TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Diss. ETH No. 15955
TIK-Schriftenreihe Nr. 68

HERBERT H. WALDER

Operating System Design for Partially Reconfigurable Logic Devices

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to the
Swiss Federal Institute of Technology, Zurich
for the degree of Doctor of Technical Sciences (Dr. sc. techn.)

Diss. ETH No. 15955

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Jürgen Becker, co-examiner
Prof. Dr. Marco Platzner, co-examiner

Examination date: April 11th, 2005

I would like to thank

- Prof. Dr. Lothar Thiele for providing a pleasant research environment and for supervising my research work,
- Prof. Dr. Marco Platzner for introducing me into the fascinating field of *Reconfigurable Computing*, and for his valuable support and inspiration during my thesis,
- Prof. Dr. Jürgen Becker for his willingness to be a co-examiner of my thesis.

для моей зайки

my parents and sisters


Seite Leer /
Blank leaf

Diss. ETH No. 15955

Operating System Design for Partially Reconfigurable Logic Devices

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY (ETH)
ZURICH

for the degree of

DOCTOR OF TECHNICAL SCIENCES (DR. SC. TECHN.)

presented by

HERBERT H. WALDER
DIPL. EL.-ING. ETH

born 3rd April 1969
citizen of Egg/Zurich, Switzerland

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Jürgen Becker, co-examiner
Prof. Dr. Marco Platzner, co-examiner

2005

Seite Leer /
Blank leaf

Abstract

The logic capacity of current SRAM-based FPGAs (Field Programmable Gate Arrays) amounts to multiple times the size of coarse-grained circuits, such as FIR-filters, spectral transformations (FFTs, DCTs), and crypto- or multimedia codecs. Furthermore, modern FPGAs are partially reconfigurable; i.e., parts of the FPGA area can be altered during run-time, while other parts not involved in the reconfiguration process continue to operate in parallel.

These characteristics allow for novel use of FPGAs as subcomponents in embedded systems: Several independent logic circuits (so-called *hardware tasks*) can be loaded, executed, and removed again after completion, whereas subsequent hardware tasks can reuse the same resources. Due to these properties, an FPGA becomes a *dynamically allocatable resource*.

In this dissertation, a novel kind of operating system, a *Reconfigurable Hardware Operating System (RHWOS)* is proposed that meets the following requirements:

- make available the reconfigurable resources of an FPGA to several applications, using a well-defined interface,
- manage the limited resources in an efficient way, in order to ensure a high utilization of the FPGA, and
- hide the complexity of the management and reconfiguration functions from the user applications.

An RHWOS can be seen as an extension of a *Real-Time Operating System (RTOS)*, that additionally manages an FPGA as a dynamic system resource and executes both software and hardware tasks. In this way, an RHWOS is in a position to exploit the advantages of task implementations in software or in hardware.

The realization of an RHWOS reveals a variety of novel problem areas in a conceptual and algorithmic context, as well as on a technological level. This work addresses all of these aspects. For a number of RHWOS-typical problems, several solutions were developed, implemented and experimentally evaluated.

RHWOS Concepts

On a conceptual and technology-independent level, the internal structure of an RHWOS is investigated, functional modules are identified and the interaction of the modules is explained.

As a result, a complete design-concept for the development of RHWOS-driven applications is introduced and a run-time environment with an exemplary partitioning of its modules in hardware and software, respectively, is proposed. The run-time environment defines dedicated interface points between the RHWOS and hardware tasks and supports their partial reconfiguration.

RHWOS Algorithms

Some of the task and resource management algorithms in an RHWOS strongly differ from those employed in an RTOS. The run-time placement of hardware tasks in combination with real-time scheduling is identified as the main problem in an RHWOS from an algorithmic point of view.

Several variants of heuristic algorithms that solve this problem are presented. All algorithms were experimentally evaluated in terms of time- and space-efficiency.

RHWOS Prototype

The implementation of an RHWOS and a case study application based on current FPGA technology and development tools is also addressed in this work.

The main components of an RHWOS were successfully implemented and tested on a tailored prototyping platform, the XF-BOARD. The run-time environment provides a bus-structure for enabling inter-task communication and supports partial reconfiguration of tasks with variable sizes.

By means of a case study application from the real-time signal-processing domain, the practicability of an RHWOS based on current FPGA technology is proved.

Kurzfassung

Die Logikkapazität heutiger SRAM-basierter FPGAs (Field Programmable Gate Arrays) übersteigt um ein Vielfaches die Grösse von grobgranularen Schaltungsblöcken, wie z.B. FIR-Filter, Spektraltransformationen (FFT, DCT), Krypto- oder Multimedia-Kodierer/Dekodierer. Zudem lassen sich moderne FPGAs partiell rekonfigurieren; d.h., einzelne Teilbereiche können zur Laufzeit gezielt verändert werden, während die von der Rekonfiguration nicht betroffenen Schaltungen parallel dazu weiterarbeiten.

Diese Eigenschaften eröffnen neue Einsatzmöglichkeiten von FPGAs als Subkomponenten in eingebetteten Systemen: Mehrere voneinander unabhängige Schaltungsblöcke (sog. *Hardware-Tasks*) können zur Laufzeit in ein FPGA geladen, ausgeführt und wieder entfernt werden, während nachfolgende Hardware-Tasks die freigewordenen Bereiche wiederverwenden.

In dieser Dissertation wird ein neuartiges Betriebssystem, ein *Reconfigurable Hardware Operating System (RHWOS)*, vorgeschlagen, dessen Aufgabe es ist,

- die rekonfigurierbaren Ressourcen eines FPGAs gleichzeitig mehreren Applikationen mittels definierten Schnittstellen verfügbar zu machen,
- die begrenzten Ressourcen effizient zu verwalten, um einen möglichst hohen Auslastungsgrad des FPGAs zu erreichen, sowie
- die Komplexität der benötigten Rekonfigurations- und Verwaltungsvorgänge von den Applikationen fernzuhalten.

Ein RHWOS kann als Erweiterung eines *Real-Time Operating Systems (RTOS)* angesehen werden, das ein FPGA als zusätzliche dynamische System-Ressource verwaltet und neben Software-Tasks auch Hardware-Tasks ausführt. Somit ist ein RHWOS in der Lage, die spezifischen Vorteile von Task-Implementationen in Software oder in Hardware auszuschöpfen.

Bei der Realisierung eines RHWOS treten verschiedene neuartige Problembereiche auf, sowohl in konzeptioneller und algorithmischer, als auch in technologischer Hinsicht. Die vorliegende Arbeit befasst sich mit allen drei Aspekten, indem für ausgewählte RHWOS-typische Problemstellungen verschiedene Lösungen entwickelt, implementiert und evaluiert werden.

RHWOS-Konzepte

Auf einer konzeptionellen und technologie-neutralen Ebene wird untersucht, in welche funktionale Blöcke sich ein RHWOS zur Kompilations- und Laufzeit gliedert und in welcher Weise diese interagieren.

Als Resultat wird ein durchgängiges Design-Konzept für die Entwicklung von RHWOS-gesteuerten Applikationen beschrieben und eine strukturierte Laufzeitumgebung mit einer exemplarischen Partitionierung der einzelnen

Komponenten in Hardware bzw. Software vorgeschlagen. Die Laufzeitumgebung definiert die Interaktionspunkte zwischen RHWOS und Hardware-Tasks und unterstützt deren dynamisch-partielle Rekonfiguration.

RHWOS-Algorithmen

Einige Algorithmen im Bereich der Task- und Ressourcen-Verwaltung eines RHWOS weisen erhebliche Unterschiede zu denjenigen eines RTOS auf. Die Laufzeit-Platzierung von Hardware-Tasks in Kombination mit zeitkritischer Ablaufplanung wurde als zentrales und RHWOS-typisches Problem identifiziert.

Verschiedene Varianten von heuristischen Verfahren und Algorithmen zur zeit- und speichereffizienten Lösung dieses Problems werden vorgestellt und mittels Simulationen bewertet.

RHWOS-Prototyp

Die Realisierung eines RHWOS und einer Beispiel-Applikation, basierend auf heute kommerziell erhältlichen FPGAs und Entwicklungswerkzeugen ist ein weiterer Gegenstand dieser Arbeit.

Auf einer eigens dafür entwickelten Plattform, dem XF-BOARD, wurden die Hauptkomponenten eines RHWOS entwickelt. Die im FPGA implementierte Laufzeitumgebung stellt eine Busstruktur zur Inter-Task-Kommunikation bereit und ermöglicht die partielle Laufzeit-Konfiguration von Hardware-Tasks mit variablen Grössen.

Mit Hilfe einer Beispiel-Applikation aus dem Bereich der Echtzeit-Signalverarbeitung wird die Realisierbarkeit eines RHWOS mit heute zur Verfügung stehender FPGA-Technologie unter Beweis gestellt.

Seite Leer /
Blank leaf

Contents

1

Introduction

Embedded computer systems are omnipresent. They can be found in almost all parts of our infrastructure that we are directly and consciously using day-to-day. Examples include mobile phones, PDAs, music or video players, ticket machines or even a variety of household appliances.

Moreover, embedded systems are also operating within infrastructures that are not consciously perceived by most of people, but that nevertheless provide important services on which modern societies strongly rely on, e.g. power plant and traffic control systems, internet and telecommunication network nodes, security and surveillance systems, automotive control systems, medical devices, building automation, etc.

Developers of such systems are faced with a number of trade-offs in the design process. Depending on the application environment, the important driving criteria can be: cost, performance, flexibility, power consumption, design complexity, risk of design faults, or time to market.

Functions executed in embedded systems often comprise a high algorithmic complexity combined with hard real-time constraints. Computing systems able to meet these requirements can be composed of a variety of different processing elements, memories, I/O devices, sensors and actuators. The choice of processing elements include instruction-set processors (DSPs, ASIPs, μCs), fixed function hardware (ASICs), and (re)configurable devices.

In addition to the system components, the application-software design, implementation, debugging, and operational support aspects must also be considered. These aspects significantly influence the development costs, the time-to-market, and the maintainability of a complete system. Due to these reasons, the usage of *Real-Time Operating Systems (RTOS)* to realize complex embedded

systems is widespread. Generally, an Operating System (OS) eases the development of applications by introducing several levels of abstraction, such as *User Tasks* and *OS-Objects*. In an OS environment, applications are no longer monolithic, but rather composed of a number of cooperating user tasks and OS objects. OS objects offer *application independent services* that can be invoked by user tasks, like FIFO-buffers, timers, semaphores, access to I/O device, etc., whereas user tasks implement the *application specific* functions. In an RTOS, user tasks and OS objects represent software code blocks. Henceforth, we denote user tasks realized in software as *Software Tasks (ST)*.

During run-time, the OS fully controls the *execution* of the application by (i) scheduling, activating, and deactivating STs, (ii) managing the limited system resources and (iii) resolving resource conflicts among the user tasks.

Compared to a general purpose OS [Win, Unx, Lnx, Mac], an RTOS can be tailored for a specific application environment and is able to meet several space and time constraints, e.g. small *foot-prints* (low memory consumption), fast task context switch, or support for application with hard real-time constraints. Examples of currently available RTOS include [VxW, TDB, WEm, ELi, ECo, QNX].

1.1 Programmable Logic in Embedded Systems Design

In the last decade, *Programmable Logic Devices (PLDs)* have emerged and have been increasingly included into embedded system designs to execute specific functions. In the majority of cases, PLDs just act as *ASIC replacements*, executing a fixed function after system has started up. In this way, the potential of modern PLDs, e.g. SRAM-based *Field Programmable Gate Arrays (FPGAs)*, is not fully exploited. Three major characteristics of modern FPGAs permit a change in regarding PLDs as building blocks in embedded systems:

- ***High Logic Capacity***
  While the logic capacity of early FPGAs was rather limited, currently available FPGAs offer up to 8 million system gates[1], which is a multiple of the gates needed to implement coarse grained functions (such as data conversion or filter algorithms, crypto codecs, etc.). Thus, several functions can be instantiated and executed at the same time on a single FPGA device.
- ***Real Execution Concurrency***
  Reconfigurable Computing often stands for a change of paradigm which is denoted as *Computing in Space*. The term *multitasking* in context with FPGAs gains a new dimension. In contrast to multitasking on a single processor system, where functions are executing only in a quasi-parallel manner, all functions in an FPGA are executing independently in parallel.

[1] Xilinx Virtex-II 8000 (XC2V8000) [XV2a]

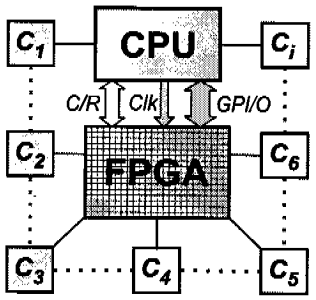


Fig. 1: General model of a *Reconfigurable Embedded System* composed of a CPU, an FPGA, and i external components $C_{1..i}$.

- ***Fast and Partial Reconfigurability***
  Modern FPGAs allow for *dynamic- and partial reconfiguration*, i.e., parts of the reconfigurable area can be quickly altered during runtime, whereas other parts remain unaffected and continue to operate undisturbed. Thus, logic circuits occupying only parts of the device can be dynamically loaded, executed and removed during runtime.

A system exploiting these three characteristics would no longer consider an FPGA as a static computing element with a fixed function, but rather as a highly flexible, dynamically allocatable computing resource. The resource offered to the system would be *basic logic elements*, so called *Reconfigurable Logic Units (RLU)*, which can be arbitrarily combined to form complex functions. We denote them as *Hardware Tasks (HT)*, analogous to STs in RTOS. The HTs could be loaded, executed, and removed from the device (i.e., deallocate the resources again) at any point in time during system execution.

This approach clearly asks for a dedicated unit that manages these limited reconfigurable resources to ensure their efficient utilization. We denote such a management unit as ***Reconfigurable Hardware Operating System (RHWOS)***. An RHWOS can be considered as an RTOS augmented by functions and services that dynamically manage the reconfigurable resources of an FPGA. These functions include scheduling and placing, loading/removing and starting/stopping of HTs. Furthermore, an RHWOS provides OS-objects that allow for communication and synchronization of HTs and STs.

Figure 1 conceptualizes the architecture of the target system which we utilize throughout this thesis. We consider it the core form of a reconfigurable embedded system, exhibiting the relevant properties we aim to investigate, and substituting a wide range of more complex architectures. The system is composed of a *CPU*, a partially reconfigurable SRAM-based *FPGA*, and a number of external devices $C_{1..i}$, either connected to the CPU or the FPGA. Three types of links are established between the CPU and to the FPGA:

- The CPU directly connects to the *Configuration/Readback Port (C/R)* of the FPGA. Thus, the CPU is in a position to completely control the FPGA at every point in time, i.e., to alter the configuration by executing a full or partial configuration, or by reading back the current state of the device.
- Several bidirectional *General Purpose I/O* wires *(GPI/O)* allow for any kind of communication between CPU and FPGA.
- A number of *Clock* signals *(Clk)*, generated by the CPU, drive the circuitry implemented in the FPGA.

Practically, such a system can be implemented either as a combination of discrete elements [XFB, Tre, Gec], or in a single hybrid device, i.e., *Configurable System on a Chip (CSoC)* [XV2b, A7S, Exc, Cha]. However, when analyzing the fundamental system characteristics from an RHWOS point of view, these variants are equivalent.

1.2 Problem Areas and Research Objectives

System architectures as indicated in Figure 1 and the use of an RHWOS induce various new relevant research areas that can be divided into three major categories:

I) RHWOS Concepts
RHWOSs can be considered on a technology independent level, disregarding any device and/or implementation specific problems to a large extent. From this conceptual point of view, the internal structure, functions and mechanisms of an RHWOS are central and raise the following questions:

- *RHWOS Structure*
  From which modules is an RHWOS composed of, during compile-time, and run-time, respectively? What functions do the modules execute and how do these modules interact with HTs and STs?
- *Application-, Task-, and Programming-Model*
  What application, task, and programming model is defined by an RHWOS? Which steps do the task and application design flow include?

In Chapter 3, we will propose our complete design concept for an RHWOS covering all above stated aspects.

II) RHWOS Algorithms
Many of the algorithms performing task- and resource-management functions in an RHWOS are radically different than to those in an RTOS.

The reconfigurable resource, provided by the FPGA, can be viewed as a bounded two-dimensional area. A HT itself has a certain shape and requires a part of this area to execute. We consider on-line scenarios, in which HTs may arrive at any point in time, have different shapes, unknown execution times, and may have execution deadlines. Based on these conditions, we identify two problem categories:

- *Hardware Task Placement and Free Area Management*
  Activating a HT leads to the geometric problem of determining a feasible location on the FPGA surface in which to *place* the HT. We denote this RHWOS function as *Hardware Task Placement.* After a HT has been successfully placed, the residual free reconfigurable area needs to be managed by the RHWOS. We call this function *Free Area Management.*

  Both functions are unique for RHWOS. Due to the dynamic nature of HTs, complex task allocation situations occur in the FPGA surface. Highly specialized algorithms and data-structures are needed in order to permit a time and space efficient implementation of these functions.
- *Hardware Task Scheduling*
  For STs in RTOS, various scheduling policies are known [Pin95, SSRC98, But00]. However, activating a ST means merely *assigning the CPU* to the ST, whereas a HT needs to be *loaded onto the FPGA* by a partial reconfiguration process prior to its execution. This operation induces some delay. Moreover, several independent hardware tasks can be running concurrently in an FPGA.

  Open questions are: To what extent can the known RTOS scheduling policies be adopted to schedule HTs? What is the influence of the reconfiguration overhead occurring in RHWOS?

Chapter 4 presents an experimentally confirms a number of novel task placement and scheduling algorithms for RHWOS.

III) RHWOS Implementation

The previously viewed issues remain on a technology independent level. The realization of an RHWOS that uses a currently available FPGA invokes problems on a very detailed technical level.

- *Reconfigurable Device and Platform Architecture Requirements*
  Running an RHWOS poses a number of requirements to the underlying reconfigurable device and the architecture of the platform. How are these requirements specified?
- *Runtime Environment*
  What kind of structural elements are needed in the FPGA to allow for partially reconfiguring hardware tasks? How can inter-task communication be realized in such a highly dynamic environment?

In Chapter 5, we present prototypical implementations of the most critical parts of an RHWOS, and prove their proper functioning by a case study application.

The main claim of the thesis is:

Reconfigurable Hardware Operating Systems (RHWOS) are required to efficiently employ partially reconfigurable logic devices in embedded systems.

1.3 Overview

Chapter 2 lays the foundation for all subsequent chapters by introducing the principles of programmable logic devices and their use in embedded systems. The major characteristics from the perspective of an RHWOS are highlighted, e.g. programmable logic device architecture, run-time reconfigurability and design approaches. A case study application is presented that motivates the use of an RHWOS.

Chapter 3 is devoted to conceptual aspects of an RHWOS. It starts with modeling the relevant characteristics of a reconfigurable embedded system from an RHWOS point of view. The application programming- and task-models are introduced, followed by an over-all description of the compile-time and run-time system of an RHWOS. In the breakdown of the systems on a module level, the elementary functions of an RHWOS are discussed and reviewed with related work.

Chapter 4 concentrates on task and resource management functions of an RHWOS. Therefore, the resource and task models introduced in the previous chapter are refined and substantiated by formal definitions. On this basis, a number of novel hardware task placement and scheduling algorithms using different modeling scenarios are developed and experimentally evaluated.

Chapter 5 reports on a prototype implementation covering the main parts of an RHWOS, based on a currently available FPGA family (XILINX VIRTEX-II). A runtime environment is realized that allows for partially reconfiguring hardware tasks during run-time and enables fast inter-task communication. On a platform tailored to RHWOS, a successful implementation of a case study application is presented that proves the feasibility of an RHWOS.

Chapter 6 concludes this thesis with a summary of the main results, and provides some starting points for further research and future architectures of partially reconfigurable logic devices controlled by RHWOS.

2

Embedded Systems and Programmable Logic Devices

Overview

This chapter provides introductory information about programmable logic devices; this information will be essential for comprehension of the research work, presented within the subsequent parts of this thesis.

We start with viewing embedded systems on a conceptual level, outlining the major properties and interaction scheme with their environments, and focus on internal design aspects. Then we introduce programmable logic devices, which represent the main issue of this thesis: we classify the device types, discuss the characteristic features and point out their different uses as building blocks in embedded systems. We further discuss a case study application that motivates the use of an RHWOS. A brief look at the market background of programmable logic rounds off the chapter and underlines the importance of this work from an economic point of view.

2.1 Conceptual View to Embedded System Design

While *General Purpose Computers* endeavor to provide highest computational power for a wide range of applications, *Embedded Systems* are tailored to a specific type of application within a well defined technical context [Car01].

Viewed from a high level of abstraction, any embedded system transforms an input vector of n signals $I_{1..n}$ to an output vector of m signals $O_{1..m}$ (as

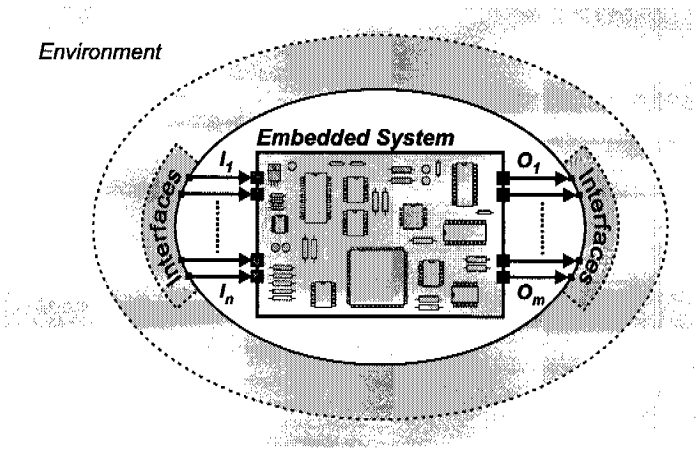


Fig. 2: Conceptual view of an embedded system.

indicated in Figure 2). The input and output signal vectors are connected by interfaces to the specific technical context, namely sensors, and actuators, respectively. Sensors and actuators can include both *Human/Machine (HMI)* as well as *Machine/Machine Interface (MMI)* devices.

At the beginning of any embedded system's life cycle, there are *requirements* mainly defined by its future operational area. We subdivide these requirements into *primary* and *secondary* ones.

The primary requirements capture the needed functional and timing behaviour that ensures the correct working of the system in the specific technical context. In addition, there may be various environmental constraints which are directly imposed by the application context, e.g. physical dimension, weight, energy consumption, or resistibility against some particular outside influence, such as temperature, humidity, or radiation, etc. All these specifications *must* be fulfilled without any variations.

The secondary requirements include aspects that are not directly visible from the outside, for instance system flexibility, extensibility, manufacturing costs, production complexity, number and kind of built in components, architectural clearness, etc. In the majority of cases, system designers encounter at this point a degree of freedom and, thus, can devise a set of self-defined requirements in order to achieve the aimed objective.

Obviously, the more precisely all requirements (and all non-requirements) are specified, the more likely it is that an optimal solution can be determined.

Design Space Exploration is the process of analyzing several functionally equivalent alternatives (all meeting the primary requirements) to identify the most suitable one [Hsi00, TCGK03]. Multi-objective optimization algorithms [Zit99, Kal01] can be used to efficiently calculate them, even if there is a high number of competing attributes.

However, as pre-requisites, detailed specifications of the application, the available devices, and communication infrastructure, and their characteristics, respectively, are needed (e.g. execution times of each algorithm of the application, when computed on a DSP or an ASIC, throughputs of busses, etc.).

Further optimization potential can be found in the use of existing devices in an unconventional way by exploiting specific device characteristics. We claim that modern *Programmable Logic Devices*, namely SRAM-based partially reconfigurable FPGAs, hold such a potential.

2.2 Programmable Logic Devices

The term *Programmable Logic Device (PLD)* refers to any type of integrated circuits that can be configured by the end-user to implement a wide range of logic circuits. Since its first introduction in 1975 by *Signetics Corporation* [Sig], this kind of logic device has evolved substantially. Over the years, a large number of manufacturers [Xil, Alt, Lat, Act, Atm] developed various device types with different architectural characteristics, various target applications and several vendor-specific designations.

2.2.1 PLD Architecture Taxonomy

According to specific architectural attributes, PLDs can be divided into the following categories:

- ***Simple Programmable Logic Devices (SPLDs)***
  The internal structure of an SPLD is straightforward: It contains two consecutive coupled matrices, each implementing logic AND and OR functions with a particular number of input lines. Both, the connections between input lines and AND plane (product lines), and between product lines and OR plane may be *programmable* or *fixed*. The result of the OR plane is forwarded to the corresponding output pin.

  There are three different types of SPLDs, depending on which planes are programmable and/or fixed (see Table 1).

SPLD Type	AND array	OR array
FPLA (Field Programmable Logic Array)	prog'able	prog'able
PROM (Programmable Read-only Memory)	fixed	prog'able
PAL (Programmable Array Logic) [Mon, AMD]	prog'able	fixed

Tab. 1: Programmable arrays of different SPLD types.

