. Note that a simultaneous activation of *EX* phases in all slots is not possible, as for such an activation all slots would have to finish reconfiguration simultaneously. However, the mutually exclusive reconfiguration port prevents simultaneous finishing of *RT* phases. We thus only have a nearly simultaneous activation of *EX* phases, which occurs when the *RT* phases of m tasks end nearly

synchronously, as depicted in Fig. 5.12. The smallest interval I of starting times of tasks nearly simultaneously activated sums up to $(m-1)\cdot\Delta$.

Based on the worst case scenario given by the nearly simultaneous activation depicted in Fig. 5.12, we can compute the capacity of the server. In detail, the server must have enough capacity C_S to schedule an aperiodic job that has the computation requirement of the m-th longest t_{EX}, thus

$$C_S = m\text{-th}\max(t_{EX}). \tag{5.10}$$

Only if this task τ_m having the m-th $\max(t_{EX})$ is scheduled in the latest activated slot m_{latest}, the complete capacity equaling t_{EX} of this task will be required. No larger capacity is required, as firstly all tasks having a shorter t_{EX} than τ_m result in a shorter capacity required if scheduled in m_{latest}. Secondly, if a task having a longer t_{EX} than τ_m is scheduled in m_{latest}, we assign τ_m (or a task with shorter t_{EX}) to one of the other slots, degrading C_S required at least by Δ.

Moreover, we can improve the server capacity, if we have a large Δ compared to the EX phases. If the largest t_{EX} is smaller than $(m-1)\Delta$, the server capacity becomes 0, as the task can be executed completely during the interval $I = (m-1)\cdot\Delta$, thus at least one slot is free when the simultaneous activation phase ends. This holds for the k-th largest t_{EX} and $t_{EX,k} < (m-k)\Delta$, while $k < m$.

As the capacity must always be available, we require a server that preserves its capacity. Furthermore, as two *fls* conditions could occur successively, only separated by one RT phase, the capacity of the server must always be replenished as soon as possible. The *Sporadic Server* (SS) [SSL89] solves our requirements. The SS algorithm creates a high-priority task for servicing aperiodic requests and preserves the server capacity at its high-priority level until an aperiodic request occurs. SS replenishes its capacity only after it has been consumed by aperiodic task execution.

Finally, the period (= relative deadline) of the server is the minimum of the above introduced $D_{S,\min}$, and the capacity + RT phase, i. e., $C_S + t_{RT}$. Thus, the server will have the highest priority among all other tasks. Furthermore, the feature of the SS allows us to have always enough capacity available, as even partially consumed capacity is replenished after the server's period. This is always early enough, as between two consecutive occurrences of *fls* conditions always a complete RT phase will be scheduled.

To conclude, we use the Sporadic Server based on the parameters discussed above to serve the delay of tasks that request the reconfiguration port, but cannot be reconfigured due to the *fls* condition. The server—referring to its parameters—is included in the schedulability test to guarantee a proper handling of *fls* scenarios. Combined with the resource access protocol for *frc* sections discussed next, we then finally derive a schedulability test to guarantee real-time execution of the task set.

5.4.3 Resource Access Protocol for Full Reconfiguration Capacity Sections

When the *frc* condition occurs, all slots are occupied and at least one slot is in RT phase. If the newly arriving instance of a task τ_i has lower priority than the just reconfiguring

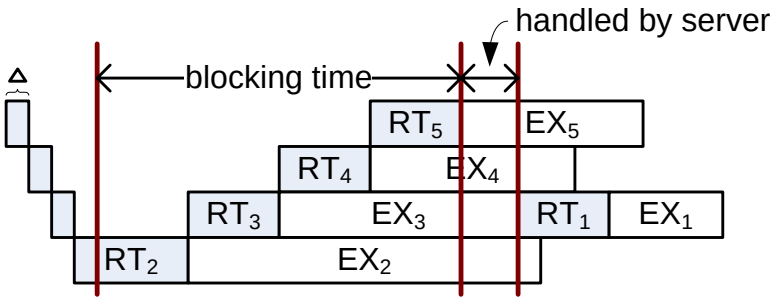

Figure 5.13: Blocking Time for *frc*: Worst Case.

task τ_j, nothing will happen and τ_i will be sorted into the list of ready tasks. However, if the priority is higher, τ_i could either be scheduled and τ_j would be killed, or delayed until the reconfiguration port is free again.

For our schedulability analysis, we disallow killing as it harms the assumptions of DM and would complicate the computation of the interference time I_j. Thus, we delay τ_i—higher priority tasks will suffer a blocking due to lower priority tasks, as the reconfiguration port is occupied. This is similar to a critical section of resource sharing. In order to avoid (unbounded) priority inversion, a resource access protocol is necessary. As we only face direct blocking and will not suffer chained blocking or deadlocks [But04], we can apply the *Priority Inheritance Protocol* PIP [SRL90]. The protocol modifies the priorities of those tasks that cause blocking. In our case, τ_j would temporarily inherit the priority of τ_i.

The schedulability analysis of the PIP is based on the response time analysis. Therefore, the blocking time B_i is added to the recurrent equation:

$$R_i = t_{RT,i} + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil t_{RT,j}. \tag{5.11}$$

Note that this test becomes only sufficient, as tasks could actually never experience blocking. In order to calculate the blocking time B_i, we rely on the worst case blocking time as displayed in Fig. 5.13. The worst case occurs when a high priority task suffers a blocking due to m lower priority tasks in their RT phase. Thus, the blocking time will not be longer than $B_{max} = (t_{RT} - \Delta)(m - p)$, while $p = m$ for the lowest priority task and decreases by 1 with every priority increase until $p = 0$.

We notice that the blocking time will be large for a large m. However, B_i will never be longer than the largest $t_{EX,j} + t_{RT} - \Delta$ among all tasks with lower priority than τ_i. In fact, the k-th longest EX phase among the tasks τ_j added to a non-avoidable fraction of $k(t_{RT} - \Delta)$ will denote B_i, if it is smaller than B_{max}. We calculate B_i by Algorithm 4.

Algorithm 4 Computation of the Blocking Time

```
1: L ⇐ Γ
2: L̄ ⇐ ∅
3: p ⇐ m
4: while L ≠ ∅ do
5:     τ_i ⇐remove_lowest_priority_task(L)
6:     B_i ⇐ (t_RT − Δ)(m − p)
7:     if p > 0 then p ⇐ p − 1
8:     for (k ⇐ 1; k < (m − 1) ∧ k < number_of τ_j in L̄; k ⇐ k + 1) do
9:         B_tmp ⇐ k-th longest t_EX,j + k(t_RT − Δ)
10:        if B_tmp < B_i then B_i ⇐ B_tmp
11:    end for
12:    L̄ ⇐ L̄ ∩ τ_i
13: end while
```

5.4.4 DM + SS + PIP Schedulability Test

For the schedulability test, we thus have to combine the response time analysis for DM with the PIP and the server. From a scheduling point of view, SS can be replaced by a periodic task having the same utilization factor. As our server will never come active during a *frc* condition, the server does not have a blocking time ($B_S = 0$). Thus, we have to solve the recurrent equation

$$R_i = t_{RT} + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil t_{RT} \tag{5.12}$$

for the task set $\Gamma' := \Gamma \cup \tau_S$.

5.4.5 Experiment

We have randomly generated periodic task sets and tested their schedulability using our simulator. Fig. 5.14 shows an example of seven tasks executed in three slots. The server becomes active, when the *fls* condition arises, depicted in the additional and virtual server slot. By explicitly displaying the server, we can evaluate the performance of the scheduling. Based on the parameters of our randomly generated task sets, we first defined the capacity and period of the server, in order to add the server to our task set ($\Gamma' := \Gamma \cup \tau_S$). Then, we performed the feasibility test of the previous subsection. By means of our simulator, we executed task sets that passed the feasibility test, as well as task sets that failed, and displayed the schedules for further analysis.

In general, our approach using the PIP access protocol and the SS server on one hand correctly guarantees the schedulability of the task sets. On the other hand, however, the approach is pessimistic, as neither the server capacity, nor the blocking time are consumed completely among the majority of our task sets. Thus, when scheduling task sets that are not feasible according to our test, but according to the standard DM response

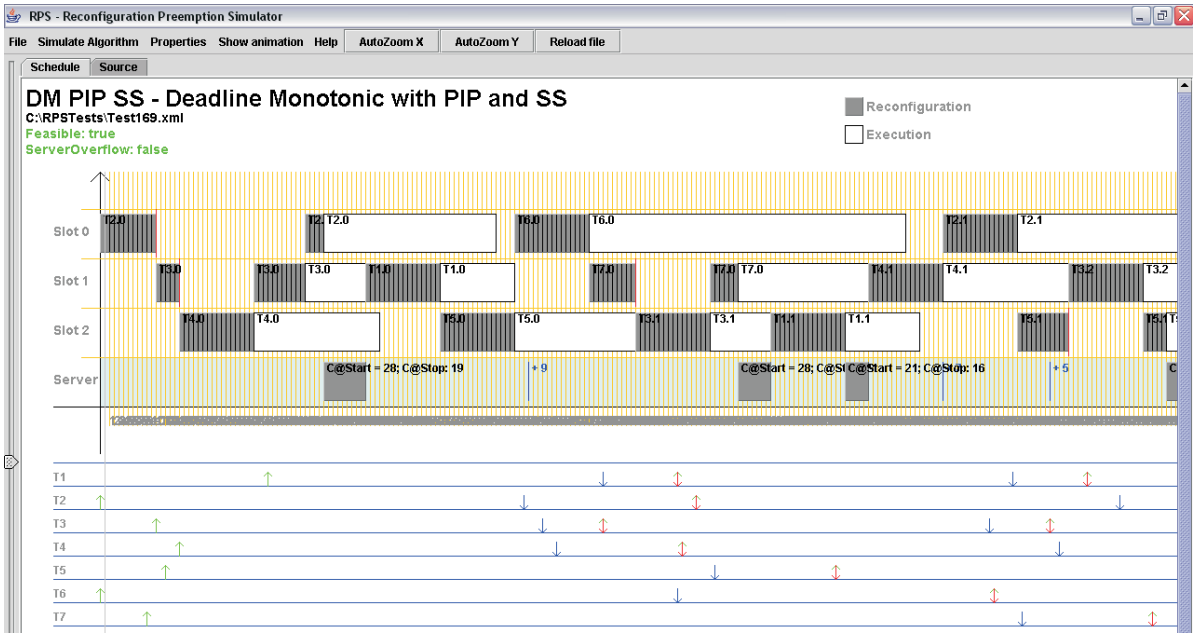

Figure 5.14: Simulator

time analysis, we still often could derive feasible results within the reconfiguration port scenario. The same empirical results from the aperiodic task sets hold: our approach performs best if $t_{EX} \leq t_{RT} \cdot (m-1)$. Furthermore, there exist few cases, where killing of tasks can increase the performance of the schedule (reducing the max. lateness) or even result in feasibility.

For thus increasing the number of periodic task sets that can be accepted for the reconfiguration port scheduling, the pre-calculation of the hyperperiod is advisable. Nevertheless, our approach of using the SS server and the PIP access protocol provides a comfortable method for providing a schedulability test that can be completely calculated offline having also a low computational complexity. In particular, the calculation of the hyperperiod can be avoided, which is beneficial, if the number of tasks is high and/or the hyperperiod is very long.

5.5 Caching

In this section, we introduce caching concepts to the scheduling of the reconfiguration port of partially reconfigurable systems. Being a powerful concept for accelerating processing in multiple execution environments, caching can also help to increase the performance of the reconfiguration port scheduling. Basically, by caching configurations, we reduce the reconfiguration overhead. We present several caching concepts in the following, including the evaluation and test of the algorithms [Fra06], [DF07a].

Caching can be applied in case of periodic systems—the RT phase between two consecutive instances of a task τ_i is omitted. We thereby have to take care of no other task starting a reconfiguration in this slot between the occurrence of the two instances of τ_i. If however the slot cannot be used otherwise, we might affect the performance of the other tasks, as their reconfiguration request could not be granted.

Therefore, we apply caching based on the current load of the execution environment, as well as on global statements like the frequency of a task. For example, two consecutive instances that are executed as late as possible ($\tau_{i,k}$) and as soon as possible ($\tau_{i,k+1}$) most often can share the same slot without idling time of this slot between the two instances. We then can avoid the intermediate reconfiguration, which belongs to $\tau_{i,k+1}$ and is located seamlessly in between the two EX phases. We start the EX phase of $\tau_{i,k+1}$ earlier, improving the response time of the task. By re-using an already loaded configuration (caching), we thus can improve the overall performance. Thereby, if a task is more often activated—in fixed priority scheduling systems it will have a higher priority—we indicate it as preferable for caching.

Problem Abstraction

To summarize our problem: We want to reduce the amount of reconfigurations when scheduling a task set Γ of n periodic tasks τ_i onto our execution environment of m equally sized slots with $m < n$ in general. Thereby, however, caching means no RT phase. We have to bear in mind that by avoiding the RT phase, we undermine the concept of reconfiguration port scheduling, as the instance when the caching pays off, is an instance without t_{RT}: $t_{RT} = 0$.

Often caching helps to improve the schedulability and/or the maximum lateness. We are interested in which methods provide the most promising solutions. Caching can even increase the feasibility of task sets. Thereby, we will also improve additional design goals like energy consumption, area consumption, resources needed, etc. For example, having fewer RT phases basically reduces the energy consumption.

Note that caching in our scenario is different to caching in von Neumann architectures, as we do not cache data, but we cache configurations. Thus, while caching is often disabled in real-time scenarios, as it complicates the predictability, caching can be of benefit for our algorithms of real-time scheduling the reconfiguration port.

Our caching algorithms are based on the priority of the tasks. We define the priority based on different characteristics of a task:

$$p_a = \frac{t_{RT}}{T} \tag{5.13}$$

$$p_b = \frac{t_{EX}}{T} \tag{5.14}$$

$$p_c = \frac{t_{EX} + T_{RT}}{T} \tag{5.15}$$

$$p_d = d^* \tag{5.16}$$

The first definition p_a is the same as in the classical mono-processor scheduling domain, where the calculation time c_i of a task divided by the period T_i stands for the

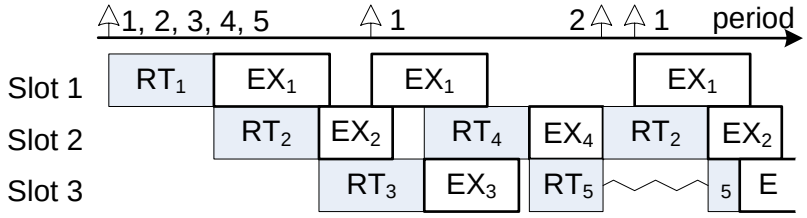

Figure 5.15: Caching: Priority Based Slot Reservation where one slot is constantly used for task τ_1.

priority. In contrast, p_b and p_c, along with p_a are defined for the purpose of static priority reconfiguration port scheduling, while p_d is suitable for dynamic priority assignment.

5.5.1 Offline caching methods

We will introduce offline caching methods first, before considering on-line methods in Sect. 5.5.2. The algorithms apply static (fixed priority) as well as dynamic scheduling.

Priority Based Slot Reservation

If we can completely avoid the reconfiguration of a slot, we will have the maximum gain concerning the saved configurations, as only the initial configuration must be loaded. Our first concept presented—the *priority based slot reservation*—follows such a goal by statically analyzing the task set prior to the dispatching of the tasks to the slots.

The applicability of the method depends on the number of slots available and on the characteristic of the task set. We prefer tasks with high priority, as those tasks get activated frequently. We reserve one or more slots exclusively for a corresponding amount of high priority tasks. The reserved slots are configured at set-up time and are from there on exclusively used each for their task only. The other tasks have to share the remaining slots. Figure 5.15 shows an example.

The selection of the tasks that exclusively occupy a slot is primarily based on the priority of the task. We can use either of the priorities p_a, p_b, or p_c, as all show good results. However, p_c particularly helps to indicate tasks that are worth to be cached. If the priority p_c is close to 1, this task would require nearly a slot on its own anyway, as the reconfiguration phase followed by the execution phase of this task would reside in a slot during the period of the task.

Among the tasks having a high p_c, we prefer the tasks with a high p_b, as such a task will exploit the exclusively reserved slot longer than a task with a lower p_b. Furthermore, such a task would block one of the slots for the reconfiguration port scheduling for a disproportional long time, as the ratio of the reconfiguration phase t_{RT} within a period of such a task is very high.

To schedule the whole task set, we always require enough slots to be able to serve all tasks. In particular, if not all tasks are assigned to their exclusive slot (as is the case with $m < n$), there must always remain at least one slot that is not exclusively reserved for a task. Having only one slot left for reconfiguration inevitably means that hiding the

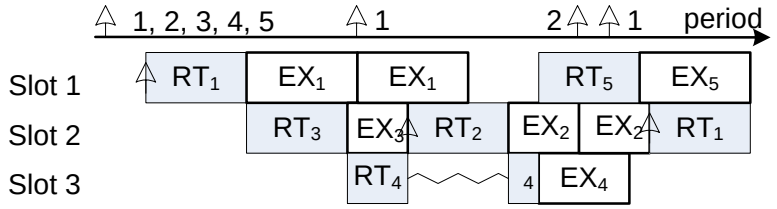

Figure 5.16: Consecutive Task Combination with modification of the release times of tasks τ_1 and τ_2.

reconfiguration latency cannot be applied any more. Therefore, we try to have at least two slots remaining.

Furthermore, concerning the set-up of the system, we have two possibilities to consider the configuration requirements of the fixed assigned tasks. Either, we denote the sum of the configuration times for these tasks as additional requirement that precedes the scheduling, or we include the configuration time into the calculation of the schedulability. The former eases to find a feasible schedule and will serve practical conditions in most cases, while the latter also allows us to consider systems without set-up time. However, the latter could extremely deform the system, if a large number of slots is exclusively reserved, as the reservation—the configuration—will only occur once.

If we consider the initial configuration of the exclusive slots within our feasibility analysis, we apply the earliest due date algorithm (see Sect. 5.3.1), as we first have to schedule a set of aperiodic tasks that have simultaneous arrival times. We then use a periodic scheduling algorithm for the remaining tasks.

Concerning the feasibility analysis of the priority based slot reservation, we thus have to first guarantee that the initialization phase can be completed without missing deadlines. Thereafter the remaining tasks have to be scheduled. Here, we can apply the method of Sect. 5.4 or schedule the task set for the hyper period. For the latter case, we schedule the task set until the first hyper-period of the whole task set is reached, or until the hyper-period of the new task set is reached, depending on whether the initial configuration of the exclusive tasks is part of the scheduling analysis or whether a set-up time is used. In both cases, the hyper-period is easier to schedule, as only the free tasks require reconfiguration, thus reducing the occupancy of the reconfiguration port.

The advantage of the priority based slot reservation scheduling algorithm can be summarized as follows: As selected tasks and/or tasks with high priorities are assigned to exclusive slots, these tasks will always meet their deadline, if their initial configuration can be guaranteed.

Consecutive Task Combination

The second offline caching method—the *consecutive task combination*—bases on the manipulation of the characteristics of the task set. If we combine two instance of a task and execute the *EX* phase of the first instance as late as possible, we can avoid the *RT* phase for the next instance. The finishing time of the first instance thereby will be at the end of the task's period, see Fig. 5.16.

If we want to combine two instances of a task τ_i, we will have to ensure that no other task τ_j will reconfigure the slot in the meantime. We therefore adapt the release time of the first instance and the deadline of the second instance. The new release time $\tilde{r}$ of task i and instance k can be calculated by $\tilde{r}_{i,k} = k \cdot T_i - (t_{RT} + t_{EX,i})$. Similar, the new absolute deadline $\tilde{d}$ for the task i and instance $k+1$ is $\tilde{d}_{i,k+1} = r_{i,k+1} + t_{EX,i}$. Moreover, for instance $k+1$, we do not have to consider the task τ_i for the reconfiguration port scheduling, as it does not require an RT phase. Thus, the relevant deadline is the deadline $d^*_{i,k}$ of the instance $\tau_{i,k}$.

Here, we denote the drawback of the approach: As the release time is pushed to the furthest possible point in time, while the deadline remains the same, there is no slack available for the modified tasks. Thus, these tasks must be scheduled as soon as they are released in order to enable the matching of the deadline. We can guarantee an immediate reconfiguration only for the task with the highest priority and if at least one slot is free when this task arrives (no *fls* condition holds).

Alternatively, we can allow the tasks some slack by releasing the task prior to the new release time $\tilde{r}$. Then, we have to block the slot after finishing the execution phase of instance k until the beginning of the instance $k+1$. Such a modification can be accepted if the blocking time is smaller than the t_{RT}, as due to the avoidance of the RT phase for the instance $k+1$, a complete t_{RT} is saved. In general, we can approach the problem by using a resource access protocol, which prevents other tasks also having higher priority from using the blocked slot for reconfiguration.

In Fig. 5.16, we depict an example where only some of the tasks are combined in order to facilitate consecutive task combination. Not modifying all tasks makes particularly sense, if a scenario arises where two or more combined tasks compete for the reconfiguration port at the same time. If we allow no slack for combined tasks, only one task could meet its deadline. We therefore select some tasks for the consecutive task combination, while leaving other tasks untouched.

The selection can be done on the basis of the absolute deadlines d^* of the instances of all tasks during a hyperperiod. If two or more of such deadlines fall into an interval of $t_{RT} - \Delta$ length, their corresponding tasks might not be schedulable. This becomes clear, if we take a closer look at the modified release times of two tasks τ_v and τ_w, whose instances are combined pairwise. If say the 3rd deadline $d^*_{v,3}$ of task τ_v and the 5th deadline $d^*_{w,5}$ of task τ_w are located within an interval $t_{RT} - \Delta$, the modified release times also fall into such an interval. The modified deadline for τ_v can be calculated by $\tilde{r}_{v,3} = 3 \cdot T_v - (t_{RT} + t_{EX,v}) = d^*_{v,3} - t_{RT}$, and for τ_w by $\tilde{r}_{w,5} = 5 \cdot T_w - (t_{RT} + t_{EX,w}) = d^*_{w,5} - t_{RT}$. If $\left|d^*_{v,3} - d^*_{w,5}\right| < t_{RT} - \Delta$, then it holds that $|\tilde{r}_{v,3} - \tilde{r}_{w,5}| < t_{RT} - \Delta$, and thus one of the task misses its deadline, as they compete for the reconfiguration port at the same time interval.

For deriving the critical deadlines, we have to consider only every second deadline $\mathfrak{D}^* = \left\{d^*_{i,1}, d^*_{i,3}, d^*_{i,5}, \dots\right\}$ of each task within a hyperperiod, as only those deadlines are relevant for scheduling tasks whose instances shall be combined pairwise. We then compare these deadlines $\mathfrak{D}^*$ of all tasks within a hyperperiod and derive a selection of tasks by building a graph. If two deadlines of two different tasks fall in the range

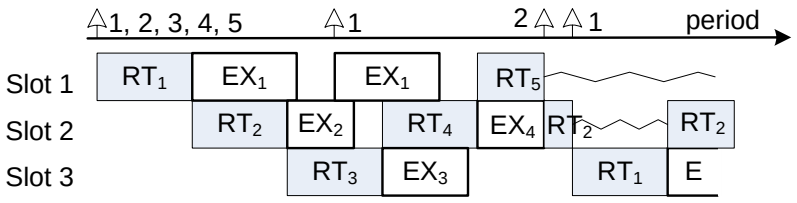

Figure 5.17: Scheduling Look Back

$t_{RT} - \Delta$, we draw a connection between the two tasks. We repeat this for all critical pairs of deadlines from $\mathfrak{D}^*$. From the final graph we then can derive the tasks whose instances can be combined pairwise, by selecting those tasks that have no direct connection. The selection is done on the basis of graph coloring. The number of colors tells us how many tasks can be scheduled according to the consecutive tasks combination. To derive the maximum number of tasks, we select the color that occurs most often within the graph.

Please note that an uneven number of instances of a task within a hyperperiod must be handled also. We can either exclude the last instance of these tasks from the consecutive task combination, or double the hyperperiod, as then all tasks have an even number of instances. Otherwise the last instance of a task within a hyperperiod would be combined with the first instance of the next hyperperiod, making this hyperperiod different to the former one.