Figure 3 shows a part of the logic diagram of a standard PAL [EPA] with

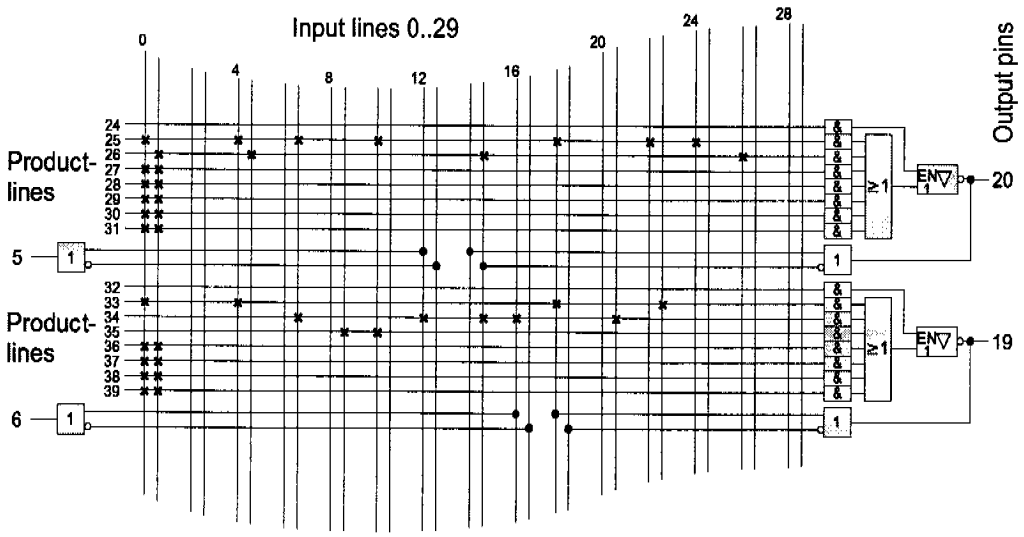


Fig. 3: Part of the logic diagram of a standard PAL with programmed interconnections in the AND-matrix, each marked with asterisks (device: Atmel ATF22V10C [EPA]).

some exemplary programmed interconnections in the AND plane, each connection marked with an asterisk.

Since any combinational logic function can be reduced to a sum of products (conjunctive normal form, CNF), i.e., groups of AND terms followed by OR terms [DM94], SPLDs optimally support the implementation of such two-level combinatorial logic. Advanced types include flip-flops that enable the implementation of state-full logic.

The strengths of SPLDs can be found in their speed (short pin-to-pin delay) and predictable timing behaviour; on the other hand, their weak points are their low logic capacity, architectural narrowness and restricted reconfiguration capabilities [Sha98]. Due to these limitations, SPLDs are not further considered within this thesis (cf. 2.2.2 and 2.2.4). Today, several vendors have established SPLDs in the market with different product labels, such as *GAL*, *ispGAL*, *PLA*, *PGA*, *EPLD*, *PEEL*, etc., which all contain similar features.

- ***Complex Programmable Logic Devices (CPLDs)***
  CPLDs attempt to overcome the architectural limitations of SPLDs with a more flexible block structure and more flexible interconnects. A CPLD comprises a number of *Function Blocks (FB)*, which again split into several SPLD-like *Macro Cells (MC)*, typically containing one dedicated flip-flop. Each programmable macro cell provides the generation of a product term. All function blocks are interconnected via a programmable switch matrix, whereas *I/O Blocks (IOBs)* allow for connectivity to the devices outside.

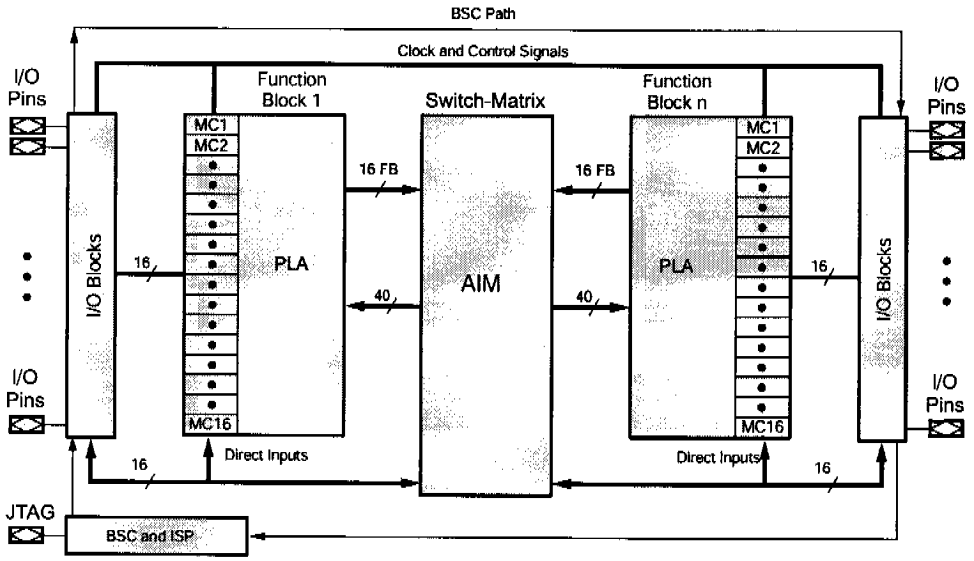


Fig. 4: Block structure of a CPLD (Xilinx CoolRunner [XCR]).

Figure 4 depicts a CoolRunner-II type CPLD [XCR] of Xilinx [Xil], which follows the architecture described above. Device types vary in the number of macro cells and I/O blocks available on chip, but also in the size of the switch matrix.

In addition to having the strengths of SPLDs, CPLDs feature more efficient resource utilization due to the flexibility of the switch matrix. A drawback arises in the complexity of the switch matrix if a high number of function blocks exists. However, the reconfiguration properties of CPLDs are rudimentary, therefore, CPLDs are not further discussed in this thesis (cf. 2.2.2 and 2.2.4). Typical vendor-specific names of CPLDs are *XPLD*, *ispXPLD*, and *EEPLD*.

- ***Field Programmable Gate Arrays (FPGAs)***
  Since FPGAs are of utmost importance within this thesis, we discuss their internal structure in more detail than those of SPLDs and CPLDs. We focus on one specific type of FPGA architecture: the *fine-grained, 2-dimensional island-style, single-context* type [BRM99, Sha98]. Most modern FPGAs follow this architecture [XV2a, XS3, Str, Cyc, ECP, isp].

  The first *island style* FPGA was developed by XILINX INC. [Xil] in 1984.[1] It consists of many identical *Reconfigurable Logic Units (RLU)*, which are placed in a two-dimensional array surrounded by *I/O Blocks (IOB)*, as de-

[1] In the meantime, several other semiconductor companies [Alt, Lat, Act, Qui] have developed their own FPGA families and use proprietary namings for the internal architectural modules and elements. For the purpose of clarity, we will consistently use in this thesis the nomenclature introduced by the FPGA inventor XILINX INC. [Xil].

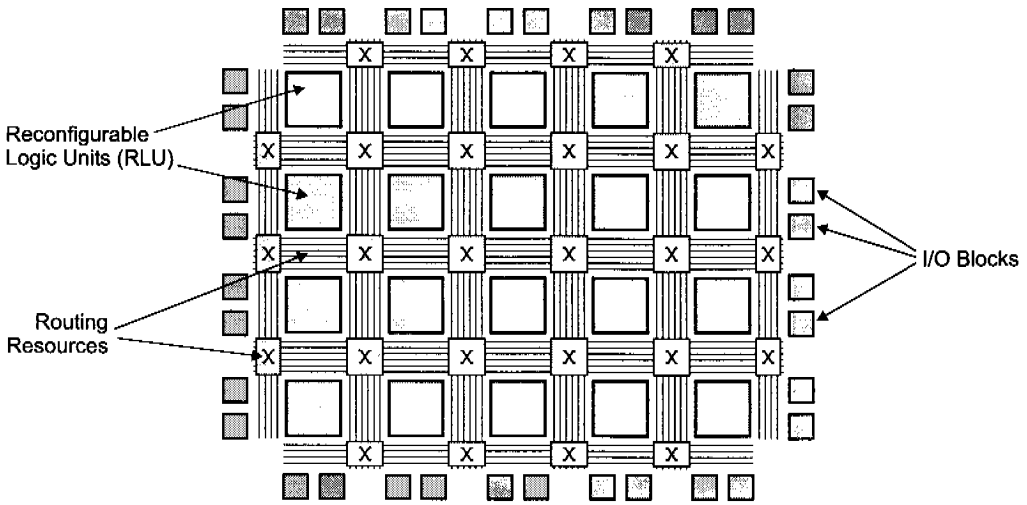


Fig. 5: FPGA island-style architecture (special function blocks and clock distribution network omitted) [BRM99].

picted in Figure 5. The channels between the RLU and I/O blocks are used for routing and provide programmable interconnection between any arbitrary location within the device. With this channel based routing strategy, much higher design complexities can be realized than by using the switch matrix approach of CPLDs. On the other hand, the high interconnection flexibility leads to non-predictable timing.

In the remaining part of this section, we focus on one specific FPGA device family to allow for discussing the relevant characteristics by means of a concrete example. We choose the VIRTEX-II [XV2a] family of the world's leading FPGA vendor XILINX INC. [Xil] which is a state-of-the-art FPGA.[2]

Configurable Logic Block (CLB)

Each VIRTEX-II type RLU, so-called *Configurable Logic Block (CLB)*, splits into four *Slices*, which are the smallest elements of logic execution. Figure 6 shows the slice internals (top half only). Slices possess a much more sophisticated structure than macro cells of CPLDs: The central elements of a slice are two 4-Input *Look Up Tables (LUTs)* and two *Registers*, whereas the LUT can alternatively be operated as 16 bit shift register or 16 bit RAM, and the register as flip-flop or latch. Each LUT is capable of implementing any arbitrarily defined boolean function of four inputs. The propagation delay is therefore independent of the function implemented.

A number of *Multiplexers (MUX)* connect the LUT outputs and/or slice inputs with either the register inputs or directly to the slice outputs. Further-

[2] as of August 2004.

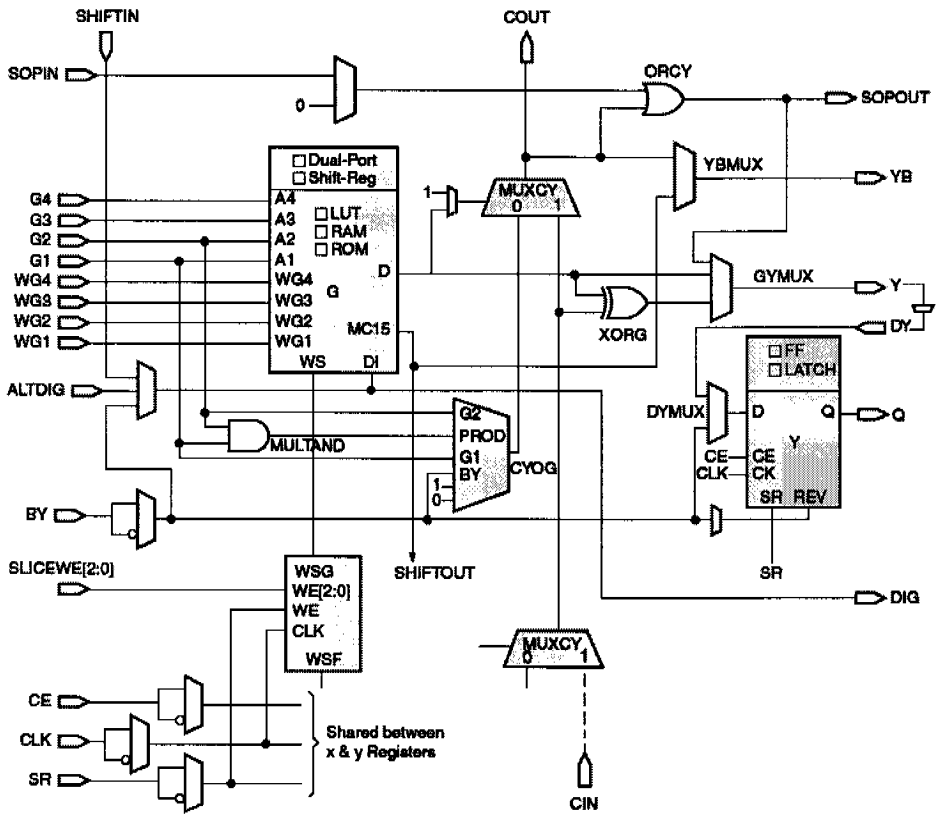


Fig. 6: Slice internals of a Xilinx Virtex-II FPGA [XV2a].

more, the MUXs allow the formation of a boolean function with up to eight inputs per slice half.

Special wires to and from neighboring slice halves, and slices, respectively, are available to implement fast carry chains used for high speed arithmetic and shift operations.

Configuration of a slice implies setting the LUT as well as the register operation mode, LUT content, and MUX bits.

I/O Block (IOB)

I/O blocks are signal ports that connect the device's inside with its outside world (see Figure 5). Beside the obvious settings such as port direction (input, output, bidirectional), more profound properties can be configured, such as logic level of single ended (TTL LVTTL, etc.) or differential signaled (LVDS, etc.) I/O standards, slew rate, single or double data rate, digitally controlled impedance drivers (serial and parallel termination), etc.

Special Function Block (SFB)

Modern FPGAs implement a number of additional modules, which can be built in designs to execute specific functions more efficiently. Virtex-II devices offer three types of SFBs: *Block RAMs*, *Multipliers*, and *Digital Clock Managers* [XV2a].

Block RAMs (BRAM) are static, high-speed, dual-ported RAMs, that are fully integrated into the device. Depending on the device type, more than a hundred BRAM blocks are available. Several single BRAM blocks can be combined to RAM blocks of higher memory depth or to form other memory organizations, such as FIFO or circular buffers.

Multiplier blocks allow for calculating the product of two 2's complement 18 bit signed integers in only one clock cycle, and hence, are best suited for being used in data-flow oriented signal processing designs, for example.

A digital clock manager offers a wide range of powerful clock management features, such as frequency synthesis, clock de-skew, and phase shifting. Several digital clock managers are available on-chip; that is, parts of the design can be run in different clock domains.

Since these special function blocks are scattered irregularly over the device, the surface homogeneity is thereby disrupted.

Routing Resources

Sophisticated routing resources are implemented to make connection of all device elements possible. Both, the form of the routing architecture and its dimensioning are critical design issues of any FPGA, due to the trade-off between interconnection flexibility and resource usage (silicon area) [CH99, TDC97]. Recent FPGAs devote more than 90% of their silicon resources to the routing [DH96].

In VIRTEX-II devices, IOBs, CLBs, Block RAMs, multipliers, and digital clock manager elements all use the same interconnect scheme and the same access to the global routing infrastructure. Routing lines of varying lengths and switch matrices are combined to provide a physical connection between two endpoints, e.g. to connect the output of a CLB with a IOB. The longest type of wires *(Long Lines)* can be employed to establish chip-spanning signals or bus structures.

The efficiency of a design, in terms of speed (timing), area and energy consumption, is strongly influenced by the capability of the routing infrastructure.

Clock Distribution

Clock distribution networks span the entire device and supply each CLB, IOB, etc. with a number of clock signals. Because of the high fan-out, the

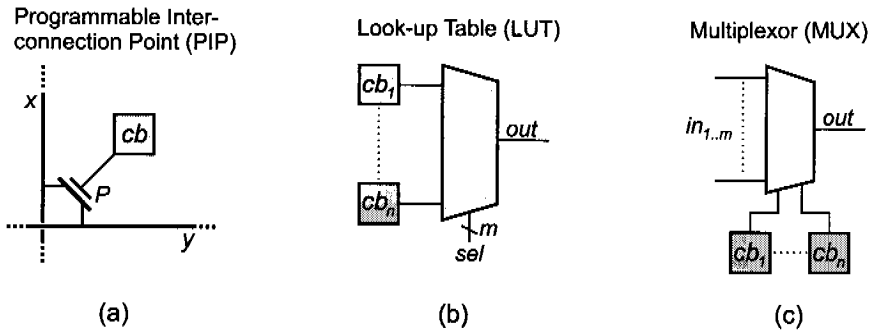


Fig. 7: Basic programmable logic structures and their belonging configuration bits cb / $cb_{1..n}$.

wires of the clock nets are physically implemented in a way to provide high signal transmission quality, primarily in terms of propagation delay and clock skew. In addition, clock nets are segmented to avoid clocking of unused parts of the device, which results in lower energy consumption of the device.

For a complete and more detailed description of FPGA architectures, we refer to the following books: [BFRV92, Tri94, OD95, Wan98, Sha98, BRM99].

FPGAs offer the highest logic capacity of all PLD categories. Currently available devices include up to 8 million system gates which represent a total of about 46'600 slices organized in an array of 112 × 104 CLBs [XV2a].

The advantage of FPGAs clearly lies in the efficient resource utilization enabled by the flexible routing structures, and the high complexity of realizable designs compared to SPLDs, and CPLDs. However, the routing flexibility causes unpredictability of timing. Furthermore, powerful CAE tools are needed for developing designs which efficiently make use of the available logic capacity.

2.2.2 Configuration Storage Technologies

Orthogonally to the logic architecture and structure of a PLD, the way how its configuration is *stored* in the device can be different. This property strongly influences the way in which the device can be operated.

- ***Fuse / Anti-fuse***
  Using the *Fuse* and *Anti-fuse* technology, respectively, changes in the physical structure of the device are made at predefined locations by applying well directed electric current: existing connections are broken (fuse), or connections are established (anti-fuse) to achieve logic paths.

  Fuse as well as anti-fuse are *One Time Programming (OTP)* technologies. The advantage of non-volatility of the device's configuration is paid by the fact that the programming process is irreversible and, thus, the device can not be reconfigured. Hence, this technology is preferably employed in small (i.e.,

low cost) devices, such as SPLDs. Compared to other technologies, one-time programming consumes little chip area and achieves faster execution speeds.

- ***EPROM / EEPROM / FlashRAM***
  Memory cell based configuration techniques follow a fundamentally different approach: the physical chip structures remain unchanged, but the content of memory cells controls the behaviour of logic elements. Figure 7 indicates basic programmable logic structures, driven by memory cell content bits cb (=configuration bit), and $cb_{1..n}$, respectively.

  Two routes x and y, as drawn in Figure 7(a), can be interconnected by a *Pass Gate* P controlled by a configuration bit cb. This structure is referred to as *Programmable Interconnection Point (PIP)* and is the basic element of any switch matrix. In Figure 7(b), $cb_{1..n}$ constitute the function of an m-*Look-up Table (LUT)*, whereas sel specifies its argument. An m-LUT is able to calculate $2^{2^m} = 2^n$ different boolean functions on its input sel, whereas n is the number of configuration bits needed. Figure 7(c) displays a circuit where $cb_{1..n}$ control a *Multiplexor (MUX)*.

  An *EPROM* memory cell can be erased by exposing it to ultraviolet radiation. A previously programmed configuration stored in the EPROM cells can be deleted and replaced by another one. *EEPROM* and *FlashRAM* allow for electrical erasing and reprogramming (in circuit), which speeds up the development cycle drastically. This flexibility is paid by a higher chip area and more complex fabrication processes [BR96].

- ***SRAM Based***
  In SRAM based FPGAs, all configuration bits are held in conventional SRAM cells. The major advantage of this solution is the very fast configuration speed (up to $400 Mbits^{-1}$) which results in configuration times ranging from $65.4ms$ down to less than $850\mu s$[3]. Furthermore, SRAM-based FPGAs can be reconfigured arbitrarily often. These properties encourage new use cases for FPGAs, allowing them to be operated in a highly dynamic and flexible way (cf. 2.3.3 and 2.3.4).

  The attained flexibility comes at a price: (i) The stored SRAM configuration is volatile, that is, external components are needed to feed a set of configuration bits into the device after power up; (ii) the resource consumption of an SRAM cell in terms of chip area and power is comparatively high; (iii) SRAM-based FPGAs are susceptible to *Single Error Upset (SEU)* [YSC02]. Due to outside influences, transient bit-flips may occur, which can lead to malfunction and/or, in worst case, to device damages.

Table 2 summarizes the different characteristics of configuration storage tech-

[3]This holds for full configurations of XILINX XC2V8000, and XC2V40 devices, respectively [XV2a].

storage technology	area requirement	volatile	configuration speed
(Anti-)Fuse	low	no	*n.a.*
EPROM	high	no	low
EEPROM	medium	no	medium
SRAM based	high	yes	high

Tab. 2: Categories and characteristics of PLD configuration storage technologies.

nologies.

2.2.3 Design Flow and Development Cycle

Before a PLD is able to execute the desired function, a sequence of design steps needs to be completed. To simplify matters, we will focus on the design flow for FPGAs (see Figure 8).

Step 1: *Design Entry*
The circuits need to be defined using a *Hardware Description Language (HDL)*, such as *ABEL* [Dat], *PALASM* [PAL] or *CUPL* [Wal66] for SPLDs and VHDL [Zai93] or Verilog [Sag98] for CPLDs and FPGAs. Alternatively, some CAE tools support state diagram tools or schematic entry.

Step 2: *Optimization / Synthesis*
This step includes architectural and logic synthesis of the design. The result is an optimized device independent *Netlist* (e.g. EDIF), representing the design based on standard logic functions and registers (Register Transfer Level).

Several manufacturers of *Logic Synthesizers* exist, such as [Syn, Men, Syo].

Step 3: *Technology Mapping*
The device independent net-list is mapped to the logic elements concretely available on the target device, e.g. look-up tables, multiplexors and registers, present in CLBs or other SFBs. The physical location of each block remains undefined.

Step 4: *Placement, Routing*
The place and route process assigns a location to each block and establishes physical connections by allocating routing resources.

Depending on the design complexity and the applied implementation constraints, this step can take from minutes up to hours.

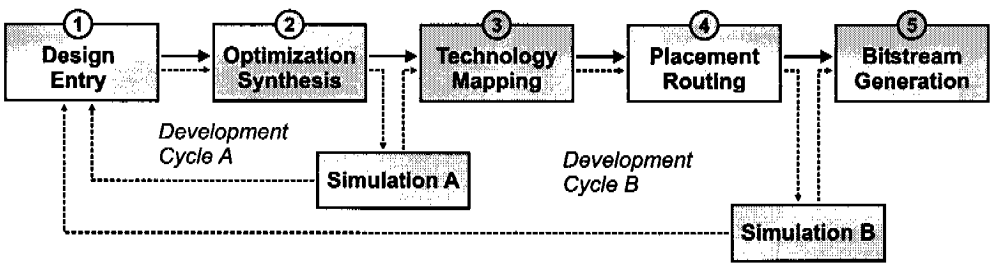


Fig. 8: General design flow for PLDs and development cycles *A* and *B*.

Step 5: Bitstream Generation
The final step includes the creation of the *Bitstream* file. A bitstream file contains all the information needed to fully configure the device.

In contrast to the bitstream files of SPLDs [Car97] (JEDEC [JED]), the internal structure of bitstreams for FPGAs is vendor specific and, in general, not disclosed.

Two development cycles *A* and *B* can be practiced: Cycle *A* is used to detect design errors on a logical- and device-independent level by simulating synthesized netlists. Simulations in cycle *B* include the target device characteristics on a physical level, such as timing behaviour.

2.2.4 Reconfiguration and Readback Mode

Each PLD is equipped with a reconfiguration port. The reconfiguration port is used to write and read configuration data in, and from the device, respectively. The implementation of the port combined with the device's configuration storage technology determines the operatable *Reconfiguration Modes*.
The reconfiguration modes can be categorized by two orthogonal attributes.

- *Temporal Attribute*
  The temporal attribute reveals the frequency of a reconfiguration process: If the device is only configured once after *Power-On Reset (POR)*, we denote this feature as *Static Reconfiguration*. In the case of *Dynamic Reconfiguration*, the device undergoes several reconfigurations after system power-up.

  Figure 9(a) illustrates a static reconfiguration scenario: During *Compile Time*, circuit *A* is being compiled *CP*; after system activation (power-on reset), circuit *A* is being loaded and *Executing (Exe)* during the whole system *Run Time*.
- *Spatial Attribute*
  The spatial characteristic describes which portion of the device is being reconfigured: A *Full Reconfiguration* alters all reconfigurable resources, whereas

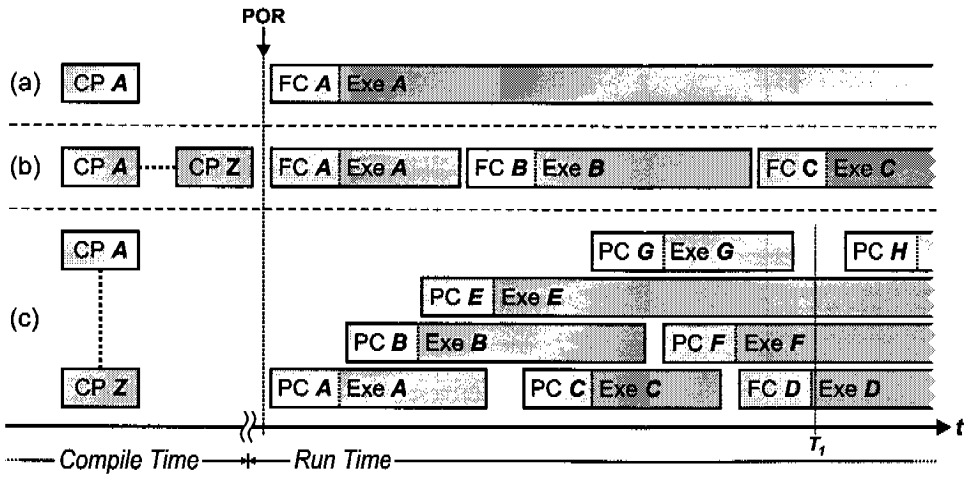


Fig. 9: Reconfiguration Modes: (a) Static reconfiguration, (b) Run-time reconfiguration (full), and (c) Run-time reconfiguration (partial) using *full configuration (FC)* and *partial configuration (PC)* to execute (Exe) the circuits $A..Z$.

a *Partial Reconfiguration* only modifies a fraction thereof. The decisive property of partial reconfiguration is that parts not involved in the reconfiguration process continue execution without being affected.

Furthermore, the *Reconfiguration Granularity* describes the amount of reconfigurable resources that can be accessed independently during a partial reconfiguration. The granularity depends on the device-architecture and can range from bit-level [XC6] to frames containing several hundred configuration bits [XAPb].