In summary, the advantage of the approach lies in the reduction of the number of reconfiguration phases required by two for those tasks whose instances are combined. Tasks with short periods may be preferred, as their overall fraction of the reconfiguration time is larger than of those tasks with longer periods. Furthermore, the approach can improve the scheduling of tasks when for the priority p_c holds $p_c \leq \frac{1}{2}$, without permanently reserving slots for specific tasks as in the previous approach.

5.5.2 Dynamic/On-line Caching Methods

While the first two algorithms manipulated the task set offline prior to starting the schedule, the remaining algorithms will focus on online conditions.

Scheduling Look Back

The first approach, *scheduling look back* (SLB), considers the condition of the slots before starting an instance of an *RT* phase. If any of the slots already holds the configuration of the newly arriving task, the *RT* phase is omitted in favor of immediately starting the *EX* phase. Figure 5.17 depicts an example, where the second reconfiguration of the frequently occurring task τ_1 can be avoided.

Additionally to tasks with short periods, we improve the schedulability of tasks that are started relatively late, as their next invocation will be not too far away from the finishing time. These tasks often have low priorities. However, despite that the next release of the task will be soon, the reconfiguration of the task is often postponed, as the priority is too low. Here, as an improvement of the algorithm, we can temporarily

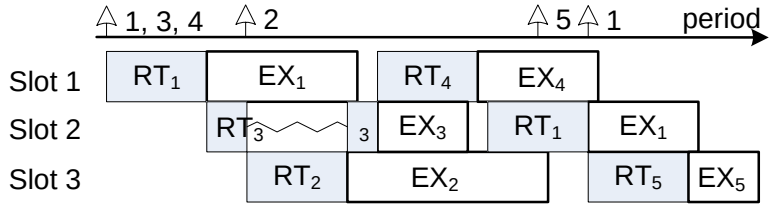

Figure 5.18: Scheduling Look Ahead

inherit the priority of a higher priority task. Such a behavior can be controlled by a resource access protocol.

Moreover, we do not have to pre-compute the task set. If we have a task set that is feasible without scheduling look back, the performance of the task set will be improved by applying SLB.

Scheduling Look Ahead

A different approach is *scheduling look ahead* (SLA). As implied by the name, it looks into the future, opposed to SLB, which only refers to the current status and former allocation of the slots. SLA becomes active each time the reconfiguration port idles. If we find a task τ_i whose next instance k will be activated in the near future, we start the reconfiguration phase for instance k at the moment the reconfiguration port idles. If we can finish the reconfiguration phase before the original release time, the next instance of task τ_i starts to execute the moment its period begins, without having to wait for its reconfiguration phase to finish. Even if the reconfiguration is not completely finished when the period begins, the actual start of the execution time still will be earlier. We thus improve the response time of instance k of this task.

Moreover, the requests of the reconfiguration port are spread over the time line, therefore reducing the cases of two tasks competing for the reconfiguration port at the same time, which in turn reduces the number of *frc* conditions. Thus, the performance of the scheduling can be improved. Figure 5.18 depicts an example.

SLB relies on several characteristics of the task set. As we know the whole task set from the beginning of the scheduling, we particularly know when a new instance of a task is released. Moreover, as the *EX* phase of a task can start at the beginning of the period, we can start with the *RT* phase before the final release of the task.

The pre-loading can only take place if (1) the reconfiguration port is free and (2) the time until the release time of the instance of the task selected is less or equal the reconfiguration time of this task. Otherwise, we might reconfigure a task τ_i, mark it as already reconfigured (so that it can be executed immediately when the next instance is released), and another task τ_j uses this slot for reconfiguration before we have executed task τ_i.

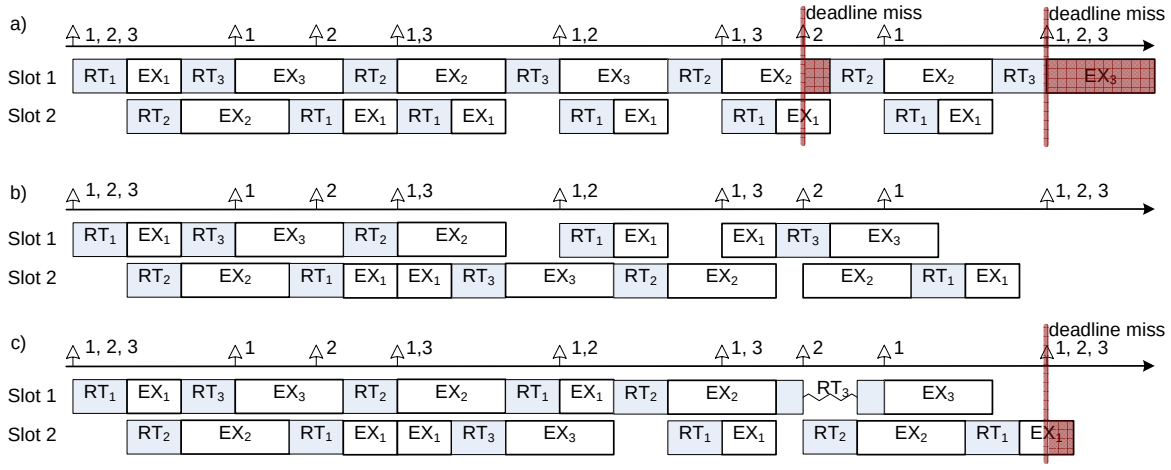

Figure 5.19: Example of the same task set on a two-slotted architecture applying different scheduling algorithms. a) DM (*deadline monotonic*), b) DM + SLB (*scheduling look back*), c) DM + SLB + SLA (*scheduling look ahead*)

5.5.3 Combination of the Methods

Again, we have used our reconfiguration port scheduling simulator to evaluate the algorithms. Thereby, the simulator helps in many aspects, as it visualizes the schedules. In particular, non-feasible tasks can be found. Additionally, it allows us to rate task sets by performing the scheduling algorithms on a set of task sets. Finally, we can also evaluate task sets by looking at the graphically displayed hyper-periods.

Figure 5.19 depicts an interesting example of applying caching in the realm of reconfiguration port scheduling. In Fig. 5.19 a) we have scheduled the task set displayed in Table 5.2 without any configuration caching, using deadline monotonic scheduling. In contrast, if we use scheduling look back, as depicted in Fig. 5.19 b), we get a feasible schedule, while a combination of scheduling look back and scheduling look ahead results in an unfeasible schedule. Figure 5.19 c) displays the latter scenario, where the scheduling is still improved comparing to the scheduling without configuration caching.

Table 5.2: Task set for Fig. 5.19

Task	t_{RT}	t_{EX}	period T_i
1	4	4	12
2	4	8	18
3	4	8	24

5.5.4 Implementation

The first configuration caching algorithm presented in 5.5.1 (*priority based slot reservation*) demands for a preprocessing of the task set. This preprocessing is done offline. The same holds for initializing the fixed tasks. After everything is set, we have to schedule the remaining tasks.

Similarly, the second algorithm in section 5.5.1 requires a pre-processing that is done in the sense of design space exploration. Afterwards, the dispatching of the tasks can be done on these off-line results.

Scheduling look back requires m additional steps when a new task arrives, with m the number of slots. We have to check all slots if the configuration of the arriving task is already loaded in one of the m slots.

In contrast, scheduling look ahead is more sophisticated. The algorithm is activated each time the reconfiguration port is idle. Then, we have to check for all slots if there is at least one slot without an EX phase being active. If we have found one, we consider the whole task set and search for the task with the next release time. If the distance between this release time and the current time is equal to or less than the reconfiguration time, we start to reconfigure this task.

SLB and SLA can be combined. Each time the reconfiguration port idles and we look for an RT phase of a task τ_i that could be pre-loaded (executed in advance), we also look whether the configuration of τ_i already is loaded. If so, then we do not have to reconfigure it once more. Therefore, we look for the next task and might prefer this one. However, we will have to block the firstly selected slot in order to not destroy the already loaded configuration resting in this slot. Nevertheless, the implementation complexity increases when SLB and SLA are combined.

Finally, due to the re-use of the same configuration, we might have to re-initialize memory (constant values, etc.) or even larger parts of the circuitry. Thus, we might have to reconfigure parts of the slot. In the average case, the reconfiguration is still shorter than t_{RT}, in the worst case it is equal to t_{RT}. Thus, we will never decrease the behavior of our algorithm, and most likely will always reduce the maximum lateness. In general, the re-initialization is particularly important for memory elements like BlockRAM that can be found on modern FPGAs, as wrong initial values can falsify the computation.

In many cases and for the majority of the reconfigurable area of a slot, we can simply reset the configuration using standard reset methods. Note that the reset mechanism should be user implemented when generating partial bitstreams on Xilinx Virtex devices.

5.6 Experiment

After discussing the capabilities of the reconfiguration port scheduling algorithms, we also want to discuss an example application, where the real-time aware execution of run-time reconfigurable tasks can be of benefit. We selected the scenario of an online recognition system that is required to identify and sort objects placed on several conveyer belts. On each conveyer belt, objects arrive periodically. We sort the objects into

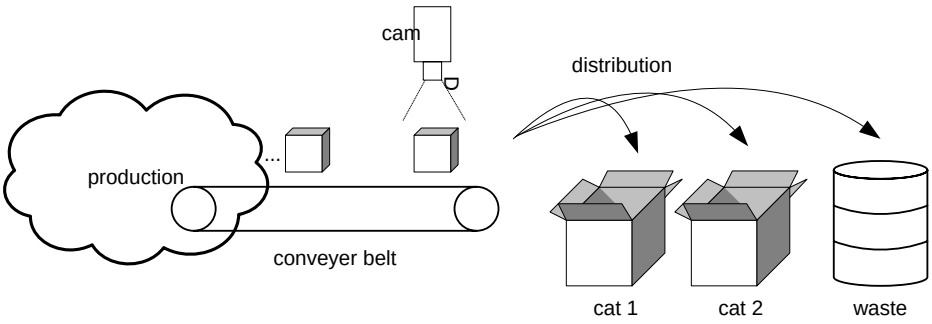

Figure 5.20: Conveyer belt

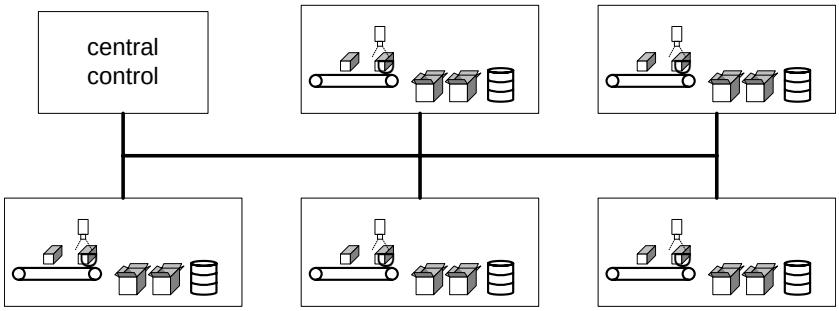

Figure 5.21: Set of conveyer belts connected to a central processing unit.

different categories (quality levels) meeting the timing constraints of each of the conveyer belts to fulfill the overall requirements of a production system.

As displayed in Fig. 5.20, the recognition shall take place at the end of a production line that turns out goods in a well-defined frequency. Therefore, several sensors are installed that derive data, which must be interpreted correctly to be able to sort the product into boxes for different quality levels. It quickly becomes obvious that for economical reasons, the production should not be halted and therefore the sorting must meet the frequency of the arriving products. As object recognition requires high processing resources, a hardware implementation often is preferred. Using reconfigurable logic therefore, we can react on changes of the production line by adapting the recognition circuit. An FPGA as processing device thus is a recommended alternative.

In addition, plants usually not only comprise of one but several production lines resulting in a set of conveyer belts, whose objects must be sorted in parallel. We thus face a set of recognition systems that all must work correctly to prevent single production lines from disruption. Moreover, we assume a centralized control for the set of recognition systems, which could be necessary due to harsh environmental conditions in the immediate vicinity of the conveyer belts. Figure 5.21 depicts an example consisting of five conveyer belts each having their own sensors for recognition and own actuators for sorting. The sensor data of each conveyer belt is forwarded to the centralized control system, which evaluates the data and sends a command to the conveyer belt. The command denotes the quality of the product and allows for correct sorting.

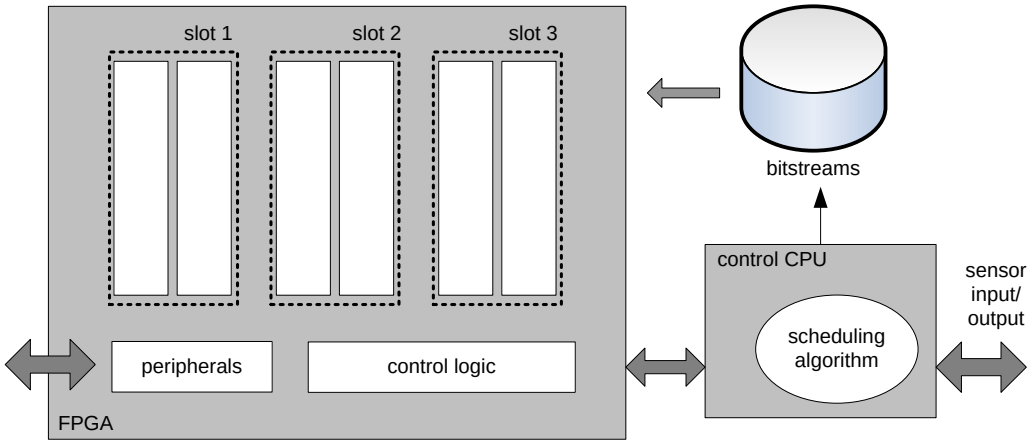

Figure 5.22: Experimental set-up on the Erlangen Slot Machine

We can keep costs low, if we equip the central control system with a reasonably small FPGA only. The area of the FPGA then must be shared by the different recognition circuits of the conveyer belts. Run-time reconfiguration solves the problem technically. However, the whole system also must be classified as a hard real-time system, as a timely processing is vital for un-interruptible production. In a worst case scenario, the so-called domino effect [But04] may even disrupt all processing lines initially only suffering one single fault. Proper scheduling thus is essential. For such a scenario, the reconfiguration port scheduling method of this chapter comes in.

Implementation

For the practical implementation, we assume an execution environment, which consists of three slots. The slots operate in parallel and can be used by any of the recognition circuits in a time-shared fashion. A simple control unit on the FPGA takes care of the correct data transmission. The recognition sensor provides a defined block of data, which must be accessible by the slot holding the corresponding processing circuit. The result, which is a simple command, is sent back directly to the conveyer belt.

For experimental investigations, we have implemented a prototyping environment on the Erlangen Slot Machine. The layout is depicted in Fig. 5.22. To also consider the preemption of the reconfiguration phase, we have divided each slot into two sub-slots that form a unity. Such an explicit partitioning of the slots helps us to overcome some technical problems, as it is tremendously difficult to implement a stop of the reconfiguration process. We store two partial bitstreams for each sub-slot and reconfigure these sub-slots in sequence, unless a preemption occurs. Also, this implementation only can offer a single preemption of each reconfiguration phase, it nevertheless allows us to explore the fundamental feasibility of the approach. As both sub-slots are located directly side by side without any space in between, we only require a single bus macro to route the signals between them. Moreover, note that in contrast to the two slot example, the sub-slots cannot be executed on their own. Only the complete reconfiguration of slot—including both sub-slots—facilitates to start execution.

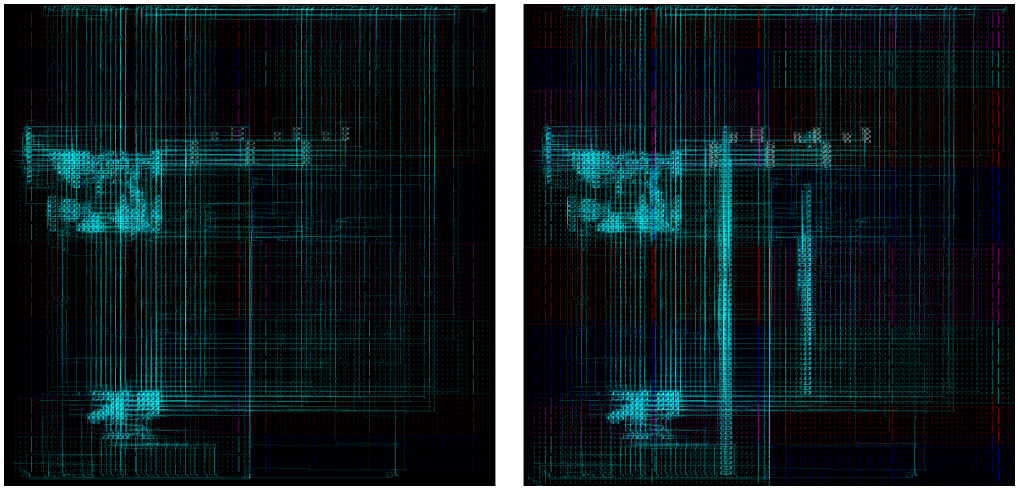

Figure 5.23: FPGA editor output of the execution environment implemented on the ESM. Left: only the basic logic, right: two slots are loaded.

For the communication, we assign dedicated wires to each of the slots. The input data are distributed by the controller to the slots. The data distributer takes care of providing the correct data to the slots. Unfortunately, the Xilinx FPGAs do not generate a done signal after finishing a partial reconfiguration. We thus cannot measure the exact reconfiguration time of one of the sub-slots, however the development team of the ESM provided us with the necessary information.

Moreover, to overcome any problems of signal integrity due to partial reconfiguration, we initiate a well-defined bitsequence that must be received by the task, prior to start execution and also by the control unit before sending out a result of a task. This mechanism to send and detect a sequence can be considered as a gate.

Finally, the ESM main board hosts a PowerPC that is used to boot a minimum Linux environment. The operating system allows for the execution of the scheduling algorithms using the ESM API. This API exposes the capabilities of the ESM through libraries that can be used by user space programs. Most notably are the ability to (re)wire the crossbar, upload the bitstreams and communicate with the FPGA through the so-called *hwswcom* modules.

Results

In Fig. 5.23, we display the layout of the execution environment on the FPGA using the Xilinx FPGA editor. The control logic is implemented in the top left area of the FPGA. The peripheral control logic for the VGA system is located in the left below the control logic. The slots are placed side by side starting in the second third of the device. Each slot is 12 CLBs wide with a height of 144 CLBs which allows a task to use about $12 \cdot 144 = 1728$ CLBs in total. Moreover, sub-slots span 6 CLBs in width.

For the control logic we require only 679 slices which reflects a utilization of 2 %. About half of this space is used by the communication interface with the PowerPC. When we also add the control logic for the camera peripheral to our design, we require

an additional 290 slices, however the amount of routing logic required increases, as the SRAM is accessed by the camera module.

The estimated reconfiguration time for one slot (12 CLBs width) of our execution environment sums up to approx. 127 milliseconds. We added a rough 10 % which lead us to a total reconfiguration time of 140 milliseconds for one slot. The reconfiguration time for a sub-slot was estimated the same way and accounts for 65 milliseconds. Adding again roughly 10 %, the minimum time interval of the reconfiguration port scheduling comprises $\Delta = 70$ ms. The relative long reconfiguration times can be accounted due to the slow flash memory used to store the partial bitstreams. The transmission times between the PowerPC and the FPGA can be neglected in this setup because only 4 bytes are necessary to upload and received data.

We used the above estimated reconfiguration times in a scheduling algorithm and were able to confirm them by verifying that the actual schedule gave the same results as the simulated one. We executed the scheduling algorithm on the PowerPC by referring to a pre-computed table of the tasks parameters.

5.7 Lesson Learned

The chapter has shown how we can exploit reconfigurable fabrics for the domain of reactive systems. To establish real-time behavior as demanded by such systems, we introduced the novel concept of reconfiguration port scheduling.

In particular, the drawback of the reconfiguration overhead can be used for establishing real-time scheduling. We therefore focus on the whole reconfiguration process, particularly including the single reconfiguration port of FPGAs, which circumvents parallel reconfiguration of areas. Both, the long duration and the sequentiality of the reconfiguration, however, allow us to apply mono-processor scheduling algorithms, which are well-researched and help to ease scheduling in particular for real-time scenarios. The appliance of the scheduling algorithms thereby is possible and valuable for such scenarios, even though some limitations have to be taken into account.

We thus can map mono-processor scheduling algorithms to a new domain. Without reinventing the wheel, we can achieve real-time behavior for tasks executing on reconfigurable fabrics. Particulary, the abstracting layered approach helps to assign responsibilities within the real-time scheduling to different experts. We rely on slotted execution environments that already exist in the literature and extend their usage.

Subsequently, we could derive that the concept of a run-time platform layer is valuable in general. We can pose exact requirements to such a platform, including the need for a sound communication concept for the execution environment, which must be given to avoid the data transfer becoming the bottleneck of the system. In the realm of the work on making reconfigurable systems mature, the reconfiguration port scheduling thus facilitates to beneficially make use of reconfigurable fabrics.

Nevertheless, for a comprehensive exploitation, some extension to the current reconfiguration techniques are necessary. Foremost, the preemption of the reconfiguration phase, as demanded by our scheduling algorithms, is not fully supported on modern

FPGAs. We solved the idea of preemption by dividing the reconfigurable slot area into pre-defined sub-slots, which are reconfigured subsequently. If arbitrary preemption would be possible, the flexibility could be increased and the work-around of sub-slots would become unnecessary.

Although posing a challenge for the implementation, the preemption of the reconfiguration phase still opens new perspectives to the appliance of run-time reconfiguration on FPGAs in the real-time domain. In general, reconfigurable devices execute tasks in parallel, which intentionally collides with the single machine principle and seems to require new methods and evaluation strategies for scheduling. By virtue of the reconfiguration port scheduling including task preemption, we can achieve real-time behavior relying on algorithms having a low computation complexity.

We have also seen that the pure reconfigurable port scheduling technique can be easily extended by caching concepts. Caching, in general, is driven by the overall design goal that reconfiguration should be avoided as often as possible. In our scenario of scheduling the reconfiguration phases, caching means that tasks will skip some instances. Thus, it is more likely that more tasks will meet their deadlines when applying configuration caching compared to the schedulability of the task set without configuration caching.

Despite that configuration caching inevitably increases the complexity of the feasibility analysis of a task set, some interested scheduling strategies of the algorithms are possible. We can include the results of the investigations of this approach into a system that heuristically tries to derive feasible schedules even if the reconfiguration port occupancy is larger than 1. In such a case, schedulability would be impossible without caching.

The drawbacks of the reconfiguration port scheduling are the need for homogeneous slots. We thereby might fail to exploit the whole capabilities of reconfigurable systems. Nevertheless, by virtue of the method shown in this chapter, reconfigurable fabrics are exploited in a sophisticated way targeting a specific application area. However, the method does not completely combine the three ideas of run-time reconfiguration: algorithmic reconfiguration, functional reconfiguration, and architectural reconfiguration.

To finally combine these three reconfiguration types, we require an approach that starts on an even higher level of abstraction as the approaches of the last three chapters. The designer should not be aware of whether its applications are executed in a time-shared fashion on a reconfiguration fabric, adapted by run-time reconfiguration on changing environmental input, or even a complete reconfiguration of the run-time environment happens. Thus, we finally want to achieve a complete transparency of the run-time reconfiguration.

5.8 Related Work

Some work has already been done in online scheduling of real-time tasks on reconfigurable architectures. Most of them divides the problem into two main problems: task scheduling and task placement. In general, scheduling tasks on partially reconfigurable FPGAs is targeted differently in the literature.

The chapter also considers caching configurations under real-time scheduling. A combination of both to the best of our knowledge was not considered so far.

Task Scheduling

Some works schedule area and time together [FKT01], while others focus on task scheduling with fragmentation handled by a specific manager [RMVC05], or on the area as an execution environment [WP04].

In the area of scheduling tasks on partially reconfigurable FPGAs, the authors of [ABT04] present an approach in the realm of our work. In their work, they optimize the area occupied, respecting the task time constraints. Tasks are not allowed to be preempted.

In the same scenario, the authors of [WP02] and [SWP04] analyze the effect of overall response time and guarantee-base scheduling when tasks comprise different shapes.

In some works, even task preemption is considered. If task preemption is allowed, the task acceptance rate is improved [ABK+04, WP03]. However, hardware task preemption represents additional costs due to still non-efficient techniques and methods available. Generally, we seldom find concepts that respect the reconfiguration time or the sequentiality of the reconfiguration phases. Usually, both are neglected due to the assumption that the execution time is much higher than the reconfiguration time [WP03].