Figures 9(b,c) display the dynamic reconfiguration modes after a POR. In (b) full configurations (FC) are loaded consecutively, one for each circuit A, B, and C, respectively; In (c) *Partial Configurations (PC)* are performed to load the circuits $A...H$. In this example, circuits $A...H$ only require a portion of the reconfigurable resources; that is, more than one circuit can be accommodated at the same time. For instance, at time T_1, circuits D, F, and E are already loaded and executing simultaneously on the FPGA.

Physically, a configuration port can be implemented as serial or parallel, and as an unidirectional or a bidirectional interface, respectively. Serial ports show advantages in terms of lower design complexity, but suffer from low speed [JTA]. Parallel ports hold clear advantages in reconfiguration speed but require more sophisticated circuitry to access. Bidirectional ports allow for *Readback*, i.e., reading configuration and status data out from the device.

XILINX VIRTEX / VIRTEX-II devices are equipped with the so-called *SelectMap* port, which allows for both serial and high speed parallel, and for full

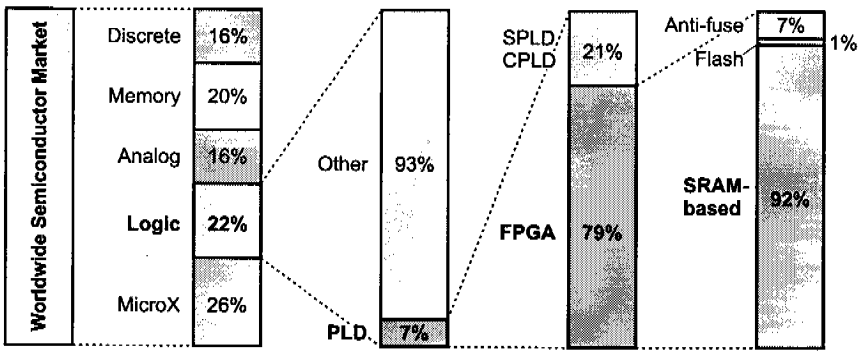


Fig. 10: Turnover breakdown of programmable logic devices and SRAM-based FPGAs in the worldwide semiconductor market. The estimated annual market volume in year 2004 amounts to USD 217 billion [Gar04, Sha04].

and partial reconfiguration/readback with a bandwidth of up to $400 Mbits^{-1}$ [XAPa].

2.2.5 Economic and Market Background of PLDs

It is estimated that the embedded system market is growing at around 25% per year [SBJ+96, TAS01, Sha04]. Compared to the worldwide semiconductor market (217 billion USD in year 2004, estimated by [Gar04]), PLDs contribute about 1.5% to its turnover[4] (Figure 10). While this absolute amount seems almost negligible, its annual growth rate is definitely notable and deserves attention. In the last decade, the PLD market grew much faster (in average 23% per year) than the semiconductor market (12% per year) [iSu, Gar04].

The communication industry is the largest end-market for the PLD industry and will mainly drive growth. However, it is expected that other markets such as industrial, consumer, automotive and aerospace/military will become drivers in near future by replacing ASIC designs. This trend is justified by the increasing need of flexibility and necessity of reducing the time-to-market.

2.3 PLDs as Building-blocks in Embedded Systems

Depending on the device characteristics, PLD's can be employed in various system formations. We briefly review the most common use cases and highlight their specific properties.

[4] In year 2003 [Sha04]

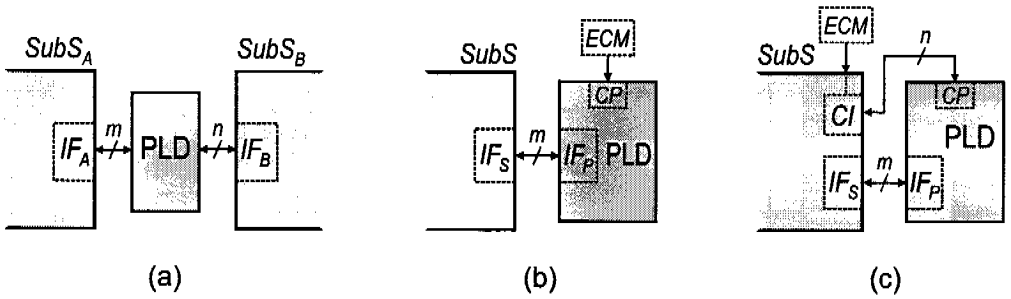


Fig. 11: Different use cases of PLDs as building-blocks in embedded systems

2.3.1 Use Case A: *Glue-Logic*

Embedded systems may be composed of a number of cooperating self-contained *Subsystems (SubS)*, each with well defined interfaces. Such prefabricated modules are commonly used in embedded system development in order to reduce engineering efforts. To integrate these modules into a complete system *Glue Logic* is needed, as exemplified in Figure 11(a). Glue logic *converts* a set of m signals of subsystem $SubS_A$ to n signals interfacing with subsystem $SubS_B$. Glue logic is usually not involved in creating any application functionality.

Typical examples for glue logic tasks are address decoding, data latching, serial to parallel (and vice versa) conversion, or simple boolean functions, respectively.

The conventional method for implementing glue logic is to compose the desired interface function out of several discrete standard logic elements. An efficient alternative for this error-prone and inflexible approach is the use of PLDs. Since the complexity of such circuitry is limited, SPLDs or CPLDs in EEPROM or FlashRAM-based versions are commonly applied.

The benefits of this solution include (i) reduction of design complexity and risk of design failure because of the low number of devices, (ii) fast circuitry, as a result of CAE optimized logic functions, and (iii) error recovery and upgradeability, due to the capability of *In-circuit (Re-)Configuration* (e.g. using the *JTAG*-interface [JTA]).

In this use case, PLDs are exclusively operated using the static reconfiguration model.

2.3.2 Use Case B: *Fixed-Function Co-Processor*

Applications executed by embedded systems often comprise computational intensive tasks e.g. video and/or audio processing (time/frequency and color space transformations [BA04, Weg04], pattern recognition, decompression algorithms [DJR01], etc.), cryptographic encryption/decryption [ER03], (de-)modulation or adaptive filtering, etc. [PEW+03]. Such primarily data-flow-oriented algorithms

can be isolated and implemented as highly optimized co-processor functions, which are parameterized and invoked by another subsystem.

Compared to SW-implementations, functions realized in HW can show speed-ups of one to three orders of magnitude [PP02, GNVV04].

Traditionally, co-processors are realized as custom designed hardware components, namely *Application Specific Integrated Circuits (ASICs)* [VMS96]. An alternative form is replacing the ASIC by a PLD which is executing the same functionality. Figure 11(b) displays this approach: *SubS* directly connects via an m-wire interface IF_S to IF_P of the PLD. The PLD itself is configured from an *External Configuration Memory (ECM)* using the PLD's *Configuration Port (CP)* after system power-up. The co-processor function is invoked using the connection interfaces IF_S, and IF_P, respectively.

Since the complexity of coprocessor functions most often exceeds the capacity of SPLDs and CPLDs, such designs are mostly implemented using FPGAs.

When comparing the two realization versions, ASIC vs. FPGA, the following characteristic aspects need to be discussed:

- *Performance / Resource Demand / Power Consumption*
  The global routing infrastructure of an FPGA offers a high degree of interconnection flexibility, but imposes significant signal delays, particularly for those lines, that are made of a high number of wire segments and are connected by switch matrices. The maximal speed of a circuit within a specific clock domain is determined by the delay of the longest routing path [XIF]. In ASIC designs, the wires are monolithic, which keeps the propagation delay low and, consequently, the circuit performance high.

  The flexibility of the routing infrastructure is also reflected in the consumption of resources needed for these structures. The logic capacity of FPGAs are given by the device type and, thus, the utilization of the FPGA may not be optimal. Circuits, particularly large ones, may not fully occupy all the logic resources offered by a specific FPGA device type.

  As a result, the power consumption of an FPGA is higher than that of an ASIC with the same functionality. In general: *For any FPGA-based application, there is an ASIC implementation of the system that is at least as fast, dense, power-efficient, and cheap in high volumes as the FPGA-based solution*[5].

- *Flexibility and Time-to-Market*
  Depending on the complexity, a complete ASIC design process can last several months and leads to fixed circuitry; that is, changes cannot be realized in the functionality of an ASIC after it is produced. This holds for correction of design failures but also for any kind of upgrades or optimizations.

[5]This sentence is known as: *'The Law of FPGAs vs. ASIC'*, formulated by Hauck [Hau98b, Hau98c].

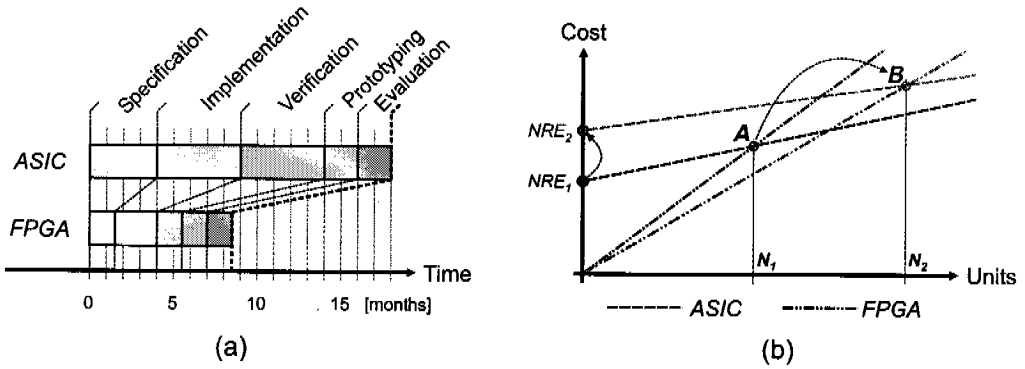


Fig. 12: Comparison of (a) Time-to-market *(today)*, and (b) Cost *(today and future trend)* between ASIC and FPGA designs [Sha04].

In this context, FPGAs show clear advantages: Since the user circuitry is configured into the device after it left the fabrication process, designs can be renewed at any point of time, correcting errors or inserting optimizations and upgrades, even after system deployment. Consequently, the time-to-market of a FPGA design is much shorter (approximately 50%) compared to that of ASICs. Especially the times necessary for the phases *Specification*, *Implementation*, and *Verification* are substantial shorter. Figure 12(a) depicts the total time and break down [Sha04].

- *Costs*
  From a customer point of view, producing an ASIC characteristicly includes high *Non-Recurring Expenses (NRE)* and low cost per chip in high volumes. In contrast, the NREs of an FPGAs are not borne by a single customer, but rather by the manufacturer. This reduces the end-user costs for low quantities. However, in high volumes, the FPGA variant shows higher costs, due to the higher device and production complexity. This characteristic is shown in Figure 12(b): For less than N_1 units, the total cost for the FPGA solution tends to be lower than for the ASIC variant.

  Due to higher design verification and photo mask costs using advanced *Process Technologies*, e.g. 90nm or 65nm, the NREs are expected to grow in the future, as indicated in Figure 12(b), $NRE_1 \rightarrow NRE_2$. Thus, the crossover-point A will move ($\rightarrow B$) to higher quantities N_2, further supporting the FPGA solution versus ASIC.

In this *ASIC-replacement* use case, FPGAs are used in a static reconfiguration model, implementing exactly one specific function. Thus, OTP, EEPROM, or SRAM configuration memory technologies are typically employed, whereas configuration speed is of no importance.

2.3.3 Use Case C: *Multi-function Co-Processor*

In contrast to use case *B*, in case *C* the external configuration memory *ECM* is not directly connected to the PLD, but rather to a subsystem. The subsystem implements a *Configuration Interface (CI)* which provides access to the configuration port *CP* of the PLD, as shown in Figure 11(c). In this case, the subsystem *SubS* is in a position to fully control the PLD's configuration. The subsystem *SubS* may write a *Full Configuration Bitstream* into the PLD, i.e., entirely change the PLD's function, at any point of time. In this system architecture, the configuration and, thus, the functionality of the PLD is no longer fixed, but may be changed arbitrarily during run-time.

New execution scenarios for PLDs become possible: Not only one, but several different co-processor functions can be loaded consecutively during run-time and made available, depending on the system's need at a particular moment [PB99, EP00, Eis02, PP03, Eka04].

As a consequence, the controlling subsystem *SubS* has the task of providing a configuration to the PLD before any functionality can be invoked. *SubS* turns from an ordinary *Function Caller* to a *Resource Manager* which governs the PLD as a dynamically allocatable resource. Furthermore, the PLD changes from a static fixed function element to a multi-functional device.

In this use case, the disadvantages of PLD's regarding performance, power consumption and cost (as discussed in 2.3.2) are compensated by their flexibility, i.e., the potential to utilize the PLD for different purposes.

The speed with which a new full configuration bitstream can be downloaded to the PLD, i.e., the function of the PLD can be changed, is crucial for the usage and feasibility of the reconfiguration model to operate. In addition, the execution time t_{exe} should be much longer than its configuration time t_{conf} in order to achieve high efficiency and utilization of the PLD. SRAM-based FPGAs with parallel configuration ports optimally meet this requirement [XAPa], since they allow for fast reconfiguration.

From an system designer's point of view, the application needs to be partitioned, in order to load and execute clusters of tasks in the PLD [PB99, ABT03]. The partitions must preserve the data dependencies defined by the DFG, and the size of the clusters must not exceed the logic capacity of the PLD.

During runtime, a *Scheduler* is in charge of determining which configuration needs to be loaded into the PLD, depending on the application execution state.

2.3.4 Use Case D: *Versatile Programmable Logic Resource*

Use case *D* exhibits identical HW architecture as use case *C*, as conceptualized in Figure 11(c), but additionally, the subsystem *SubS* executes *partial reconfiguration operations* on the FPGA. This leads to the following consequences regarding the internal structure of the FPGA and its functionality, viewed from the cooperating subsystem:

- *Accessibility to Single Functions*
  The scenario in use case *C* determines that all functions on the device can only be controlled *en bloc*; that is, *all* functions start after a reconfiguration process has completed, and all functions stop at the latest, when a new reconfiguration process starts, respectively. Partial reconfiguration allows for loading/removing *single* functions to/from the device. In this way, the device can be used in a much more flexible and efficient way.
- *Reconfigurable Area Structure*
  Whereas in full configuration mode, where each configuration establishes *all* the logics and structures needed to execute the circuitry, partial reconfiguration scenarios induce the necessity of permanently available infrastructural elements supporting this operational mode.
- *Management Complexity*
  More sophisticated functions need to be executed in the device which controls the FPGA, such as *resource management*, *task management*, etc.

This use case is the most promising and contains the highest potential for further exploitation of the abilities of SRAM-based FPGAs.

2.4 Motivating Case Study

In order to experimentally investigate the benefits of use case *C* and *D* (as described in the previous section), we have realized a case study system. The case study system implements a set of sample applications from the networking, signal processing, and multimedia domain, respectively.

2.4.1 Case Study System and Sample Applications

Figure 13 depicts the block schematics and data-flow diagram of our case study system. It consists of an FPGA and a number of external devices $C_{1..7}$, ranging from simple push-buttons (as $SW_{1..3}$) to audio codec C_7 and physical ethernet transceiver C_2. The system executes seven different applications $A_{1..7}$. All applications split into multiple tasks, e.g. $T_{1..10}$, and interact with FIFO-buffers $Q_{1..8}$, and/or external devices, respectively.
Table 3 lists the applications $A_{1..7}$ and the respective tasks they consist of. For example, application A_7 implements a simple *audio waveform generator*. As soon as the push-button SW_3 is pressed, the waveform generator task (WavGen) T_8 is enabled and fills Q_5 with the generated audio waveform data. The device driver task (CDCDrv) T_6 reads out Q_5 and drives the audio codec (C_7). For a detailed description of the complete case study system and its applications, we refer to [WP03b, WP03c], and [LZ02, Rup03, ER03], respectively.

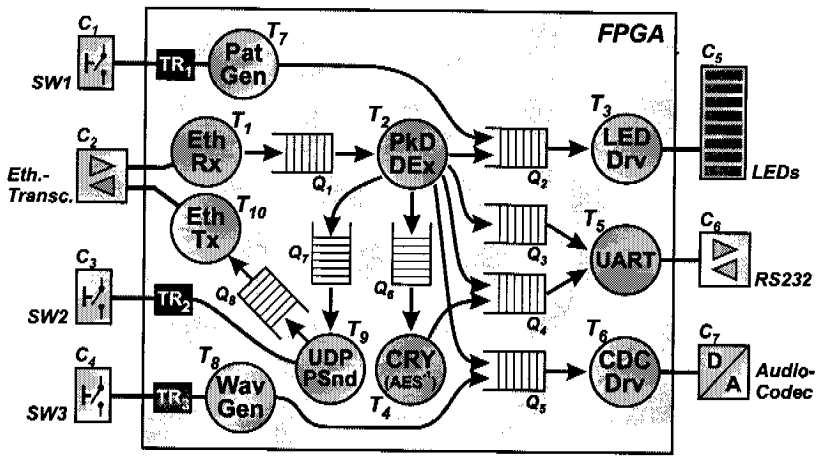


Fig. 13: Data-flow graph and block schematics of the case study system.

Application	Involved Tasks, Buffers, and Ext. Devices
A_1: Local Visual Pattern Generator	$C_1, TR_1, T_7, Q_2, T_3, C_5$
A_2: Remote Visual Pattern Display	$C_2, Q_1, T_2, Q_5, T_6, C_7$
A_3: Remote ASCII Dump (plain)	$C_2, T_1, Q_1, T_2, Q_3, T_5, C_6$
A_4: Remote ASCII Dump (crypt.)	$C_2, T_1, Q_1, T_2, Q_6, T_{4b}, Q_4, T_5, C_6$
A_5: Network Audio Player	$C_2, T_1, Q_1, T_2, Q_7, T_9, Q_8, T_{10}, Q_5, T_6, C_7$
A_6: Telemetry Data Sender	$C_3, TR_2, T_9, Q_8, T_{10}, C_2$
A_7: Local Audio Generator	$C_4, TR_3, T_8, Q_5, T_6, C_7$

Tab. 3: Case study applications $A_{1..7}$ and involved tasks $T_{1..10}$, triggers $TR_{1..3}$, FIFO buffers $Q_{1..8}$, and external devices $C_{1..7}$.

As target platform for the case study, we used the XESS XSV-800 prototyping board [XSV, XES] featuring a XILINX VIRTEX XCV-800 FPGA and a multitude of external components. This FPGA type includes an array of 56×84 ($= 4704$) CLBs .

Figure 14 depicts the floor-plan of the implemented case study system [XFE] and the mapping of the regions to the appropriate tasks.

Table 4 lists the size of each task $T_{1..10}$ (including queues $Q_{1..8}$) in percentage of the available reconfigurable area on the XILINX XCV-800 FPGA.

2.4.2 Application- and Task-Activity Analysis

We present two approaches to reduce the amount of required reconfigurable area which we denote as *Selective Application Activation* and *Sequential Task Activation*, respectively.

Task	CLBs	Area [%]
T_1 : Eth RX (ethernet packet receiver, MII), incl. Q_1	763	16.2%
T_2 : PktD DExt (packet discriminator and data extractor)	297	6.3%
T_3 : LED Drv (LED driver), incl. Q_2	104	2.2%
T_{4a}: CRY K (AES sub-key generator)	447	9.5%
T_{4b}: CRY D (AES decoding engine), incl. Q_6	509	10.8%
T_5 : UART (serial interface driver), incl. Q_3 and Q_4	62	1.3%
T_6 : CDC Drv (audio codec driver), incl. Q_5	189	4.0%
T_7 : PatG (Pattern generator)	85	1.8%
T_8 : Wav Gen (Waveform generator)	190	4.1%
T_9 : UDP PSnd (UPD packet sender), incl. Q_7	236	5.0%
T_{10}: Eth TX (ethernet packet sender), MII, incl. Q_8	400	8.5%
Total	**3282**	**69.7%**

Tab. 4: Area requirements of each task $T_{1..10}$ in CLBs and percentage of the full reconfigurable area of a XILINX VIRTEX XCV-800 device.

I) Selective Application Activation

Figure 15 shows the *task activity pattern* for (a) the system in idle state and (b)..(f) for selected applications A_1, A_7, A_6, A_5, and A_4. The regions marked black are active (i.e., the circuits are actually executing), whereas the circuits in the gray regions are inactive.

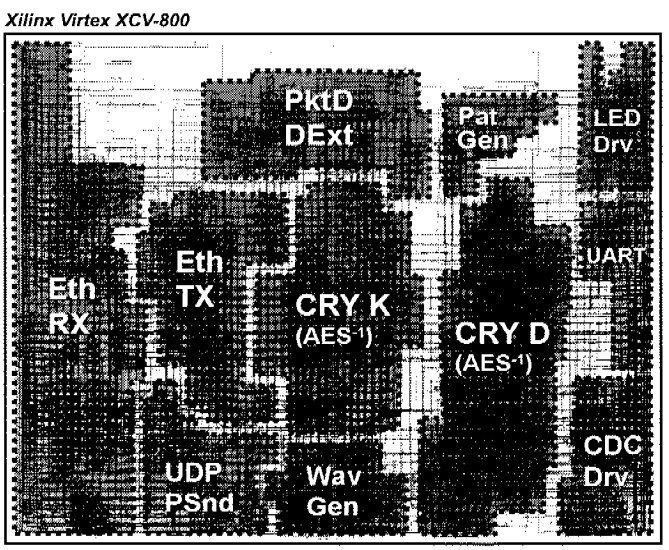


Fig. 14: FPGA floor-plan of the case study system implemented on XILINX VIRTEX XCV-800.

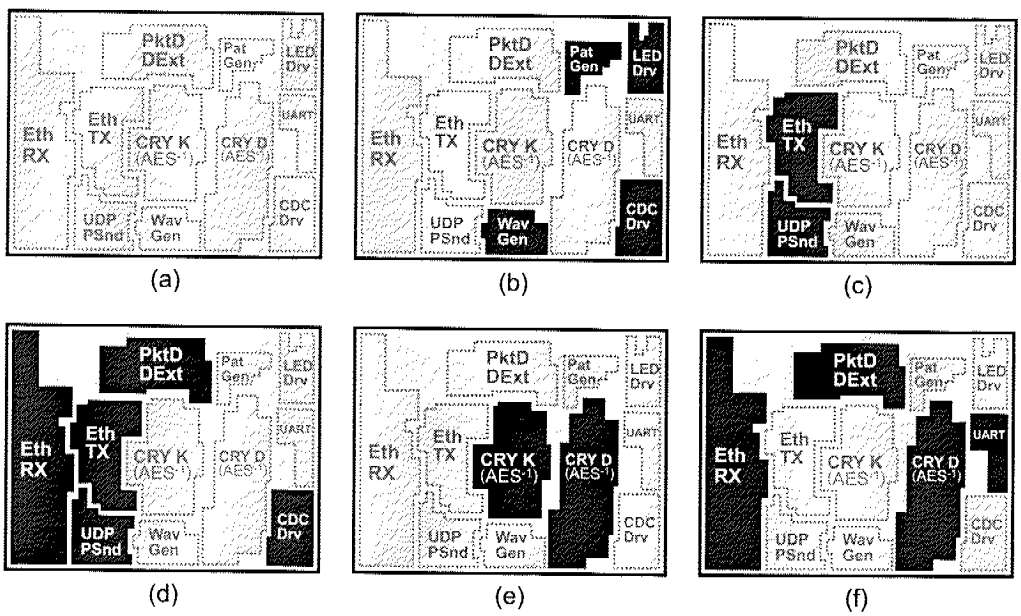


Fig. 15: Task activity pattern when (a) system is in idle state, (b) local visual and audio generators are running (A_1 and A_7), (c) telemetry data sender executes (A_6), (d) network audio player (A_5) is running, (e) the AES decoder is initializing, (f) remote ASCII dump (decrypted) application (A_4) executes.

Table 5 itemizes the amount of CLBs that each application $A_{1..7}$ requires for execution, and its percentage of the total FPGA area[6]. It follows, that the utilization of the FPGA is at most 40.1%, i.e., when the network audio player application A_5 is running. This observation is correct, as long as only *one* application at a time is actually executing in the system.

We conclude that an FPGA offering a logic capacity of only 40.1% of the capacity of an XCV-800 would be sufficient to achieve the same system functionality. Hence, a XCV-400 providing 2400 CLBs could be used, assuming that there is an additional *configuration control unit* responsible for reconfiguring the FPGA as soon as a different application should execute. This scenario corresponds with use case *C*: All bitstreams representing the applications $A_{1..7}$ are stored in an external configuration memory. To change from one application to another, the appropriate bitstream is downloaded as a *full reconfiguration process* to the FPGA. All circuits executing in the FPGA are replaced during the full reconfiguration, even those which could be reused by applications to be executed next.

The fact that a smaller FPGA can be employed has a significant impact on

[6]Note that task T_4 splits into two subtasks T_{4a} (sub-key generator) and T_{4b} (AES decoding engine). Since the sub-key generator is only required during T_4's initialization, the area requirement for application A_4 can be considered including T_{4a}, or excluding T_{4a} (in brackets).

Application	Required CLBs	Area [%]
A_1: Local Visual Pattern Generator	189	4.0%
A_2: Remote Visual Pattern Display	486	10.3%
A_3: Remote ASCII Dump (plain)	1122	23.8%
A_4: Remote ASCII Dump (crypt.)	(1184) 1631	(25.2%) 34.6%
A_5: Network Audio Player	1885	40.1%
A_6: Telemetry Data Sender	636	13.5%
A_7: Local Audio Generator	379	8.0%

Tab. 5: Area requirements of each application $A_{1..7}$ in CLBs and percentage of the full reconfigurable area of a XILINX VIRTEX XCV-800 FPGA.

the commercial side: The price for a XCV-400 device is $> 57\%$ *lower* than for a XCV-800, as shown in Table 6. Even if a XCV-600 were chosen to implement the complete system, the cost reduction for the FPGA would be $> 40\%$.

The advantageous characteristic of such a system is the flexible use of the FPGA which can lead to significant cost savings in terms of the reconfigurable device. The disadvantage is that only *one* application can be executing at a time.