In [DP06], we find an approach that also considers real-time scheduling on partially reconfigurable FPGAs. In contrast to our approach, the scheduling is solved by algorithms of the parallel scheduling domain. Additionally, the authors present adapted and new algorithms for the special requirements of scheduling in space and time. Furthermore, we focus on the reconfiguration phase as main scheduling problem.

In general, in our approach, the placement problem must not be considered since we assume that every task comprises the same size.

Caching

Caching in the domain of reconfigurable computing is considered in e. g. [DeH96a]. In general, if applications to be executed are similar, caching is desirable. Furthermore, if tasks are periodic, the reuse of already present stages improves a system. Successful *configuration caching*, presented in [LCH00], is also effective if the reconfigurable device is attached as a co-processor. [LH02] proposes to combine *configuration prefetching* [Hau98] and *configuration caching*, thereby detailing the subject in some more extend. Besides these conceptual works, authors also investigate caching in coupled FPGA-processor systems [SNG01]. History-based algorithm or specific constructs to denote stages worth to be cached are introduced. In all the works on configuration caching, however, real-time requirements are not considered.

In all the works on configuration caching, the reconfigurable fabric operates as co-processer and real-time requirements are not considered.

5.9 Summary

In this chapter, we have investigated scheduling strategies known from the single machine environment and applied them on reconfigurable devices. Therefore, we introduced a real-time scheduling layer for partially reconfigurable FPGAs. The layer accepts task sets and schedules the reconfiguration port of execution environments. Due to the mutual exclusiveness and the sequentiality of the port, we can apply scheduling algorithms from the monoprocessor domain. We discussed assets and drawbacks of this approach on the basis of aperiodic task sets. With this background, we derived scheduling algorithms for aperiodic and periodic tasks.

As main result for both algorithms applied (EDD and EDF), we note that the approach performs best if $t_{EX} \leq t_{RT} \cdot (m-1)$. On the gained results of aperiodic tasks, we have derived a scheduling algorithm for periodic tasks. The algorithm bases on the deadline monotonic (DM) scheduling concept and is extended by a server to handle *full load of slots* conditions, and a resource access protocol to handle *full reconfiguration capacity* conditions. For the schedulability test, we combine the response time analysis for DM with the priority inheritance protocol and a server. We have to solve the recurrent equations $R_i = t_{RT} + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil t_{RT}$ for the task set $\Gamma' := \Gamma \cap \tau_S$.

Furthermore, we have presented caching concepts for reconfiguration port scheduling. The concepts improve the performance of real-time scheduling of tasks onto reconfigurable devices. We have presented two offline algorithms, as well as on-line algorithms.

To summarize, we have shown the benefits of applying mono-processor scheduling algorithms, as well as considered drawbacks, which can harm the predictability and must be considered by slight modifications of the scheduling algorithms.

6 Algorithmic Skeletons for Dynamic Reconfiguration

In the last three chapters, we have shown how reconfigurable hardware can be exploited for different application domains. Most notably, challenges like intermodule communication or device fragmentation thereby can be met if an underlying run-time environment serves as a hardware wrapping layer of abstraction. On top of such a layer, scheduling or mapping algorithms can show their strength as they can rely on a well-designed platform for their specific needs. As the approaches proved to soundly serve their application range, we are interested in how to generalize this concept towards providing a suitable abstracting framework for arbitrary applications. In this chapter, we therefore describe a new solution, which gathers information about the structure of an application on a high level by providing so-called *algorithmic skeletons* as implementation templates. The functional and non-functional information, which is inherent in each of this skeletons, is used to exploit the capabilities of reconfigurable devices, particularly in the dynamic case. Furthermore, the approach facilitates to offer a comprehensive bridge between a typical application/software engineer and a partially run-time reconfigurable FPGA as execution platform.

6.1 Introduction

All over the domain of computer science, the programming level of computing devices is raised if the technologies start to become mature. The basic von Neumann single instruction stream based processor may serve as immediate example: nowadays object-oriented programming languages like Java or C# are the number one choice for implementing applications on such machines, with template- or model-driven software development as presumably next evaluation step. The written code thereby is way above the final machine operations that are eventually executed on the CPU.

Despite a performance drawback that is always discussed when abstraction is applied, the overall benefits usually outperform this loss of speed. Programming on a high level of abstraction facilitates faster development times, efficient code generation, error-resistance, etc. Moreover, the applications themselves can improve their performance, if the complexity of the application under development becomes too large to be overlooked. Therefore, depending on the requirements of an application under development, a problem formulation (modeling and implementation) on a high level of abstraction can result in high-performance code, which will outperform manual implementations, as meta-information like the structure of the application can be exploited beneficially.

We gain similar results for this trend in all domains of processing devices. For example, ASICs are often designed using standard cells, whereby a low-level VLSI-layout is encapsulated into an abstract logic representation. Standard cells allow designers to scale IC design beyond simple single-functions to complex multi-million gate devices. For example, Systems on a Chip are a result of this technique.

Whatever brings areas of computer science to their specific matureness, always additional concepts are introduced. Object-oriented design, for example, significantly extends imperative and procedural programming. As reconfigurable computing increases its presence, the question how to proceed in this area arises. When considering the combination of time and space as given for reconfigurable computing, a simple extension of the classical design flow for FPGAs quickly could become tedious and lack optimal results. We thus need a powerful concept that stands above traditional steps, maybe even orthogonal to them, and supports processing in space and time. So far, an all-encompassing and satisfactory solution is missing, as can be drawn as conclusion from the discussion of a model of computation for reconfiguration computing conducted in [KKS04]. Among others [Sin07] discusses the challenges of designing and programming reconfigurable systems under the scope of future programming models for parallel systems, owing a concrete answer.

Parallel Computing Domain

Based on the closeness to temporal and spatial execution on reconfigurable devices, the domain of parallel programming—executing applications on cluster computers, etc.—is of particular interest. Foremost, parallelism is associated with a boost in performance, as many processing units operate on the same problem in parallel. The core requirement thereby is the organization of the parallelism—the distribution of tasks/threads to the processing nodes must be managed. This task requires special care and even today is an active subject of research [MMadH06]. Several challenges arise, among them the distribution and synchronization of data. Basically, two main concepts were introduced and are used therefore: the shared memory model and the message passing model.

Shared memory basically means that all processing nodes comprise one single memory. This memory is used for the communication and synchronization between the tasks/threads. Also, a distributer can put values to the shared memory and collect the results afterwards. The second approach—MPI (message passing interface)—basically facilitates to abstract from the network. It offers a language-independent communications protocol and targets performance, scalability, and portability. To exploit the parallelism of the underlying machines, both concepts still require the programmer to constantly bear the parallelism of the implementation on his/her mind. They are also often criticized as being fairly low-level [GGKK03]. Applying shared memory or MPI to the domain of reconfigurable computing thus most likely would yield similar results.

The domain of parallel computing however also provides higher level concepts to exploit parallelism. Thereby, the concepts integrate the specific requirement of spatial computation, which seem to be desirable also for reconfigurable fabrics: Basically, the awareness of the parallelism must be present in the programming framework used, if reasonable code offering high performance shall be the result. Thereby application engi-

neers however should be supported continuously, including the abstraction of low-level details of the parallel execution. Moreover, to eventually industrialize the design and therefore utilize the concept of abstraction, sophisticated concepts must be provided.

Several of such concepts exist in the literature and have been evaluated. Among current research are patterns that consider facets of concurrency and parallelism [Dan01, MMS00]. However, the fairly old concept of *Algorithmic Skeletons* introduced in the 1980s by Cole [Col89] gets several characteristics for a mature design of parallel programs to the point. It is close to the application under development, provides implementation guidelines for efficient code and thereby enables programmability and portability. Recently, skeletons are also captured in object-oriented languages [GSP02, MSSB00]. Moreover, authors conduct discussions on the closeness of algorithmic skeletons and design patterns [RG02].

Partially Reconfigurable FPGAs

Undoubtedly, modern partially reconfigurable FPGAs can be compared to parallel machines. Foremost, their processing in space resembles the processing on multiple nodes of parallel machines. If we also consider partial run-time reconfiguration capabilities of FPGAs, the adaptability given for parallel machines can be found in FPGAs as well. Thus, concerning the temporal and spatial capabilities of reconfigurable devices, it is worth to be investigated whether the ideas and concepts of the parallel machine domain can be assigned to partially run-time reconfigurable FPGAs.

Similar to the parallel computing domain, benefits are most likely if the design is done in an architecture aware manner—close to the technical (hardware) characteristics of the FPGAs. As the latter is challenging for the application oriented designer, we show how we can raise the level of abstraction by using the above mentioned *algorithmic skeletons*. Basically, they are programming templates that guide designers to efficiently implement algorithms by separating the structure from the computation itself, which—appropriately applied to reconfigurable fabrics—can make the reconfigurability transparent for the algorithm under execution. The programmer can thereby evaluate the system under design on a high level, as well-defined algorithmic skeletons allow for high-level estimation. For each skeleton (not for the application), experts have set the implementation on the reconfigurable fabric. Additionally, dynamic reconfiguration, which—when brought to its basics—allows for dynamic acceptance of previously unknown tasks, gets supported by algorithmic skeletons in a very sophisticated way.

The reason to focus on algorithmic skeletons can also be found along the wish for a lightweight approach. We therefore do not want to introduce the complex shared memory model or the heavyweight MPI model to reconfigurable computing. Algorithmic skeletons seem to be a suitable and promising alternative. As will be shown in the remainder of this chapter, algorithmic skeletons particularly allow FPGA design specialists to valuably add to the challenging task of industrializing reconfigurable computing.

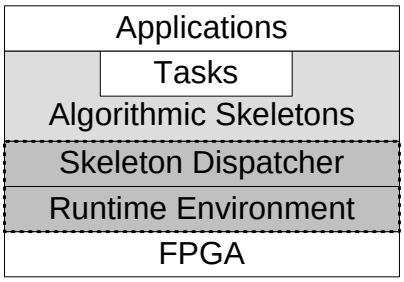

Figure 6.1: Layered model

6.1.1 Layered Approach

We again describe the approach by virtue of a layered model, see Fig. 6.1. Applications, which built the entrance layer, are described by a set of tasks. These tasks must be implemented using algorithmic skeletons, in particular describing their interaction. An execution environment that executes the tasks on an FPGA accepts the tasks described by a set of skeleton instances only. The set of skeletons is processed by a dispatcher that is deeply connected to its execution environment.

6.1.2 Remainder of the Chapter

This chapter is organized as follows: We first formulate the problem and conceptually describe the proposed solution. The approach bears numerous potential for detailed research. In this work, we refine the concept proposed by detailing three skeletons of the stream parallel computing paradigm. Moreover, dynamic reconfiguration by virtue of algorithmic skeletons is discussed in Sect. 6.5.

We consider applications of algorithmic skeletons for the design of reconfigurable systems. In general, as the approach of this chapter is a rather large research, we mainly introduce the major ideas theoretically with some supporting examples. Finally, we look at the lessons learned, review related work, and summarize the chapter.

6.2 Concept

To appreciate the elegance of using algorithmic skeletons for reconfigurable systems, we want to consider the design of reconfigurable systems from two contrary points of view: platform design and application design. In retrospect, both points get manifested in the previous chapters.

Concerning the *platform design*—the design of the execution environment—FPGAs fundamentally are hardware that allows for executing arbitrary circuits in space. Therefore, a firm background in hardware design is desired, including communication and I/O requirements. Moreover, we have to respect the critical path information of circuits, clock skew, etc. Partially reconfigurable FPGAs increase this complexity as they allow

the modification of hardware over time—including all the obstacles mentioned in the previous chapters.

In contrast, the *application design*—the design of applications to be executed basically on any computational device—is driven by achieving high performance and short time to market. Application designers therefore explore the theory behind applications and search for algorithms that serve the problems best. Moreover, they try to abstract from the executing devices, mostly due to reasons of programmability and portability. Application engineers are used to describe the problems and solutions on a high level of abstraction. In particular, the details of hardware and FPGAs thereby are of secondary focus, as development takes place more in the terms of the software world, even if special requirements of embedded systems are respected in some design methodologies. Generally, application engineers do not know how to program reconfigurable systems or even explore run time reconfiguration.

Although FPGAs themselves bear the capability of fast turnaround time in their inherent nature of being in-field adaptable, only a rarely found combination of application expert and hardware engineer seems to exhibit the knowledge to satisfactory produce valuable results. Considering the rising complexity of modern FPGAs—particularly the challenging run-time reconfigurability—new approaches are inevitably. If these features shall be exploited beneficially, expert knowledge encompassing temporal and spacial behavior of algorithms executing on hardware is required. In the previous chapters, we have shown sound concepts for exploiting reconfigurable fabrics in several application domains, however owing a general solution.

Basically, synthesis from behavioral problem description to reconfigurable hardware targets this issue. In the domain of partial run-time reconfigurable hardware however, automatic synthesis still lacks good results [Bob07]. Most of all, the input given as pure algorithm only focusing on the algorithmic behavior can hardly produce reasonable results. For example, partitioning (as presented in Chap. 3) or Pareto-optimal mapping (Chap. 4) may be required, both struggling to detect application inherent parallelism. Furthermore, if iterative design, which is very costly for hardware, is required due to performance evaluation, or portability is an issue, we require a more suitable design methodology that supports designers on a high level of abstraction.

Algorithmic skeletons, which inherently provide parallelism by making structure and computation (behavior) explicit, are a technique that facilitates to overcome several of the constraints mentioned above. Moreover, algorithmic skeletons can serve as bridge between circuit design and application development for FPGAs.

6.2.1 Algorithmic Skeletons

Already briefly discussed in the introduction of this chapter, we want to detail algorithmic skeletons in the following, before proposing them for reconfigurable system design.

Algorithmic skeletons were introduced by Cole in the 1980s [Col89] under the motivation of providing a means for the design and implementation of software systems, which allow for coordination of concurrent activities of large-scale parallel systems. The major aims of algorithmic skeletons therefore are to simplify programming, enhance

portability, improve performance, and offer a scope for static and dynamic optimization [Col04]. Thereby, the approach addresses many of the traditional issues within the parallel software engineering process.

The basic idea of algorithmic skeletons is to separate the structure of a computation from the computation itself. Algorithmic skeletons free the programmer from the implementation details of the structure, such as how to map it to the available processors, as the inherent structure of each algorithmic skeleton constraints the physical distribution of the processing so that sound performance is possible. By providing a structured management of parallel computation, they can be used to write architecture independent programs, shielding application developers from the details of a parallel implementation.

The idea of parallel programming with skeletons thus is to separate two basic concerns of parallelism—application and implementation. The user can specify the potentially parallel parts of an application using predefined programming patterns (skeletons) and leaving the actual organization of parallelism to the skeleton implementation, for example provided by a compiler or a library [RG02]. Algorithmic skeletons also can be nested, thereby increasing their expressiveness [Pel98].

The closest analogy of algorithmic skeletons are higher-order functions of functional languages. More precisely, algorithmic skeletons are similar to polymorphic higher-order functions representing common parallelization patterns. The higher-order functions therefore are implemented in parallel. However, unlike the concept of higher-order functions might suggest, skeleton programming is not functional programming. In contrast, algorithmic skeletons can be used as the building blocks of parallel and distributed applications by integrating them into a sequential language [BK96].

The purpose of every skeleton is to abstract a pattern of activities and their interactions. Applications are expressed as an instance of one or more skeletons. They thereby provide a precise means of implementation, which also basically separates them from design patterns. The latter are mostly used during the design phase and offer only orientation for the final implementation. Initially, patterns are the structuring concepts of sequential computing/programming. In contrast, skeletons are a concrete implementation aid.

Concerning design space exploration, skeletons and their level of abstraction enable to explore a variety of parallel structurings for a given application. Thereby, a clean separation between structural aspects and the application specific details can be achieved by virtue of the algorithmic skeletons. Thanks to the structural information provided, static and dynamic optimization of implementations is possible.

Consequently, there has to be a balance between generality (allowing re-use for different architectures and user kernels) and specificity (for efficient implementation and interfaces to the user kernels). Moreover, we denote the so-called *trap of universality*—providing a skeleton that is generic in itself and can be used if no other skeleton might fit. Such a skeleton would increase the complexity of any run-time environment. In order to avoid this trap, there is usually the restriction of the acceptable input for a system to a set of valid algorithmic skeletons only, see also [Col04, RG02].

Finally, reliability is increased because skeleton programs are stripped of irrelevant details, often making them easier to construct.

6.2.2 Application for Reconfigurable Computing

As sketched in the introduction, reconfigurable computing on FPGAs basically bears similarity to processing on parallel systems, as execution of algorithms on hardware like FPGAs also means processing in parallel. When reconfiguring FPGAs, we usually define exchangeable regions and apply different modules to these regions. Several such regions can be marked on the same FPGA, comprising an execution environment. The regions then are comparable to the nodes of a computing cluster. The intermodule communication, so still a challenging research area, enables various ways of data exchange. We can distribute applications into the regions as it is done in the parallel computing domain. For efficient execution and beneficial exploitation of the capabilities, both systems need structure, which is provided by algorithmic skeletons.

Having applications described by virtue of algorithmic skeletons, we can extract structural information of the application that is viable for executing the application in time and space. For example, parts of an application that can be processed in parallel, may reside spatially distributed on a reconfigurable fabric, thereby improving the overall response time of the application. However, if space is limited, we may also execute these parts in sequence applying run-time reconfiguration.

Algorithmic skeletons therefore become deeply connected to the execution environments. Applications then use a well-defined set of skeletons that allows for comprehensive exploitation of the processing capabilities of the hardware encapsulated by the environment. In fact, as the implementation of skeletons is tied to the execution environment—each execution environment, which was specifically designed for its reconfigurable fabric, offers a set of skeletons having pre-defined characteristics—the application designer can evaluate the application under development on a high level of abstraction, by mapping the application to a (set of) skeleton(s).

Moreover, algorithmic skeletons facilitate to react on dynamic requirements. For example, if several applications share the same reconfigurable fabric, we can offer different quality of service to the application. Referring to the structural information of the applications, we may execute parts of an application spatially or temporarily, depending on a negotiation of the applications. Also, bottleneck stages of a pipeline-like application might be executed in parallel on a reconfigurable fabric to improve the performance.

We thus use algorithmic skeletons as means of abstraction for partial run-time reconfiguration. The skeletons foremost provide a sound method to abstract reconfigurable computing on a high level. The core idea of the skeletons thereby is to force users to add valuable information for reconfiguration to the system description/implementation by referring to algorithmic skeletons. We then can extract the inherent information about parallelism from the algorithmic skeletons and use it to generate optimized results.

In a straight forward approach, algorithmic skeletons therefore are offered as a library that is used by the algorithms of the application under development. Tasks of these applications described by virtue of skeletons can be executed on a run-time environment that accepts the skeletons. The usage of algorithmic skeletons constrains the design of algorithms to a set of templates understood by the environment.

Trap of Universality for FPGAs

Algorithmic skeletons intentionally constraint the design space of application developers and force him/her to express the algorithms by referring to a given set of skeletons solely. A common question thereby is the one on problems that might fail to be expressed by the set of skeletons given. Cole coined this problem the *trap of universality* [Col89], see above. He raised the question whether it is advisable to offer a skeleton for all requirements of the application designer or whether the processing platform should be optimized for a set of skeletons that must be used but provide high performance.

In case of applying algorithmic skeletons to FPGAs, the trap of universality can be approached in a different manner. Besides the possibility to dedicate some area for arbitrary algorithms (a kind of *generic skeleton*), we can also make use of a soft or hard core CPU mounted on the fabric. Applications, which are complex to be covered by a skeleton, then can be executed in software on the CPU. For example control flow intensive parts of an application, which are furthermore generally costly when implemented in hardware, directly benefit from the additional possibility to execute software within our otherwise algorithmic skeleton based system.

6.3 Run-time Execution Environment

Being the fundament of the whole concept, the execution environment must be considered in sufficient depth. The design of the environment heavily influences the acceptance and performance of applications given by virtue of skeletons. Acceptance, as each environment will only accept a defined variety of skeletons including a maximal nesting. Performance, as execution environments can be optimized to prefer selected skeletons including a guarantee to execute them in a best-way.

In addition to the pure physical and structural layout of an execution environment, a reconfiguration manager (dispatcher or scheduler) must be designed. This manager resembles the entity that is responsible for accepting skeletons and allocating them onto the reconfigurable fabric. The manger therefore must hold track of the current status of the substrate. As the skeletons heavily constrain the variety of the applications, the reconfiguration manager must only know how to handle its specific subset of skeletons combined with the specific run-time environment. Again, the design of the reconfiguration handler will be a trade-off between performance and flexibility.

Together, the environment and the reconfiguration manager thus make up a pair that is finally responsible for dispatching applications. Thereby, we inevitably will have to solve a trade-off between generality and specificity. On one hand, the execution environment may be optimized for executing algorithms given as a specific type of algorithmic skeleton that have to fulfill area constraints. Applications not meeting these constraints would have to be rejected. On the other hand an execution environment may offer a broad rang of skeletons that themselves have few area or communication constraints. However, the performance of such a general system quickly might drop down.

To face the trade-off between generality and specificity, we present and evaluate three variations of execution environments in the following. The first—tile-based—marginally

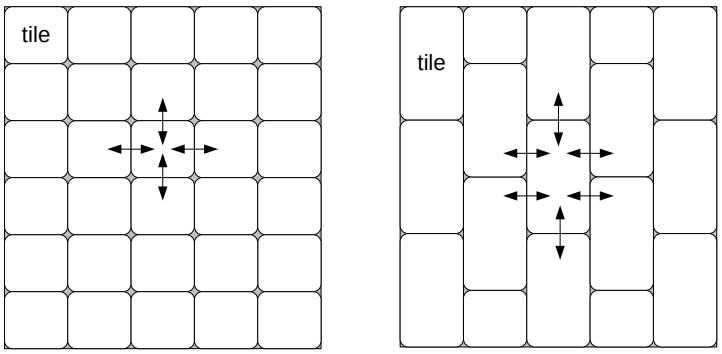

Figure 6.2: Two tile-based run-time execution environments

constraints the reconfigurable device by only dividing the substrate into tiles comprising mainly direct neighbor communication only. The architecture accepts basically all skeletons, however, the design of an appropriate reconfiguration manager quickly can become a sophisticated task. A nice combination of generality and specificity depicts the second—skeleton-centric—example, that accepts a well-defined subset of algorithmic skeletons. These skeletons themselves may be implemented quite freely in the range of the environment. The idea of having a very tight environment including a nearly static reconfiguration manager is presented last, called application-centric. Here, the platform is designed offline with a task set given, therefore allowing to execute the task set with a high performance.

Moreover, in-field updates to the environment or the reconfiguration manager that affect the behavior of the pair can be possible and welcome.

Tile-based Execution Environment

For this very general execution environment, we rely on architectural ideas of reconfigurable computing, like the tile-based structure of Xilinx Virtex-4 and Virtex-5 FPGAs. Basically, the substrate is divided into tiles, which can be of but do not have to be necessarily restricted to equal size, see Fig. 6.2. We therefore consider two different tile arrangements: the first being a purely quadratic organization, while the second one offers more direct communication possibilities due to an underlying hexagonal structure. For the IO, each boarder tile can accept new values.

A run-time reconfiguration manager takes care of the execution. The manager allocates area dynamically for the different problems. As tasks are given by virtue of skeletons, we know the structural requirements of the tasks. They thus are not loaded arbitrarily into the tiles, but by referring to the skeletons. The reconfiguration manager can also initiate an online relocation. As already stated in previous chapters, however, this relocation is hardly cost efficient. In general, as the freedom of the allocation and scheduling also depends on the FPGA architecture used, both the environment and the manager form a close unity.

The approach offers high flexibility on homogenous FPGAs, as the reconfiguration manager can exploit the whole area. Each tile comprises the same logic resources, therefore completely harmonizing the execution environment. However, also heterogeneous

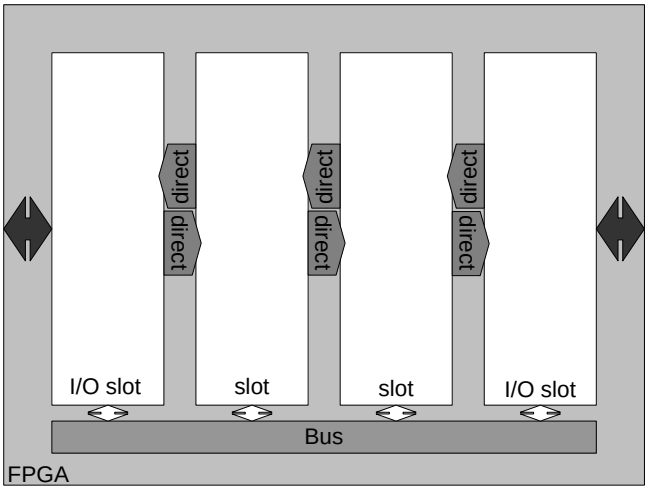

Figure 6.3: Skeleton-centric execution environment

execution environments are possible. Here, not all tiles will accept all types of skeletons, however the performance of skeletons can be better.