However, since only *full reconfiguration processes* are executed, the potentialities of modern FPGAs are not fully exploited.

II) Sequential Task Activation

We claim that *streaming oriented applications*, such as networked audio players [DW02, DPP02, HL04], voice over IP (VoIP) phones, or internet TV players [DJR01], hold additional potential to further reduce the required amount of reconfigurable resources.

We consider a sample application as depicted in Figure 16, which is similar to A_5 (network audio player) of the case study system. A device driver DD_1 receives data packets containing audio data from an external device C_1, e.g. from

FPGA Type	CLB Array	Price [USD]
XCV-300	32 × 48 (=1536)	269.–
XCV-400	40 × 60 (=2400)	378.–
XCV-600	48 × 72 (=3456)	639.–
XCV-800	64 × 96 (=4704)	896.–

Tab. 6: Prices of XILINX VIRTEX devices, according to XILINX' price list, 05.10.2001, for speed-grade 4 (-BG432C) type.

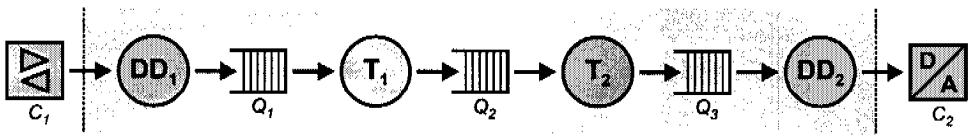


Fig. 16: Streaming oriented sample application consisting of two device drivers $DD_{1..2}$, two tasks $T_{1..2}$, and three FIFO buffers $Q_{1..3}$

a network transceiver, at an average rate ($\frac{1}{\Delta t_p}$). DD_1 forwards the data-packets to FIFO-buffer Q_1. The tasks T_1 and T_2 (coupled by Q_2) process the packet's data, e.g. MP3-decompressing and/or audio filtering. T_2 writes its output to Q_3 which acts as play out-buffer for device driver DD_2. DD_2 itself is connected to an external device C_2, e.g. to an audio codec.

Ordinary Application Execution Scenario

Figure 17(a) shows the status of $DD_{1..2}$, $T_{1..2}$, and $Q_{1..3}$ over time: In time intervals Δt_p, data-packets are received by DD_1 and stored in Q_1. As soon as Q_1 contains data, T_1 starts to readout Q_1, process the data and forwards it into Q_2. This immediately causes T_2 to readout data from Q_2, process and store it in Q_3, and so on. The decisive observation is that the activation of both tasks T_1 and T_2 is controlled by the fill-level of Q_1, and Q_2, respectively. For a certain amount of time, T_1 as well as T_2 are *concurrently* active. This means that the FPGA has to offer enough reconfigurable area to accommodate both tasks at the same time. The depth of the FIFO-buffers Q_1 and Q_2 are dimensioned such that the content of *one* received data-packet can be stored (in case of Q_1), and the amount of data produced by T_1 can be stored (in case of Q_2), respectively. The buffer dimensioning assumes that T_1 has completely read out Q_1 before the next data-packet arrives, and T_2 has read Q_2 before T_1 stores new data of the next packet. Since Q_3 acts as play-out buffer for DD_2, the depth must suffice to prevent buffer underflows.

Event Controlled Application Execution Scenario

The timing behaviour of the same application, but controlled by *events*, is illustrated in Figure 17(b): In this scenario, tasks are activated as a result of events. Events are generated based on buffer fill-levels of $Q_{1..3}$ and thresholds $Th^+_{1..2}$, Th^-_3, and sent to a *configuration control unit*. As soon as the fill-level of Q_1 exceeds threshold Th^+_1, event E_1 is generated (cf. time 11). The task scheduler (which is part of the configuration control unit) receives E_1 and decides to activate T_1. The configuration controller loads T_1 by a *partial configuration process* in the FPGA. At time 20, the fill-level of Q_2 exceeds Th^+_2 which causes Event E_2. Although T_1 has not yet completely read out Q_1, the scheduler decides to stop

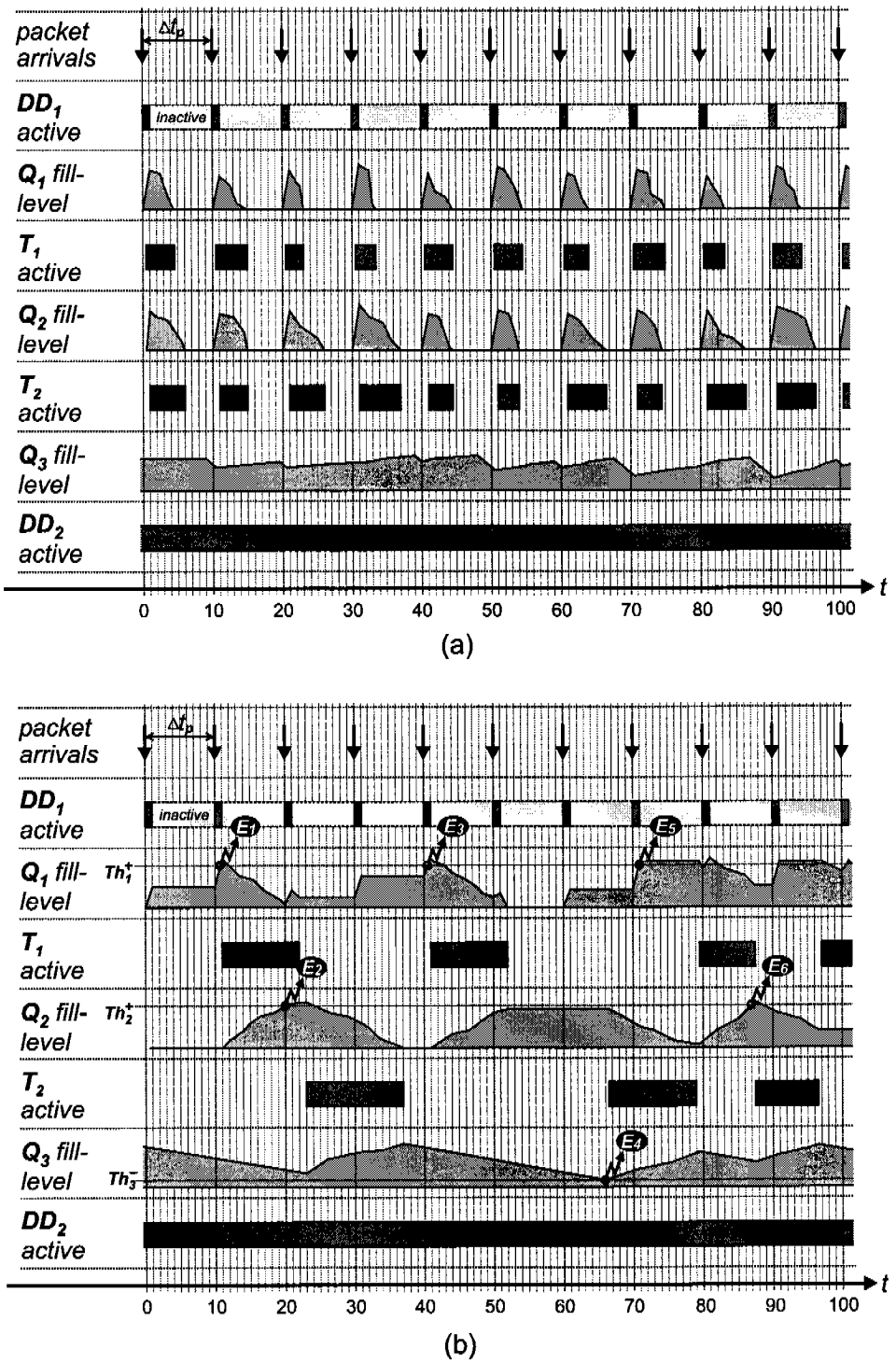


Fig. 17: Timing behaviour of the sample application: in (a) the ordinary application execution scenario, and (b) the event controlled application execution scenario (events $E_{1..6}$).

T_1, frees the reconfigurable resources occupied by T_1, and loads task T_2 (again by a partial reconfiguration process) to the same physical location that was previously used by T_1. At time 37, T_2 finishes, since all data from Q_2 are processed. The resources occupied by T_2 are freed. At time 41 Q_1's fill-level again exceeds Th_1^+, which causes the scheduler to load T_1 (to the same location). At time 66, event E_4 occurs, since the fill-level of the play-out buffer Q_3 underruns threshold Th_3^-. The scheduler stops T_1 and loads T_2 again, to prevent a play-out buffer underflow, and so on.

The scenario in Figure 17(b) shows, that at any point in time, either task T_1 *or* T_2 is active. At no point in time, both tasks need to be active. Both device drivers $DD_{1..2}$ and the FIFO-buffers $Q_{1..3}$ need to be present in the system at all times, since DD_1 needs to be permanently ready to receive an incoming data-packet, and DD_2 is in charge of continuously driving C_2.

We claim that the required reconfigurable resources can be reduced (compared to the previous scenario) if a configuration control unit pro-actively manages the reconfigurable area of an FPGA by loading and unloading the hardware tasks T_1 and T_2, respectively. The FPGA merely needs to be large enough to accommodate $DD_{1..2}$, $Q_{1..3}$, and the larger one of T_1 and T_2.

The load process of a hardware task involves *partial reconfiguration* of the FPGA. Circuits representing $DD_{1..2}$ and $Q_{1..3}$ reside in a *static part* of the FPGA and are not altered by the reconfiguration.

2.4.3 Conclusions

Based on the analysis of the case study system, we draw the following conclusions:

- The task execution sequence is no longer controlled locally by each task or application itself, but in a centralized form by configuration controller.
- The configuration controller needs system-wide information in order to make decisions, such as task hardware task scheduling.
- In order to collect status information and to execute control functions, the configuration controller needs connection to each system element, e.g. to $Q_{1..3}$, to continuously monitor thier fill-levels.
- A configuration controller consists of several functional modules, such as *task scheduler* or *reconfigurable resource manager.*
- As a consequence of the delayed activation of tasks (caused by reconfiguration times and resource conflicts) the required buffer-depth and the application latency increase.
- Several applications consisting of different tasks may execute concurrently.

The configuration control unit plays a central role in this event driven execution scenario. Analogous to an *Operating System (OS)* that manages resources and controls the execution of software tasks, we henceforth denote the configuration control unit as ***Reconfigurable Hardware Operating System (RHWOS)***.

2.5 RHWOS Vision

Adopted from insights in the previous section, we derive a set of characteristics and visionary requirements of an RHWOS.

An RHWOS should

- manage and execute several software and hardware tasks concurrently.
- offer similar services to hardware tasks as an RTOS provides to software tasks, e.g. inter-task communication, memory services, or access to I/O devices.
- efficiently manage the reconfigurable area of the FPGA as a dynamically allocatable system resource.
- hide the complexity of partial reconfiguration processes and the underlying FPGA technology from the application developer.
- support application developers in the debugging process by providing monitoring and triggering facilities that give insight into the interaction between the application objects.

In the following chapters we will elaborate on different conceptual, algorithmic and realization issues related to RHWOS and present our solutions for a number of novel problems which are typical for RHWOS.


Seite Leer /
Blank leaf

3

RHWOS Concepts and Architecture

Overview
In this chapter, we will describe the concepts of an RHWOS on a technology-independent level. We analyze its architecture, identify the necessary modules, and describe how these modules interact internally and cooperate with the user applications.

Problem Statement
The functional range of an RHWOS exceeds that of an RTOS by two major aspects: an RHWOS (i) manages a partially reconfigurable logic device as a dynamically allocatable system resource, and (ii) manages both software *and* hardware tasks. These two differences have a major impact on the application design, as well as on the run-time architecture of an RHWOS compared to an RTOS. Open questions are: Which OS modules and functions are new? How do the RHWOS modules interact with each other? What application, task, and programming model is defined by an RHWOS?

Contributions and Results
First, we model the relevant characteristics of the target platform. On this basis, we define our RHWOS concept consisting of an application programming model, hardware and software task models, and services offered by the RHWOS. Then, we present our approach for both a compile-time and run-time system. The compile-time system defines a methodology for the design of RHWOS driven applications, whereas the run-time system constitutes the environment in which the applications are actually executed. Finally, we discuss performance and benchmarking issues typical for RHWOS which allow for comparing different imple-

mentations of complete RHWOS or single functions thereof.

The distinguishing point of this work compared to existing work is the *completeness* and *consistency* of the presented concepts.

3.1 Background and Related Work

Related work in reconfigurable operating system research generally splits into two categories. One category describes concepts and frameworks disregarding detailed implementation or optimization problems. The other category deals with isolated operating system functions that try to develop methods and algorithms focussed on a particular sub-problem, embedded in a certain (often arbitrarily) model environment.

Related Work in Operating System Concepts

Brebner [Bre96, Bre97] was among the first to propose an operating system approach for partially reconfigurable hardware. He defined *Swappable Logic Units (SLUs)*, which are position independent hardware tasks that are swapped in and out by the operating system.

Burns et al. [BDH+97] realized that, instead of adding run-time support for the reconfigurable logic in every application, it is more efficient to develop a separate *run-time system*. The run-time system provides a set of support functions that addresses the common requirements of several applications. Their OS approach defined various modules as *virtual hardware manager*, a *transformation manager*, and a *configuration manager*. These modules include several essential functions, which we will break down into more specific ones (cf. Section 3.5).

Jean et al. [JTY+99] discussed an online scenario where a resource manager schedules arriving hardware tasks to a farm of FPGA. In [PB99], Purna et al. present methods to partition a single application (that is too large to fit in a reconfigurable device), into several smaller tasks to form a sequential set of configurations. This can be viewed as an operating system function. However, since each task occupies exactly one FPGA, partial reconfiguration is not an issue, thus, we do not further consider this research direction.

Wigely et al. [WK01a] identified a list of modules and services that must be offered by a reconfigurable operating system, such as dynamic partitioning, allocation, placement and routing. In [WK01b] functions providing circuit protection, cache management and inter-process communication were added. In [WK02b] the authors present *MARC.1*, an architectural concept that provides services for loading and execution of pre-designed user applications which are made of several *application modules*. The modules have a dependency relationship that is represented by a *task graph*. MARC.1 divides the task graph into *application*

partitions, which are then executed on the FPGA within *logic frames*. This seems to be the first example of an RHWOS which covers the most important aspects, from application design to a run-time system. However, the descriptions remain on a very high level of abstraction. The functions and modules presented do not constitute a complete system and their interactions are not clearly defined.

The reconfigurable computing research group at *IMEC* [IME] are active in several areas related to RHWOS:

In [MMB+03, NMB+03, NCV+03] IMEC presents their *OS4RS*[1]. The main focus of this operating system is the runtime support for multimedia applications. In [MMB+03], Marescaux et al. denote typical functions of their OS4RS, as *task creation/deletion, dynamic heterogeneous task relocation*, etc. They also point to an important characteristic of an RHWOS, the *debug ability* and *observability* of OS elements and user tasks supporting the application development process. In [MNC+03], IMEC focuses on one specific property of tasks, which they call *relocatability*, and present a design environment *OCAPI-XL* that supports the development of applications in such a way that hardware and software versions of tasks with equivalent internal state representations are automatically generated. By means of so-called *switch points*, tasks previously running in hardware can interrupted and restarted in software, and vice versa. This represents a unique principle of an RHWOS: A specific function can be performed in different quality levels by choosing its *execution engine*, i.e. CPU or FPGA. IMEC presented a prototypical implementation called *T-ReCS Gecko* [Gec].

In conclusion, a complete and consistent conceptual description of an operating system that dynamically manages a partially reconfigurable device is inexistent so far in related work. With this work, we aim at closing this gap.

3.2 Reconfigurable Embedded System Model

The subsequent modeling of reconfigurable embedded systems provides the basis for our RHWOS concepts.

3.2.1 Target Architecture Model

Figure 1 (on page 3) conceptualizes the target architecture we consider for a *Reconfigurable Embedded System*. A CPU and a partially reconfigurable device, e.g. a SRAM-based FPGA, are tightly coupled. Both are connected to a number of external components C_i, such as memories and I/O devices.

The tight connection of CPU and FPGA is implemented by three different interfaces:

[1] Acronym, stands for *O*perating *S*ystem *for* *R*econfigurable *S*ystems

- The *C/R* interface enables the CPU to fully control the FPGA's *Reconfiguration/Readback Port*. In this way, the CPU is in a position to write configuration data to the FPGA, and read configuration and status information from the FPGA, respectively, at every point in time.
- A number of wires form the *General-Purpose Interface (GPI/O)* which allows for high-bandwidth communication between the CPU and FPGA.
- The *Clock Interface (Clk)* allows the CPU to fully control all clock nets present in the FPGA.

The CPU may be attached to some additional logic components to physically implement the interfaces with the FPGA, and external devices, respectively. To simplify matters, we disregard these components and denote the entire group as CPU.

The target architecture can be technically realized in different variants: As a multi-component system, the CPU and FPGA are discrete components mounted on a PCB [XFB, Tre, Gec], whereas the *Configurable System-on-a-Chip (CSoC)* solution comprises both components in a single device. The two variants are equivalent from an RHWOS conceptual point of view.

The CPU in a CSoC can be implemented either as a hardwired (silicon) IP core [XV2b, A7S, Exc, Cha], or as an instance of a soft CPU core [LEOa, XMB, NIO, LEOb]. Even though the CPU is part of the FPGA's reconfigurable area, it is assumed that the CPU remains unchanged during whole operation.

In this work, we focus on single processor architectures, rather than multi processor systems.

3.2.2 CPU Device Model

A *CPU* is a processing element which executes a defined set of instructions. In our concept, we make no distinction between devices with specialized processor internal architectures, such as *Micro-controllers*, *DSPs*, *stand-alone CPUs*, etc. We focus on the following properties, relevant for RHWOS:

- ***Sequential Instruction Execution***
  The CPU executes each instruction in a strictly sequential manner[2]. Therefore, *multitasking* can only be practiced in a *pseudo-multitasking* form, i.e. by frequently switching from one stream of execution to another, driven by time or events.

  Modern CPU devices include on-chip peripherals, such as I/O modules, timers, interrupt and DMA controllers, etc., that can be triggered by the CPU and execute functions as intelligent sub-systems in *parallel*.

[2]Multi-scalar, multi-threaded, or VLIW (Very Large Instruction Word) processors show different variants of execution parallelism. However, we do not consider such processors in this work.

- ***Code, Data and Configuration Memory***
  Various types of memories may be attached to the CPU, in order to store the executable code, application data, and configuration bitstreams, respectively.

  As in an RTOS, the concepts of *Virtual Memory Management* are not implemented. Data and code to be executed are assumed to be present in the memory during the whole run-time in order to allow for hard real-time scheduling. Effects caused by *page faults* would introduce unpredictable delays in code execution.
- ***Interrupts***
  Interrupts with different sources and priorities can be handled by the CPU. An Interrupt is the basic mechanism to give an RTOS full control over the system, i.e. to cancel the execution flows of tasks.
- ***Constant Clock Speed***
  The CPU is driven by a clock with a constant frequency f_{Clk}.

This model is consistent with that used in commercially available RTOS [VxW, Vir, TDB, QNX, WEm, ELi].

3.2.3 FPGA Device Model

SRAM-based partial reconfigurable FPGAs are the central part of the system (and main focus of this thesis). From a conceptual point of view, the following characteristics are relevant to discussing FPGAs as components in an RHWOS:

- ***Reconfigurable Resources***
  An FPGA contains a well defined number of *Reconfigurable Logic Units (RLUs)*, *Input/Output Blocks (IOBs)*, and *Special Function Blocks (SFBs)* which can be interconnected by *Routing Elements*. Several combined RLUs and SFBs form circuits that jointly execute any complex logic function. IOBs provide connectivity to external devices physically attached to the FPGA.

  Each RLU contains storage elements which represent the current state of the circuit.

  The RLUs, IOBs, and SFBs are the main resources of an FPGA. Each of them is either fully *occupied* or *free* at a given point in time. If a resource is occupied, it means that it is currently integrated into a circuit, otherwise it is considered as free.
- ***Configuration Memory / Configuration/Readback Port (C/R)***
  Each reconfigurable element gets its current configuration settings from the *Configuration Memory* which is modeled as *Static RAM (SRAM)*.

  The *Configuration/Readback Port* provides write (configuration) and read (readback) access to the configuration memory. The granularity of write and read accesses can range from one configuration bit to indivisible groups of

bits. *Full Configuration/Readback* denote operations in which the whole configuration memory is accessed, whereas *Partial Reconfiguration/Readback* only affect a part thereof.

The *C/R* is modeled as a half-duplex communication channel with a limited bandwidth in each direction. Configuration and readback processes each occur sequentially.

- ***Configuration/Readback Bitstream File***
  Configuration bits are organized in *Bitstreams*. Besides the configuration and status bits, a bitstream holds additional structural information, such as configuration memory addresses and checksum bits.
- ***Clock Nets (Clk)***
  An FPGA may offer up to c physically independent *Clock Nets* ($Clk_{1..c}$). Each RLU, SFB, and IOB can be configured to be driven by one of these clock sources.

Most modern FPGAs follow this model [Xil, Alt]. In contrast, the CHAMELEON SYSTEM CS2000 family [Cha, PLP+03] can reconfigure all its units in only one clock cycle. This is realized by two configuration planes and a configuration controller, which can switch the background plane into the active plane in just a cycle. However, there is also a bandwidth limit in configuring the background plane. In this work, we do not consider this device type.

In order to focus on the task and resource management aspect of an RHWOS in more detail, we will refine the FPGA model in Chapter 4.

3.3 RHWOS Model

Based on the model of a reconfigurable embedded system, we construct an architectural concept of an RHWOS and define modules and functions which are executed inside.

3.3.1 Terminology

For the sake of clarity, we define the following terms in connection with RHWOS. The terms are for the most part consistent with the already well-established wording in the RTOS community [Tan87, SG98].

- ***OS / RTOS / RHWOS***
  OS generally denotes the entirety of all modules forming the operating system. *RTOS* is a generic term for a class of operating systems that provide support for real-time applications. An RTOS exclusively executes software

tasks. An *RHWOS* is an RTOS augmented with functions to additionally manage partially reconfigurable devices and execute hardware tasks.

- ***OS Module***
  Subsystem of an OS with a well-defined function and interface. May execute either in software or in hardware.
- ***OS Function***
  Well-defined function executed by a OS module. OS modules are application independent.
- ***OS Object, OS Service***
  An *OS Object* is an instance of an OS module. It only exists during run-time (run-time object). An *OS Service* is a function of an OS object visible to a user task.
- ***Hardware/Software Task, User Application***
  A *Hardware* or *Software Task* executes application specific functions, either implemented in hardware or in software. Tasks are the result of an *application decomposition* step. A *user application* is a set of hardware/software tasks cooperating with OS objects.
- ***Platform***
  Hardware on which the RHWOS and user applications run.
- ***Compile / Synthesize***
  Processes to translate the source code of software or hardware to a platform specific, executable form. We only use the term *compile* for both processes.

3.3.2 Application and Programming Model

An RHWOS offers an application and programming model, in the sense that

- applications are composed of a set of cooperating user tasks and OS objects, as depicted in Figure 18,
- during run-time, a task scheduler decides which task needs to be activated in order to ensure the entire application's functionality. If more than one task is ready to execute, the task activation follows a certain scheduling policy.

A *Task* is the main abstraction in any OS driven system. A single task executes a specific sub-function of an application and exposes a well-defined interface to the OS. Each task is connected to one or more OS objects which provide essential services to tasks, such as task communication, task synchronization, etc.

In contrast to an RTOS, an RHWOS can execute both software tasks and hardware tasks at the same time. An application programmer may decide to implement the same task (with exactly the same functionality) in both software and

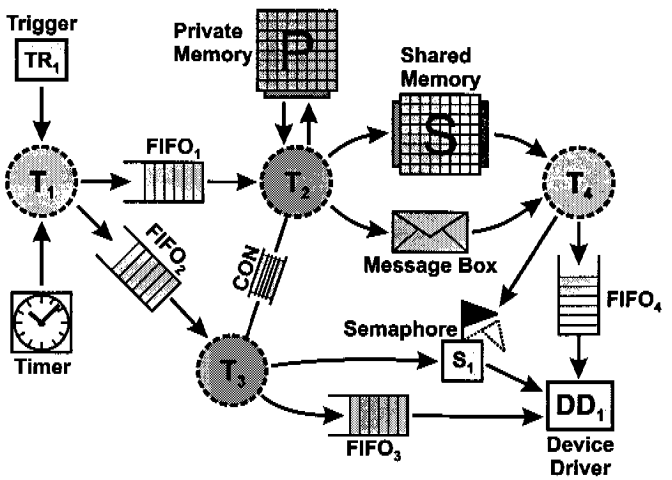


Fig. 18: Example application consisting of cooperating user tasks $T_{1..4}$ and OS objects $FIFO_{1..4}$, TR_1, DD_1, *CON*, memory objects (private and shared), a message box, timer, and a semaphore S_1.

hardware. The RHWOS determines at run-time which version to choose. The decision is based on criteria such as (i) current resource situation, (ii) performance requirements, and (iii) timing constraints. As a result of a dynamic application situation, the RHWOS may decide to stop the execution of a software task and restart it in hardware, and vice versa.

3.3.3 User Task Model (Hardware- and Software Tasks)

User tasks implement the application specific functions of the system. Tasks are the result of the *application decomposition*, which is the fist step in the design of any OS driven application. In contrast to an RTOS, an RHWOS environment allows tasks to be implemented in hardware or software.

From an RHWOS point of view, there is no difference in activating a hardware task or a software task. An OS has no knowledge about the internal functions of a task. An OS is just aware of in what sequence, dependency and timing constraints the tasks need to be executed.

The concept of a *Hardware Task (HT)* is the major difference between an RTOS and an RHWOS. A HT features a set of ports, as shown in Figure 19. The function carried out by a HT is invisible to the RHWOS.