Xilinx Virtex-4 devices support the proposed execution environments as they support 2D style partial reconfiguration. Furthermore, on these devices, the external communication—the I/O pads—is separated and not part of a slice. Therefore, the reconfiguration manager must respect the IO pinning. Only applications that can feed in their data at free tiles may be accepted. We can also implement the tile-based execution environment on the Erlangen Slot Machine.

In Sec. 6.4, we show how skeleton dispatching can look like on such an environment.

Skeleton-centric Execution Environment

We can also design the execution environments towards accepting only a specific set of skeletons, denoted as *skeleton-centric execution environment*. By focusing on a selection of skeletons, we can offer very well-performing execution of these skeletons on an FPGA. We may also design the execution environment with respect to hardware cores of the FPGA like multipliers, etc. Well-placed dedicated cores then add to the performance of the skeletons. The reconfiguration manager is optimized to accept and schedule a set of skeletons onto the execution environment including the specific computational resources.

The exemplary execution environment depicted in Fig. 6.3 shows how a skeleton-centric platform may look like. The platform was designed to host both pipe and farm skeletons. Each stage of a pipe skeleton can be loaded into one of the slots. The first stage should be placed in the rightmost or leftmost slot and immediate successors should be placed in neighboring slots. For the data transfer between the stages, the direct communication links can be used. The communication for farm skeletons in contrast is given by the bus that spans the whole width of the slots. Therefore, the placement of workers of a farm skeleton can be done more freely.

The design of a reconfiguration manager directly follows the implementation constraints of the skeletons. For the example given in Fig. 6.3, the reconfiguration manager

may even rely on predefined scenarios of having pipe, farm or a combination of both hosted on its substrate. For example, the scheduling concepts of the previous chapters could be used, for example deriving a schedule for the pipe skeleton. In Sect. 6.5 we show how dynamic reconfiguration can be achieved on such an environment, thereby also detailing the two skeletons pipe and farm.

In general, there always may be an overlap of areas that are shared by different skeletons. However, compared to the tile-based approach, not all areas of this execution environment are suitable for all algorithmic skeletons. Furthermore, the communication infrastructure of the skeleton-centric approach is optimized for the intended set of skeletons. Therefore, it often provides superior performance compared to the tile-based approach, however excluding applications that do not meet the communication requirements.

Moreover, we can also couple the skeleton-centric approach with a specific board the FPGA is mounted on. In general, on such boards the pinning is fixed—having dedicated areas for different types of input. By taking these physical constraints into account, the skeletons can be optimized concerning their physical arrangement on the FPGA.

To conclude, skeleton-centric execution environments nicely resemble the idea of platform-based design. However, they not necessarily must stay fixed over time. In-field updates of both the partitioning and the dispatching may extend the idea of platform-based design as observed in [Ram07].

Application-centric Execution Environment

In the case of an application-centric execution environment, the platform and the reconfiguration manager will be fixed for a given task set. The task set must be structured by virtue of algorithmic skeletons. Algorithmic skeletons then are used to ease the design process of reconfigurable systems. The structuring given also eases portability of the applications, as the non-functional information given by the skeletons helps to find a suitable placement on multiple execution environments.

The performance of such a system thereby can be estimated and evaluated on a high level of abstraction, as the algorithmic skeletons facilitate to estimate the behavior.

For example, applications that are too large to fit on the reconfigurable substrate available can share the resources (exploiting hardware virtualization). If they have been designed using algorithmic skeletons, even an automatic generation of both, execution environment and reconfiguration manager can be performed. This idea under the primary focus of developing a framework for the design of partial dynamic reconfigurable circuits by virtue of algorithmic skeletons was carried out in a diploma thesis supervised by us [Fra07].

Summary

Despite the application area for the execution environment, the final decision also depends on the FPGA designer and the FPGA itself, including the selection of the communication structure. In this work, we use the introduced run-time environments to abstractly exploit the feasibility of our approach. It is most likely that FPGA specialists would implement different and better solutions. In fact, thanks to the clear

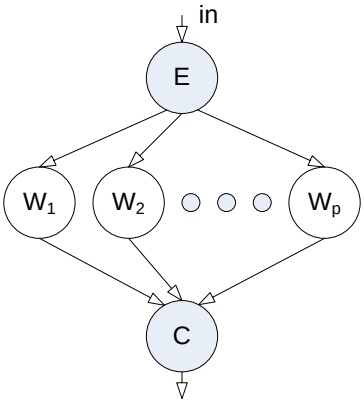

Figure 6.4: Farm parallelism

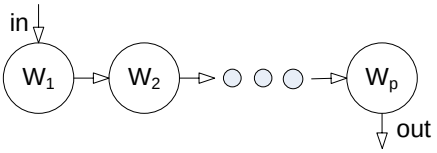

Figure 6.5: The pipeline paradigm

requirements of providing an optimal run-time environment for a set of fixed skeletons that will be executed on the FPGA, the FPGA specialist can focus on this limited set. Nevertheless, in order to approach a truly optimal solution, both the algorithmic and the FPGA specialist should work together as close as possible.

6.4 Stream Parallelism

To detail the idea of algorithmic skeletons for reconfigurable computing, we discuss the domain of stream parallelism in this section. We link the problem to the idea of having a very flexible tile-based execution environment.

Basically, stream parallelism is a very close idea of parallel computing that matches the ideas of execution on FPGAs [Pel98]. Stream computation can be described as applying $f : \alpha \rightarrow \beta$ on a stream of input values $\alpha_1, \alpha_2, \ldots$. The idea is to exploit the parallelism within the computation of f on different (and unrelated) elements of the input stream. As an example, we can consider a vision system that explores images. The images enter the system abstracted as a stream and must be handled differently.

6.4.1 Farm Paradigm

An algorithm that computes the same f on all of the elements of a stream $\alpha_1, \alpha_2, \ldots$ exploits the *farm paradigm*. The computations $f(\alpha_1), f(\alpha_2), \ldots$ can be executed in parallel using a pool of parallel processing modules. Figure 6.4 depicts the concept. The major characteristic to observe is that the workers $W_1, W_2, \ldots, W_p$ can be executed independently of each other as they are all operating on a different data set.

As an application example, we can assume a stream of two video channels that should be output alternatingly to one single channel. However, the switch between the two channels should not be abrupt but smoothly—a fading between the two channels. We thus have a function f that is applied on a stream of three input values $\langle x_1, y_1, c_1 \rangle, \langle x_2, y_2, c_2 \rangle, \ldots$, while x_1 and y_1 denote the two video streams and c_1 the

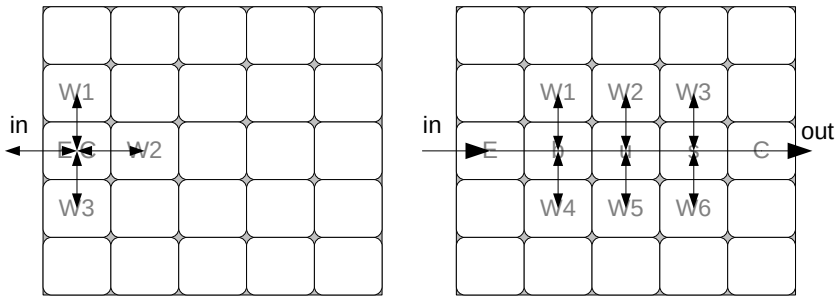

Figure 6.6: Two possible execution schemes of applications using the farm skeleton.

dominance of the one stream over the other (an increasing/decreasing number of e. g., 8 Bit width). Both streams arrive at the node E which distributes the single images to the worker processes $W_1, W_2, \ldots, W_P$ adding the number c_i. These functions can be computed independently of each other in different worker nodes W_i. The results then are propagated to the combining node C that forwards the stream to the video screen. The more workers we have, the higher the frequency of the input values can be.

If we describe our algorithm using a skeleton for the farm paradigm, the structure of the application is given. Thus, we know how to execute the algorithm on a tile-based execution device. Figure 6.6 shows two examples how the skeleton can be mapped. In the left approach, we use the same tile for the input and output of the nodes. However, as we rely on direct communication links, the number of possible worker tiles is limited. Therefore, the right approach of Fig. 6.6 spans the farm skeleton over the whole width of the FPGA.

Dynamic run-time reconfiguration is needed if the amount of worker modules should be adapted during run-time. External stimuli therefore could be a requirement to adapt the quality of service, etc. Further details are discussed in Sect. 6.5.

To summarize the farm skeleton, a structural concept is given that facilitates to distribute workers of an application on different tiles of a partially and run-time reconfigurable FPGA. The execution of the workers including their reconfiguration take part under the supervision of the run-time environment and its dispatcher. By describing an application on basis of the farm skeleton, the number of workers is not set. Depending on the resources available, a different quality of service can be achieved. The optimal solution—a solution that avoids the blocking of workers, etc. due to overload conditions—must be derived carefully by evaluating the execution times of the function f and the distribution time of the initial node E.

6.4.2 Pipeline Paradigm

The pipeline paradigm comprises a composition on n functions $f_1 \ldots f_n$ such that

$$f_1 : \alpha \rightarrow \gamma_1, \ldots, f_i : \gamma_{i-1} \rightarrow \gamma_i, \ldots, f_n : \gamma_{n-1} \rightarrow \beta \tag{6.1}$$

Figure 6.5 shows the concept as a graph.

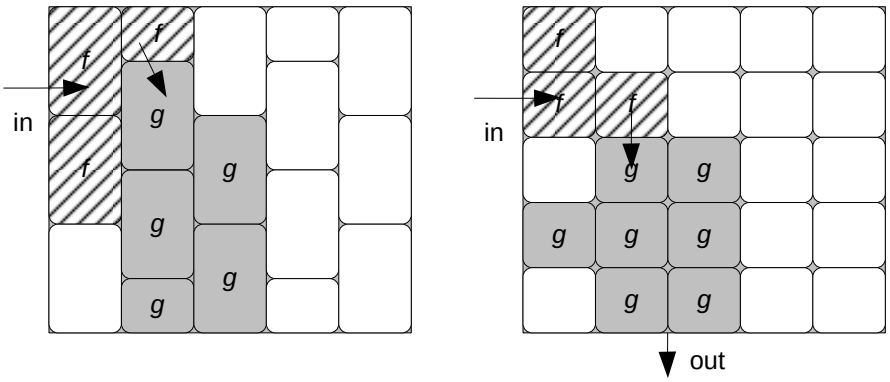

Figure 6.7: Two implementations of a pipe skeleton with different area requirements.

As an example within our image processing environment, we consider a scenario of a stereo vision system. We receive the input of two cameras $\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \ldots$ and want to extract valuable information out of the system. We therefore compute the composition of two functions f and g. Function f will result in a combination of the two images, having the pixel combined into the means ($g(\langle x_i, y_i \rangle) = z_i$), while f will produce the histogram on the resulting image z_i. The functions f and g can be executed in parallel each on a subsequent data set, thus exploiting pipeline parallelism.

In Fig. 6.7, we depict two possible implementation assuming the second function g to consume more area than function f. Here we can see that different stages can consume more area than available on one single tile by simply combining tiles. The dispatcher of the run-time environment may react on the different requirements of the functions within the pipeline. If enough area is available, the dispatcher may also built up a second pipeline in parallel in order to increase the throughput of the systems.

Describing a problem using the pipeline skeleton, we can further exploit the characteristics of stream processing. As the stages of the pipeline get activated in sequence, we can decrease the reconfiguration latency of the overall system. We successively load the bitstreams of the pipeline stages in their order given. After reconfiguring the first stage, this stage may start its execution before the complete pipeline is loaded. The same holds for the subsequent stages. Thus, the fastest possible response time can be guaranteed. Additionally, if less area than required by the stages is available, we may apply hardware virtualization. Therefore, only parts of the overall pipeline are loaded on the FPGA at the same time. These parts may also perform block processing of a block of input sets in order to hide the reconfiguration overhead.

In general, the adaptability given by FPGAs can be explored best if the streams do not exist indefinitely but for a given period of time. Then, we can share the same reconfigurable fabric to host these streams in a time-shared fashion, also including the possibility to host multiple applications at the same time and to interweave two or more applications. Moreover, algorithmic skeletons can be composed to built more complex parallel structures. For example, a farm skeleton may be nested in a pipe skeleton denoting farm parallelism of one of the stages.

6.5 Dynamic Reconfiguration

Basically, different applications each using their own instances of skeletons can be executed on the same FPGA, exploiting a multi task environment. Depending on the specific needs (quality of service, etc.) of the applications behind the skeletons, we can react and dynamically adapt the organization of the skeletons on our execution environment by virtue of (partial) run-time reconfiguration.

In particular, it becomes possible to exploit true dynamic reconfiguration—the hosting of applications not known at the design time of the platform. If these applications are given by virtue of algorithmic skeletons and the execution environment accepts the skeletons used, a successful mapping can take place. Thereby, not only the structural information but also the resource requirements of the application have to be taken into account. A reconfiguration manager that conducts the dynamic reconfiguration of its execution environment therefore must react very flexibly and verify arriving applications. In the worst case, the manager might also have to reject the application. Nevertheless, if applications for dynamic execution were given completely without any structural information, we would have to cope with fragmentation and on-line routing issues that can be tremendously challenging.

In the following, we present two examples of dynamic reconfiguration by virtue of algorithmic skeletons. The first one extends the tile-based execution environment example from above, discussing the general procedure for dynamically hosting applications not known at the design time of the platform. The second example discusses dynamic reconfiguration on a skeleton-centric execution environment.

6.5.1 Dynamic Reconfiguration on a Tile-Based Execution Environment

On the homogeneous tile-based execution environment dynamic reconfiguration is both flexible and challenging. Basically, tasks given by virtue of skeletons can be loaded to multiple possible locations. Nevertheless, the skeleton describes the general requirements of the behavior of the application in space and time including the physical requirements like communication, etc. We thus get a reasonable guideline for the dispatching of the applications, which can be used by the reconfiguration manager.

In Fig. 6.8, we display an example of three applications dynamically arriving and sharing a reconfigurable fabric. We assume a scenario where one application using a farm skeleton is executed in the left side of the FPGA and a second application also using a farm skeleton that occupies the right side of the FPGA. The former one can use four worker tiles, while the latter has six tiles on its dispose. The workers of the second farm skeleton however occupy two physical tiles, thus only three workers are executed in parallel using six tiles. Moreover, the emitter and collector of the second application are each placed in their own tiles. Note also that this arrangement additionally requires two tiles for the data transfer of worker *W1* and *W3* to the collector, as we support direct neighbor communication only.

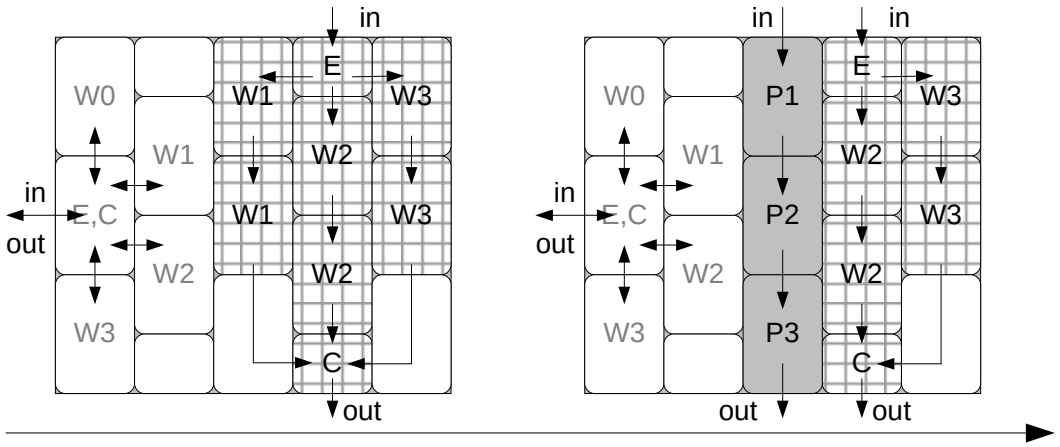

Figure 6.8: Run-time reconfiguration of different skeletons

At some point in time, a new application requests to enter the system. The application is given as an instance of a pipe skeleton, comprising of three stages. As there is no free space on the device to host the application, we dynamically alter the current allocation of the fabric. We decrease the amount of workers of the second (right) farm skeleton, thus freeing area in the middle of the FPGA. This area is then used to execute the new application that is implemented referring to the pipeline skeleton.

The high degree of freedom given by the tile-style execution environment makes application dispatching a challenging task. As can be imagined by the example of this section, the reconfiguration manager has to consider multiple contradictory requirements like I/O, response time, resource requirements, etc. In particular, the scheduling of the reconfiguration phases for dynamic reconfiguration on the tile-based execution environment becomes complex, as many alternatives are possible. For example the dispatcher can operate on a first come first serve basis using a queue for the reconfiguration of the tiles of every application. Having a single queue for each application allows for interweaving the reconfiguration phases of different applications based on the priorities or on the current state of the applications. Nevertheless, the generation of an optimal schedule that minimizes the response times of all applications quickly can become complex, in particular if execution times of tasks are not known in advance. The subsequent example therefore focuses on reducing the complexity of the reconfiguration manager, however sacrificing the flexibility of the tile-style platform.

6.5.2 Dynamic Reconfiguration on a Skeleton-centric Execution Environment

Based on the exemplary skeleton-centric execution environment presented in Sect. 6.3, we discuss dynamic reconfiguration characteristics for a second example. Dynamically arriving tasks shall be executed on the given environment. The execution environment was designed to host two types of skeletons: pipeline skeleton and farm skeleton.

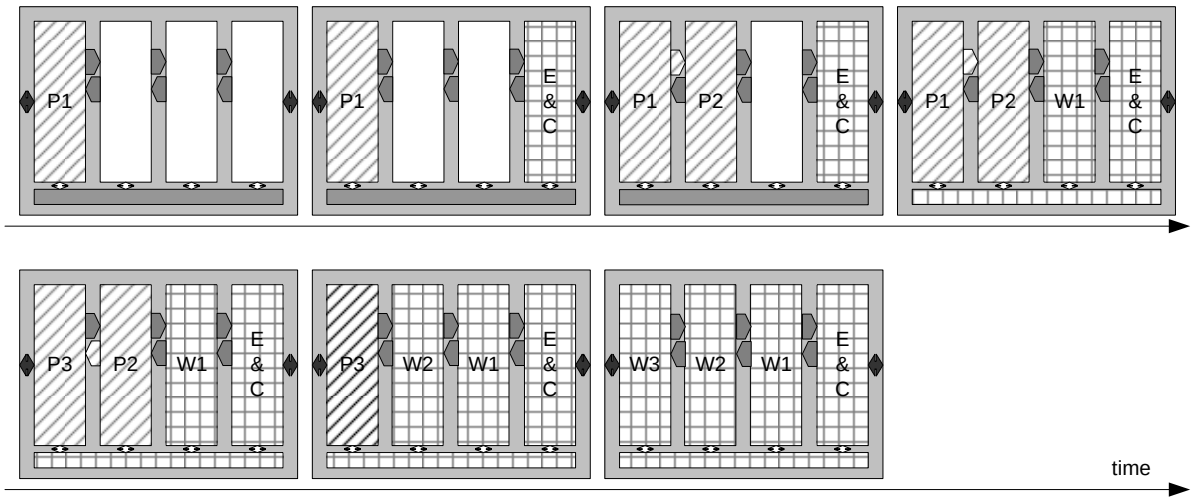

Figure 6.9: Two applications given as farm and pipeline skeleton executing on a skeleton-centric execution environment

For the example discussed, two independent applications arrive for execution on the platform. Both applications have a block of input data to be processed—they thus will have to reside on the reconfigurable fabric until the complete data could be transformed.

The two applications are given by virtue of algorithmic skeletons. The first application is given referring to the pipeline skeleton. The block of data has to undergo three transformation steps (*P1, P2,* and *P3*), making up the stages of the pipeline. In contrast, the second application is given using the farm paradigm, which denotes that each value of the input data stream can be executed independently. Therefore, a worker (W) has been designed, which gets input data from an emitter (E) and sends the result to a collector (C) node. To maximize the response time, as much workers as possible should be executing in parallel.

Figure 6.9 displays the scenario on the skeleton-centric execution environment, assuming the pipeline skeleton arriving before the farm skeleton. Shortly after reconfiguring the first stage of the pipeline (*P1*), the rightmost slot gets reconfigured with the emitter/collector logic of the second application (*E/C*). Before loading the first worker (*W1*) of the farm application, we start to reconfigure the second stage of the pipeline (*P2*). As now all four slots are occupied, we reuse the slot of the first stage of the pipeline to host the third stage (*P3*). Therefore, the block of input data must have completely passed the first stage of the pipeline. The execution environment support direct data forwarding in both directions so that the execution direction of the stages of the pipe skeleton can change. After the data has also passed the second stage of the pipeline, we can reuse the slot of the second stage, to add a second worker to the execution environment (*W2*). The same holds for the slot formerly used for (*P3*), once this stage also has finished. Then, we gain a well-performing second application exploiting task level parallelism. The emitter/collector tile will have to take care of balancing arriving data to the three possible workers.

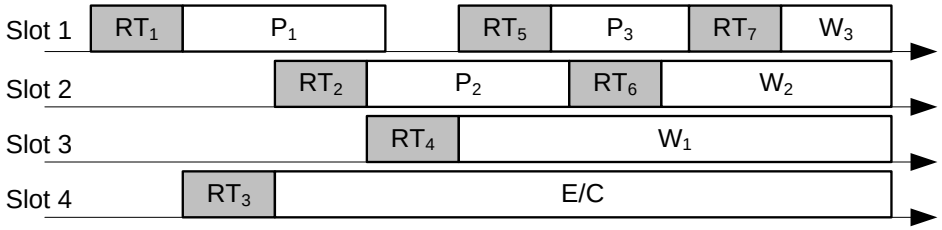

Figure 6.10: Scheduling example of a farm and pipe skeleton on the skeleton-centric execution environment

Concerning the intermodule communication of the different parts of one skeleton, we can apply a physical separation on this execution environment. In detail, for the pipeline skeleton, direct communication is used. Without any intermediate buffer or control logic, the data is transmitted from one slot to its direct neighboring slot. Obviously, the constraints of such a direct communication arise that have been discussed in Chap. 3, for example concerning the limited data width. Moreover, note that the number of communication resources has been fixed at design time of the execution environment. Therefore, the pipeline stages must be adapted to meet the constraints.

For the data transfer of the farm skeleton, we use the bus of the execution environment. The emitter sends data to the worker tiles and also collects the results. Obviously, the benefit of using a bus is that communication also between non-neighboring slots can take place. However, for both communication concepts the applications must be constrained. Appropriate wrappers that may become part of the execution environment can help to reduce the user requirements. For example, the concept of using the wishbone bus as discussed in Chap. 3 can become valuable.

The schedule of such an example looks like the one depicted in Fig. 6.10. As the slot size of this exemplary execution environment is homogeneous, we derive equal reconfiguration times. Moreover, we have to consider the single reconfiguration port, which requires a ordering of the reconfiguration requests. However, as the skeletons allow us to rate the requirements very easily, the complexity of the scheduling can be reduced. For example, the three stages of the pipeline must be scheduled in sequence without any optional stages. In contrast, the farm skeleton only requires the emitter/collector slot and one worker tile to be reconfigured to guarantee correct behavior. Additional workers are optional, however increase the performance on average.

To summarize, the implementation of applications by virtue of algorithmic skeletons enables a sophisticated and dynamic execution of applications on FPGAs. The run-time environment allows us to load tasks not known at the design time of the environment. As the usage of algorithmic skeletons enforces the applications to be well-formed, we thereby can prevent fragmentation of the devices and guarantee communication requirements. Nevertheless, the design of an appropriate reconfiguration manager is required, which is a sophisticated task, even if the execution environment has been designed skeleton-centric.