The *control port* P^c connects a HT to the run-time manager of the RHWOS. Using this interface, the RHWOS and the HT exchange control information, i.e. all commands and information except application data. Examples for commands issued from the RHWOS toward the HT are: task management commands, such as reset, start or stop signals. In the opposite direction, the HT signals information about its current internal state to the RHWOS, e.g. *ready for execution*, after

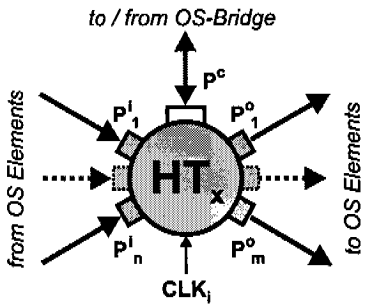


Fig. 19: Hardware task HT_x with n input ports $P^i_{1..n}$, m output ports $P^o_{1..m}$, control port P^c, and i clock inputs CLK_i.

a reset/initialization sequence, *finished*, after the HT has completed execution, or *critical section*, when the HT is currently executing a part in which the RHWOS must not interrupt (preempt) the task's operation. In addition, a HT implements a number of *input ports* $P^i_{1..n}$ and *output ports* $P^o_{1..m}$. Input and output ports are implemented as independent parallel interfaces and allow a task to access operating system objects in order to transfer application data. Finally, a *clock port CLK* is required to drive the task with a proper clock signal.

Tables 7 and 8 compare the properties of software and hardware tasks in our RHWOS concept, during compile-time, and run-time, respectively.

3.3.4 RHWOS Objects and Services

RHWOS objects offer services in such a way that they can be conveniently accessed by tasks. The rational behind services is to hide the complexity of one or more functions, and to define a level of abstraction. The interface between an RHWOS object and a task is well-defined, but it is *invisible* to the task *how* a service is actually realized.

An RHWOS provides a rich set of OS objects and services (like an RTOS). The services can be categorized in *task management*, *task communication*, *resource management*, *I/O management*, and *time management*, respectively. All OS objects and services together constitute the *application programming interface (API)* of an RHWOS. Examples of basic RHWOS objects and services include:

- ***Private and Shared Memory***
  Private and shared memory are services through which a task can access memory. The interface offered to the tasks is RAM-like, i.e. implementing address, data, and control wires. The RHWOS transforms the memory requests from tasks to physical memory accesses. The transform process includes recalculation of virtual addresses into physical ones. *Private memory*

Characteristic	*Software Task (ST)*	*Hardware Task (HT)*
Creation / Development	High-level procedural programming language (C, C++, etc.) or Assembler.	Hardware description language (VHDL, Verilog, etc.)
Size	Measured in number of bytes. Usually occupies a contiguous block of memory.	Measured in number of RLUs. Usually occupies a contiguous block of RLUs.
Building blocks	Consists of a number of instructions to be sequentially executed.	Consists of a number of RLUs which form a logic circuit.
Execution Engine	CPU	FPGA
Starting and End Point	Defined entry point. Can have several end points. Activated by setting the PC to the address of the entry point.	Implements a control wire to reset the circuit. Activated by resetting the circuit.
Execution Structure	Sequential (no real multitasking, no real parallelism within task). Can call subroutines.	Several parallel areas of activity (real multitasking, computing in space). No subroutine calls.
Execution Speed	Defined by single clock frequency. Maximal clock frequency defined by CPU.	Defined by frequency of the clock net. HTs may have several clock domains with different clock frequency requirements (clock range).
Context	Context is represented by content of CPU registers and stack.	Context is represented by all storage elements within the RLUs.
Stack	Needs a stack to store return addresses within subroutines.	Has no stack.
Critical Section	Can have critical sections, but can only be in one at the same time. ST calls OS kernel routine to signalize entry and exit.	Can have critical sections. Can be in several critical sections at the same time. HT signalizes CS by a wire.
OS Interaction	By OS kernel calls (subroutine call).	OS service: via hardware port (data and control wires); Status: via control port.

Tab. 7: Comparison of single hardware and software task compile-time characteristics.

area can only be accessed by a designated task; accesses from other tasks to this area is prohibited by the RHWOS (memory protection). In contrast, *shared memory* is accessible by several tasks.

- ***FIFO Buffer / Message Box***

Function	*Software Task (ST)*	*Hardware Task (HT)*
Activation	Jump to entry point	Load bitstream, reset, activate clock signal(s)
Abort	Interrupt and remove task from task list	Stop clock(s) and mark the previously occupied RLUs as free
Preempt	Interrupt, save registers (context)	Stop clock, readback and extract context (storage elements)
Restore	Restore registers (PC, SP, etc.)	Insert context into raw-bitstream, load bitstream, activate task

Tab. 8: Comparison of hardware and software task run-time functions.

A *FIFO Buffer* is a memory and communication service, optimal for data stream oriented applications, offering write and read operations on constant data size (word). *Message Boxes* allow for transferring data with variable lengths from one task to another. Both services are available in either synchronous or asynchronous versions.

- ***Connector***
  Connectors are used to interlink tasks which may not be present in the system at the same time. However, the connections wires are controlled by the RHWOS.
- ***Timer***
  Timer objects offer time-based services to tasks, such as one-shot or periodic timer events.
- ***Semaphore***
  A *Semaphore* is an OS object used for task synchronization, i.e. to ensure mutual exclusion of tasks when accessing shared resources.
- ***Device Drivers***
  Device Drivers implement circuits that control external devices and offer services to user tasks. Encapsulating access to external devices in device drivers offers similar advantages as in RTOS: the access functions are independent of the actual I/O device and mutual exclusion issues can be resolved by the RHWOS. Furthermore, time-critical I/O protocols are handled by permanently resident optimized driver functions. This is an important issue for an RHWOS, as loading a driver function on demand with the same reconfiguration latency as a user task could easily violate timing constraints imposed by the external device.

RHWOS objects can be implemented either in software or in hardware. In the case of a hardware implementation, the service is accessed by a hardware interface.

Like hardware tasks, RHWOS elements implemented in hardware exhibit *input ports* $P^i_{1..n}$ and *output ports* to transfer data between the task and RHWOS element, and a *control port* P^c which allows the RHWOS to exchange control and status information with the OS element.

Software services are invoked by software calls. In any variant, a user task simply *calls* a service; then the RHWOS decides whether to execute the functions producing the service in software or hardware, i.e. actual execution is hidden from the user task. This decision can be taken during compile-time or run-time.

3.4 RHWOS Compile-Time System

The objective of this section is to present an integrated design methodology for applications to be executed in an RHWOS. We denote the entire environment in which all design and development phases takes place as the *Compile-Time System* of an RHWOS. We specify

- the design flow, and the information an application developer has to provide to the RHWOS in order to ensure proper operation of the entire system
- all input-, intermediate- and output-files which are used or are generated
- modules and functions executing during compile time.

The scope starts with analysis of the user application(s) and ends when all files needed to operate the system in run-time are created.

All descriptions remain on a conceptual level, pointing out the relevant relationships and mechanisms, but not addressing detailed implementation problems.

3.4.1 CTS Development Cycle (Overview)

We outline all the steps that must be carried out by an application developer. To simplify matters, we assume that all platform specific files are given. Figure 20 provides an overview; a detailed description of all steps and involved files follows in Sections 3.4.2 and 3.4.3.

- ▷ All applications need to be decomposed in a set of cooperating tasks and OS objects. The application developer defines which tasks will be implemented in hardware, or in software, or in both, respectively.
- ▷ The precedence, timing and data relations between tasks and OS objects, and on the application level need to be defined by the developer.

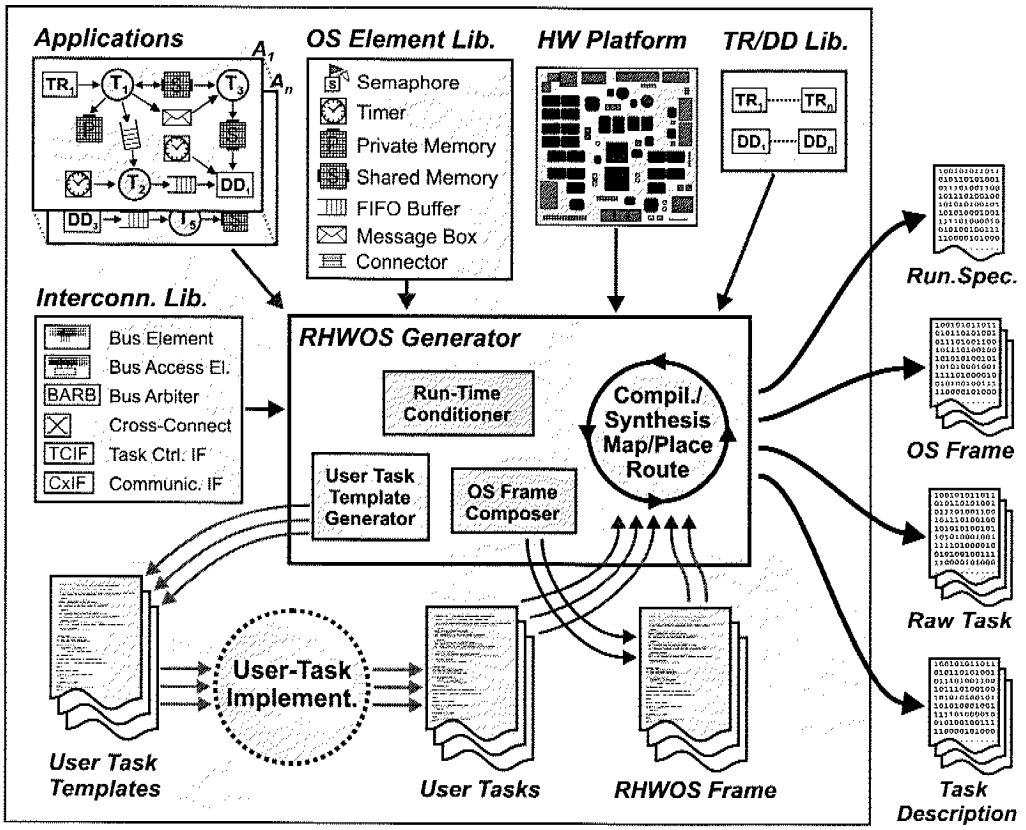


Fig. 20: Modules of the RHWOS Compile-Time System.

- ▷ Based on this information, together with the platform description and several RHWOS libraries, the *RHWOS Generator* creates the *User Task Templates* for each user task. These are skeleton-files containing interface declarations of the OS objects to which the respective user tasks are associated. User task templates are software source code or hardware description language files.

- ▷ The developer now is in charge of implementing all user tasks using the user task template files as starting points.

- ▷ All user task source files are compiled/synthesized by the RHWOS Generator. In this step, several RHWOS library functions and circuits are included to generate all run-time executable files.

After completion of these steps, all run-time files necessary to execute the system are available. The debug cycle can comprise all the above described steps, or only the task implementation step.

3.4.2 Compile-Time System User Files

The following files need to be created by the user of an RHWOS, or are provided by the supplier of a target platform, respectively.

I) Application Specification File

As a first step in the development cycle, each of the n user applications $A_{1..n}$ needs to be *decomposed* into a set of m tasks $T_{1..m}$ and s OS objects $O_{1..s}$. Each task carries out a part of the application's functionality. Depending on the internal functions of a task and the application requirements, a developer decides whether to implement a task T_x in software, or hardware, or even in both. The rational behind implementing a task targeting to a CPU as well as a FPGA is to give the task scheduler a higher degree of flexibility during run-time.

Wherever applicable, tasks access services which are offered by RHWOS objects, such as storage, I/O, communication or synchronization services.

The result of the decomposition step is a formation consisting of a set of software and hardware tasks and OS objects with well-defined connection relations between each other. A connection relation includes (i) direction, (ii) interface type, and (iii) interface parameterization. An example arrangement of a decomposed single application is shown in Figure 18.

In addition to the interconnections of each task and OS objects, the processing time, memory requirement and data-throughput specifications (if applicable) for each task also needs to be specified. This information is used by the *RHWOS Generator* to dimension RHWOS elements, such as the size of private/shared memory or FIFO buffer depths.

Since a system may run different applications concurrently, each application needs a priority assigned in order to give the scheduler the ability to take decisions at run-time.

All the information listed above need to be defined in the *Application Specification File* and is used by the RHWOS Generator.

II) RHWOS Element Library

The RHWOS element library contains a set of pre-implemented RHWOS elements ready to be used for application development (application decomposition).

The RHWOS elements are parameterizable and available in various realizations. For example, a FIFO buffer can be parameterized in its width, depth and kind of interface (synchronous or asynchronous). The RHWOS element library stores RHWOS elements to be instantiated in both hardware and software, and with different run-time qualities, e.g. optimized for execution speed or resource efficiency. In addition to the functionality, the RHWOS element library stores for each variant of an element its specific timing information.

An application developer chooses objects out of this library to conduct the application decomposition step.

III) Interconnection Library
According to the application model, tasks always connect to RHWOS objects. They never connect directly to other tasks. Therefore, an important service of an RHWOS is to provide a flexible and scalable inter-task communication infrastructure.

The interconnection library provides various elements which are used to compose the communication infrastructure. The requirements for this communication infrastructure strongly depend on the application, i.e. complexity of the communication structure, data through-put, timing constraints, etc. Therefore, realizations may range from simple multiplexors, to packet-switched networks on a chip, bus-architectures, up to non-blocking cross bar circuits. The choice of a particular version is a trade-off between flexibility and resource consumption.

Various researchers present solutions for OS controlled networks on a chip (NoC) which can be used to establish an inter-task communication infrastructure [MVVL02, MMB+03, NMV90, ABF+04, HUBB04].

IV) Target Platform Description
Detailed information about the target platform is required in order to derive the executable files of the RHWOS. The target platform description contains information about

- CPU and FPGA type (footprint, speed-grade, etc.)
- the maximal clock frequency of the CPU
- external components C_i and their connection to the CPU, and FPGA, respectively, including details about address decoding of memory devices (memory map)
- any restrictions which may apply concerning signal delays, slew-rates, etc.

This information is used by the RHWOS Generator to create the *User Constraint File (UCF)* which is needed for the generation of the RHWOS-frame bitstream.

V) Device Driver Library
The device driver library is a repository of files, each representing pre-compiled code or circuit to implement a device driver.

Device drivers may be present in the device driver library in different variations per device, each of them optimally implementing a certain design objective, such as speed or required space.

The hardware structure of the target platform dictates whether a device driver is implemented in software or hardware. There might be no reason that justifies a software driver for an external device physically connected to the FPGA.

The device driver library is specific to the target platform. In the RTOS en-

vironment, platform manufacturers provide complete device driver libraries tailored for a specific platform, so-called *Board Support Packages*.

3.4.3 Compile-Time System Modules and Run-Time Files

The compile-time system modules are activated only after all user files defined in the previous section have been successfully created. The CTS modules are in charge of translating the user files into run-time files which are used later on to operate the entire system.

I) RHWOS Generator

The *RHWOS Generator* combines the information of all the above mentioned files (cf. Figure 20) and generates a number of intermediate, and output files. Internally, the RHWOS generator splits into four sub-modules:

- ***User Task Template Generator***
  Based on the information in the application specification file, the RHWOS element library, and interconnection library, the user task generator produces for each user task defined in the application specification file a user task template. Task template files are ASCII-files containing skeletons of user tasks written in a programming language (e.g. C/C++) in case of STs, or a hardware description language (e.g. VHDL) for HTs.

  User task templates are used by the application developers as a starting point to implement the user tasks. Initially, they contain no functionality, but include the interface declaration of the RHWOS objects to which the task is connected. Application developers are supposed to create the desired functionality and to invoke RHWOS services using the given interfaces.

  Merino et al. [MLJ98] first proposed using task templates as pre-defined wrappers to ease hardware task development.

- ***RHWOS Run-Time System Composer***
  This sub-module generates the RHWOS frame. An RHWOS frame constitutes all application independent files and circuits, such as software code to be executed by the CPU and hardware circuits running on the FPGA. It includes device drivers, RHWOS elements (software and hardware) and communication infrastructure. The information about the number and kind of RHWOS elements is derived from the application specification file.

  The output files generated include (i) source code files of the software part, written in C/C++, (ii) files written in a hardware description language, e.g. VHDL, which represent the RHWOS circuits on the FPGA, and (iii) the respective user constraint files.

- ***Run-Time Conditioner***
  The *Run-Time Conditioner* analyzes the timing information of all elements (user tasks, RHWOS objects, device drivers, etc.) and stores it in a run-time

optimal form as run-time specification file.

- ***Compilation/Synthesis, Map/Place/Route***
  These modules are activated in order to translate the source files, e.g. user tasks and RHWOS frame, into executable code blocks, and bitstream files, respectively. Standard CAE design tools like [XIF, XMD, EDK] can be used.

II) Run-Time Files

The following output files are produced by the RHWOS Generator and define the necessary pre-requisites for running the system.

- ***Run-Time Specification***
  The timing information and constraints of the user applications, tasks and OS objects are available in this file. It is used by run-time modules, such as the Run-Time Manager and Task Scheduler to properly operate the system.
- ***OS Frame***
  The *OS Frame* consists of two files. One file constitutes the code which is to be executed by the CPU. All RHWOS objects running in software are included in this code. The other file is a bitstream file to be configured into the FPGA. It represents a full configuration bitstream which is initially configured into the FPGA and establishes all static run-time objects in the FPGA such as RHWOS objects and communication infrastructure.
- ***Raw Task Bitstream***
  For each hardware task, a *Raw Task Bitstream* file is generated. This is a partial bitstream file containing the routed hardware task. Raw task bitstreams are stored in the *Raw Task Repository* of the run-time system.
- ***Task Description File***
  The contents of the *Task Description File* complements the information in the run-time specification. It contains specifications about the physical implementation and the task's run-time behaviour, i.e. all task properties that were not yet available during the application decomposition step. Examples include task execution times, task context information (later on used to perform functions such as task preemption and run-time parameterization), or maximal clock frequencies.

The files listed above are made available to the run-time system, which is described in the next section.

3.5 RHWOS Run-Time System

All modules and mechanisms which are necessary during application execution time constitute the *Run-Time System* of an RHWOS.

In contrast to an RTOS in which all OS modules are running exclusively in software (RTOS micro-kernel), an RHWOS may run modules in both software and hardware. According to the target architecture (as defined in Section 3.2.1) the processing elements of the RTS architecture are a CPU and an FPGA. The mapping of RHWOS modules and functions to either the CPU or the FPGA is not given a priori. This partitioning depends on the particular internal functions of the modules and needs to be analyzed for each system separately. Figure 21 displays the internal structure of the RTS we propose in an exemplary partitioning.

Unlike our partitioning, Kuacharoen et al. [KSM03] propose the implementation of configurable task scheduler in hardware. Brebner et al. [BD01] present a method to integrate a task placer as an FPGA circuit.

Basically, the reconfigurable area of the FPGA is divided into a *static* and *dynamic* region, whereas the static region accommodates RHWOS modules implemented in hardware, and the dynamic region is reserved for dynamically loaded hardware tasks.

In the subsequent sections, we first describe the mechanism of the run-time system on a high level of abstraction; then, we focus on each OS module and explain its specific internal functions.

3.5.1 RTS Mechanism (Overview)

As a precondition for operation, the RTS assumes that all files and information (as defined in Section 3.4) are present in the system.

An RTS can be in one of two run-time states.

State I: System Boot Phase (System Initialization)
After start-up, the system enters into the *System Boot Phase* and sequentially passes the following steps:

- ***CPU Initialization***
  The CPU, external devices and OS modules are initialized. Software tasks are loaded into the memory device(s) attached to the CPU. Then, control is passed to the *Run-Time Manager*.
- ***FPGA Initialization***
  The RTM uploads the full configuration bitstream representing the OS frame(s) to the FPGA, starts the *Clock Manager* to drive the clock net required by the OS frame, and resets the circuitry by the *OS Bridge*. The reset activates the

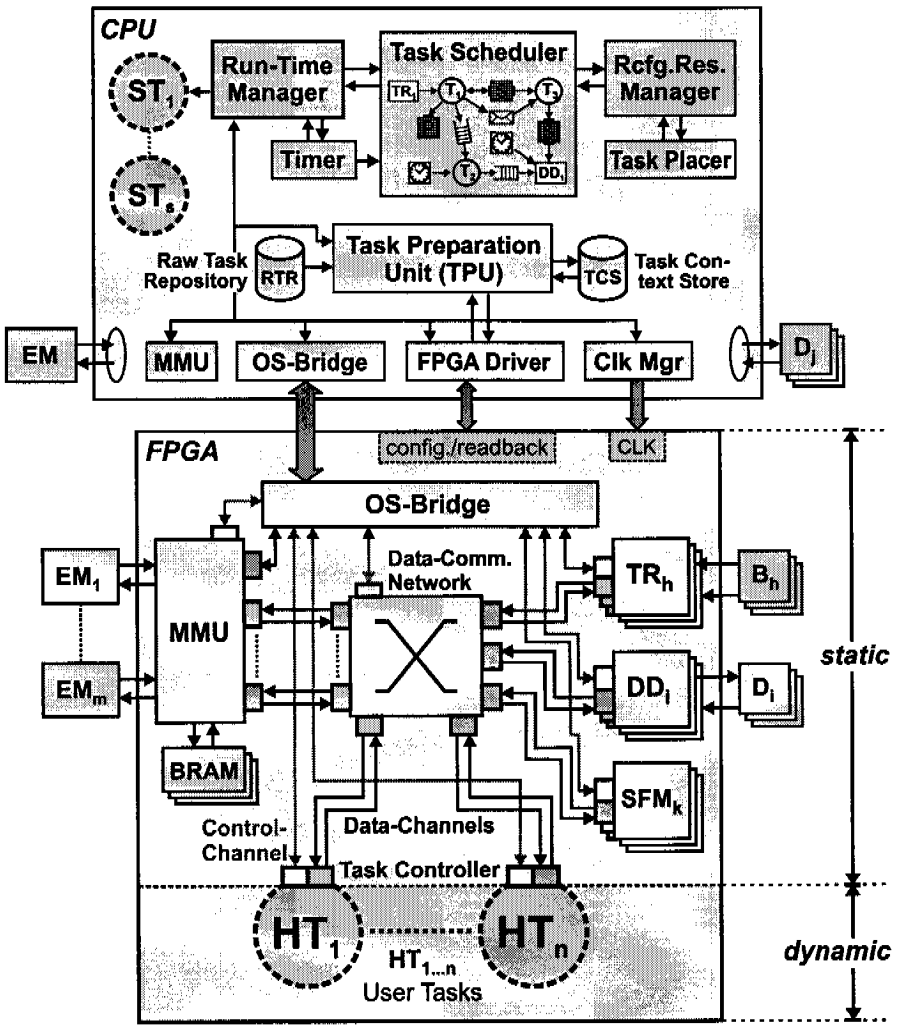


Fig. 21: Modules of the RHWOS Run-Time System with an exemplary hardware/software partitioning (CPU/FPGA) of the RHWOS modules.

initialization circuits of each OS module running in the OS frame. As soon as each OS module has completed its initialization, the start-up/initialization phase ends.

Upon completion of this phase, the RTS is then ready to execute user applications consisting of tasks, either implemented in software or in hardware, respectively. Software task (STs) are executed by the CPU, whereas hardware tasks (HTs) are executed by the FPGA.

State II: Application Execution

Application execution in an RHWOS environment functions just like in an RTOS; it is basically *event driven* [Ver01, VxW, TDB]. Events are generated from different sources and sent to a central unit, which actively manages the whole system activity, based on the received events.

The *Run-Time Manager* constitutes this control unit. The run-time manager is connected to all other RHWOS modules. The modules located in the FPGA are connected via the *OS Bridge*. Commands and data can be sent from the run-time manager to each module. For instance, the run-time manager can start, stop, preempt, and resume tasks, or parametrize OS modules, respectively.

The following example illustrates the application execution process and the cooperation of the OS modules by means of an activation process of a ST, and a HT, respectively. (This example concentrates on the conceptual steps, and does not consider detailed problems. These aspects are discussed in subsequent sections, in which the OS module functions are explained in detail):

- ▷ An event is generated, e.g. by the *Timer* module, and sent to the RTM.
- ▷ Based on the knowledge of the application, i.e. DFG, the run-time manager determines the task associated with this event (e.g. T_x) and calls the *Scheduler* to run task T_x.
- ▷ If the scheduler does not detect either a violation of the scheduling policy, or a resource or time conflict, T_x is approved to execute.
- ▷ In the case that T_x is a ST, the task is started by performing a context switch and setting the CPU's *Program Counter* to the memory address, where task is located. When T_X is a HT, the *Reconfigurable Resource Manager* and *Placer* are called to identify a location on the FPGA, in which to load T_X.
- ▷ T_x is available in the *Raw Task Repository* as a partial bitstream file. This file is modified by the *Task Preparation Unit* prior to being uploaded to the FPGA by the *FPGA Driver*.
- ▷ After completion of the configuration process, the *Clock Manager* supplies the task with the required clock signal, and a reset signal is asserted via the OS-bridge at the *Task Controller* to ensure stable conditions at the task's interfaces.
- ▷ The *Data Communication Network* is set up in order to connect the input and output ports of T_x with the OS objects, as defined in the application's DFG.
- ▷ From this point in time, all preconditions for T_x to execute are fulfilled; the reset signal for T_x is released, and T_x starts its execution.

All steps described so far are initiated and fully controlled by the run-time manager. During whole system execution, this process of *collecting events*, *triggering actions*, and *monitoring the current system state* is done continuously by the run-time manager.

3.5.2 Run-Time System Modules

In this section, we present all modules that constitute the RTS. We describe the primary function of each module and its interactions with other elements. We describe modules or functions that are typical for an RHWOS (i.e. not existing in an RTOS), in more detail. Wherever applicable, we refer to other sections or to related work in which the particular modules or functions are further discussed.