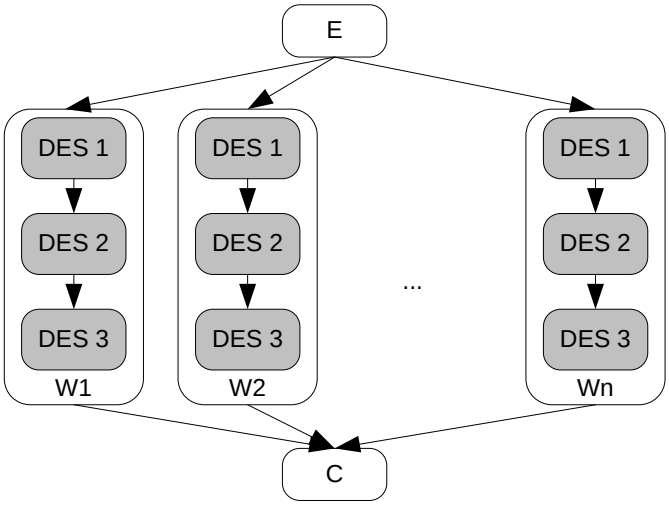

Figure 6.11: Triple DES given as a pipe and farm skeleton

6.6 Experiment

To finally approach the concept practically, as well as to consider yet another example where algorithmic skeletons show advantages, we report on conducted experiments in this section. We first show an example of the cryptography domain that resembles the triple DES example of Chap. 3. Here, we can make use of the pipeline paradigm as well as the farm paradigm. Subsequently, we report on a master thesis that exploits algorithmic skeletons for application-centric design of reconfigurable computing.

6.6.1 Cryptography Experiment

For the first experiment a similar scenario to the triple DES example as presented in Sect. 3.6 shall be used. Triple DES—three basic DES cores that are processed in sequence to raise the level of security—basically resembles a pipeline. Moreover, the algorithm usually operates on a stream of data, by encrypting (decrypting) blocks. These blocks are independent, which allows us to embrace the pipe skeleton by a farm skeleton.

If we thus describe the cryptography example triple DES by virtue of algorithmic skeletons, we get a combination of pipe and farm skeleton as depicted in Fig. 6.11. In the figure we show the graphical description of the cryptography scenario as it might also serve as input for our method. In detail, the three steps *DES 1, DES 2,* and *DES 3* make up a pipe skeleton. This skeleton is embraced by a farm skeleton consisting of the worker nodes $W1$, $W2$, ..., Wn, adding additional emitter E and collector C nodes. Behind each of the nodes *DES 1, DES 2,* and *DES 3* a description of the behavior must be given. In our case, a VHDL code is added. The code simply denotes the behavior of the steps and offers appropriate communication as the external interface—*data in*, *data out*, as well as *start* and *stop* (*data ready*).

For implementing this application to be executed on a reconfigurable run-time environment, a pre-processing must take place that wraps the pure VHDL description and adds

surrounding control logic. Of particular interest are the variations for the intermodule communication. Here, we can basically rely on the research carried out in the Chap. 3, where we have investigated direct, buffer-based, memory-based, etc. communication alternatives. The appropriate wrapping should be offered by the execution environment, as it is platform dependent. Based on the task description and the communication between the tasks given, we can generate the wrappers.

Two slot framework

Basically, we can also consider the two slot framework as an execution environment that can exploit applications given as an instance of the pipe skeleton. The difference now is that we do not constrain the two slot framework to exactly match the triple DES application, but to accept a multiple of applications described as a pipe skeleton. If we thus map the triple DES application as given in Fig. 6.11 onto the two slot environment, we wrap the VHDL code behind the stages *DES 1, DES 2,* and *DES 3* so that the data communication with the intermediate memory in the controller slot can take place.

Moreover, in difference to the example of Chap. 3, we now consider a stream of blocks of data. Due to the long reconfiguration time it is therefore advisable to host the stages as long as possible, as this can maximize the throughput. In detail, we execute *DES 1* until the memory in the controller slot is full, having accepted a number of blocks. In the meantime, we have reconfigured *DES 2*, so that now *DES 2* can process the blocks residing in the memory. Finally, *DES 3* will process the data, so that *DES 1* can be reconfigured and the procedure can start over.

The example of the two slot framework already shows that in addition to the structural information as given by the skeletons, communication information is required. In particular, the frequency of arriving blocks of the data stream must be known to adapt the two slot framework. Here, the algorithms for real-time scheduling of tasks on reconfigurable fabrics discussed in Chap. 5 can help. By performing a response time analysis, we can accept or reject an application.

Skeleton-centric execution environment

We have also designed a completely new skeleton-centric prototypical execution environment on a Xilinx Virtex-4 FPGA. The environment, depicted in Fig. 6.12, consists of eight partially reconfigurable regions. The regions are physically arranged to match the architectural constraints of the FPGA. For example, the height of each of the regions sums up to 16 CLBs or a multiple thereof. Basically, the environment was designed as an exemplary skeleton-centric platform that is capable of accepting tasks given as pipe and farm skeletons and some specific combinations thereof.

In the left side of the FPGA, we have placed six regions that match the requirements of a pipe skeleton. The direct communication going from top to down facilitates an immediate forwarding of data to be processed following the pipeline paradigm. The *E/C* regions on the right side can help to also reduce the pipeline to less than six stages as described by mapping the triple DES application onto the platform in the following.

Moreover, we have implemented the infrastructure for two farm skeletons on the execution environment. By dividing the substrate into an upper and a lower half, the

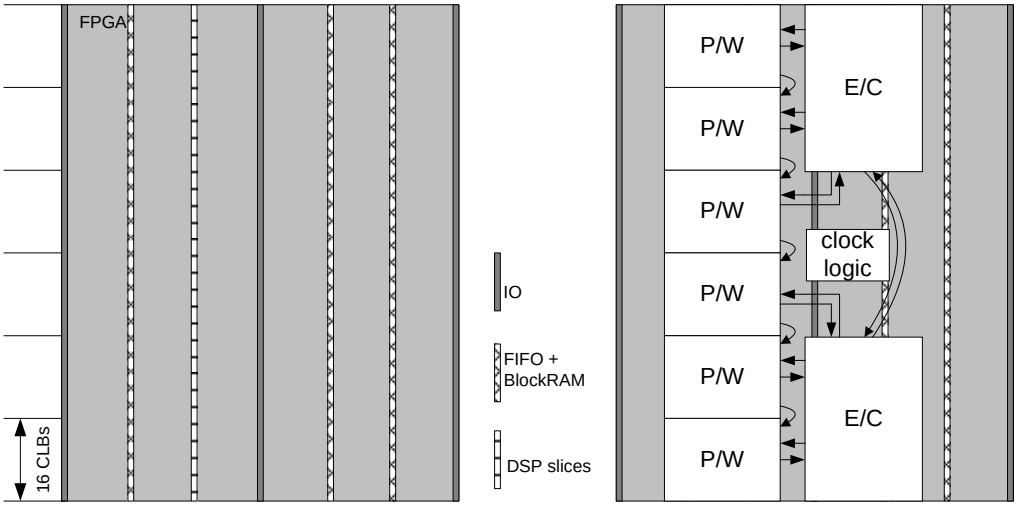

Figure 6.12: Skeleton-centric platform on a Xilinx Virtex-4 FPGA

arrangement becomes clear. Basically, the *E/C* regions can host emitter and collector logic to access the three regions on their left side. We can also combine the two *E/C* regions to make up one large emitter and collector, which can access six worker regions.

For mapping the triple DES application given by virtue of a farm and pipe skeleton as depicted in Fig. 6.13 onto the skeleton-centric execution environment, we exploit the entire regions. The two *E/C* regions are combined to distribute the data to the two workers. Each worker again consists of three stages, thus embracing the pipe skeleton by a farm skeleton exploiting parallelism by implementing multiple pipelines. In Fig. 6.13, we have marked the communication infrastructure used. Again the pure VHDL code was wrapped to match the requirements of the execution environment. The result is given in Fig. 6.14, which depicts an Xilinx FPGA editor snapshot of the triple DES application.

6.6.2 Application-centric

In a master thesis under our supervision [Fra07], the practicability of using algorithmic skeletons for hardware virtualization on partially reconfigurable FPGAs was investigated. The author targeted on how to efficiently calculate offline the placement and scheduling for temporally allocated modules of partially reconfigurable FPGAs. He was motivated by the fact that mapping applications to partially run-time reconfigurable devices is a complex and challenging task.

The work shows that algorithmic skeletons dramatically ease the design process and allow for high-level exploration of the design under development. Basically, applications, which are described by HDL modules and their connections that both together resemble algorithmic skeletons, use patterns designed by experts to built up their specific run-time execution environment. For the scheduling, the arrangement, etc., always the inherent information of the underlying algorithmic skeleton is used. Therefore algorithms for the automatic scheduling and placement of the modules on the FPGA are investigated.

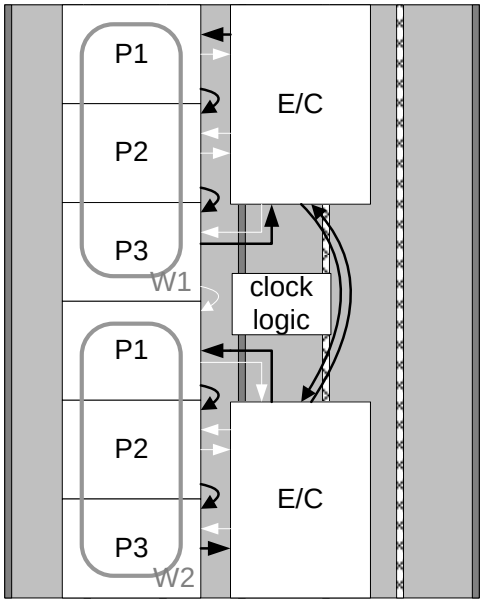

Figure 6.13: Mapping the triple DES application onto the skeleton-centric platform

Foremost, the farm and pipe skeleton get a slightly different interpretation: The stages of a pipe skeleton primarily denote tasks of a task dependence graph, which are executed in sequence, always activating the next stage after the current stage has finished its computation. For the farm skeleton, the amount of workers is given. Moreover, each worker comprises a different function requesting its data from the emitter. An implementation of a farm circuit is given in Fig. 6.15. In addition to the already known emitter, collector, and worker tiles, also a dedicated memory for intermediate data storage is used. Furthermore, Fig. 6.15 depicts the communication infrastructure including data busses and control signals.

In the design flow proposed, the design is given using standard VHDL and a descriptive language designed in the master thesis influenced by the work on algorithmic skeletons on massively parallel systems of [Kal02]. The descriptive language basically allows for instantiating skeletons:

```
FARM(emitter, worker1, worker2, ..., workerN, collector);
PIPE(stage1, stage2,..., stageN);
```

Using the language, we do not have to specify the signals between the slots. In contrast, an automatic generation takes place. The VHDL files therefore have to fulfill certain specifications including well-declared *data in, data out, reset, start,* and *done* signals so that a wrapper can be built automatically around the original files.

Within the design flow, always a preliminary wrapper version will be generated. These wrapper files are not the final ones because for generating the final ones a placement of the modules is required. However, they approach the final version very well and help to evaluate the area requirements of the modules. Hence, it is possible to estimate the

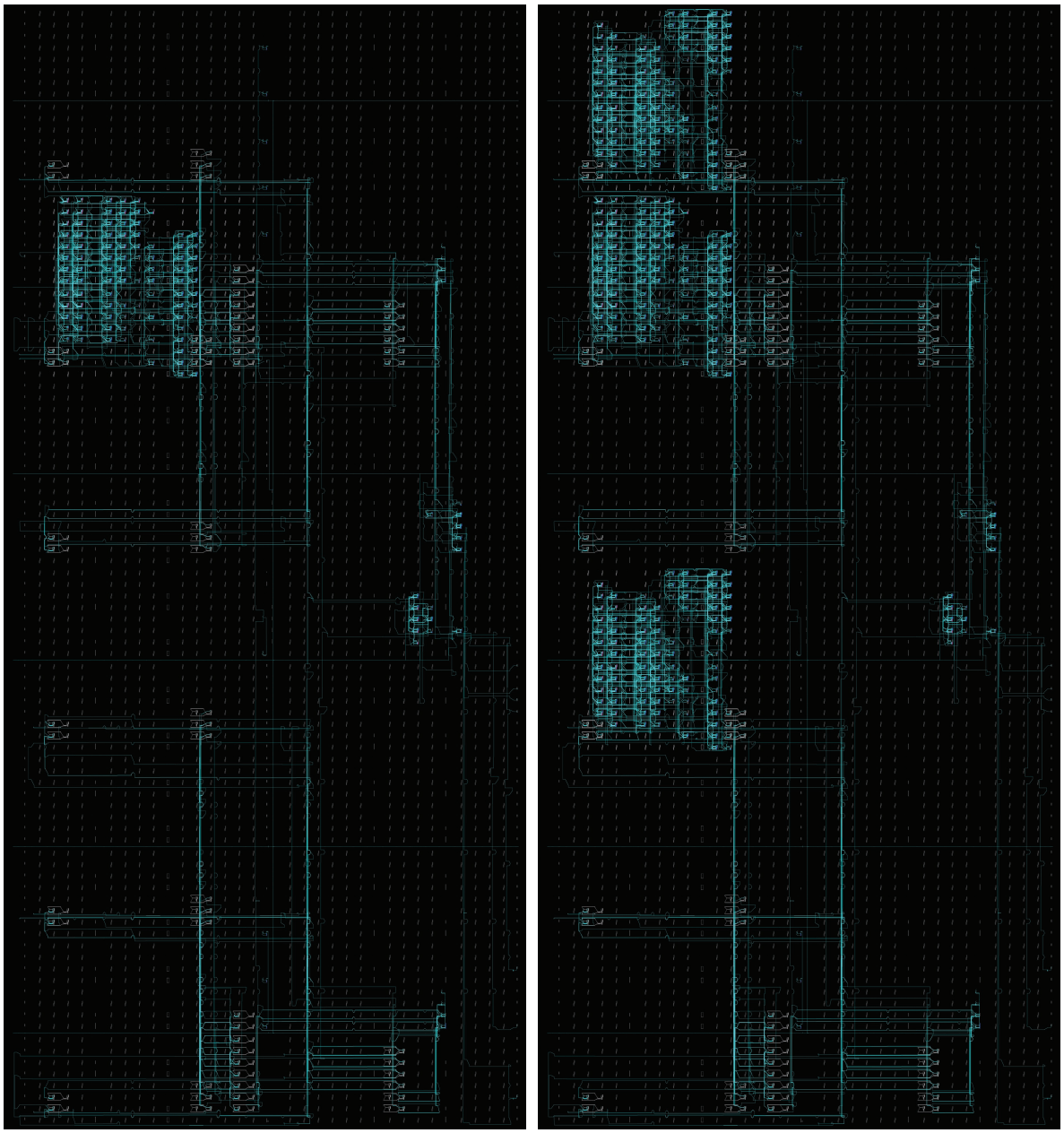

Figure 6.14: Two Xilinx FPGA editor outputs of the triple DES application mapped to the skeleton-centric platform of Fig. 6.12. Left: only *P 2* of *W 1* is loaded, right: *P 1, P 2, and P 4* are loaded.

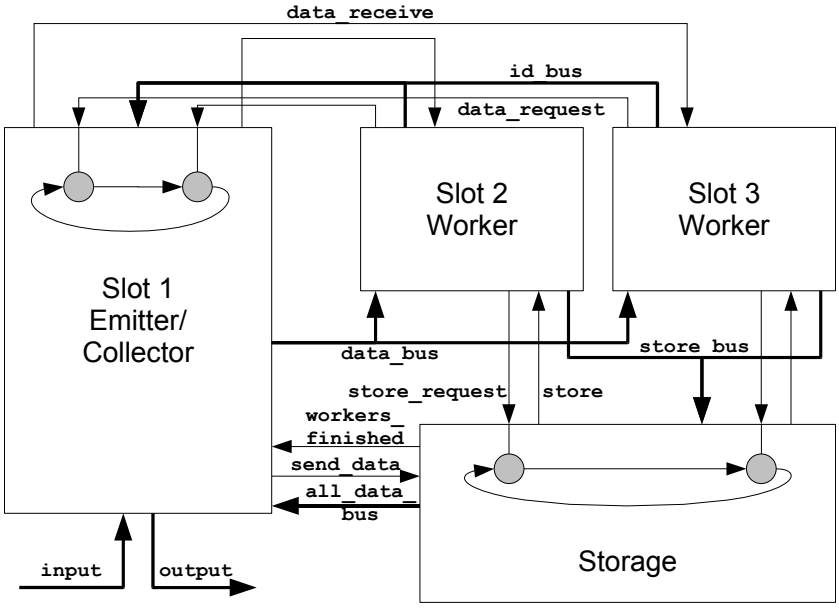

Figure 6.15: Detailed view of the farm circuit [Fra07]

number of slices (CLBs) required of every module using the Xilinx XST tool or similar synthesis tools.

Based on these estimations, a scheduling and mapping can take place relying on so-called patterns. These are skeletal execution environments designed by experts, which give placement suggestions for reconfigurable regions and communication. They thereby only give communication guarantee—wires can be routed. The design environment finally decides on the communication.

In a pattern database designed within the thesis, a couple of patterns are stored for several FPGA types. An algorithm selects the best pattern for the design of the application designer and maps the tasks. The optimization aim is the overall completion time of all tasks. Based on the used skeletons it is possible to extract a good placement of the jobs on the FPGA.

For the eventual generation of the bitstreams, firstly final wrappers based on the location of the tasks are generated. Then, the communication infrastructure (Xilinx busmacros), the placement of the reconfigurable regions (according to the Xilinx Early Access Design Flow), and the *top* file a generated.

Moreover, a tool—PaReToFAS (**Pa**rtial **Re**configuration **To**ol for **F**PGAs using **A**lgorithmic **S**keletons)—was implemented, which provides a design environment that combines the possibility to design patterns and to define the applications as a well-described combination of VHDL modules. To select the best pattern for the application, Frank has developed a genetic and a simulated annealing algorithm.

6.7 Lesson Learned

Algorithmic skeletons are a sophisticated concept to push reconfigurable computing into a new dimension. Basically, they allow us to raise the level of abstraction to meet the requirements of application engineers, while also constraining the physical implementation complexity of execution environments for reconfigurable fabrics. We can thus achieve sophisticated results concerning both, ease of development and well-performance. The key factor is the inherent separating of the structure from the computation itself. The structure given can be exploited for distributing applications into the dimensions time and space in a systematic manner.

In this chapter, we have shown the fundamental applicability of algorithmic skeletons for reconfigurable computing. We basically could see that temporarily and partially adaptable—run-time reconfigurable—FPGAs are close to the concept of parallel machines, as the nodes—reconfigurable regions—can change their behavior and thereby their contribution to the overall application over time. Hence, we investigated several concepts how algorithmic skeletons can match the reconfigurable computing domain. We considered stream parallelism in more detail, which is close to many application fields of reconfigurable systems.

To implement the concept of algorithmic skeletons, we have also realized that suitable execution environments are of great help. Such environments should be combined with a reconfiguration manager forming a close unity. The reconfiguration manager then is responsible for dispatching and rejecting applications. On one hand, as these applications are given by virtue of algorithmic skeletons, the selection and scheduling is reduced. On the other hand, however, the reconfiguration manager becomes a key part of the concept as its competence decides on the acceptance and if so also performance of the applications on the execution environment. In this thesis, we have classified the execution environments into *tile-based*, *skeleton-centric*, and *application-centric*, which each facilitate to explore a different degree of flexibility.

Within the chapter, we could identify two major benefits of using algorithmic skeletons for the design of reconfigurable systems, which are hardly covered by other design methods so far. First, the evaluation of the design on a very high level of abstraction can be achieved by virtue of algorithmic skeletons, as they allow for exploration of designs in a structured manner. In particular, if the performance of an execution environment is known for a set of skeletons, we can estimate the performance of applications that match one or more of these skeletons if they are executed on the environment. Moreover, for the application-centric execution environment, we have shown how an offline mapping technique can be achieved, to derive and evaluate designs on a high level of abstraction.

Secondly, the concept of algorithmic skeletons for reconfigurable systems facilitates to achieve a *true* dynamic reconfiguration, in the sense that applications, which are not know at the design time of the execution environment, can be executed on the fabric. Applications are given in structured manner through instances of skeletons, thus also dynamic executable on a fabric. Here, the skeleton approach is very sophisticated and helps to manage and deal with a lot of challenges of reconfigurable computing design. For example, the execution environment provides generic communication that

is addressed by instances of skeletons. Moreover, the given reconfigurable regions of the environments refer to skeletons and therefore constrain the mapping of applications. We thereby avoid fragmentation and still achieve a good performance, as the environment has been designed for applications that come described as skeletons in mind.

In general, the research on using algorithmic skeletons for reconfigurable computing still is in its infancy. In this work, we therefore have shown the approach of applying algorithmic skeletons to reconfigurable computing design conceptually. Among others, the best way how to formally describe a design using algorithmic skeletons remains open. Despite first steps using a descriptive language, the complete integration of algorithmic skeletons into a design language has to be solved. SystemC might be a natural choice, as algorithmic skeletons can be introduced to this language by creating a specific library. Additionally, we can implement the idea of algorithmic skeletons on top of our Part-E design environment (see Appendix A).

To conclude, algorithmic skeletons are in line with the requirement of raising the level of abstraction for the design of reconfigurable systems. The core benefits of the approach are the main motivation points of algorithmic skeletons: programmability, portability, and performance.

6.8 Related Work

In the literature, we find some works on designing reconfigurable systems on a higher level of abstraction than hardware description languages (HDLs). Most of these works do not target partial run-time reconfigurable systems. Additionally, the models proposed assume the designer to have reasonable knowledge of the system under development.

The work of DeHon *et al.* [DAD+04] on design patterns for reconfigurable systems is a sophisticated approach on providing canonical solutions to common and recurring design challenges of reconfigurable systems and applications. The authors intend on providing a mean to crystallize out common challenges of reconfigurable system design and the typical solutions. However, their work focuses more on providing a layer of abstraction to the reconfigurable systems community than to application engineers.

Some years earlier, SCORE (Stream Computations Organized for Reconfigurable Execution) was proposed in [CCH+00]. The approach focusses on providing a unifying computation model to abstract away the fixed resource limits of devices. Therefore, the resources are virtualized, which can ease the development and deployment of reconfigurable applications. Again, the addressees of the SCORE approach are mainly reconfigurable computing engineers.

Modern languages for embedded systems like SystemVerilog or SystemC also aim at raising the level of abstraction. These approaches can be seen as extended HDLs that introduce design principles to the hardware world, such as re-use, polymorphism, etc. For example, SystemC as language to model dynamic reconfigurable hardware is used in [APC03]. However, the languages are often used for simulation only and the generation of executable code is challenging. Nevertheless, these languages make up a good starting point for future research.

[BSC+00] describes a Matlab compiler that targets heterogeneous systems, including FPGAs. User functions become a process in VHDL. The system also targets multiple FPGAs, however assuming the SPMD style of programming.

Finally, in low-level hardware design, [BC04] focus on a high-level hardware description called hardware skeletons. Considering the idea of separation of structure from the algorithm, this approach is closest to our work. Moreover, the authors motivate their work similar to us—an increase of abstraction in order to open the field of hardware design to a broader audience. Basically, they derive their ideas from FPGA based image processing. They present an approach for developing a general framework for FPGA based image processing. This framework is based on a library of hardware skeletons. However, the amount of skeletons is very limited and they are still very low-level and will often be too far away from algorithm designers. Moreover, we do not find the paradigm of reconfigurability in their work.

6.9 Summary

In this chapter, we have introduced algorithmic skeletons for dynamic reconfigurable computing. Algorithmic skeletons separate structure from the behavior of an algorithm. By providing a library of skeletons to implement applications for reconfigurable systems, we can beneficially explore partial run-time reconfiguration of reconfigurable fabrics. Therefore, solutions—hardware implementations—of the skeletons are applied to various applications. We have discussed the important aspect execution environment, which should come with an appropriate reconfiguration manager specifically implemented for each environment. As a first introduction to this approach, we have considered stream parallelism paradigms including their composition in some more detail. We have also discussed concepts how dynamic reconfiguration can be achieved by virtue of algorithmic skeletons. In general, the approach is a hopeful mean to provide an interface between the hardware platform (FPGA) and applications, raising the level of abstraction.