Basically, we split the reconfigurable area of the FPGA into two parts, a *static* and a *dynamic* part.

- The static part accommodates the modules that constitute the run-time part of the RHWOS. This part remains unchanged during the whole systems operation.
- The dynamic area is devoted to accommodating the application specific functions, implemented in form of hardware tasks. Hardware tasks are dynamically allocated to this area, occupy a certain amount of the reconfigurable resource during their execution, and free it again after completion.

I) Run-Time Manager

The run-time manager acts as a control unit in the entire RHWOS run-time system. It is connected to the control port of *all* OS modules and user tasks (via OS bridge, using the *control-channels*). All information about the system status, e.g. operating states of OS objects or user tasks, or utilization of the system resources, come together in this module.

Based on this overall knowledge of the system and the applications, the run-time manager is the originator of all actions taking place in the system. However, to actually carry out most of the functions, the RTM calls other specialized sub-modules. To be able to coordinate the system activities, the run-time manager needs to manage all the system's resources (except the reconfigurable device, which is managed by the reconfigurable resource manager, as explained below).

In related work, the run-time management unit is called *Virtual Hardware Manager* [BDH+97], or *Dispatcher* [Eis02], or is included in general terms as *HOS* [MJL98], *OS4RC* [WK01a], or *OS4RS* [NMB+03][3].

II) HW/SW Task Scheduler

The task scheduler decides which software or hardware task has to be executed next, among all tasks ready to run. This decision is based on a particular scheduling policy. Our design concept supports a wide range of scheduling policies. Scheduling can be off-line or on-line. An off-line schedule is suitable for

[3]Acronyms: HOS = *H*ardware *O*perating *S*ystem, OS4RC = *O*perating *S*ystem *for* *R*econfigurable *C*omputer, OS4RS = *O*perating *S*ystem *for* *R*econfigurable *S*ystems

statically-defined applications and is reflected by a rather simple task sequence table. On-line schedulers are priority-driven and split into non-preemptive or preemptive schedulers. The task scheduler receives events that are generated by different sources during run-time, e.g. by queues, timers, device drivers, and triggers.

Several researches analyzed the problem of hardware task scheduling, e.g. Mei et al. [MSV00], Handa et al. [HV04b] or Ahmadinia et al. [ABT04, ABK+04].

Compared to a scheduler in an RTOS, the task scheduler in an RHWOS possesses an additional degree of freedom: A user task can be available in both variants, as either a software or hardware task. That is, the task scheduler may decide to start a task in hardware, when the CPU is busy, or vice versa. In a preemptive system, the task scheduler may even cause a task currently running in software to be preempted and resume execution in hardware, or vice versa. This scenario can make sense in order to significantly shorten a task's execution time, or to adapt the quality of service (QoS) of a task.

Mignolet et al. [MNC+03] have successfully experimented with this function, which they denote as *task relocation*[4].

III) Reconfigurable Resource Manager

This module keeps track of the dynamically assigned reconfigurable resources in the FPGA. The responsibility of the reconfigurable resource manager is to *efficiently* manage the free reconfigurable area on the FPGA in order to make placement of hardware tasks possible. The trade-off seen by the RRM is between *placement quality* and *resource consumption* to provide the quality.

Many researchers developed methods for efficient area management, such as Bazargan et al. [BKS00], Ahmadinia et al. [ABT04, ABBT04, ABK+04, ABF+04], or Handa et al. [HV04a].

In Section 4.4, we look in detail at the internals of the reconfigurable resource manager and present several algorithms which allow the reconfigurable resource to be efficiently managed.

IV) HW Task Placer

Each time the task scheduler decides to execute a hardware task which is currently not residing in the FPGA, the hardware task placer is invoked by the reconfigurable resource manager to determine a *feasible location* where this task can be currently placed.

A task placer can follow different placement strategies, such as *first fit*, *best fit*, etc. if more than one location is available in which the task fits in. The chosen strategy of the HW task placer, together with the method applied in the reconfigurable resource manager, strongly influences the utilization of the FPGA.

[4] We use the term *relocation* in an other context, i.e. for altering the position of a hardware task within the reconfigurable area of the FPGA (cf. Section 4.2.1).

Hardware task scheduling, reconfigurable resource management and task placement are strongly coupled functions, which is in contrast to scheduling software tasks in a single processor RTOS. In Sections 4.3 and 4.5, we investigate placement strategies and run-time placement methods to increase the utilization, and to find task placements in a short time, respectively.

V) HW Task Preparation Unit
The task preparation unit generates and analyzes partial bitstreams that represent the hardware tasks. Two more modules are directly connected to the task preparation unit:

- ***Raw Task Repository***
  This repository stores task circuits in their raw form, i.e. in a position-independent form which is generated by the task design flow. Before a raw task can be downloaded to the FPGA, it must be relocated to a specific location in the user area.
- ***Task Context Store***
  This modules holds the contexts that have previously been extracted from preempted hardware tasks.

The following services are provided by the task preparation unit:

- ***Task Relocation***
  Task relocation takes a raw hardware task from the raw task repository and a position in the FPGA and generates a partial bitstream that can be downloaded to the FPGA.

  Horta et al. [HLK02] report on a successful implementation of run-time task relocation using the *PARBIT* tool[5].
- ***Context Extraction / Insertion***
  Context extraction takes a readback partial bitstream and extracts the context of a hardware task. In contrast, context insertion takes a raw task and its previously stored context and generates a partial bitstream to be downloaded to the FPGA.

  For example, resuming a task in a preemptive scheduling scenario requires calling the context insertion service and, subsequently, the task relocation service. Fornaciari et al. [FP98], Simmler et al. [SLM00], and MacBeth et al. [ML01] analyzed the problem of preemption in FPGAs; Lerjen [LZ02] presented an running implementation of hardware task preemption.
- ***Parameter Insertion / Result Extraction***
  Another service is parameter insertion, which takes a raw task and inserts

[5]PARBIT is an acronym which stands for *Par*tial *Bit*stream *T*ransformer.

a set of parameters at predefined locations (as a preset of the storage elements in RLUs). This service allows for load-time parameterization of tasks, e.g. for filters (for setting filter parameters) or crypto codecs (encryption/decryption keys).

This service would also be a conceivable way of parameter passing and result fetching which avoids the need for dedicated inter-task communication infrastructure. However, it suffers from inefficiency.

In related work, this entity is often called *Transformation Manager* [BDH+97], or is considered as an implicit part of the *Task Placer* [WK01a], or *Task Loader* [MJL98].

VI) Time Manager (Timer)
This module offers time-based services, such as one-shot and periodic timer events. The events are targeted to the run-time manager.

VII) Clock Manager (Clk Mgr)
Hardware tasks may require different clock signals in order to properly execute. The clock manager is in charge of generating several independent clock signals.

For debugging purposes, the clock manager can provide more sophisticated clock signals, as single stepping or bursts of an defined number of clock impulses.

VIII) FPGA Driver
This driver provides device-independent configuration and readback services to the TPU. The services comprise full and partial configuration as well as full and partial readback. Physically, the driver connects to the FPGAs configuration and readback port.

IX) RHWOS Bridge
Since both CPU and FPGA accommodate part of the operating system, a communication channel between the two devices is required. The operating system modules use this channel to exchange commands and data. The OS bridge provides a device-independent command interface. Physically, the communication channel is mapped to the GPI/O port.

X) Device Driver and Triggers
The function of device Drivers (DD) is explained in Section 3.3.4.

Triggers (TR) are basically a special form of device drivers that are used for rather simple external devices, e.g. switches, that can only generate events which

are then routed to the real-time manager.

XI) Special Function Manager (SFM)
The special function manager offers services to tasks, such as multipliers, based on hardware functions implemented in the FPGA.

XII) Memory Management Unit (MMU)
The memory management unit offers memory services to the tasks, such as FIFO queues with specific access modes (blocking/non-blocking), private memory blocks, or shared memory blocks. The memory structures are implemented with the FPGAs internal memories and externally connected memory devices.

XIII) Data Communication Network
The data communication network provides connection between all run-time modules within the FPGA in order to transfer application data (data-channels). As stated in Section 3.4.2 the implementation of the data communication network strongly depends on the specific needs of applications concerning communication bandwidth, etc.

There is a lot of related work dealing with communication in reconfigurable devices:

In [MVVL02, MBV+02, NMV90], the IMEC group present a solution for inter-task communication on CSoC based on a *Network on a Chip (NoC)*, a packet switched network that can be controlled by an operating system. They basically identified the need to separate *communication* area from *computation* area as a precondition to make task creation/deletion using partial reconfiguration possible. Their communication infrastructure is arranged around fixed blocks, in which hardware tasks can be loaded. They further identified three types of communication in a OS controlled system: (i) reconfiguration data, (ii) so-called *OS Operation And Management (OAM)* data, and (iii) application data. Since the application's data require high bandwidth, but OAM require low latency, they decided to implement separate networks to meet these requirements.

Bobda et al. improved IMEC's concept in [BMK+04] by a so-called *Dynamic NoC (DyNoC)* approach which allows for communication among dynamically placed tasks of variable sizes and arbitrary locations.

Kalte et al. [KLV+02a] present on-chip reconfigurable interconnect structures interfacing to the industry de-facto bus standard *AMBA* [ARM, AMB].

A solution based on XILINX VIRTEX was implemented by Huebner et al-[HUBB04] defining several *module slots* which can communicate across a shared bus.

XIV) Hardware Task Controller
Hardware task controllers are *transfer points* provided by the RHWOS to physically connect to the hardware tasks. For each hardware task, a controller is instantiated. A hardware task controller implements both an interface to the control port P^c, and to each of the n input ports ($P^i_{1..n}$), and m output ports ($P^o_{1..m}$) of the hardware task, respectively (cf. Section 3.3.3).

3.5.3 Operational Modes

An RHWOS can be in one of two *operational modes*, whereas each mode aims at optimizing a specific characteristic of the over-all system.

- ***Debug Mode***
  During the application and task development phase, the RHWOS can be operated in the *debug mode*. This mode provides services to conveniently develop tasks, and test/debug their functions and interactions with OS elements.

  As described in Section 3.5.2 the run-time manager can cause the clock manager to supply specific clock nets with single clock impulses in order to *trace* a hardware task's function. Moreover, the run-time manager can access the contents of memory (e.g. FIFO buffers or shared memory) by the control port of a particular OS element to dump its content or to insert test patterns.
- ***Execution Mode***
  In the execution mode, the RHWOS aims at providing the highest performance, in terms of execution speed and resource economization.

  Nevertheless, the run-time manager can log information about the run-time behaviour of the entire system, the applications, or user tasks and OS objects, e.g. peak values of buffer fill-levels, resource utilization, etc. In this way, the dimensions of the RHWOS elements (e.g. FIFO buffer depths) can be verified and optimized to further improve the over-all system utilization.

Knowing which operational mode is currently selected is a priori invisible to the user tasks. The interfaces and services offered to the tasks are functionally equivalent in both modes.

However, changing the operational mode in any case requires recompilation of the RHWOS, because other versions of RHWOS library elements may be integrated in the system.

3.6 RHWOS Performance and Benchmarking Aspects

A concept for an RHWOS must include the evaluation of its performance. Such an evaluation is necessary to motivate the use of an RHWOS at all, to characterize single RHWOS functions, and to compare different RHWOS implemen-

tations. An evaluation consists of a well-defined set of metrics and benchmark applications, together with a description of the benchmarking procedure to derive quantitative data.

Generally, any operating system that manages resources is faced with two kinds of challenges in terms of performance: (i) to minimize overheads, and (ii) to prevent losses.

I) RHWOS Overheads
Overheads occur because the OS itself consumes a certain amount of the resources which it manages and because the OS takes influence in the operation of the system. Mapped to an RHWOS, this leads to the following overhead metrics:

- ***Area Overhead***
  This metric measures the amount of reconfigurable resources and memory that is additionally needed to implement the runtime modules in the FPGA and the CPU. To actually calculate the area overhead, the difference between the implementations of two systems performing identical functionality, one *with* and one *without* RHWOS support, must be determined.
- ***Runtime Overhead***
  Runtime overheads occur in an RHWOS mainly due to two reasons:

  The first reason is that the RHWOS needs to place some structural elements in the processing path of an application, e.g. hardware task controller and communication infrastructure. These elements introduce signal delays which lower the over-all performance of the system. In an RTOS, this kind of overhead appears when a task calls a kernel-service.

  The second reason is because the process of hardware task activation takes some time, i.e. time is wasted between an external event and the start of the hardware task loaded in response. This process can include several steps, such as task relocation, context insertion/extraction, task configuration, etc. In an RTOS environment, this delay is called *interrupt latency*. For software tasks, the interrupt latency ranges in a few clock cycles (micro-seconds), whereas for hardware tasks, the delay can be some milliseconds.

  Obviously, the time wasted in activating a hardware task is compensated by a much higher execution speed (efficiency) compared to a software task executing the same functionality.

The *ideal* RHWOS features no area nor runtime overheads. Practically, this situation is never achievable.

In Section 5, we present our prototypical implementation of an RHWOS and discuss measured overhead data.

II) RHWOS Resource Losses

The problem of resource losses can be (i) caused by non-optimal algorithms that manage these resources, and/or (ii) the result of constraints imposed by the environment in which the RHWOS executes.

- ***Algorithm Non-Optimality***
  The most critical point in which non-optimal RHWOS algorithms can cause resource losses is connected with the management of the reconfigurable resource. The basic problem is *fragmentation*[6] connected with task scheduling/placement. An optimal scheduling/placement algorithm could (in the best case) place all hardware tasks in a tightly packed way, such that no area is wasted. However, such an algorithm is costly in space and time. In reality, other algorithms must be used, which prevent an ideal solution but keep the cost within reasonable limits. The lower algorithm quality inevitably leads to losses.
- ***On-line Constraints***
  Embedded systems are integrated in technical contexts which often require on-line decisions in response to external events. These decisions typically have to be taken within a hard time-limit (deadline) and without any knowledge about future events. Algorithms working in such an environment belong to the class of *On-line Algorithms* [BEY98]. An on-line algorithm can produce *at most* the same quality as an off-line algorithm, but most often provides a significantly less quality.

  In connection with RHWOS in an on-line scenario, scheduling and placement algorithms cause resource losses due to the unpredictable arrival of hardware tasks.

The *average utilization* of the reconfigurable device is the most straightforward metric to quantify the loss.

In Section 4.2.4, we formally define an number of metrics for different types of fragmentation and utilization which reflect the specific characteristics of an RHWOS.

[6]For the definition of *area fragmentation*, we refer to Section 4.2.4.

4

Task and Resource Management in RHWOS: Scheduling and Placement Techniques

Overview
In this chapter, we focus on the problem of task and resource management in an RHWOS, notably on task scheduling and task placement in on-line scenarios.

First, we identify new problem areas arising exclusively in context with task and resource management in an RHWOS. Then, we refine the resource and task model that we introduced in the previous chapter and define a number of RHWOS specific metrics. This lays the foundation for systematically investigating the novel problems. In each of the following Sections 4.3 to 4.6, we present our solutions for several selected task and resource management problems in an RHWOS. Each of these sections starts with a *problem statement* and an outline of its content, namely the *contributions and results*.

4.1 New Problem Areas

Executing an RHWOS as described in Section 3.5 raises a number of novel algorithmic problems. We have identified three main groups:

I) Hardware Task Placement
Hardware task placement denotes the problem of determining a feasible location in the FPGA's reconfigurable area, at which to load and execute a newly arrived hardware task.

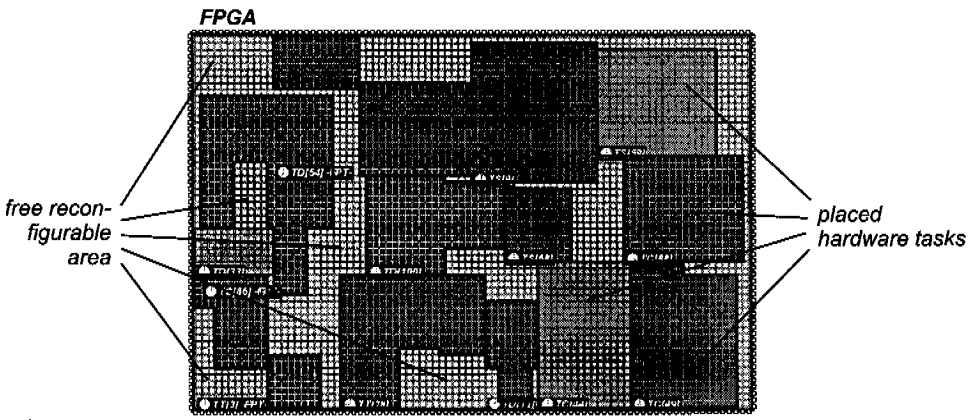


Fig. 22: Complex hardware task allocation in the reconfigurable area of an FPGA.

Several HTs may be simultaneously running on an FPGA, each with different arrival, execution and finishing times. Highly dynamic and complex allocation situations may therefore appear on the reconfigurable surface. A task placement algorithm in an RHWOS

- is in charge of determining placements for newly arrived tasks. If there are several placement possibilities, a well-considered strategy should be applied to choose the right placement, in order to make successful placements of subsequent tasks more likely.
- should find a placement in a short time, to cope with the real-time constraints (on-line scenario).

The quality of the task placement algorithm directly influences the utilization of the FPGA as a system resource. Moreover, the placement algorithm is crucial for the over-all system performance and determines whether the FPGA can be profitably employed as a dynamically allocatable system resource.

II) Reconfigurable Area Management

Reconfigurable Area Management includes the problem of (i) how the free area on the FPGA surface is represented by means of data-structures, and (ii) what algorithms are used to efficiently maintain these data-structures.

The realization of this part is critical for the performance of an RHWOS. Depending on the implemented algorithms and data-structures, the time and space complexity may significantly vary and influence the efficiency of an RHWOS.

In Sections 4.4 and 4.5 we present algorithms and data-structures tailored for being employed in an RHWOS.

III) Real-Time Hardware Task Scheduling
The term *Real-Time Hardware Task Scheduling* denotes the problem of deciding which hardware task(s) among several ready hardware tasks to execute next, in order to meet timing constraints. Basically, hardware task scheduling shows similarities to the scheduling of software tasks in an RTOS. However, there are major differences:

- In an RTOS, the executable code block, representing a software task, is present in the memory at every point in time. Activating a task just means setting the program counter (PC) to the memory location where the task resides.

  Prior to activating a HT in an RHWOS, it may be necessary to first load it into the FPGA. This step involves time consuming resource management and data transfer processes which must be considered when scheduling HTs with real-time constraints.
- In contrast to an RTOS that manages a single computational resource (CPU) that is either *fully* allocated by a software task or empty at any given point in time, an RHWOS manages a resource (FPGA) which can be occupied *partially*.

  This inevitably leads to situations in which the scheduler decides to activate a hardware task, but the task does not fit in the FPGA due to the lack of reconfigurable area at this very moment.
- Whereas only one task can execute in an RTOS at a given point in time, in an RHWOS several hardware tasks may be running concurrently (in parallel) on the FPGA.

These fundamental differences have an influence on the real-time scheduling algorithms required for an RHWOS, compared to those in an RTOS.

An important question is: to what extent are real-time scheduling algorithms known from RTOS applicable in an RHWOS, or in what way do they need to be adapted. In Section 4.6 we investigate several scheduling algorithms feasible for an RHWOS.

Table 9 provides an overview of the underlying model parameters we used in each of the following sections.

Sample Task Sets
An efficient and accurate method to evaluate the performance of the proposed task and resource management algorithms would be a trace-based simulation. Unfortunately, such traces are not available, as RHWOS systems are a new area where only first prototypes are being built.

As a consequence, we have to resort to the simulation of randomly generated,

	Device & Area Model						Task Model					
	Reconfig. Area Model				Surface Homogeneity	Reconfig. Port Charact.	Task Shape Model		Task Transformablilty	Execution Mode		Execution Deadline given
	1 Dim.	2 Dim.	Variable	Slotted			Rectangle	Polyomino		Completion	Preemptive	
4.3 Proactive Hardware Task Placement		X	X		X			X	X	X		
4.4 Partitioning-based Free Area Mgmt.		X	X		X		X			X		
4.5 Run-Time Hardware Task Placement		X	X	X	X		X			X		
4.6 Sched. & Place. in Slotted Area Model	X			X		X	X	X		X	X	X

Tab. 9: Underlying modeling parameters of the presented task and resource management algorithms in Sections 4.3 to 4.6.

i.e., synthetic, workloads.

Therefore, we have generated task sets in six different classes, varying in the task size. The classes are denoted by C_i and contain tasks of equally distributed size in the interval $[50, i]$ RLUs. The classes are C_{100}, C_{300}, C_{500}, C_{900}, C_{1600}, and C_{2700}, respectively. These classes have been chosen taking into consideration the area of typical FPGA cores (cf. Table 10) and the size of the FPGA.

Simulation Framework

In order to experimentally evaluate the algorithms, we have constructed the *Task Placement and Scheduling Simulation System (TPS* [3]*)*, an integrated time discrete simulation framework, which allows various scheduling and placement parameters for randomly generated hardware task sets to be measured.

Our simulation framework comprises the simulator module (incl. HT scheduler and placer), a hardware task generator (incl. rectangular and polyomino shapes), a module for data collection and statistical analysis including a Gantt chart viewer, and a graphical display of the allocation situation and queue loads. Several FPGA types with different RLU array sizes can be selected for simulation.

For a detailed description of our simulation framework, we refer to [TPS]. The Figures 22, 28, 29, 32, 58, and 63, respectively, show GUI screen-shots of our simulation framework.

4.2 Model Refinement and Metric Definitions

We refine the hardware task and FPGA models which we have introduced in Section 3.3. The objective of the refinements is to lay the foundations to experimentally investigate particular task and resource management problems arising exclusively in an RHWOS.

We provide several variants of models with different abstractions. Wherever applicable, we indicate and evaluate the discrepancies between the models and their feasibility using currently available FPGA technology[1] and we consider related work.

In order to particularize the run-time system model (cf. Section 3.5), we focus on the essential modules dealing with task and resource management and subdivide them into more specific functions.

Furthermore, we define a set of terms and notations which we are using throughout this chapter.

4.2.1 Hardware Task Model Refinement

The modeling of HTs as presented in Section 3.3.3 remains on a conceptual level, disregarding implementation and technology related details. In order to approach the reality, we further extend the model by adding the following structural and timing characteristics:

- ***Shape***
  Whenever a HT is actually being *implemented*, a certain physical shape results. The shape is determined by all reconfigurable resources, i.e., RLUs and routing elements, involved in the task's circuitry. Current design tools allow the outline to be influenced to a certain extent by defining *area constraints* [XIF, XMD] prior to the final implementation step.

  However, area-constraining a design significantly affects its timing, i.e., it may lower the upper bound of the clock range in which a task can be run. Kalte et al. [KKKR04] analyzed this interrelationship and measured a multiplication of the *worst path delay* of up to 10, when *compacting* a design to $\frac{1}{6}$ of its original width.

  The resulting task's shape can be modeled on two different levels of abstraction.

  i) Bounding Rectangle
  The shape can be modeled as the *smallest possible rectangle* including all RLUs and routing elements occupied by the task, whereas this rectangle must have the same orientation as the RLU array. While this version is simple in terms of data structures representing the task, it leads to *internal fragmenta-*

[1] Based on the features of Xilinx Virtex-II FPGAs, according to the state as of 2004.

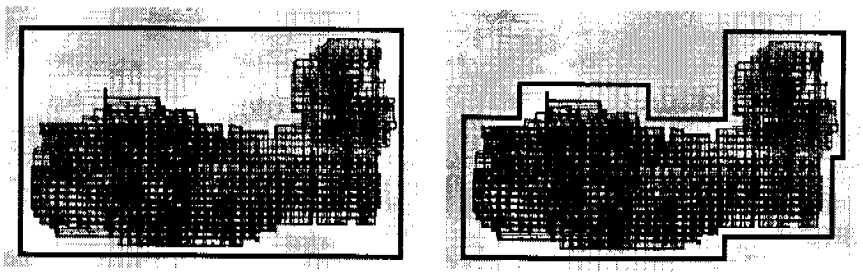


Fig. 23: Variants of task shape models: (a) Bounding rectangle or (b) Polyomino. The RLUs and routing elements involved in the HTs circuitry are marked black, whereas all other resources not part of the HT are grey.

tion. (We refer to Section 4.3 for a detailed discussion of this metric.) The internal fragmentation can be substantial for large tasks that significantly differ from rectangular shapes.

Nevertheless, this model is widely used in research work dealing with task placement problems, e.g. in [BKS00, CCKH01, FKT01, WSP03, HV04c, ABF+04].

ii) Polyomino

A more sophisticated model of the task shape is the polyomino. A polyomino is a connected subset of the square lattice (RLU) tiling of a plane (FPGA surface). Generally, this model can better approach the real shape of the task and, thus, keeps the internal fragmentation lower than the bounding rectangle. On the other hand, this model (i) requires more complex data structures to represent it, and (ii) induces more difficulties on a algorithmic level, i.e., regarding placement and resource management. Due to this fact, this model was less frequently considered in related work [EGJ99, CCKH01].

Figure 23 depicts both variants of HT shape models, (a) the bounding box, and (b) the polyomino.

- ***Relocatability***

  The relocatability of a HT describes the degree of freedom in terms of its placement in the reconfigurable area[2].

  Generally, a HT can be relocatable or not. A non relocatable task must be placed exactly on its predefined position, e.g., to access special function blocks which are in-homogeneously distributed on the FPGA. Note that current FPGA design tools produce exactly this kind of task.

[2]Nollet and Mignolet et al. [NCV+03, MNC+03] use the term *Task Relocation* in a different context, namely to move a task that was previously executing in software to hardware, and vice versa. Other researchers, such as Compton et al. [CCKH01], Wigley et al.[WK02a], Horta et al. [HLK02], etc. are compliant with our naming.