7 Conclusion and Outlook

7.1 Conclusion

In this thesis, we have presented four design methods for exploiting reconfigurable fabrics under the motivation of making reconfigurable systems mature.

The first design method investigated in detail was termed the two slot framework, which basically offers two reconfigurable slots that can be executed and reconfigured alternatingly. The architecture brings reconfigurable computing to its basics and therefore served very well as introduction to the general requirements and challenges of exploiting reconfigurable systems. Foremost, we derived the requirement to explicitly consider the reconfiguration overhead within the design of reconfigurable computing systems. Therefore, we introduced the separation of a task to be processed by a reconfigurable device into reconfiguration (RT) and execution (EX) phase, which form a close unity and both must be included into any design method.

Despite their simplicity, two slot architectures by no means have so far become visible processing resources in modern designs. When integrating the architecture into a framework, as was done within this thesis, we could identify additional challenges that must be solved for a beneficial exploitation of the two slot architecture. Among them, we discussed the intermodule communication, the partitioning of applications to match the sequential execution requirement of the two slots, and the scheduling algorithm.

The two slot framework also is open for extensions. We introduced concepts of low power design to the framework, exploiting multiple frequency domains as given on modern FPGAs. Furthermore, we discussed the possibility to include the two slot framework as an IP core into a processor-based system, offering hardware virtualization for SoCs.

In the second method, we broadened the reconfigurable architectures covered to include multi-slot environments also comprising heterogeneous slots. We therefore extended a high level design method for embedded systems, which so far excluded partially run-time reconfigurable fabrics from its processing resources. We showed how partial reconfigurability can be included into this so-called specification graph approach. Basically, we added the reconfiguration phases to the task graph and the reconfiguration port to the architecture graph. To prevent overwriting of reconfigurations whose corresponding tasks are delayed for execution due to precedence constraints, we introduced life periods. Life periods can be seamlessly integrated into the scheduling algorithm of the design space exploration of the specification graph approach.

The approach is of value for the so-called platform-based design, where applications and execution environment meet in the middle. The seamless integration of partially run-

time reconfigurable systems introduced by us helps to extend the processing resources and exploit adaptability. Moreover, the specification graph approach offers design space exploration on a high level of abstraction, therefore attracting application designers of multiple domains. By virtue of our extensions, these designers can make use of partial reconfigurability without having to learn a new design method.

The third method we discussed was the new and within this thesis developed concept of real-time reconfiguration port scheduling. Based on a fixed execution environment, we investigated real-time scheduling of tasks into homogeneous slots particularly including the single and mutually exclusive reconfiguration port. As the characteristics of the port resemble the conditions of mono-processor scheduling, we applied well-researched mono-processor scheduling algorithms to the domain of real-time execution of tasks on reconfigurable systems based on the fact that these tasks all have to pass the single reconfiguration port. We showed several scheduling algorithms for the reconfiguration phase, including the new concept of preempting reconfiguration phases in order to improve schedulability. The algorithms base on the newly introduced deadline d^*, which denotes the latest finishing time of a reconfiguration phase in order to be able to guarantee a task to meet its deadline d.

To derive the basic requirements of the reconfiguration port scheduling, we first investigated aperiodic task sets having synchronous as well as asynchronous arrival times of tasks. We thereby showed the fundamental applicability of the reconfiguration port scheduling concept to real-time task execution on reconfigurable fabrics. Two conditions—the *full load of slots (fls)* and the *full reconfiguration capacity (frc)* situation—additionally arise. We showed how to include them into the scheduling algorithms EDF and EDD, and derived guarantee tests respecting *fls* and *frc*.

We then introduced a solution for periodic task sets that allows us to rate task sets concerning their schedulability only based on the parameters of the tasks. To cover the two conditions *fls* and *frc*, which also occur during periodic task scheduling, we introduced a virtually executing server to include *fls* conditions and a resource access protocol for *frc* scenarios. Both can be included into the schedulability test.

Moreover, we investigated caching techniques for the reconfiguration port scheduling. Caching means the avoidance of the reconfiguration phase and therefore is particulary beneficial for the overall performance of a task set, as it helps to reduce the response times. We derived four caching methods, two for offline manipulation of the task set and two for online optimization of the reconfiguration port occupancy. By caching, we can reduce the reconfiguration time required for an instance of a task on average. We therefore can also achieve schedulability of task sets, whose scheduling demand of the reconfiguration port exceeds 1.

In general, real-time scheduling of tasks on reconfigurable fabrics is of value for the domain of reactive systems. Here tasks constantly interact with the environment, often requiring real-time behavior. Thanks to its closeness to the well-researched domain of mono-processor scheduling, our approach offers a sophisticated and lightweight method to exploit real-time scheduling on slot-based execution environments implemented on reconfigurable fabrics.

Finally, we introduced algorithmic skeletons as a new and very sophisticated means to design reconfigurable systems. Based on the conceptual closeness of reconfigurable devices to parallel clusters, algorithmic skeletons can be applied to partially run-time reconfigurable devices. Algorithmic skeletons basically separate the structure from the computation itself, and thereby offer a implementation guideline to design parallelism. We then can employ the given parallelism to execute applications in time and space on reconfigurable devices in a structured manner.

On the reconfigurable fabrics we provide execution environments that accept applications given by virtue of algorithmic skeletons. The design of the environments is constraints by a well-selected set of skeletons that is supported by the environment. We discussed how different levels of granularity of the design environments can be achieved, thereby denoting that the environment and a reconfiguration manager, which supervises the environment, should be designed in close unity.

The approach still is in its infancy. Nevertheless, we already could show two major benefits of employing algorithmic skeletons for the design of reconfigurable systems. Firstly, applications given by virtue of algorithmic skeletons facilitate to explore the design on a very high level of abstraction in a structured manner. Secondly, we can achieve dynamic reconfiguration, as algorithmic skeletons offer to become the integrating means of applications and reconfigurable fabrics. Reconfigurable fabrics can accept applications, which are not known at the design time of the execution environment, if the applications are provided using the supported skeletons of the execution environment.

The four methods presented in this work all raise the level of abstraction for the design of reconfigurable computing systems. Moreover, we embedded them into a generic layered approach that allows us to target the different design challenges separately. We consider several layers in between applications and the reconfigurable fabric. In common for all methods, a reconfiguration environment constraints the reconfigurable fabric and offers reconfigurable regions, which dynamically accept tasks. The last method based on algorithmic skeletons thereby exploits execution environments in a very flexible way, also allowing dynamic reconfiguration.

The temporal domain given for reconfigurable computing can be conveniently covered by a scheduling layer. Scheduling of applications to execution environments then means execution in time and space. By virtue of reconfiguration, hardware is re-used. Within the chapter on the two slot framework, we could see that even such a simple reconfigurable architecture demands for scheduling, at least a proper temporal partitioning where partitions respect precedence constraints. The newly introduced concept of reconfiguration port scheduling then showed how to exploit real-time scheduling on run-time reconfigurable execution environments hosted on reconfigurable fabrics.

Besides these considerations on time and space, we also covered the important problem of external and internal communication on reconfigurable fabrics that dynamically host tasks in different locations. For example, we selected the specification graph approach, as communication is seamlessly integrated by dedicated communication vertices using this method. Moreover, our methods all consider the reconfiguration time, which is hardly done by others. As could be derived by several experiments, the reconfiguration phase

accounts for a significant impact on the temporal behavior of run-time reconfiguration. For the two-slot framework, we therefore investigated concepts for frequency scaling balancing reconfiguration and execution phase. The reconfiguration port scheduling even particularly exploits the reconfiguration phase to guarantee real-time behavior.

However, developing reconfigurable systems still remains a challenging task in many aspects. Among others, it is difficult because of the need for both software and hardware expertise to determine how best map applications onto reconfigurable fabrics. Nevertheless, we consider our work as a building block that particularly focused on raising the level of abstraction, towards making reconfigurable computing mature.

7.2 Outlook

The four methods presented bear various room for extensions. First of all, the two slot framework should be finally industrialized along the idea of providing it as an IP core. Moreover, additional extensions, for example, the presented concepts on reducing the size of the partial bitstreams by comparing the difference between subsequent configurations, could be particularly investigated for the simple concept of only having two slots.

Concerning the specification graph approach, we have fundamentally added the requirements for partially reconfigurable devices. Thereby, the scheduling algorithm, which we have implemented to prove the applicability, offers potential for improvement. Moreover, the specification graph approach initially targets at hardware/software codesign. A more detailed evaluation of the method including our extensions therefore would be of high interest.

For the reconfiguration port scheduling approach, we have introduced algorithms for aperiodic as well as periodic task sets. To complete the domain covered, we should investigate additional task sets including the applicability of the mono-processor scheduling algorithms. Among others, periodic tasks having dynamic priorities or task sets comprising precedence constraints are of interest. Moreover, scheduling concepts for task sets having periodic as well as aperiodic task requests can be investigated.

Finally, algorithmic skeletons, which we mainly introduced conceptually, bear the most potential for future research. The design of applications by virtue of algorithmic skeletons allows us to achieve dynamic reconfiguration in a systematic manner, which should be applied to multiple experiments. Moreover, the idea of also adapting the execution environment during run-time, reacting on requests of applications that are given by virtue of skeletons, should be investigated in more detail. Concerning the usage of the method for high performance computing, the application engineers, who usually restrain from hardware, must be convinced to employ reconfigurable devices. Algorithmic skeletons therefore must be offered in a manner the application engineers trust. Using SystemC and a library for algorithmic skeletons may be a good start. To eventually improve the data transmission from and to FPGAs, which is particularly required for the algorithmic skeleton approach, for example the Xilinx Virtex-5 front side bus approach offers a sophisticated possibility.

A The Design Tool Part-E

Due to the lack of an appropriate design tool for the design of reconfigurable systems that exploits partial run-time reconfiguration, we developed a tool that helps to abstract the reconfigurable systems under development. Among others, the tool also was motivated by providing a vendor independent design environment.

However, as Xilinx FPGAs are the only reasonable devices for exploiting partial run-time reconfiguration, the tool mainly targets on these devices. In particular, the tool automates the generation of partial bitstreams of Xilinx FPGAs, easing the otherwise often cumbersome invocation of synthesize commands.

Part-Y

The precursor of the current tool Part-E thereby was Part-Y (see Fig. A.1 for a screenshot) implemented by us in Java. The design of Part-Y was mainly motivated for providing a user-friendly means to ease the design of partial bitstreams of Xilinx FPGAs. The tool based on the modular design flow of the Xilinx Application Note 290 [Xil04] and was particularly developed to combine all the necessary information for a final bitstream generation of such a design. This core idea of combining all information in one model still is in use in the current Part-E tool. Figure A.2 displays the reduced class diagram of this model. The user builds an instance of this model by gradually providing necessary information that is required for the design of partial bitstreams. Once complete, automatic generation of partial bitstreams according to the Xilinx Application Note 290 could be invoked.

As partially reconfigurable designs require a combined consideration of time and space characteristics, the initial tool offered various views to the same model, including the possibility for consistent modification of the design from all views. The overall guideline for such structuring the different views was the Y-chart by Gajski and Kuhn [GK83]. Following the core ideas of the Y-chart, we sorted characteristics of the design into behavior, structure and geometry. For example, area/slot assignment was part of a geometrical view, dependency of and between modules could be defined in a structural oriented view, while the loading sequence of modules was part of the behavioral view. Thereby, also the design entry points were left open to the designer. More details on the tool can be found in [DRS05, DR06], [Bob07].

Part-E

To increase the maintainability of Part-Y, we decided to merge the tool into the Eclipse framework and offer it as a plug-in [Sch07]. In particular, Eclipse allowed us to abstract from fundamental details like loading and storing of designs. Moreover, to also exploit modern programming and modeling techniques of software engineering, we employed the Eclipse Graphical Modeling Framework (GMF) as framework for Part-E. GMF provides

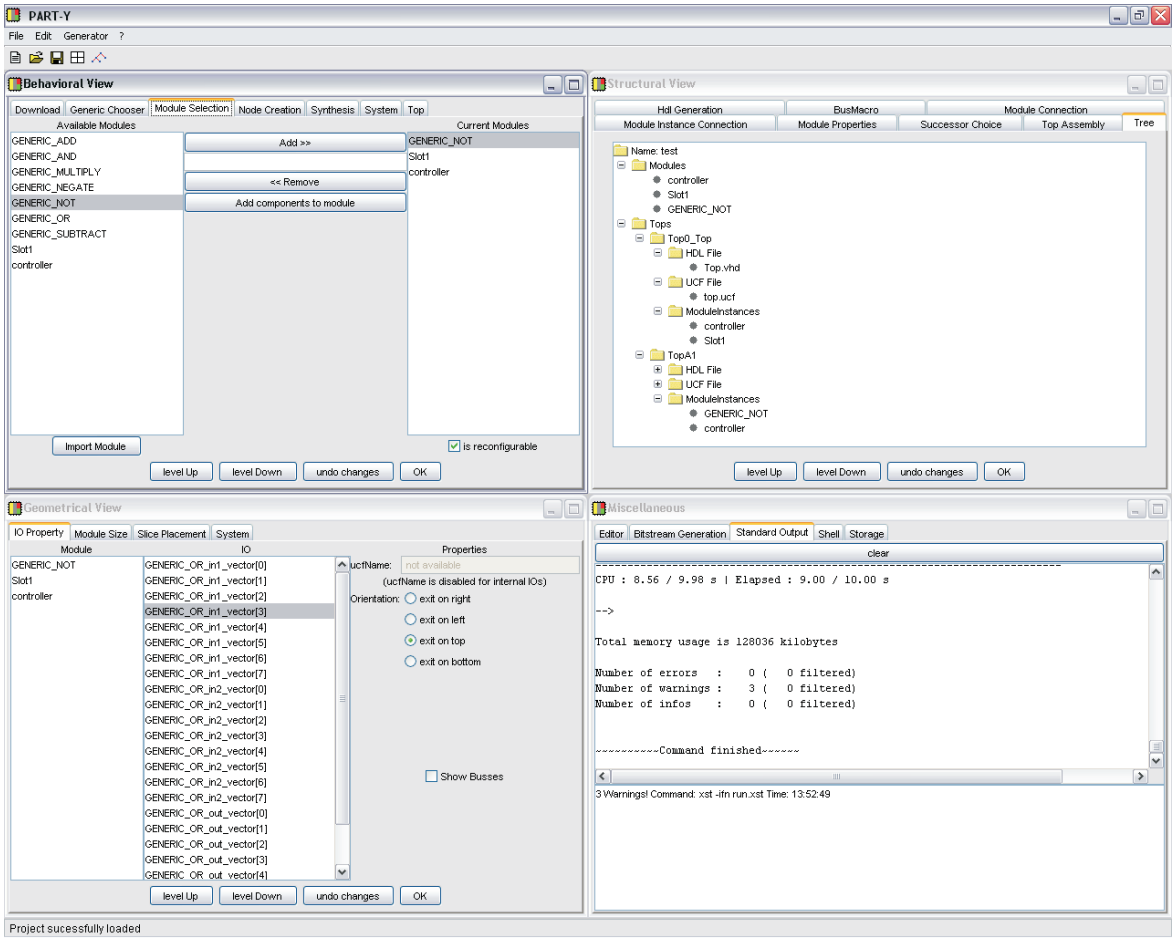

Figure A.1: Screenshot of Part-Y

design and code generation facilities for the creation of graphical editors based on an Eclipse Modeling Framework (EMF) model. Therefore, the programmer generates basic tool definitions and graphical representations of classes. Both are mapped together in a model, which then allows for generation of Java code of an editor.

For Part-E, we used an extension of the model depicted in Fig. A.2 to become the input for the GMF. Based on this model, we then fundamentally created an editor, see Fig. A.3. The editor facilitates to model reconfigurable designs. Modules, a top entity and their interconnections are described graphically. Also the reconfigurable device and its partitioning is provided in the same way. For describing the temporal domain, we added the concept of sequences. A sequence denotes the binding of a set of modules to a partitioning at a point in time. The graphical representation and design front-end thereby offers a convenient way of manipulating and evaluating the design under development. Thanks to the openness of GMF, we can easily customize this view, also accepting textual input for designs too large for a graphical representation.

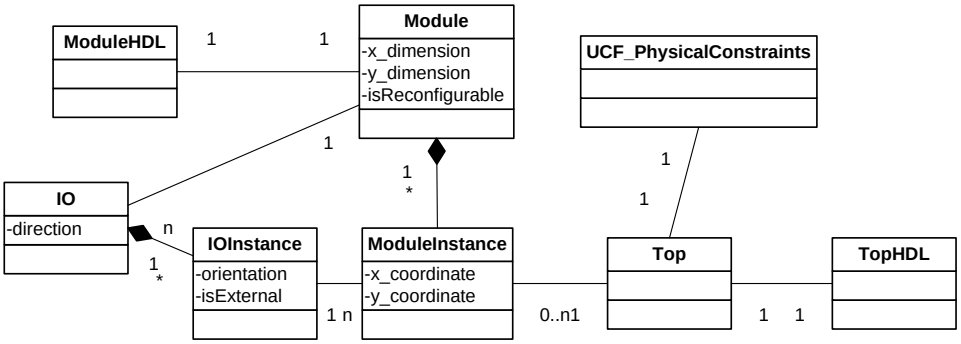

Figure A.2: Reduced class diagram of the model.

Again the single model that represents the whole characteristics of the reconfigurable system under development abstractly proved valuable. By accumulating the entire knowledge of a design in one single instance facilitates to model reconfigurable systems in a comprehensive manner on a high level of abstraction.

The tool enables to combine HDL-modules, which each represent a task to be executed dynamically on the FPGA, to so-called top entities, which assemble and structure the design including global logic, etc. Additionally, physical constraints like the position of the tasks on the FPGA are necessary. The tool eases the generation of those physical constraints, including the placement of inter-module connections. If necessary, wrapper VHDL files are generated that facilitate to meet the specific requirements of partial designs. The generation can easily be extended to meet future reconfigurable devices, also of different vendors than Xilinx. Finally, the integrated call of automatically generated commands following the Xilinx EarlyAccess Design Flow enables the convenient generation and downloading of the partial and full bitstreams.

A screenshot of Part-E is given in Fig. A.3. We depict a simple design that illustrates most of the modeling features of Part-E. In detail, the model consists of one reconfigurable and one static area. The reconfigurable area comprises of two modules onto which tasks can be loaded alternatively. Moreover, we model the entire design, including the connections between the modules and to the external connection via the top design entity. The default Xilinx Bitfile Generator generates the complete VHDL for the design including the wiring on the top level, synthesis and implements it and finally generates the corresponding bitfiles based on the Xilinx EarlyAccess design flow [LBM+06].

In the meantime, several improvements could be added to the design environment. Among others, the extension points of the Eclipse framework allow for virtually any design flow and HDL parser, which is given as a plug-in. For example, we use a VHDL parser to derive the information for external communication given in VHDL files. We also use a plug-in to highlight the syntax of the input language. Moreover, the possibility to execute the synthesize, mapping, place & route, and the final bitstream generation remotely on several machines in parallel was implemented. Thereby, the partial design flow can easily be parallelized, as the steps towards bitstream generation must be done individually for every partially reconfigurable module.

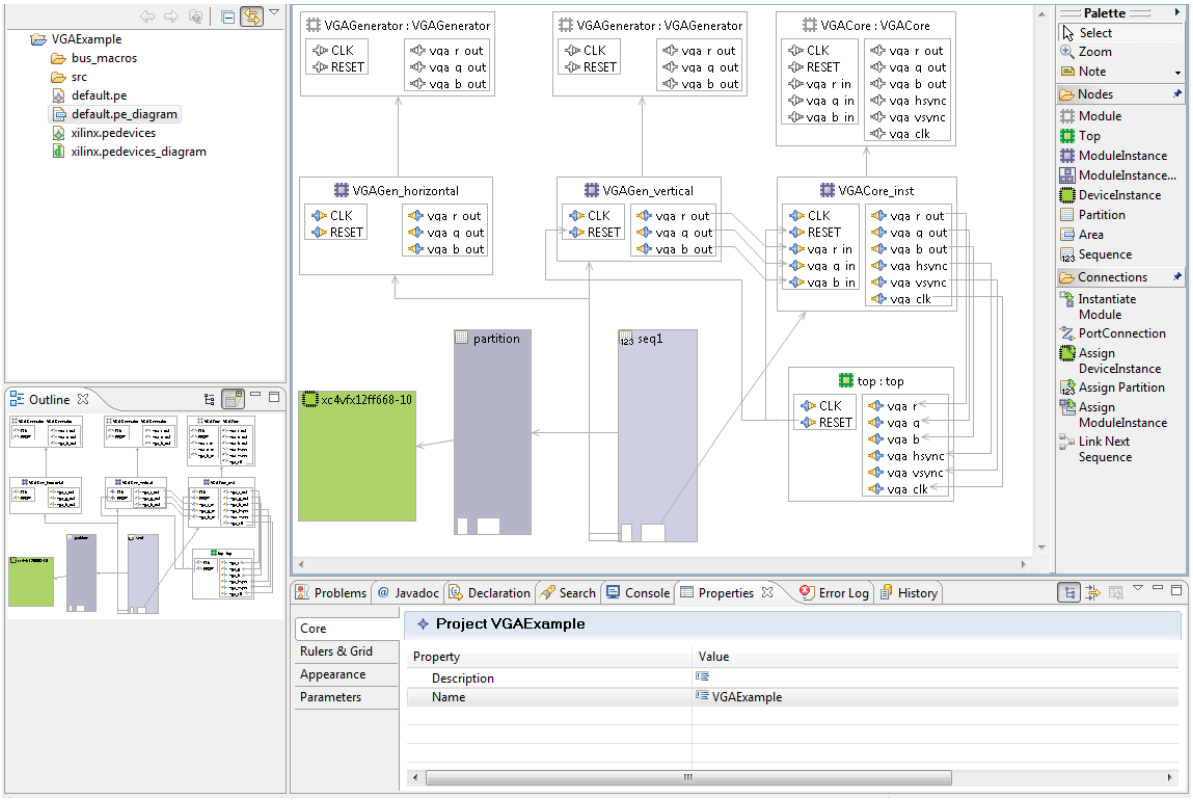

Figure A.3: Screenshot of Part-E

To summarize, Part-E basically focuses on automated generation of partial run-time reconfigurable designs, particularly including the generation of (partial) bitstreams. Therefore, Part-E takes raw HDL input files and generates wrappers based on the graphically given information of the structure and behavior of the design under development. Primarily, it wraps the obstacles of partial bitstream generation. Part-E thus was developed to provide an integrated means of designing partially reconfigurable systems.

The current version of Part-E can be accessed on Sourceforge: `parte.sf.net`.

Author's Own Publications

[DB05] Florian Dittmann and Christophe Bobda. Temporal Placement on Mesh-Based Coarse Grain Reconfigurable Systems Using the Spectral Method. In Achim Rettberg, Mauro C. Zanella, and Franz Josef Rammig, editors, *From Specification to Embedded Systems Application, Proceedings of the IESS*, pages 301–310. Kluwer Academic Publishers, 15 - 17 August 2005.

[DF07a] Florian Dittmann and Stefan Frank. Caching in Real-time Reconfiguration Port Scheduling. In *Proceedings of the FPL 2007*. IEEE, 27 - 29 August 2007.

[DF07b] Florian Dittmann and Stefan Frank. Hard real-time reconfiguration port scheduling. In *Proceedings of the Design, Automation and Test in Europe*, San Jose, CA, USA, 2007. EDA Consortium.

[DG06a] Florian Dittmann and Marcelo Götz. Applying Single Processor Algorithms to Schedule Tasks on Reconfigurable Devices Respecting Reconfiguration Times. In *13th Reconfigurable Architectures Workshop (RAW 2006)*, 25 - 26 April 2006.

[DG06b] Florian Dittmann and Marcelo Götz. Reconfiguration time aware processing on fpgas. In *In Proceedings of the Dagstuhl Seminar N^o 06141 on Dynamically Reconfigurable Architectures*, 2006.

[DGR07] Florian Dittmann, Marcelo Götz, and Achim Rettberg. Model and Methodology for the Synthesis of Heterogeneous and Partially Reconfigurable Systems. In *14th Reconfigurable Architectures Workshop (RAW 2007)*. IEEE, 25 - 26 March 2007.

[DH05] Florian Dittmann and Markus Heberling. Placement of Intermodule Connections on Partially Reconfigurable Devices. In *Proceedings of the 18th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 236–241, 4 - 7 September 2005.