Relocatable HTs allow for translation, i.e., they can be placed at arbitrary positions with different row- and column offsets. Translation at the granularity of CLBs assumes homogeneous FPGAs. Current FPGA architectures, especially their routing resources, are not totally homogeneous.

Most of related work investigating placement problems, assumes that tasks are relocatable. [DE98, DEM+00, BKS00, CCKH01, BD01, FKT01, WSP03, ABF+04].

In practice, Horta et al. [HLK02] present the *PARBIT* tool, which allows for task translation by modifying bitstreams. This is a promising approach especially for run-time translating tasks, since (i) there is no need for time-consuming place and route processes, and (ii) the timing inside the task remains unchanged.

- ***Transformability***
  The characteristics of task transformability captures a task-internal structural aspect. Transformability can be modeled on two levels:

  i) En-bloc Transformability
  En-bloc transformable tasks can undergo geometric transformations, e.g. rotations by an integer multiple of 90 degrees, and horizontal or vertical flips, respectively, before their instantiation. Burns [BDH+97] and Compton et al. [CCKH01, CLC+02] define a number of such operations.

  En-bloc transformations leave all task internal structures unaffected, notably the configuration of the RLUs and the routing in between them. The practical feasibility of these operations strongly depends on the FPGA-internal structures.

  ii) Subtask Transformability
  HTs often split up into several subtasks. Subtasks express locality of operation. The number of wires connecting subtasks is usually much smaller than the number of wires running exclusively inside the subtasks.

  We argue that such internal task structures are the natural result of a core-oriented design style. Figure 24 demonstrates this in the example of a communication protocol stack that implements Ethernet-MAC, IP (Internet Protocol) and UDP layers [LZ02]. The protocol stack divides into a number of subtasks that handle the functions of the receive and send paths of the different protocol layers.

  Subtask-transformable HTs provide to the operating system (i) a list of relocatable subtasks, and (ii) a list of connections between the subtasks.

  A subtask transformation first applies changes in the position of one or more subtasks and then *re-routes* the connections between the subtasks. As a result, the complete task executes the same functionality as before, but exhibits a different shape. This feature can be beneficially employed in resource management algorithms. In Section 4.3 we will discuss the method of

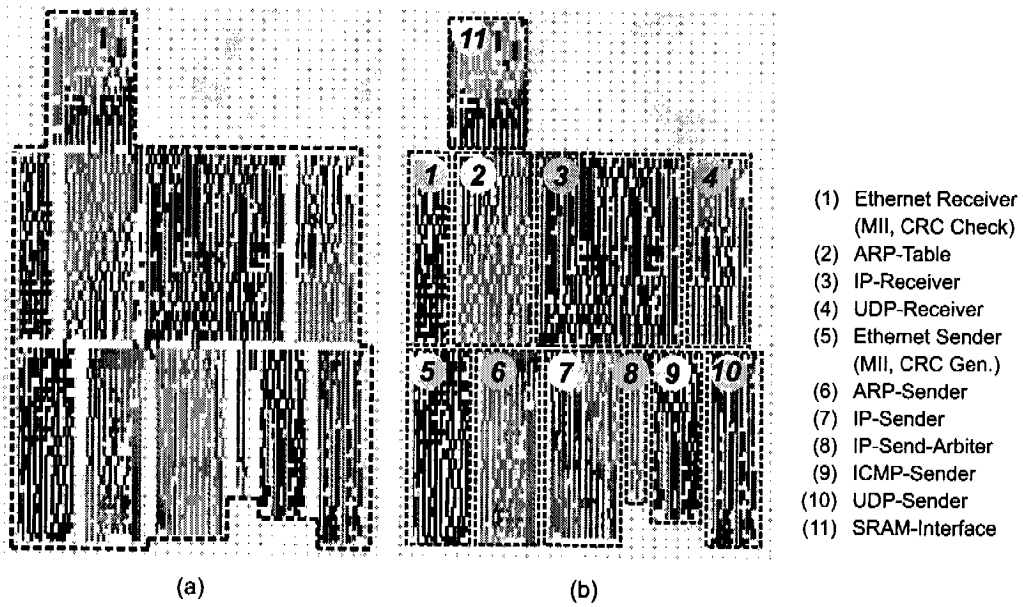


Fig. 24: Floorplan of a communication protocol stack and decomposition in subtasks $S_{1..11}$.

subtask transform and its application in more detail by presenting algorithms exploiting this technique.

- ***Spatial Critical Sections***
  A spatial critical section marks a contiguous sub-area of a HT. To guarantee proper functioning of the task, this sub-area must not be changed. An example are the *fast carry chain* resources. If a HT includes spatial critical sections, it is usually non-transformable; i.e., the HT must be placed in the same shape and orientation as implemented by the design tool.
- ***Time Domain Characteristics***
  Each task has an arrival time and an execution time (which may be known in advance or not). In real-time systems, a task may have and a deadline assigned. These parameters are used by the hardware task scheduler.
- ***Required Cycles and Clock Ranges***
  A HT requires a certain number of clock cycles, which might or might not be known in advance. The actual execution time is determined by the number of clock cycles and the clock frequency at which the task runs.

  The clock range defines a minimal and maximal clock frequencies at which the task can run. Design tools usually report an upper bound for the clock rate. A task may, however, require a specific clock rate, for example to derive a timer object that relates events to physical time. A task might further require a clock rate in a certain interval to preserve timing requirements of I/O devices or memory.
- ***Area Requirements***

Hardware Task / Device Driver	*Size* [RLU]
UART [LZ02]	60
Audio Codec Device Driver (for AK4563A) [HL04]	24
100-tap FIR Filter 12 bit data & coefficients [XCG]	250
ADPCM [DW02]	250
DCT [Amp]	600
Triple-DES processor [XCG]	800
AES processor [ER03]	950
256 point complex FFT [XCG]	850
Minimal protocol stack Ethernet-MAC, IP, UDP [LZ02]	1160
Discrete Wavelet Transform [Amp]	1800
LEON Sparc-V8 core, 32 bit mem I/F 2Kbit I-cache, 2Kbit D-cache [Gai, LEOa]	2000
MPEG2 video decoder [Amp]	3650

Tab. 10: Size of typical hardware tasks and device drivers (implemented in XILINX VIRTEX).

HTs have a certain area requirement, depending on their internal complexity. The area consumption is measured in the number of RLUs, effectively occupied by the HT (cf. Section 3.3.3). We consider coarse-grained tasks with typical sizes as listed in Table 10.

4.2.2 FPGA Area Model Refinement

The reconfigurable elements incorporated in an FPGA constitute the resources which are to be managed by the RHWOS. Physically, these elements are arranged in a two-dimensional rectangular array. The main management operation of the RHWOS to these resources is HT placement.

Generally, we assume the surface to be uniform; i.e., a task can potentially be placed at an arbitrary position. Nowadays, FPGAs most often show non-uniform surfaces. Special function blocks and routing resources are irregularly distributed over the reconfigurable area.

There are several possibilities in modeling the FPGA surface in terms of its capability to accommodate HTs. We define two major aspects which are orthogonal to each other:

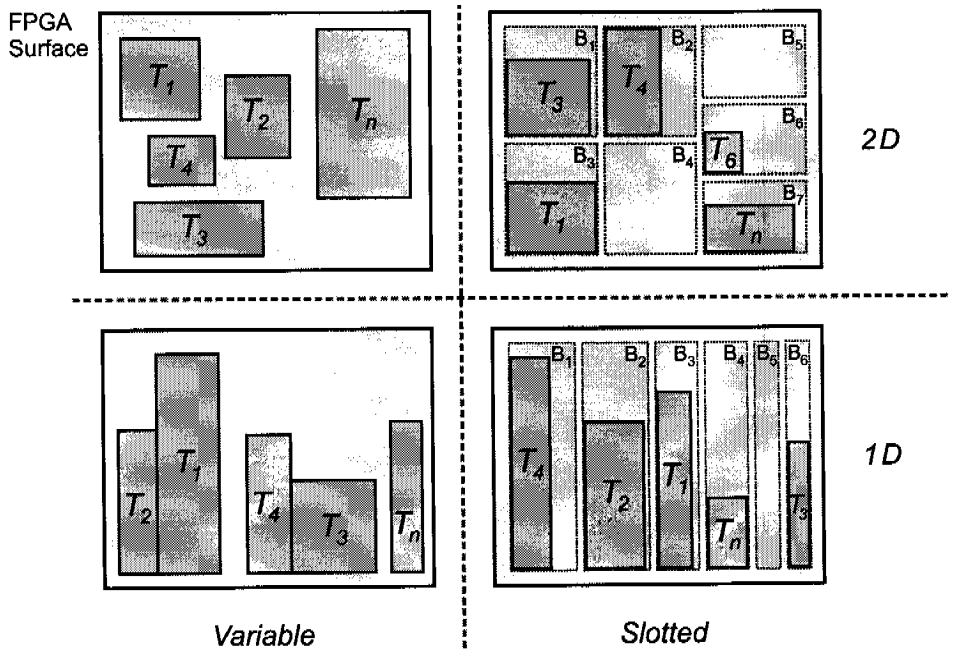


Fig. 25: Reconfigurable area model categories.

- ***1D/2D Area Model Aspect***
  The 1D/2D aspect describes the degree of freedom in determining the location of a task placement.

  In a 1D model, the location can only be chosen along one (e.g. the horizontal) dimension, that is, the vertical placement offset is always set to zero.

  The 2D model allows for varying the horizontal as well as the vertical position offset.
- ***Variable/Slotted Area Model Aspect***
  The *variable* and *slotted* area models differ in the granularity of the locations where a task can be placed.

  The variable model supports placement on a RLU level of granularity. In contrast, the slotted model partitions the surface into a number of predefined *Blocks (B)*; that is, tasks can only be placed into free blocks.

Figure 25 provides an overview of the resulting model categories based on the two aspects.

In the following, we discuss in detail each of these categories and highlight their specific characteristics and the feasibility for a practical implementation using todays FPGA technology.

I) 2D-Variable Area Model

The 2D-variable area model (see Figure 25, top-left) assumes a homogeneous FPGA surface in which HTs are freely relocatable. It offers the highest degree of freedom in task placement; that is, HTs can be placed anywhere in the two-dimensional FPGA area at the finest granularity (on a RLU level). This model may allow for a high utilization of the reconfigurable area, as tasks can be packed tightly.

However, the high flexibility of this model comes at a price when analyzing the complexity of the (i) scheduling, (ii) placement and (iii) resource allocation algorithms necessary to efficiently manage the reconfigurable area resource.

Considering a scenario in which n tasks $T_{1..n}$ arrive sequentially in time, each with a different size, shape, and a variable execution time, highly dynamic task allocation situations may result. Sophisticated algorithms are required to ensure placement efficiency, i.e., to prevent the undesirable situation that a newly arrived task T_x can not be placed, even if there are enough unused RLUs available on the FPGA. Such a situation can occur when the free RLUs are scattered over the FPGA surface (in a non-contiguous form), or in a form in which T_x does not fit.

From an implementation point of view, this model involves severe difficulties: First, the tasks will require external wires to connect to OS elements. Related work either assumes that this communication is established inside the HT via configuration and readback operations (which is feasible but presumably inefficient) or proposes to leave some space between tasks for communication channels. Second, the connection wires must be dynamically (re-)routed and the timing must be reanalyzed during run-time; this is not supported by current commercial design tools [JBi]. Moreover, current FPGA devices don't allow for (re-)configuration accesses on a RLU level.

Regardless of the implementation problems inherent to this model, it has been used by most of the authors investigating placement problems [Bre96, BKS00, DEM+00, FKT01, ABT04, HV04c].

II) 1D-Variable Area Model

The 1D-variable area model still allows for placing a HT on a RLU granularity but limits the degree of freedom to one dimension. As indicated in Figure 25 bottom-left, all tasks are placed with a vertical position offset of zero.

As a consequence, the reconfigurable area above a HT that is smaller than the FPGA's height can not be used. This means implicitly that tasks can not be packed as tightly as in the 2D model, which inevitably leads to a lower device utilization. The model suffers from both internal and external fragmentation.

On the other hand, this model simplifies scheduling, placement and resource management. In particular, the placement and resource management problem is identical to the *dynamic storage allocation problem* which was exhaustively investigated by many authors. The main reference is Wilson et al. [WJNB95].

Due to the fact that vertical stripes do not contain more than one task, this model is practical for implementation using modern FPGAs. Xilinx Virtex devices support column-oriented (re-)configuration. Brebner and Diessel [BD01] first described task management functions based on this model.

III) 2D-Slotted Area Model

Slotted area models follow the approach of partitioning the reconfigurable surface into several allocation sites, so-called *Blocks (B)*, with given locations and sizes. The 2D-slotted model provides n two-dimensionally distributed blocks $B_{1..n}$. The layout is arbitrary but fixed during system operation. Blocks are place-holders for HTs and each block can accommodate no more than one HT at a time.

Figure 25 top-right shows a possible partitioning of the reconfigurable area into $n = 7$ blocks $B_{1..7}$.

Merino et al. [MLJ98, MJL98] first presented such a concept. Recently, Marescaux et al. [MBV+02] and Bobda et al. [BMK+04] refined this concept and presented practical implementations.

Partitioning the area simplifies scheduling and placement and makes a practical implementation on current FPGA technology more realistic. As the blocks have fixed positions, the remaining area can be made an operating system resource. Communication channels and I/O are provided exclusively by the operating system. With fixed interfaces between the tasks and the operating system, there is no need for online (re-)routing and timing analysis [DPP02].

The disadvantage of a partitioned area model is the *block internal fragmentation*, i.e., the area wasted when a task is smaller than a block.

Xilinx Virtex FPGAs are partially reconfigurable only in vertical chip-spanning columns. Hence, the configuration of a task potentially interferes with other tasks allocated in blocks within the same reconfiguration columns.

IV) 1D-Slotted Area Model

Finally, Figure 25 bottom-right depicts the 1D-slotted area model. The n blocks $B_{1..n}$ all exhibit the same height as the FPGA and are arranged in one dimension.

This variant combines the simplified scheduling and placement of the 2D-slotted model with the implementation advantages of the 1D-variable model. Again, the disadvantage lies in the high block internal fragmentation.

4.2.3 System Model Refinement

Figure 26 conceptualizes a part of the *Run-Time System (RTS)*, as introduced in Section 3.5, focusing on the modules performing hardware task and reconfigurable resource management. We refine the system model by specifying the following properties:

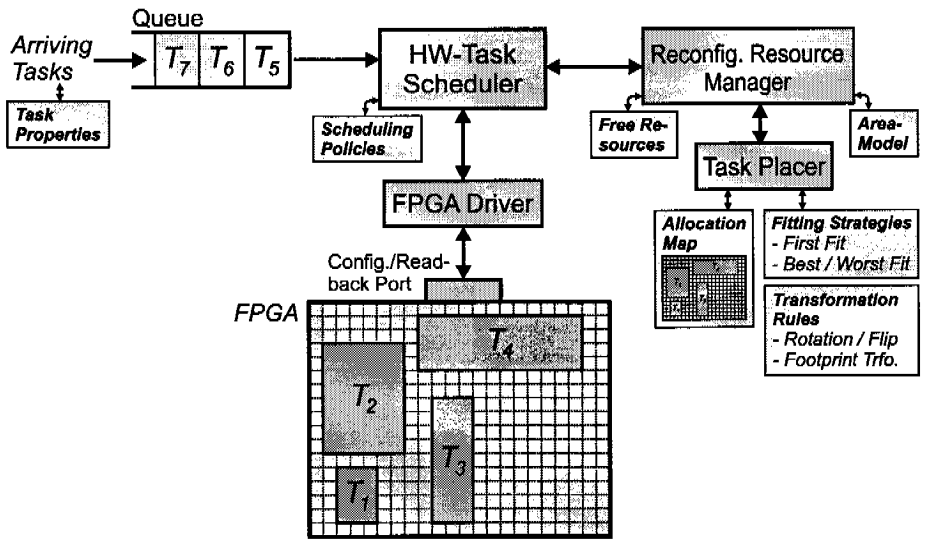


Fig. 26: Task and resource management part of the Run-Time System (RTS)

- ***Constant Reconfigurable Area Model***
  The system runs one of the previously defined 1D/2D, or variable/slotted area models, respectively. We assume that the model can not be changed during system operation. The modules present in the system are only capable of handling one area model.
- ***Restriction to Hardware Task***
  In contrast to the run-time system presented in Section 3.5, we consider applications consisting exclusively of hardware tasks. Additionally, we concentrate on task sets that hold no precedence constraints between the tasks; that is, all tasks are independent from each other.
- ***On-line Scheduling Scenario***
  We consider an on-line scenario in the sense that the hardware task scheduler does not know in advance at what time tasks arrive and what their properties will be. The scheduler applies a certain policy which remains constant during system operation.
- ***Specific System Modules and I/O***
  The configuration/readback port controlled by the *FPGA Driver* can be used in either unidirectional or bidirectional (half-duplex) mode with a limited bandwidth. This allows for non-preemptive as well as preemptive scheduling scenarios.

  The *Reconfigurable Resource Manager* keeps track of all reconfigurable elements allocated in the FPGA, utilizing one of the previously defined area models. Task placement algorithms are performed by the *Task Placer* unit, making use of specific *fitting strategies* and *task transformation rules*.

All I/O related issues such as I/O routing and timing constraints are not considered in this model.

All modules indicated in Figure 26, except the FPGA Driver, are assumed to be implemented in software and to run on a single CPU. (Obviously, the CPU needs to provide a physical interface to the FPGA's configuration/readback port.)

4.2.4 Definitions and Metrics

We introduce a number of definitions and metrics which we will use throughout the subsequent sections to formally describe our models and algorithms.

Figure 27 depicts the floor-plan of an exemplary hardware task implemented in an FPGA. In this illustration, all elements which are not relevant for the following considerations (such as I/O block, special function blocks, block-RAMs, etc.) are omitted. The assumed internal structure and configuration characteristics rely on that of the XILINX VIRTEX (-II) FPGA device family [XVI, XV2a], which we have ascertained by means of design-tools [XIF, XFE, JBi] and experimentally verified. To visualize the source and meaning of some definitions by means of a concrete example, we will always refer to this figure.

I) Hardware Task Related Definitions

For any hardware task T_i we define the following notations and interrelations:

a_i : *Arrival Time* of task T_i

s_i : *Starting Time* of task T_i

d_i : *Deadline* of task T_i

f_i : *Finishing Time* of task T_i

e_i : *Execution Time* of task T_i (without being preempted) as

$$e_i = f_i - s_i \tag{4.1}$$

w_i : *Waiting Time* of task T_i as

$$w_i = s_i - a_i \tag{4.2}$$

r_i : *Response Time* of task T_i as

$$r_i = f_i - a_i \tag{4.3}$$

W_i,H_i : *Width and Height* of task T_i, measured in number of RLUs. For both task shape models, *bounding rectangle* and *polyomino*, W_i and H_i denote the horizontal and vertical dimension of the task's bounding box. Task T_i in Figure 27 holds $W_i = 10$, and $H_i = 6$, respectively.

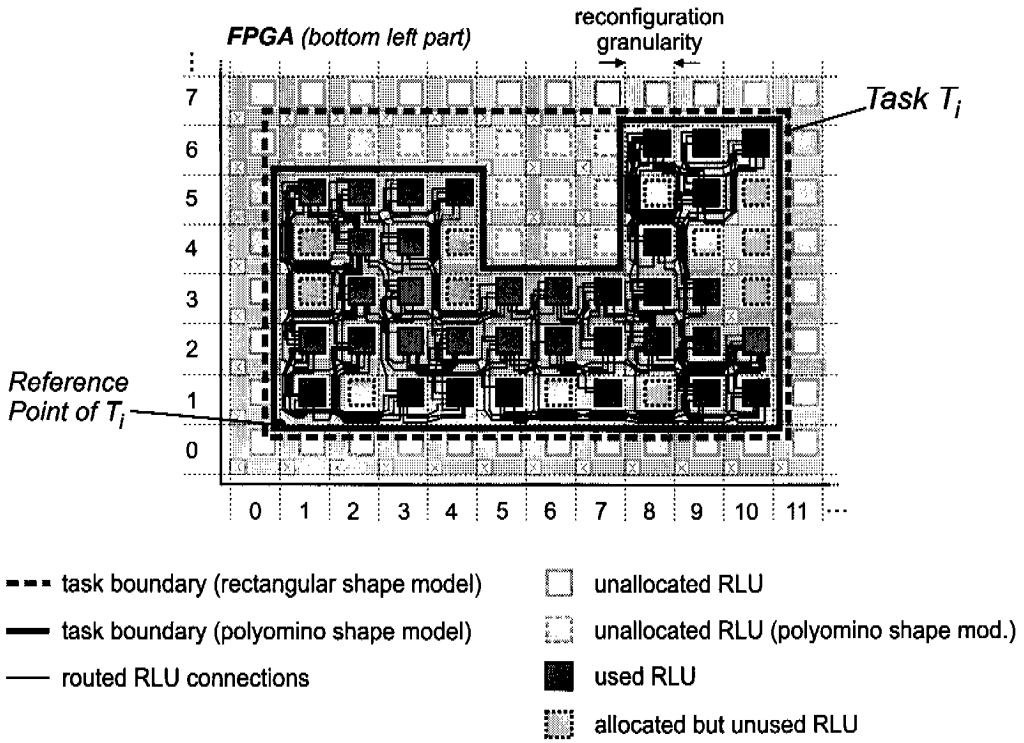


Fig. 27: Exemplary implementation (floor-plan) of a sample hardware task T_i with $H_i = 6$, $W_i = 10$, A_i^m(rect.) $= 60$, A_i^m(poly.) $= 47$, $A_i^p = 45$, and $E_i = 35$, respectively. $\mathcal{M}(T_i, R_{1,3}) = 1$ since the RLU at location $(1,3)$ is member of T_i's shape model (rectangular and polyomino), whereas $\mathcal{M}(T_i, R_{6,5}) = 0$ for the polyomino model, but $= 1$ for the rectangular model. $\mathcal{U}_R(R_{3,3}) = 1$ since the RLU at location $(3,3)$ is involved in the circuit of T_i, whereas $\mathcal{U}_R(R_{1,3}) = 0$ because $R_{1,3}$ is not involved in the circuit.

A_i^p : *Physical Area Requirement* of task T_i including routing resources, measured in integer multiples of the area requirement of one RLU.

A_i^p of task T_i in Figure 27 amounts to 45 RLUs, which results from adding up the number of *used*, and the number of *allocated but unused* RLUs. The term *allocated but unused* means that the RLU-logic itself is not part of the task's circuit, but there are routing elements involved in T_i within the reconfiguration frame of this RLU. The size of the reconfiguration frame determines the *granularity* in which the device can be reconfigured. Therefore, the membership of a single configuration bit (regardless of routing or logic bit) to a task's circuit assigns the whole frame to this task.

A_i^m : *Modeled Area Requirement* of task T_i, measured in integer multiples of the area requirement of one RLU. This metric specifies the area requirement of T_i with respect to a chosen task shape model.

Using the *bounding rectangle* shape model, the area requirement in any case yields to

$$A_i^m = W_i \cdot H_i \tag{4.4}$$

whereas in the *polyomino* shape model, A_i^m denotes the number of RLUs actually comprised by the task's outline.

Note that A_i^m may have different values for the same physical task implementation, depending on the chosen task shape model. Considering T_i in Figure 27, A_i^m using the bounding rectangle shape model amounts to 60 RLUs, whereas A_i^m for the polyomino shape results in only 47 RLUs.

The term

$$A_i^m - A_i^p \tag{4.5}$$

describes the loss of reconfigurable resources due to the task shape modeling. This value is always ≥ 0 and specifies the number of RLUs that are not used at all (neither RLU-logic, nor routing), but are contained in the modeled task. Examples are $R_{10,3}$ and $R_{10,4}$ in T_i (Figure 27).

E_i : *Number of Effectively Involved RLUs* in the logic of task T_i. Obviously, $E_i \leq A_i^p \leq A_i^m$ holds for any task and any task shape model. For a task T_i, the term

$$1 - \frac{A_i^p - E_i}{A_i^p} \tag{4.6}$$

expresses a kind of *area efficiency* of a particular task implementation. We argue that it's not a primary duty of an RHWOS to maximize this value, because it strongly depends on (i) the quality of the design tool which physically implements the task, and (ii) on the area model, which is dictated by the FPGA's (re-)configuration capability and granularity.

Applying Equation 4.6 to T_i in Figure 27 yields to an area efficiency of 71%. Real implementations can show significantly higher values of $> 90\%$ [LZ02, Rup03, Nob04, Ste04] using [XIF].

II) Task Set Related Definitions

For a task set consisting of n unrelated tasks $T_{1..n}$ we define

t_{tot} : *Total Execution Time* of the task set as

$$t_{tot} = \max_{i=1...n}(f_i) - \min_{i=1...n}(a_i) \tag{4.7}$$

$\overline{w}$: *Average Waiting Time* as

$$\overline{w} = \frac{1}{n} \sum_{i=1}^{n} w_i \tag{4.8}$$

$\overline{r}$: *Average Response Time* as

$$\overline{r} = \frac{1}{n} \sum_{i=1}^{n} r_i \tag{4.9}$$

III) Reconfigurable Device Related Definitions

For a given type of an FPGA device D, we define

W_D, H_D : *Width and Height of Device* D, measured in number of RLUs. The device contains a two-dimensional array of RLUs. We consider the array as a Cartesian (x/y)-coordinate system with its origin $(0/0)$ in the bottom-left corner of the device.

A_D : *Area of Device* D, measured in number of RLUs. Since we only consider rectangular device shapes, $A_D = W_D \cdot H_D$ follows.