[DH06] Florian Dittmann and Tales Heimfarth. Clock Frequency Variation of Partially Reconfigurable Systems. In *19th International Conference on Architecture of Computing Systems: Workshop Proceedings*, 16 March 2006.

[Dit03] Florian Dittmann. Eine High Level Synthese für eine neue synchrone bitserielle Pipeline-Datenfluss-Architektur. In *Informatiktage 2003 - Fachwis-*

senschaftlicher Informatik - Kongress. Gesellschaft für Informatik in Zusammenarbeit mit der Computer Zeitung, Konradin-Verlagsgruppe, 7 - 8 November 2003.

[Dit05] Florian Dittmann. Efficient execution on reconfigurable devices using concepts of pipelining. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 717–718, 24 - 26 August 2005.

[Dit07] Florian Dittmann. Algorithmic skeletons for the programming of reconfigurable systems. In *Proceedings of the SEUS 2007*, 7 - 8 May 2007.

[DKR03] Florian Dittmann, Bernd Kleinjohann, and Achim Rettberg. Efficient Bit-Serial Constant Multiplication for FPGAs. In *Proceedings of the 11th NASA Symposium VLSI Design*, May 2003.

[DR04] Florian Dittmann and Achim Rettberg. A Self-Controlled And Dynamically Reconfigurable Architecture. In Bernd Kleinjohann, Guang R. Gao, Hermann Kopetz, Lisa Kleinjohann, and Achim Rettberg, editors, *Proceedings of IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'04)*. Kluwer Academic Publishers, 23 - 26 August 2004.

[DR06] Florian Dittmann and Achim Rettberg. Design of Partially Reconfigurable Systems: From Abstract Modeling to Practical Realization. In *1st International Workshop on Reconfigurable Computing Education*, 1 March 2006. available at: http://isvlsi06.itiv.uni-karlsruhe.de/RCe-22.pdf.

[DRLZ04] Florian Dittmann, Achim Rettberg, Thomas Lehmann, and Mauro C. Zanella. Invariants for Distributed Local Control Elements of a New Synchronous Bit-Serial Architecture. In *Proceedings of the Delta*, 28 - 30 January 2004.

[DRS05] Florian Dittmann, Achim Rettberg, and Fabian Schulte. A Y-Chart Based Tool for Reconfigurable System Design. In *Workshop on Dynamically Reconfigurable Systems (DRS) 2005*, 17 March 2005.

[DRS+07] Florian Dittmann, Franz J. Rammig, Martin Streubühr, Christian Haubelt, Andreas Schallenberg, and Wolfgang Nebel. Exploration, partitioning and simulation of reconfigurable systems. *it - Information Technology*, 7(3):149–156, June 2007.

[DRW05] Florian Dittmann, Achim Rettberg, and Raphael Weber. Path concepts for a reconfigurable bit-serial synchronous architecture. In *Proceedings of the 2005 IFIP International Conference on Embedded And Ubiquitous Computing (EUC'2005)*, 6 - 9 December 2005.

[DRW06] Florian Dittmann, Achim Rettberg, and Raphael Weber. Towards the implementation of path concepts for a reconfigurable bit–serial synchronous architecture. In *Proceedings of the 3rd International Conference on ReConFigurable Computing and FPGA's*, pages 262–269. IEEE, September 2006.

[DRW07a] Florian Dittmann, Achim Rettberg, and Raphael Weber. Latency optimization for a reconfigurable, self-timed and bit-serial architecture. In *Proceedings of the ERSA 2007*, 25 - 28 June 2007.

[DRW07b] Florian Dittmann, Achim Rettberg, and Raphael Weber. Optimization techniques for a reconfigurable self-timed and bit-serial architecture. In *Proceedings of the SBCCI 2007*, 3 - 6 September 2007.

[GD06a] Marcelo Götz and Florian Dittmann. Reconfigurable microkernel-based RTOS: Mechanisms and methods for run-time reconfiguration. In *Proceedings of the 3rd International Conference on ReConFigurable Computing and FPGAs 2006 (ReConFig'06)*, pages 12–19. IEEE, 2006.

[GD06b] Marcelo Götz and Florian Dittmann. Scheduling reconfiguration activities of run-time reconfigurable RTOS using an aperiodic task server. In *Proceedings of the ARC 2006*, 1 - 3 March 2006.

[GDP06] Marcelo Götz, Florian Dittmann, and Carlos E. Pereira. Deterministic mechanism for run-time reconfiguration activities in an RTOS. In *Proceedings of the 4th International IEEE Conference on Industrial Informatics (INDIN 2006)*. IEEE, 2006.

[GDX07] Marcelo Götz, Florian Dittmann, and Tao Xie. Dynamic relocation of hybrid tasks: A complete design flow. In *Proceedings of Reconfigurable Communication-centric SoCs (ReCoSoc'07)*, 18 - 20 June 2007.

[ID05] Stefan Ihmor and Florian Dittmann. Optimizing interface implementation costs using runtime reconfigurable systems. In Toomas Plaks, editor, *Proceedings of the 2005 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05)*, pages 85–91, Monte Carlo Resort, Las Vegas, Nevada, USA, 26 - 30 June 2005.

[RDLZ04] Achim Rettberg, Florian Dittmann, Thomas Lehmann, and Mauro C. Zanella. A New High-Level Synthesis Approach of a Synchronous Bit-Serial Architecture. In Dominik Stoffel and Wolfgang Kunz, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 34 – 43. Shaker Verlag, 2004.

[RDZL03] Achim Rettberg, Florian Dittmann, Mauro C. Zanella, and Thomas Lehmann. Towards a High-Level Synthesis of Reconfigurable Bit-Serial Architectures. In *Proceedings of the 16th Symposium on Integrated Circuits and System Design (SBCCI)*, 8 - 11 September 2003.

[RDZL04] Achim Rettberg, Florian Dittmann, Mauro C. Zanella, and Thomas Lehmann. MACT – A Reconfigurable Pipeline Architecture. In Technology Siemens ICM MP CTO TI and Inovation, editors, *Technologies-to-Watch*, No. 21, pages 15 – 17, München, Germany, August 2004. For internal use only.

[WD06] Alexander Warkentin and Florian Dittmann. Data Transfer Protocols for a Two Slot Based Reconfigurable Platform. In *Proceedings of the Reconfigurable Communication–centric SoCs (ReCoSoC)*, 2006.

Bibliography

[ABH+94] Andreas Ast, Jürgen Becker, Reiner W. Hartenstein, Rainer Kress, Helmut Reinig, and Karin Schmidt. Data-Procedural Languages for FPL-based Machines. In *Proceedings of the 4th International Workshop on Field-Programmable Logic and Applications*, pages 183–195. Springer-Verlag, 1994.

[ABI06] John Augustine, Sudarshan Banerjee, and Sandy Irani. Strip packing with precedence constraints and strip packing with release times. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 180–189, New York, NY, USA, 2006. ACM Press.

[ABK+04] Ali Ahmadinia, Christophe Bobda, Dirk Koch, Mateusz Majer, and Jürgen Teich. Task scheduling for heterogeneous reconfigurable computers. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 22–27, Pernambuco, Brazil, 2004. ACM Press.

[ABR+93] A. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[ABRW91] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, 1991.

[ABT04] Ali Ahmadinia, Christophe Bobda, and Jürgen Teich. A Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *ARCS*, volume 3894 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2004.

[Ama06] Hideharu Amano. A survey on dynamically reconfigurable processors. *IEICE Transactions on Communications*, E89-B:3179–3187, 2006.

[APC03] Kostas Masselos Antti Pelkonen and Miroslav Cupák. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. *Proceedings of International Symposium on Parallel and Distributed Processing (Reconfigurable Architectures Workshop)*, pages 174–181, April 2003.

[AS93] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, 1993.

[Atm02] Atmel. FPSLIC on-chip partial reconfiguration of the embedded AT40K FPGA. Technical report, San Jose, CA, USA, 2002.

[AW02] Amir H. Abdekhodaee and Andrew Wirth. Scheduling parallel machines with a single server: some solvable cases and heuristics. *Comput. Oper. Res.*, 29(3):295–315, 2002.

[AWG06] Amir H. Abdekhodaee, Andrew Wirth, and Heng-Soon Gan. Scheduling two parallel machines with a single server: the general case. *Comput. Oper. Res.*, 33(4):994–1009, 2006.

[BAK96] Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder. *Splash 2: FPGAs in a custom computing machine*. IEEE Computer Society Press, Los Alamitos, California, USA, 1996.

[BAM+05] Christophe Bobda, Ali Ahmadinia, Mateusz Majer, Jürgen Teich, Sándor P. Fekete, and Jan van der Veen. Dynoc: A dynamic infrastructure for communication in dynamically reconfigurable devices. In *Proceedings of the FPL*, pages 153–158, 2005.

[BAR+05] Christophe Bobda, Ali Ahmadinia, Kurapati Rajesham, Mateusz Majer, and Adronis Niyonkuru. Partial Configuration Design and Implementation Challenges on Xilinx Virtex FPGAs. In Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Christian Hochberger, Thomas Martinetz, Christian Müller-Schloer, Hartmut Schmeck, Theo Ungerer, and Rolf P. Würtz, editors, *ARCS Workshops*, pages 61–66. VDE Verlag, 2005.

[BBD05] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil Dutt. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 335–340, San Diego, California, USA, 2005. ACM Press.

[BBHN04] Brandon Blodget, Christophe Bobda, Michael Hübner, and Adronis Niyonkuru. Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *Proceedings of the FPL 2004*, volume 3203 of *Lecture Notes in Computer Science*, pages 801–810. Springer, 2004.

[BC04] Khaled Benkrid and Danny Crookes. From application descriptions to hardware in seconds: a logic-based approach to bridging the gap. *IEEE Trans. VLSI Syst.*, 12(4):420–436, 2004.

[BD98] Gordon J. Brebner and Adam Donlin. Runtime Reconfigurable Routing. In *IPPS/SPDP Workshops*, pages 25–30, Orlando, Florida, USA, 1998.

[BDD+99] Kiran Bondalapati, Pedro C. Diniz, Phillip Duncan, John Granacki, Mary Hall, Rajeev Jain, and Heidi Ziegler. DEFACTO: A Design Environment for Adaptive Computing Technology. In *Parallel and Distributed Processing. 11th IPPS/SPDP Workshops held in conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing.*, pages 570–578. Springer Verlag, 1999.

[BDFK+02] Peter Brucker, Clarisse Dhaenens-Flipo, Sigrid Knust, Svetlana A. Kravchenko, and Frank Werner. Complexity results for parallel machine problems with a single server. *Journal of Scheduling*, 5:429–457, 2002.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, New York, NY, 1999.

[BEGGK07] Duncan Buell, Tarek El-Ghazawi, Kris Gaj, and Volodymyr Kindratenko. High-performance reconfigurable computing. *IEEE Computer*, 40(3), March 2007. Guest Editors' Introduction.

[BGD+04] Nikhil Bansal, Sumit Gupta, Nikil Dutt, Alex Nicolau, and Rajesh Gupta. Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures. In *Proceedings of the DATE*, February 2004.

[BHH+07] Jürgen Becker, Michael Hübner, Gerhard Hettich, Rainer Constapel, Joachim Eisenmann, and Jürgen Luka. Dynamic and partial fpga exploitation. In *Proceedings of the IEEE*, volume 95, pages 438–452, February 2007.

[BJRK+03] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillian, and Prasanna Sundararajan. A Self-reconfiguring Platform. In *Proceedings of the International Conference on Field Programmable Logic*, September 2003.

[BK96] George Horatiu Botorog and Herbert Kuchen. Efficient Parallel Programming with Algorithmic Skeletons. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 718–731, London, UK, 1996. Springer-Verlag.

[BLPG01] Jürgen Becker, Nicolas Liebau, Thilo Pionteck, and Manfred Glesner. Efficient Mapping of Pre-synthesized IP-Cores onto Dynamically Reconfigurable Array Architectures. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pages 584–589. Springer-Verlag, 2001.

[BMA+05] Christophe Bobda, Mateusz Majer, Ali Ahmadinia, Thomas Haller, André Linarth, Jürgen Teich, and Jan van der Veen. The Erlangen Slot Machine: A Highly Flexible FPGA-Based Reconfigurable Platform. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 319–320. IEEE, 2005.

[Bob03] Christophe Bobda. *Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement*. PhD thesis, University Paderborn, Heinz Nixdorf Institute, 2003.

[Bob07] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications*. Springer, 2007.

[Bon02] Kiran Bondalapati. *Modeling and Mapping for Dynamically Reconfigurable Architectures*. PhD thesis, University of Southern California, 2002.

[BRV89] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. pages 301–309. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[BSC+00] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A matlab compiler for distributed, heterogeneous, reconfigurable computing systems. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[BTT98] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using Evolutionary Algorithms. *J. Design Automation for Embedded Systems*, 3(1):23–58, January 1998.

[But97] Girogio C. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[But04] Girogio C. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2004.

[BW06] Andrew Butterfield and Jim Woodcock. A "hardware compiler" semantics for handel-c. *Electr. Notes Theor. Comput. Sci.*, 161:73–90, 2006.

[CCEBM04] Ewerson Carvalho, Ney Calazans, ao Eduardo Bri and Fernando Moraes. PaDReH - A Framework for the Design and Implementation of Dynamically and Partially Reconfigurable Systems. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 10–15, New York, NY, USA, 2004. ACM Press.

[CCH+00] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (score). In *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 605–614, London, UK, 2000. Springer-Verlag.

[CCKH00] Kathrin Compton, James Cooley, Stephen Knol, and Scott Hauck. Configuration Relocation and Defragmentation for FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 279–280, Napa Valley, CA, USA, April 2000.

[CDF+86] W. S. Carter, K. Duong, R. H. Freeman, H. C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze. A User Programmable Reconfigurable Gate Array. In *IEEE Customs Circuits Conference*, pages 233–235, 1986.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[CKRB03] Chen Chang, Kimmo Kuusilinna, Brian Richards, and Robert W. Brodersen. Implementation of bee: a real-time large-scale hardware emulation engine. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 91–99, New York, NY, USA, 2003. ACM Press.

[Cla07] Peter Clarke. Tilera starts shipping 64-way multiprocessor. *EE Times*, August 2007.

[CLC+02] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol, and Scott Hauck. Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):209–220, 2002.

[CMS06] Christopher Claus, Florian Helmut Müller, and Walter Stechele. Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro. In Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, and Erik Maehle, editors, *ARCS Workshops*, volume 81 of *LNI*, pages 122–131. GI, 2006.

[Col89] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/The MIT Press, London, UK/Cambridge, Massachusetts, USA, 1989.

[Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[CSR+99] Paul Chow, Soon Ong Seo, Jonathan Rose, Kevin Chung, Gerard Páez-Monzón, and Immanuel Rahardja. The design of a sram-based field-programmable gate array - part II: circuit design and layout. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(3):321–330, 1999.

[CWB05] Chen Chang, John Wawrzynek, and Robert W. Brodersen. Bee2: A high-end reconfigurable computing system. *IEEE Des. Test*, 22(2):114–125, 2005.

[CWG+98] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Donald E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 55–64. ACM Press, 1998.

[CZMS07] Christopher Claus, Johannes Zeppenfeld, Florian Müller, and Walter Stechele. Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 498–503, San Jose, CA, USA, 2007. EDA Consortium.

[DAD+04] André DeHon, Joshua Adams, Michael DeLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael C. Vanier, and Michael G. Wrighton. Design patterns for reconfigurable computing. In *FCCM*, pages 13–23, 2004.

[Dan01] Marco Danelutto. On skeletons and design patterns. In *Proceedings of the International Conference PARCO 2001*, 2001.

[Dan04] Klaus Danne. Distributed Arithmetic FPGA Design with Online Scalable Size and Performance. In *Proceedings of 17th Symposium on Integrated Circuits and Systems Design (SBCCI 2004)*, pages 135–140. ACM Press, 7 - 11 September 2004.

[Dan06] Klaus Danne. *Real-Time Multitasking in Embedded Systems Based on Reconfigurable Hardware*. PhD thesis, University Paderborn, Heinz Nixdorf Institute, 2006.

[DB04] Klaus Danne and Christophe Bobda. Dynamic Reconfiguration of Distributed Arithmetic Controllers: Design Space Exploration and Trade-off Analysis. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW'04)*, Santa Fe, USA, 26 - 27 April 2004. IEEE Computer Society.

[DBK03] Klaus Danne, Christophe Bobda, and Heiko Kalte. Run-time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL 2003)*, September 2003.

[DeH96a] André; DeHon. DPGA Utilization and Application. In *Proceedings of the 1996 ACM fourth international symposium on Field-Programmable Gate Arrays*, pages 115–121. ACM Press, 1996.

[DeH96b] André DeHon. Reconfigurable architectures for general-purpose computing. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[DEM+00] Oliver Diessel, Hossam ElGindy, Martin Middendorf, Hartmut Schmeck, and Bernd Schmidt. Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs. *IEE Proceedings – Computer and Digital Techniques (Special Issue on Reconfigurable Systems)*, 147(3):181–188, 2000.

[DH98] Oliver Diessel and ElGindy Hossam. On Scheduling Dynamic FPGA Reconfigurations. In Kenneth A. Hawick and Heath A. James, editors, *Proceedings of the Fifth Australasian Conference on Parallel and Real-Time Systems (PART'98)*, pages 191–200. Springer-Verlag, 1998.

[dLKNH+04] Fernanda Gusmão de Lima Kastensmidt, Gustavo Neuberger, Renato Fernandes Hentschke, Luigi Carro, and Ricardo Reis. Designing and testing fault-tolerant techniques for sram-based fpgas. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 419–432, New York, NY, USA, 2004. ACM Press.

[DP06] Klaus Danne and Marco Platzner. Executing hardware tasks on dynamically reconfigurable devices under real-time conditions. In *Proceedings of the FPL 2006*, 2006.

[DSG05] Nij Dorairaj, Eric Shiflet, and Mark Goosman. Planahead software as a platform for partial reconfiguration. *Xcell Journal*, (4), 2005.

[DST99] Deepali Deshpande, Arun K. Somani, and Akhilish Tyagi. Configuration Caching Vs data caching for Striped FPGAs. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 206–214. ACM Press, 1999.

[DW99] André; DeHon and John Wawrzynek. Reconfigurable computing: what, why, and implications for design automation. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 610–615, New York, NY, USA, 1999. ACM Press.

[EBTB63] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Paralelle processing in a restructurable computer system. *Transactions on Electronic Computers*, pages 747–755, 1963.

[EH94] James G. Eldredge and Brad L. Hutchings. RRANN: A hardware implementation of the backpropagation algorithm using reconfigurable FPGAs.

In *IEEE World Conference on Computational Intelligence*, Orlando, FL, 1994.

[EP02] Michael Eisenring and Marco Platzner. A Framework for Run-time Reconfigurable Systems. *The Journal of Supercomputing*, 21:145–159, 2002.

[ESS+96] H. ElGindy, A. K. Somani, H. Schroder, H. Schmeck, and A. Spray. RMB – A Reconfigurable Multiple Bus Network. In *HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96)*, page 108, Washington, DC, USA, 1996. IEEE Computer Society.

[ESSA00] John M. Emmert, Charles E. Stroud, Brandon Skaggs, and Miron Abramovici. Dynamic fault tolerance in fpgas via partial reconfiguration. In *FCCM*, pages 165–174. IEEE Computer Society, 2000.

[Est02] Gerald Estrin. Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer. *IEEE Ann. Hist. Comput.*, 24(4):3–9, 2002.

[EV62] G. Estrin and C. R. Viswanathan. Organization of a "fixed-plus-variable" structure computer for computation of eigenvalues and eigenvectors of real symmetric matrices. *J. ACM*, 9(1):41–60, 1962.

[FC05] W. Fu Fu and Katherine Compton. An execution environment for reconfigurable computing (abstract only). In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 149–158. IEEE, 2005.

[FKT01] Sandor P. Fekete, E. Köhler, and Jürgen Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Proc. DATE 2001, Design, Automation and Test in Europe*, pages 658–665, Munich, Germany, March 13-16 2001. IEEE Computer Society Press.

[Fra06] Stefan Frank. Evaluierung preemtiver Scheduling-Algorithmen für die Rekonfigurierung von FPGAs. Studienarbeit, Heinz Nixdorf Insitute, University of Paderborn, Germany, June 2006.

[Fra07] Stefan Frank. Einsatz algorithmischer Skelette zur Generierung partiell dynamisch rekonfigurierbarer Schaltungen. Diplomarbeit, Heinz Nixdorf Insitute, University of Paderborn, Germany, 2007.

[GCL02] J. Gause, P. Y. K. Cheung, and W. Luk. Reconfigurable Shape-Adaptive Template Matching Architectures. In *Proceedings 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. FCCM 2002*, pages 98–107. IEEE Comput. Soc., 2002.

[GG05] Maya Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 2005.

[GGKK03] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[GHK+91] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, Douglas Sweely, and Daniel Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1):81–89, 1991.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Interactability: A Guide to the Theroy of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

[GK83] Daniel D. Gajski and R. H. Kuhn. Guest Editor's Introduction: New VLSI Tools. *IEEE Computer*, December 1983.

[Goe02] Richard Goering. Platform-based Design: A Choice, Not a Panacea. *EE Times*, September 2002.

[Göt07] Marcelo Götz. *Run-time Reconfigurable RTOS for Reconfigurable Systems-on-Chip*. PhD thesis, University Paderborn, Heinz Nixdorf Institute, 2007.

[GP06] Björn Griese and Mario Porrmann. A reconfigurable ethernet switch for self-optimizing communication systems. In *Proceedings of the 1st IFIP International Conference on Biologically Inspired Cooperative Computing (BICC 2006)*, volume 216 of *IFIP International Federation for Information Processing*, pages 115–124, Boston, MA, USA, August 2006. Springer.

[GR01] Varghese George and Jan M. Rabaey. *Low-energy FPGAs: architecture and design*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[GRP06] Marcelo Götz, Achim Rettberg, and Carlos E. Pereira. Run-Time Reconfigurable Real-Time Operting System For Hybrid Execution Platforms. In *Proceedings of the 12th IFAC Symposium on Information Control Problems in Manufacturing*, 2006.

[GSAK00] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[GSB+00] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33:70–77, April 2000.

[GSC04] Sathish Gopalakrishnan, Lui Sha, and Marco Caccamo. Hard Real-Time Communication in Bus-Based Networks. In *RTSS*, pages 405–414, 2004.

[GSP02] Dhrubajyoti Goswami, Ajit Singh, and Bruno Richard Preiss. From design patterns to parallel architecture skeletons. *Journal of Parallel and Distributed Computing*, 62(4):669–695, April 2002.

[GTF+02] Eike Grimpe, Bernd Timmermann, Tiemo Fandrey, Ramon Biniasch, and Frank Oppenheimer. SystemC Object-Oriented Extensions and Synthesis Features. In *Forum on Design Languages FDL '02*, Sept 2002.

[GV00] Satish Ganesan and Ranga Vemuri. An Integrated Temporal Partitioning and Partial Reconfiguration Technique for Design Latency Improvement. In *Proceedings of the IEEE Design, Automation and Test in Europe (DATE '00)*, Paris, France, 2000.

[GZR99] Varghese George, Hui Zhang, and Jan Rabaey. The design of a low energy FPGA. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 188–193, New York, NY, USA, 1999. ACM Press.

[HAA+96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[Hal70] Kenneth M. Hall. An r-dimensional Quadratic Placement Algorithm. *Managment Science*, 17(3):219–229, November 1970.

[Har01a] Reiner Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 01)*, March 2001.

[Har01b] Reiner Hartenstein. Coarse Grain Reconfigurable Architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC 2001*, pages 564–569, January 2001.

[Har03] Reiner Hartenstein. Toward Reconfigurable Computing via Concussive Paradigm Shifts, 29 August 2003. invited presentation.

[Hau98] Scott Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74. ACM Press, 1998.

[Hau05] Christian Haubelt. *Automatic Model-Based Design Space Exploration for Embedded Systems – A System Level Approach.* PhD thesis, University of Erlangen-Nuremberg, Germany, July 2005.

[HBB04] Michael Huebner, T. Becker, and Jürgen Becker. Real-time LUT-based Network Topologies for dynamic and partial FPGA Self-Reconfiguration. In *Proceedings of 17th Symposium on Integrated Circuits and Systems Design (SBCCI 2004)*. ACM Press, 7 - 11 September 2004.

[Hei08] Tales Heimfarth. *NanoOS: a Distributed Operating System for Sensor Networks Based on Biologic Inspired Methods*. PhD thesis, University Paderborn, Heinz Nixdorf Institute, 2008.

[HFHK97] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. The Chimaera Reconfigurable Functional Unit. In *Proceedings of the 1997 IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[HHHN00] Reiner Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array. In *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*, pages 163–168, New York, NY, USA, 2000. ACM Press.

[HK95] Reiner W. Hartenstein and Rainer Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Proceedings of the 1995 conference on Asia Pacific design automation (ASP-DAC'95)*, page 77, Makuhari, Chiba, Japan, September 1995. ACM Press.

[HKKP07] Jens Hagemeyer, Boris Kettelhoit, Markus Köster, and Mario Porrmann. Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs. In *Proc. of the Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA '07)*, Las Vegas, USA, 25 - 28 June 2007.

[HLTP02] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 343–348, New York, NY, USA, 2002. ACM Press.

[HMM04] J. Harkin, T. M. McGinnity, and L. P. Maguire. Modeling and optimizing run-time reconfiguration using evolutionary computation. *ACM Transactions on Embedded Computing Systems*, 3(4):661–685, November 2004.

[Hoa93] D. T. Hoang. Seachring genetic databases on Splash 2. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (Cat. No. 93TH0535-5)*, pages 185–91. IEEE Comput. Soc. Press., 1993.

[HOGT05] Christian Haubelt, Stephan Otto, Cornelia Grabbe, and Jürgen Teich. A System-Level Approach to Hardware Reconfigurable Systems. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 298–301, Shanghai, China, January 2005. IEEE.

[HOH+07] A. Herrholz, F. Oppenheimer, P.A. Hartmann, A. Schallenberg, W. Nebel, C. Grimm, M. Damm, J. Haase, F. Herrera F. Brame, E. Villar, I. Sander, A. Jantsch, A.-M. Fouilliart, and M. Martinez. The ANDRES Project: Analysis and Design of Run-Time Reconfigurable, Heterogeneous Systems. In *17th International Conference on Field Programmable Logic and Applications*. IEEE, 27 - 29 August 2007.

[Hol92] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[Hor74] W. Horn. Some simple scheduling algorithms. *Naval Reasearch Logistics Quarterly*, 21, 1974.

[HPS00] Nicholas G. Hall, Chris N. Potts, and Chelliah Sriskandarajah. Parallel machine scheduling with a common server. *Discrete Appl. Math.*, 102(3):223–243, 2000.

[HTRE02] Christian Haubelt, J"urgen Teich, Kai Richter, and Rolf Ernst. Modellierung rekonfigurierbarer Systemarchitekturen. In Jürgen Ruf, editor, *GIITGGMM-Workshop 2002 - Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 163–171, Tübingen, Germany, February 2002. Shaker Verlag.

[Hüb07] Michael Hübner. *Dynamisch und partiell rekonfigurierbare Hardwaresystemarchitektur mit echzeitfähiger On-Demand Funktionalität*. PhD thesis, Universität Karlsruhe (TH), 2007.

[HV04] Manish Handa and Ranga Vemuri. An efficient algorithm for finding empty space for online FPGA placement. In *Proceedings of the 41st annual conference on Design automation*, pages 960–965. ACM Press, 2004.

[Ihm06] Stefan Ihmor. *Modeling an Automated Synthesis of Reconfigurable Interfaces*. PhD thesis, University Paderborn, Heinz Nixdorf Institute, 2006.

[Jac07] Brian Jackson. Partial Reconfiguration Design with PlanAhead 9.2. Technical report, Xilinx, 2 August 2007.

[Jai95] A. K. Jain. Convolution on Splash 2. In *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, page 204, Washington, DC, USA, 1995. IEEE Computer Society.

[JN03] ao M. P. Cardoso Jo and Horácio C. Neto. Compilation for fpga-based reconfigurable hardware. *IEEE Des. Test*, 20(2):65–75, 2003.

[JR97a] Nikolaj Janzen and Franz Josef Rammig. Timing Analysis with FPGA-based Emulation Systems. In *Proceedings of the 9th European Simulation Symposium*, 1997.

[JR97b] Nikolaj Janzen and Franz Josef Rammig. Zeittreue prototypenanalyse in fpga-basierten emulationssystemen. In *5. GI/ITG/GMM Workshop. Methoden des Entwurfs und der Verifikation digitaler Systeme*, 1997.

[JSM93] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemtive scheduling of periodic tasks with varying execution priority. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 129–139, December 1993.

[JYLC00] Byungil Jeong, Sungjoo Yoo, Sunghyun Lee, and Kiyoung Choi. Hardware-software cosynthesis for run-time incrementally reconfigurable FPGAs. In *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*, pages 169–174, New York, NY, USA, 2000. ACM Press.

[Kal02] Bodo Kalthoff. *Einsatz von algorithmischen Skeletten im Scheduling massiv paralleler Systeme*. Dissertation, University Paderborn, 2002.

[Kal04] Heiko Kalte. *Einbettung dynamisch rekonfigurierbarer Hardwarearchitekturen in eine Universalprozessorumgebung*. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Schaltungstechnik, 2004. EUR 38,-, ISBN 3-935433-48-4.

[KCR06] Fernanda Lima Kastensmidt, Luigi Carro, and Ricardo Reis. *Fault-Tolerance Techniques for SRAM-Based FPGAs*, volume 32 of *Frontiers in Electronic Testing*. Springer, 2006.

[Keb06] Udo Kebschull. Applications of FPGA reconfiguration for experiments in high energy physics. In Wolfgang Karl, Jürgen Becker, Karl-Erwin Grosspietsch, Christian Hochberger, and Erik Maehle, editors, *ARCS Workshops*, LNI, page 172. GI, 2006.

[KGS+07] Mahendra Kumar, Angamuthu Ganesan, Sundeep Singh, Frank May, and Jürgen Becker. H.264 Decoder at HD Resolution on a Coarse Grain Dynamically Reconfigurable Architecture. In *Proceedings of the FPL 2007*. IEEE, 27 - 29 August 2007.

[KIM78] H. Kise, T. Ibaraki, and H. Mine. A solvable case of the one machine scheduling problem with ready and due times. *Operations Research*, 26(1):121–126, 1978.

[KKMB02] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4):605–627, 2002.

[KKP05a] Markus Köster, Heiko Kalte, and Mario Porrmann. Run-Time Defragmentation for Partially Reconfigurable Systems. In *Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 109–115, 2005.

[KKP05b] Markus Köster, Heiko Kalte, and Mario Porrmann. Task placement for heterogeneous reconfigurable architectures. In *Proceedings of the IEEE 2005 Conference on Field-Programmable Technology*, pages 43–50. IEEE Computer Society Press, 11 - 14 December 2005.

[KKP06] Markus Köster, Heiko Kalte, and Mario Porrmann. Relocation and defragmentation for heterogeneous reconfigurable systems. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 70–76. CSREA Press, 26 - 29 June 2006.

[KKS04] Ryan Kastner, Adam Kaplan, and Majid Sarrafzadeh. *Synthesis Techniques and Optimizations for Reconfigurable Systems*. Kluwer Academic Publishers, 2004.

[KL78] H. T. Kung and C. E. Leiserson. Systolic Arrays for VLSI. In *Sparse Matrix Proceedings*, pages 245–82, Philidelphia, PA, 1978. Society of Industrial and Applied Mathematicians.

[KMN+00] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *In IEEE Trans on Computer-Aided Design*, 19(12):1523–1543, 2000.

[Koc02] Andreas Koch. Compilation for Adaptive Computing Systems Using Complex Parameterized Hardware Objects. *The Journal of Supercomputing*, 21:179–190, 2002.

[KP87] F. J. Kurdahi and A. C. Parker. Real: a program for register allocation. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 210–215, New York, NY, USA, 1987. ACM Press.

[KPC94] P. H. Kelly, K. J. Page, and P. M. Chau. Rapid prototyping of asic based systems. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 460–465, New York, NY, USA, 1994. ACM Press.

[KPR02] Heiko Kalte, Mario Porrmann, and Ulrich Rückert. A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*, 2002.

[Kre96] Rainer Kress. *A Fast Reconfigurable ALU for Xputers*. Ph.D. Thesis., University of Kaiserslautern, 1996.

[KSW+07] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre-Yves Droz. Ramp blue: A message-passing manycore system in fpgas. In *17th International Conference on Field Programmable Logic and Applications*. IEEE, 27 - 29 August 2007.

[KV99] Meenakshi Kaul and Ranga Vemuri. Integrated Block-Processing and Design-Space Exploration in Temporal Partitioning for RTR Architectures. In *Proceedings of the Reconfigurable Architecture Workshop, RAW '99*. Springer, 1999.

[KVGO99] Meenakshi Kaul, Ranga Vemuri, Sriram Govindarajan, and Iyad Ouaiss. An Automated Temporal Partitioning and Loop Fission Approach for FPGA Based Reconfigurable Synthesis of DSP Applications. In *Design Automation Conference*, pages 616–622, 1999.

[KW97] S. A. Kravchenko and Frank Werner. Parallel Machine scheduling Problems with a Singel Server. *Mathematical and Computer Modelling*, 26(12):1–11, December 1997.

[KZP03] S. Kim, C. H. Ziesler, and M. C. Papaefthymiou. Fine-grain real-time reconfigurable pipelining. *IBM Journal of Research and Development: Power-efficient computer technologies*, 47(5/6):599–609, September/November 2003.

[LBM+06] Patrick Lysaght, Brandon Blodget, Jeff Mason, Jay Young, and Brendan Bridgeford. Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration on XILINX FPGAs. In *Proceedings of the FPL 2006*, 2006.

[LCD03] Jong-eun Lee, Kiyoung Choi, and Nikil D. Dutt. Compilation Approach for Coarse-Grained Reconfigurable Architectures. *IEEE Design & Test of Computers*, 20(1):26–33, 2003.

[LCH00] Zhiyuan Li, Katherine Compton, and Scott Hauck. Configuration Caching Management Techniques for Reconfigurable Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 22–38, Napa Valley, CA, USA, April 2000.

[LD93] P. Lysaght and J. Dunlop. Dynamic reconfiguration of FPGAs. In W. Moore and W. Luk, editors, *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications*, pages 82–94, Oxford, England, 1993.

[LH02] Zhiyuan Li and Scott Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 187–195. ACM Press, 2002.

[LLM95] L. Lundheim, I. Legrand, and L. Moll. A Programmable Active Memory Implementation of a Neural Network for Second Level Triggering in Atlas. *International Journal of Modern Physics C*, 6:561–566, 1995.

[LRK77] J. K. Lenstra and A. H. G. Rnnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1977.

[LRKB77] J. K. Lenstra, A. H. G. Rnnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[LW82] J. Leung and J. W. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[Mak00] Tsugio Makimoto. The Rising Wave of Field Programmability. In *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 1–6, Villach, Austria, 2000. Springer-Verlag.

[MAV+02] T. Marescaux, Bartic A., D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs. In *Proceedings of the International Conference on Field Programmable Logic*, Montpellier, France, September 2002.

[MBKS01] S. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh. Sps: A strategically programmable system. In *Proceedings of Reconfigurable Architecture Workshop*, 2001.

[MFK+00] Rafael Maestre, Milagros Fernandez, Fadi J. Kurdahi, Nader Bagherzadeh, and Hartej Singh. Configuration management in multi-context reconfigurable systems for simultaneous performance and power optimizations. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 107–113, Washington, DC, USA, 2000. IEEE Computer Society.

[Mit95] J. Mitola. The software radio architecture. *Communications Magazine, IEEE*, 33(5):26–38, 1995.

[MMadH06] Burkhard Monien and Friedhelm Meyer auf der Heide. *New Trends in Parallel & Distributed Computing*, volume 181. HNI-Verlagsschriftenreihe, 2006.

[MMS00] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A pattern language for parallel application programs (research note). In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 678–681, London, UK, 2000. Springer-Verlag.

[Mot02] M. Motomura. A dynamically reconfigurable processor architecture. In *Microprocessor Forum*, October 2002.

[MSSB00] Steve MacDonald, Duane Szafron, Jonathan Schaeffer, and Steven Bromling. Generating parallel program frameworks from parallel design patterns. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 95–104, London, UK, 2000. Springer-Verlag.

[MSV00a] Bingfeng Mei, Patrick Schaumont, and Serge Vernalde. A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems. In *4. GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Meißen, Germany, November 2000.

[MSV00b] Bingfeng Mei, Patrick Schaumont, and Serge Vernalde. A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems. In *ProRISC workshop on Circuits, Systems and Signal Processing*, November 2000.

[MTAB07] Mateusz Majer, Jürgen Teich, Ali Ahmadinia, and Christophe Bobda. The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-Based Computer. *Journal of VLSI Signal Processing Systems*, 46(2):15–31, March 2007.

[Nag01] Ulrich Nageldinger. *Coarse-Grained Reconfigurable Architecture Design Space Exploration*. PhD thesis, University of Kaiserslautern, CS department (Informatik), 2001.

[NB04] Juanjo Noguera and Rosa M. Badia. Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling. *Trans. on Embedded Computing Sys.*, 3(2):385–406, 2004.

[NTI04] Jari Nurmi, Hannu Tenhunen, and Jouni Isoaho. *Interconnect-Centric Design for Advanced SOC and NOC*. Kluwer Academic Publishers, 2004.

[OGS+98] Iyad Ouaiss, Sriram Govindarajan, Vinoo Srinivasan, Meenakshi Kaul, and Ranga Vemuri. An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures. In *IPPS/SPDP Workshops*, pages 31–36, 1998.

[OSF+07] Tobias Oppold, Thomas Schweizer, Julio Oliveira Filho, Sven Eisenhardt, and Wolfgang Rosenstiel. CRC – concepts and evaluation of processor-like reconfigurable architectures. *it - Information Technology*, 7(3):157–164, June 2007.

[PAC06] PACT. XPP-III Processor Overview. White paper, PACT Informationstechnologie GmbH, 13 July 2006.

[PAK+07] Thilo Pionteck, Carsten Albrecht, Roman Koch, Erik Maehle, Michael Hübner, and Jürgen Becker. Communication Architectures for Dynamically Reconfigurable FPGA Designs. In *IPDPS, Reconfigurable Architectures Workshop*, pages 1–8. IEEE, 2007.

[Pel98] Susanna Pelagatti. *Structured development of parallel programs*. Taylor & Francis, Inc., Bristol, PA, USA, 1998.

[PEW+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.

[PHA+07] Katarina Paulsson, Michael Hübner, Günther Auer, Michael Dreschmann, and Jürgen Becker. Implementation of a Virtual Internal Configuration Access Port (JCAP) for Enabling Partial Self- Reconfiguration on Xilinx Spartan III FPGAs. In *Proceedings of the FPL 2007*. IEEE, 27 - 29 August 2007.

[PHB+07] K. Paulsson, M. Hübner, J. Becker, J.-M. Philippe, and C. Gamrat. On-line Routing of Reconfigurable Functions for Future Self-Adaptive Systems—Investigations within the AETHER Project. In *17th International Conference on Field Programmable Logic and Applications*. IEEE, 27 - 29 August 2007.

[PKG86] P. G. Paulin, J. P. Knight, and E. F. Gircycy. HAL: A Mulit-Paradigm Apporach to Automatic Data Path Synthesis. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 263–270, Piscataway, NJ, USA, 1986. IEEE Press.

[PT05] D. Pellerin and S. Thibault. *Practical FPGA Programming in C*. Prentice Hall, 2005.

[Ram77] Franz J. Rammig. A Concept for the Editing of Hardware. In *Proceedings of the 14th Design Automation Conference (DAC'77)*, New Orleans, USA, June 1977.

[Ram07] Franz Josef Rammig. Software-hardware complexes: Towards flexible borders. In *Proceedings of the IFIP TC 10 Working Conference: International Embedded System Symposium (IESS)*, pages 433–436, 30 May - 1 June 2007.

[Ret07] Achim Rettberg. *Low Power Driven High-Level Synthesis for Dedicated Architectures*. PhD thesis, University Paderborn, 2007.

[RG02] Fethi A. Rabhi and Sergei Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.

[RLZB04] Achim Rettberg, Thomas Lehmann, Mauro Zanella, and Christophe Bobda. Selbststeuernde rekonfigurierbare bit-serielle Pipelinearchitektur. Deutsches Patent- und Markenamt, December 2004. Patent-No. 10308510.

[RM07] Markus Rullmann and Renate Merker. A Reconfiguration Aware Circuit Mapper for FPGAs. In *14th Reconfigurable Architectures Workshop (RAW 2007)*, pages 1–8. IEEE, 25 - 26 March 2007.

[RMVC05] Javier Resano, Daniel Mozos, Diederik Verkest, and Francky Catthoor. A reconfiguration manager for dynamically reconfigurable hardware. *IEEE Design and Test of Computers*, 22(5):452–460, 2005.

[SC06] Manuel Saldaña and Paul Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *Proceedings of the FPL 2006*, pages 1–6. IEEE, 2006.

[Sch07] Simon Schütte. Eclipse Plug-In für den Entwurf Rekonfigurierbarer Rechensysteme. Diplomarbeit, Heinz Nixdorf Insitute, University of Paderborn, Germany, 2007.

[Sch08] Fabian Schulte. Entwicklung einer Two-Slot-Machine als rekonfigurierbarer, dynamischer Hardwarebeschleuniger integriert in ein FPGA basiertes System on Chip. Diplomarbeit, Heinz Nixdorf Insitute, University of Paderborn, Germany, 2008. to appear.

[SH07] Peter Sobe and Volker Hampel. FPGA-Accelerated Deletion-Tolerant Coding for Reliable Distributed Storage. In Paul Lukowicz, Lothar Thiele, and Gerhard Tröster, editors, *ARCS*, volume 4415 of *Lecture Notes in Computer Science*, pages 14–27. Springer, 2007.

[Sin07] Satnam Singh. Integrating FPGAs in high-performance computing: programming models for parallel systems—the programmer's perspective. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 133–135, New York, NY, USA, 2007. ACM Press.

[SJ02] Li Shang and Niraj K. Jha. Hardware-Software Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 345, Washington, DC, USA, 2002. IEEE Computer Society.

[SKB02] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. Dynamic power consumption in Virtex-II FPGA family. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 157–164, New York, NY, USA, 2002. ACM Press.

[SMR+04] Donatella Sciuto, Grant Martin, Wolfgang Rosenstiel, Stuart Swan, Frank Ghenassia, Peter Flake, and Johny Srouji. Systemc and systemverilog: Where do they fit? where are they going? In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10122, Washington, DC, USA, 2004. IEEE Computer Society.

[SNG01] Suraj Sudhir, Suman Nath, and Seth Copen Goldstein. Configuration caching and swapping. In *FPL '01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pages 192–202, London, UK, 2001. Springer-Verlag.

[SNRC06] Manuel Saldaña, Daniel Nunes, Emanuel Ramalho, and Paul Chow. Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI. In *Proceedings of the 3rd International Conference on ReConFigurable Computing and FPGA's*, pages 270–279. IEEE, September 2006.

[SON06] Andreas Schallenberg, Frank Oppenheimer, and Wolfgang Nebel. OSSS+R: Modelling and Simulating Self-Reconfigurable Systems. In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 177–182. IEEE, August 2006.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SS05] Alireza Shoa and Shahram Shirani. Run-time reconfigurable systems for digital signal processing applications: A survey. *J. VLSI Signal Process. Syst.*, 39(3):213–235, 2005.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, June 1989.

[STB06] Hayden Kwok-Hay So, Artem Tkachenko, and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 259–264, New York, NY, USA, 2006. ACM Press.

[SVCBS04] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and challenges for platform-based design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 409–414, New York, NY, USA, 2004. ACM Press.

[SVM01] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, 18(6):23–33, 2001.

[SWP03] Christoph Steiger, Herbert Walder, and Marco Platzner. Heuristics for Online Scheduling Real-time Tasks to Partially Reconfigurable Devices. In *Proceedings of the 13rd International Conference on Field Programmable Logic and Application (FPL'03)*, pages 575–584. Springer, September 2003.

[SWP04] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Comput.*, 53(11):1393–1407, 2004.

[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 IEEE International Real-Time Systems Symposium*, page 224. IEEE Computer Society, 2003.

[SWT+02] Herman Schmit, David Whelihan, Andrew Tsai, Matthew Moe, Benjamin Levine, and R. Reed Taylor. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 63–66, 2002.

[TB01] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *J. VLSI Signal Process. Syst.*, 28(1-2):7–27, 2001.

[TB05] Alexander Thomas and Jürgen Becker. Multi-grained reconfigurable datapath structures for online-adaptive reconfigurable hardware architectures. In *ISVLSI '05: Proceedings of the IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05)*, pages 118–123, Washington, DC, USA, 2005. IEEE Computer Society.

[TB07] Alexander Thomas and Jürgen Becker. New adaptive multi-grained hardware architecture for processing of dynamic function patterns. *it - Information Technology*, 7(3):165–173, June 2007.

[TCJW97] Steve Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 22, Washington, DC, USA, 1997. IEEE Computer Society.

[Tel05] Anil Telikepalli. Performance vs. Power: Getting the Best of Both Worlds. *Xcelljounral*, 54, 2005.

[TKB+07] F. Thoma, M. Kühnle, P. Bonnot, E. Moscu Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schüler, K.D. Müller-Glaser, and J. Becker. MORPHEUS: Heterogeneous Reconfigurable Computing. In *17th International Conference on Field Programmable Logic and Applications*. IEEE, 27 - 29 August 2007.

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[Tre95] Nick Tredennick. Technology and Business: Forces Driving Microprocessor Evolution. *Proceedings of the IEEE*, 83(12), December 1995.

[Tri94] Stephen M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[UHGB04a] Michael Ullmann, Michael Hübner, Björn Grimm, and Jürgen Becker. An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. In *IPDPS*. IEEE Computer Society, 2004.

[UHGB04b] Michael Ullmann, Michael Hübner, Björn Grimm, and Jürgen Becker. On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities. In *Proceedings of the International Conference on Field Programmable Logic and its Applications (FPL2004)*, pages 454–463. Springer, 30 August - 1 September 2004.

[VBR+96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.

[VKKR02] Ranga Vemuri, Srinivas Katkoori, Meenakshi Kaul, and Jay Roy. An efficient register optimization algorithm for high-level synthesis from hierarchical behavioral specifications. *ACM Trans. Des. Autom. Electron. Syst.*, 7(1):189–216, 2002.

[VM05] Nikolaos S. Voros and Konstantinos Masselos. *System Level Design of Reconfigurable Systems-on-Chip*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[VS07] Stamatis Vassiliadis and Dimitrios Soudris, editors. *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2007.

[VT03] Ramachandran Vaidyanathan and Jerry L. Trahan. *Dynamic Reconfiguration: Architectures and Algorithms (Series in Computer Science)*. Kluwer Academic/Plenum Publishers, 2003.

[Wan98] Markus Wannemacher. *Das FPGA-Kochbuch*. MITP-Verlag, 1998.

[War06] Alexander Warkentin. Two-slot based reconfiguration environment. Studienarbeit, Heinz Nixdorf Insitute, University of Paderborn, Germany, 2006.

[Waw07] John Wawrzynek. Adventures with a reconfigurable research platform. In *17th International Conference on Field Programmable Logic and Applications*. IEEE, 27 - 29 August 2007.

[WH95] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In Peter Athanas and Kenneth L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[WOK+00] Karlheinz Weiß, Carsten Oetker, Igor Katchan, Thorsten Steckstor, and Wolfgang Rosenstiel. Power estimation approach for SRAM-based FPGAs. In *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 195–202, New York, NY, USA, 2000. ACM Press.

[WOK+06] John Wawrzynek, Mark Oskin, Christoforos Kozyrakis, Derek Chiou, David A. Patterson, Shih-Lien Lu, James C. Hoe, and Krste Asanovic. Ramp: A research accelerator for multiple processors. Technical Report UCB/EECS-2006-158, EECS Department, University of California, Berkeley, Nov 2006.

[WP02] Herbert Walder and Marco Platzner. Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 24–30, June 2002.

[WP03] Herbert Walder and Marco Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, pages 290–295. IEEE Computer Society, March 2003.

[WP04] Herbert Walder and Marco Platzner. A Runtime Environment for Reconfigurable Hardware Operating Systems. In *Proceedings of the 14th International Conference on Field Programmable Logic and Application (FPL'04)*, pages 831–835. Springer, August 2004.

[Xil04] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. Application Note, Xilinx, September 2004.

[YMHB00] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA*, pages 225–235, 2000.