$R_{x,y}$: *RLU at device location* (x, y), whereas $x \in [0, W_D - 1]$ and $y \in [0, H_D - 1]$

IV) Function Definitions

$\mathcal{U}_R(R_{x,y})$: *Usage of the RLU at location* (x, y)

$$\mathcal{U}_R(R_{x,y}) = \begin{cases} 1, \text{ if } R_{x,y} \text{ is involved in a circuit;} \\ 0, \text{ otherwise.} \end{cases} \tag{4.10}$$

Considering T_i of Figure 27, $\mathcal{U}_R(R_{1,1}) = 1$, whereas $\mathcal{U}_R(R_{2,1}) = 0$.

$\mathcal{M}(T_i, R_{x,y})$: is $R_{x,y}$ *Member of task* T_i

$$\mathcal{M}(T_i, R_{x,y}) = \begin{cases} 1, \text{ if } R_{x,y} \text{ is part of the modeled task } T_i; \\ 0, \text{ otherwise.} \end{cases} \tag{4.11}$$

Note that $\mathcal{M}(T_i, R_{x,y})$ may return different values for the same physical implementation of a task if different task shape models are applied, i.e., $\mathcal{M}(T_i, R_{6,5}) = 0$ (see Figure 27) using the polyomino shape model, whereas $\mathcal{M}(T_i, R_{6,5}) = 1$ for the bounding rectangle model.

Both functions $\mathcal{U}_R(R_{x,y})$ and $\mathcal{M}(T_i, R_{x,y})$ can be applied to any arbitrary location within the reconfigurable area of the FPGA.

The relation between E_i, A_i^m, $\mathcal{U}_R(R_{x,y})$, and $\mathcal{M}(T_i, R_{x,y})$ is defined for any task T_i as

$$A_i^m = \sum_{x=0}^{W_D-1} \sum_{y=0}^{H_D-1} \mathcal{M}(T_i, R_{x,y}) \tag{4.12}$$

and

$$E_i = \sum_{x=0}^{W_D-1} \sum_{y=0}^{H_D-1} \mathcal{U}_R(R_{x,y}) \cdot \mathcal{M}(T_i, R_{x,y}) \tag{4.13}$$

respectively. Note that both definitions hold for rectangular *and* polyomino shaped task models.

v) Device Allocation and Task Placement Constraint

At every given point in time t , either *none* or $n \geq 1$ tasks $T_{1..n}$ are placed on a device D. We define

$\Lambda_D(t)$: *Allocation of Device D* at point in time t.

All placed tasks $T_{1..n}$ of an allocation Λ_D must fulfill the *Task Placement Constraint* which can be formulated as:

$$\sum_{i=1}^{n} \mathcal{M}(T_i, R_{x,y}) \leq 1, \quad \forall x \in [0, W_D - 1], \forall y \in [0, H_D - 1] \tag{4.14}$$

The placement constraint is an invariant which assures that none of the RLUs is used by more than one task at the same time. In other terms, the tasks $T_{1..n}$ must not overlap in space.

A newly arrived task T_i can only be accommodated in an existing allocation Λ_D, if T_i does not overlap with any of the already placed tasks $T_{1..n}$. We define

$\mathcal{P}(T_i, x_p, y_p)$: *Task Placement Operation.* This operation defines the set of RLUs that are occupied by a relocatable task T_i after it is placed to a specific device location $P = (x_p, y_p)$. Task T_i's reference point is the bottom left corner of its bounding rectangle (see Figure 27). Obviously, $x_p \in [0, W_D - W_i]$ and $y_p \in [0, H_D - H_i]$ holds, because the complete task T_i must be placed within the reconfigurable area of device D. A placed instance of task T_i to location $P = (x_p, y_p)$ can be written as:

$$T_i^P = \mathcal{P}(T_i, x_p, y_p) \tag{4.15}$$

In consideration of Equation 4.14, a valid placement for task T_i in a given allocation only exists, if $\exists x_p \in [0, W_D - W_i]$ and $\exists y_p \in [0, H_D - H_i]$:

$$\sum_{x=x_p}^{x_p+W_i} \sum_{y=y_p}^{y_p+H_i} \mathcal{U}_R(R_{x,y}) \cdot \mathcal{M}(T_i^P, R_{x,y}) = 0 \tag{4.16}$$

VI) Fragmentation

Fragmentation is a major problem in the reconfigurable resource management of an RHWOS. Many researchers issued verbal descriptions of several types of fragmentation, e.g. [DE98, DW99, DEM+00, BKS00, WK01b, GASF02] etc., but only a few stated clearly quantified definitions [WK02b, HV04c, TSMN04]. We distinguish the following categories of fragmentation:

$\mathcal{F}_T(T_i)$: *Task-Internal Fragmentation* of a task T_i, measured in percentage. Task-internal fragmentation occurs as a consequence of a chosen task shape model. We define it as the ratio between the physical area requirement of a task (including RLUs and routing resources) and the area requirement of the modeled task T_i:

$$\mathcal{F}_T(T_i) = 1 - \frac{A_i^p}{A_i^m} \tag{4.17}$$

If $\mathcal{F}_T(T_i) = 0$, it means that the task shape model ideally represents the physical outline of task T_i as produced by the design tool. However, the same implementation of a task may exhibit different task internal fragmentations for each shape model applied; e.g. for T_i of Figure 27 the task-internal fragmentation $\mathcal{F}_T(T_i) = 4.2\%$ using the polyomino shape model, whereas $\mathcal{F}_T(T_i) = 25\%$ for the bounding rectangle shape model.

The basic principle of this fragmentation type is consistent with related work. The naming chosen by Wigley et al. [WK02b] is *logic block internal fragmentation*, whereas Handa et al. [HV04c] denote the same metric as *internal fragmentation*.

$\mathcal{F}_B(B, T_i)$: *Block-Internal Fragmentation* of a block B caused by task T_i, measured in percentage. The block-internal fragmentation can only be calculated in a 1D/2D-slotted area model. It expresses the waste of RLUs in a block by a specific task. If B denotes the size of a block in which task T_i is placed, we define the block-internal fragmentation as:

$$\mathcal{F}_B(B, T_i) = 1 - \frac{A_i^p}{B} \tag{4.18}$$

For a task T_i that perfectly fits into a block B, the block-internal fragmentation results to $\mathcal{F}_B = 0$.

[WK02b] and [HV04c] use the term *partition* instead of *block* but mean the same type.

$\mathcal{F}_A(\Lambda_D)$: *Area Fragmentation* of allocation Λ_D, measured in percentage. The area fragmentation describes in which form the free reconfigurable area around placed tasks is available on a device. Extremal values of area fragmentation occur in situations when the free area is available in one contiguous piece and has a rectangular shape. Then, intuitively, the area fragmentation is low. If the free area is divided into a lot of small non-contiguous portions that are irregularly distributed over the FPGA's surface, a high area fragmentation results. Wigley et al. [WK02a] identified the *checkerboard pattern* as worst case scenario, in which every second RLU is free, and assign this situation a fragmentation value of 100%.

This metric is of interest in connection with task placement. A *placement algorithm* aims at keeping the area fragmentation as low as possible. Low fragmentation promises successful placement of tasks. This potentially leads to a high task packing density, and thus, a high device utilization.

Only a few researchers stated clear formulas to calculate the area fragmentation [WK02b, TSMN04, HV04c]. In Section 4.3, we present our own area fragmentation metric and review related work.

In [HV04c], Handa et al. identified *virtual fragmentation* as one more category, describing the situation, in which a *placement algorithm* is not able to locate contiguous area to place a task, even if such an area is available. We call this incident a *placement mistake* (see Section 4.4) and claim that this is characteristic of the placer module, rather than an area property.

VII) Area Utilization Metrics

The area utilization is the most important metric in order to analyze the efficiency of an RHWOS. To get solid values of the metric, the underlying architectural details of an FPGA must be analyzed.

In static (RHWOS-less) FPGA designs, the device utilization is defined as the ratio between the number of used RLUs to the totally available number of RLUs in the device D. This can be expressed as:

$$\frac{1}{A_D} \sum_{x=0}^{W_D-1} \sum_{y=0}^{H_D-1} \mathcal{U}_R(R_{x,y}) \tag{4.19}$$

Standard design tools, such as [XIF], report this kind of static area utilization;

which is meaningful in this context.

In an RHWOS environment, we define

$\mathcal{U}_A^s(D)$: *Static Area Utilization* of a device D, measured in percentage, as

$$\mathcal{U}_A^s(D) = \frac{1}{A_D} \sum_{i=1}^{n} E_i \tag{4.20}$$

assuming that n tasks $T_{1..n}$ are running at a given point in time.

The determination of the device utilization in a dynamic RHWOS environment needs some more consideration. There are two decisive points to take into account:

- ***RLU /Routing Configuration Nexus***
  The (re-)configuration architecture of current FPGAs does not allow for separately modifying configuration bits affecting RLUs, and routing elements, respectively. When defining the boundary of a hardware task, all RLUs *and* routing resources belonging to the task must be included. If an RLU that is even not involved in the task's circuitry overlaps with an active routing element, this RLU must be considered as part of the task. In consequence, a task may contain unused RLUs in order to ensure proper routing. From this, it follows that for any allocation of n tasks $T_{1..n}$, the condition

  $$\sum_{i=1}^{n} \mathcal{M}(T_i, R_{x,y}) \geq \mathcal{U}_R(R_{x,y}), \quad \forall x \in [0, W_D - 1], \forall y \in [0, H_D - 1] \tag{4.21}$$

  holds. Due to this fact, the utilization of the device executing dynamic hardware tasks, must be calculated based on the area requirement, rather than based on allocated RLUs. Hence, we propose the utilization metric to be calculated as follows

  $$\mathcal{U}_A^m(D) = \frac{1}{A_D} \sum_{i=1}^{n} \sum_{x=0}^{W_D-1} \sum_{y=0}^{H_D-1} \mathcal{M}(T_i, R_{x,y}) \tag{4.22}$$

  Thus, $\mathcal{U}_A^m(D)$ represents a *momentary* utilization, only valid at given point in time, when n tasks $T_{1..n}$ are running in D. Applying Equation 4.4, in which A_i^m denotes the total area covered by task T_i, the utilization simplifies to

  $$\mathcal{U}_A^m(D) = \frac{1}{A_D} \sum_{i=1}^{n} A_i^m \tag{4.23}$$

- ***Dynamic Hardware Task Property***
  In an RHWOS environment, hardware tasks are dynamically placed in the

FPGA, and removed again after completion. Hence, the device utilization changes over time and the dynamic aspect needs to be included into the calculation of the utilization.

A first approach would be to average the utilization over the total execution time t_{tot} of a task set consisting of n tasks $T_{1..n}$ like

$$\overline{U}_A^m(D) = \frac{1}{t_{tot} \cdot A_D} \sum_{i=1}^{n} A_i^m \cdot e_i \tag{4.24}$$

As described in Section 3.5, a hardware task passes through several (non-productive) activation steps before actually executing, i.e., configuration, initialization, etc. During these phases, the reconfigurable area must already be assigned to the task, which would increase the utilization, according to Equation (4.24). But in fact, the task is not yet making use of the resource. Hence, this non-productive period of time should not be considered in calculating the utilization. The amount of time spent in non-productive states during task activation can be significant, particularly for tasks with short execution times. To correct this we further define function

$p_i(t)$: Task T_i is *productive* at point in time t as

$$p_i(t) = \begin{cases} 1, \text{ if task } T_i \text{ is productive at time } t \\ 0, \text{ otherwise.} \end{cases} \tag{4.25}$$

Applying $p_i(t)$, the non-productive task states can be excluded and a more accurate utilization metric can be defined,

$\overline{U}_A^p(D)$: *Dynamic Area Utilization* of device D, measured in percentage,

$$\overline{U}_A^p(D) = \frac{1}{t_{tot} \cdot A_D} \sum_{t=a_{min}}^{f_{max}} \sum_{i=1}^{n} A_i^m \cdot p_i(t) \tag{4.26}$$

whereas

$$a_{min} = \min_{i=1..n}(a_i) \quad \text{and} \quad f_{max} = \max_{i=1..n}(f_i)$$

The ratio between productive and non-productive time is mainly caused by (i) the (re-)configuration capabilities of the FPGA, and (ii) the performance of the RHWOS run-time modules responsible for executing task management functions, such as *Run-Time Manager*, *Task Preparation Unit*, *FPGA-Driver*, etc. (cf. Section 3.5). Since the RHWOS keeps full control over the entire FPGA at any point in time, the RHWOS is able to determine $p_i(t)$ of all tasks $T_{1..n}$.

The metric $\overline{U}_A^p(D)$ thus includes both the characteristics of the reconfigurable device D and the efficiency of the RHWOS.

In Section 3.5 we described the over-all structure of the RHWOS run-time system, which basically partitions the reconfigurable device D into a static and dynamic part. We denote these parts D_s (static), and D_d (dynamic), respectively. Due to the above discussed reasons, both parts need to be individually evaluated regarding their utilization.

Applying the dynamic area utilization $\overline{U}_A^p(D)$ exclusively to the dynamic part provides a further enhanced metric, as

$$\overline{U}_A^p(D_d) = \frac{A_D}{A_{D_d}} \cdot \overline{U}_A^p(D) \tag{4.27}$$

Consequently, $\overline{U}_A^p(D_d)$ expresses how efficiently the dynamically used reconfigurable area is exploited by the RHWOS. This metric is only defined for $A_{D_d} > 0$. Since $A_{D_d} \leq A_D$ it follows that $\overline{U}_A^p(D_d) \geq \overline{U}_A^p(D)$. This is meaningful because the statically used area is *excluded*, which increases the utilization. $\overline{U}_A^p(D_d)$ characterizes the quality of the resource management algorithms in an RHWOS.

To specify the same utilization metric for static RHWOS part D_s makes no sense evidentially. An RHWOS implementation must in any case aim at minimizing the amount of statically used reconfigurable resources A_{D_s}. Hence, we define

$\mathcal{L}_A(D)$: *Area Loss Ratio* caused by an RHWOS in device D, measured in percentage,

$$\mathcal{L}_A(D) = \frac{A_{D_s}}{A_D} \tag{4.28}$$

Our simulation framework which we use in Section 4.6 considers the task activation states and reports the utilization according to $\overline{U}_A^p$.

In Chapter 5, we present our prototypical realization of an RHWOS executing a case study application. The runtime environment of this RHWOS implementation follows the static/dynamic splitted approach. Hence, we will apply the metrics, numerically calculate and discuss them in a real environment (cf. 5.5.3). Many authors of related work [BS99, EMSS00, BKS00, BD01, WK01a, GASF02, KLV+02a, ABT04, HV04c, TSMN04, PMW04] are aware of the importance of considering the device utilization/efficiency as a metric to evaluate a run-time reconfigurable system. However, only a few state clear formal definitions or consider dynamic effects, whereas the use of static-based utilization metric, as U_A^s, is wide-spread.

Eldredge and Wirthlin et al. [EH96, WH97, Wir97] introduced the term *Functional Density*, which is a metric describing the speed of a circuit in relation its area consumption. The time wasted during (re-)configuration processes can be included in the calculation. The functional density can be seen as a kind of utilization metric, but does not implicitly respect the problem of different area models. The sense of this metric is included in $\mathcal{U}_A^p$, since the area consumption and the time aspects are considered in the same way.

Throughout this chapter, we will employ this formalism and further extend it in the following sections.

4.3 Pro-active Hardware Task Placement

Problem Statement
Hardware task placement in an on-line scenario inevitably leads to *area fragmentation*; that is, the free reconfigurable area is divided into several small portions which are irregularly distributed over the FPGA surface. A high fragmentation lowers the chance of further successful task placements, and thus, significantly reduces the device utilization and the over-all system performance. Therefore, an RHWOS is required to take measures against area fragmentation.

Contributions and Results
We present two on-line placement methods that pro-actively combat area fragmentation, and evaluate their benefits in comparison with common placement strategies.

We introduce a new area fragmentation metric that over-weights large area-contiguous portions of free space compared to smaller and scattered ones. Based on this metric, the placer module runs a *best-fit (=lowest fragmentation)* strategy in determining a location for each newly arrived task. In addition, we define a novel task transformation rule, the so-called *footprint-transform*, a technique which allows for further improving the placement quality by adapting a task's shape (its footprint) to the current allocation situation. In contrast to related work, we use a task model that allows polyomino shaped tasks instead of rectangles. Each task consists of a number of subtasks. We claim that this model better reflects the core-oriented design style and show better execution performance.

The experimental evaluation of our methods show performance gains of up to 18.4% compared to previously known placement policies.

4.3.1 Background and Related Work

We consider a non-preemptive on-line system (cf. Figure 26) using the 2D-variable area model and polyomino shaped independent tasks. This model induces a high complexity in determining task placements and is susceptible for area fragmentation.

In off-line scenarios, one can afford to spend the time to derive optimal or near-optimal placements with low area fragmentation. Fekete et al. [FKT01] present such a method determining compact arrangements of 3D boxes on a given area. Two dimensions define the task's rectangular size, and the third represents the execution time.

On-line task placement and 2D bin-packing are related problems. Hence, online algorithms used for 2D bin-packing problems, such as first-fit, best-fit, or different bottom-left heuristics, have been considered for task placement in [EGJ99, BKS00].

Measures against area fragmentation in an on-line scenario can be divided into *pro-active* and *re-active* ones.

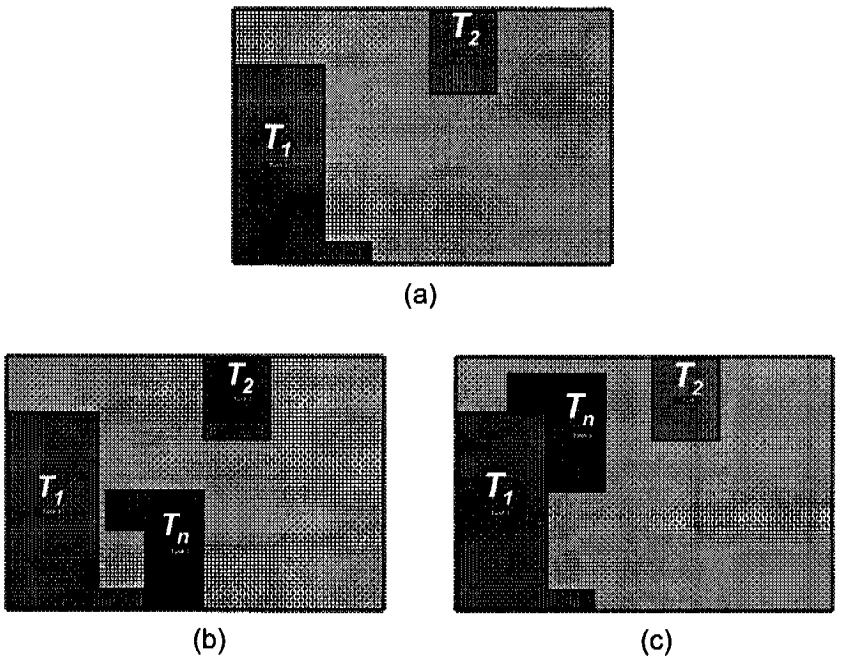


Fig. 28: Screen shots from our simulation framework: (a) Initial allocation with placed tasks T_1 and T_2, (b) First-fit placement of task T_n with $\mathcal{F}_A(\Lambda_D) = 43\%$, and (c) Best-fit placement of task T_n with $\mathcal{F}_A(\Lambda_D) = 35\%$.

Re-active strategies apply when a task cannot be placed due to fragmentation. Such a method is presented by Diessel et al. [DE98, DEM+00] by performing rearrangement techniques denoted as *local re-packing* and *ordered compaction*. Currently running tasks are packed more closely in order to free larger contiguous areas, and thus, lower the fragmentation. Compaction requires preemption, where running tasks are stopped and, continued at a different location. With current FPGA technology, preemption is rather costly in terms of time.

Another approach is presented by Brebner et al. in [BD01], where an FPGA itself computes the positions to which a task is mapped to and also the compaction. This approach bases on 1D-variable model.

A pro-active strategy is a placement technique that makes the successful placement of subsequent tasks more likely. In other terms, such a strategy aims at keeping the fragmentation as low as possible.

Compton et al. [CCKH01] discuss task relocations and transforms to reduce fragmentation. Task transforms consist of a series of rotation and flip operations.

In this section, we evaluate the performance of first-fit, best-fit, and best-fit combined with transformation operations in terms of placement quality.

4.3.2 Placement Methods

First-Fit Placement Method

First-fit searches the $W_D \times W_H$ RLU-array row-wise from the bottom-left to the top-right corner. Overlapping the search positions with the bottom-left RLU of the task, first-fit tries to match the task shape with free RLUs. The new task is mapped to the first matching position found. An example for a first-fit placement of a task T_n is shown in Figure 28(b).

Best-Fit Placement Method

The rationale behind best-fit is to place a task in a way that makes the successful placement of subsequent tasks more likely. It is reasonable to assume that this is the case when the residual areas on the surface form maximal large rectangles.

To quantify allocation situations, we introduce an area fragmentation metric of an allocation Λ_D, the *area fragmentation grade* $\mathcal{F}_A(\Lambda_D)$.

Best-fit determines $\mathcal{F}_A(\Lambda_D)$ for all possible placements and selects the position that minimizes $\mathcal{F}_A(\Lambda_D)$.

For a given allocation Λ_D, we place the largest possible free rectangle r_1 into the residual area, note its dimensions $A(r_1)$ and mark it as considered. This process is iteratively continued with the next-largest rectangle r_2, r_3, ..., r_i until the complete free area has been marked. The result is a histogram of i free rectangular areas $A(r_{1..i})$. The histogram consists of j different area-classes, each denoting n_j rectangles of size $A(r_{1..j})$. Figure 29 depicts the decomposition of the free area into free rectangles $r_{1..i}$ according to the process described above.

Based on the n_j area-classes $A(r_{1..j})$, we define the area fragmentation grade of an allocation Λ_D as follows:

$$\mathcal{F}_A(\Lambda_D) = \begin{cases} 1 - \dfrac{\sqrt{\sum_j n_j \cdot A^2(r_j)}}{\sum_j n_j \cdot A(r_j)}, & \text{if } j \geq 1; \\ 1 & \text{, otherwise.} \end{cases} \tag{4.29}$$

The rationale behind our area fragmentation metric is that it gives higher weights to *large* free rectangles compared to *small* ones. This is achieved by (i) finding maximal free rectangles r_j and (ii) adding up their *squared* areas $A^2(r_j)$. To make the fragmentation metric independent of the device size, it is normalized to the total available free area $\sum_j n_j \cdot A(r_j)$.

The area fragmentation grade $\mathcal{F}_A(\Lambda_D)$ is bounded by $0 \leq \mathcal{F}_A(\Lambda_D) < 1$. We argue, that the lower $\mathcal{F}_A(\Lambda_D)$ is, the higher is the probability that a future task T_n can be mapped into D. $\mathcal{F}_A(\Lambda_D) = 1$ for 100% utilized FPGAs.

A possible improvement of this metric would be to consider the adjacency of free rectangles, such as r_1 and r_2 in Figure 29(a,b). That is, an allocation consisting of neighboring free rectangles should yield a lower fragmentation grade compared to an allocation in which the free rectangles are scattered.

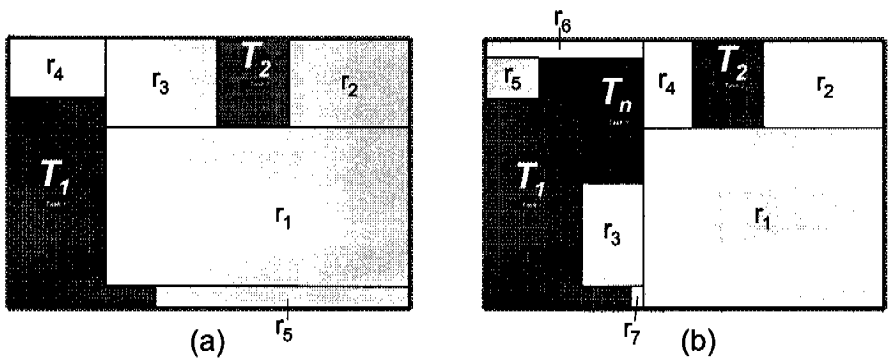


Fig. 29: Decomposition of the free area into rectangles r_i of (a) the allocation in Figure 28(a), and (b) the allocation in Figure 28(c), respectively.

Wigley et al. [WK02b] used the square of shorter side (which they call *characteristic dimension*) of an empty rectangle as a measure of fragmentation. They calculate the distribution of characteristic dimension, and the mean of that distribution is used as a measure of fragmentation. In contrast to this approach, our fragmentation grade is normalized to the actual free area and is thus device-independent.

In [TSMN04], Tabero et al. calculate the product of *suitability-values* of each piece of free area. The suitability-value itself represents a normalized value which is proportional to the ability of a free area to accommodate a rectangular task.

Recently, Handa et al. [HV04a] proposed a fragmentation metric that sums up the *fragmentation contribution* of all free RLUs. The contribution of a single RLU R is calculated based on the number of other free RLUs that are in *vicinity* to R. Two RLUs are considered to be in vicinity if they can be connected by a straight line drawn horizontally or vertically, and that line is not intersected by any occupied RLU. Hence, this metric prefers connected regions over unconnected ones.

Best-Fit (Lowest Fragmentation) Placement Example

Figure 28(a) shows an allocation with two tasks T_1 and T_2 in an array of size 64×96 RLUs[3]. The area fragmentation grade for this allocation amounts to $\mathcal{F}_A(\Lambda_D) = 35.6\%$. A third task T_n is to be placed. Figure 30 presents the resulting area fragmentation grades for the different positions of the task in the RLU array. In this plot, $\mathcal{F}_A(\Lambda_D)$ is set to 0% when T_n cannot be placed.

Due to the dimensions of the new task T_n, only a subset of the array has to be checked. Figure 28(b) displays the first-fit placement of T_n resulting in a $\mathcal{F}_A(\Lambda_D) = 43\%$, whereas Figure 28(c) shows the best-fit placement which

[3]According to a XILINX VIRTEX XCV-1000 FPGA.

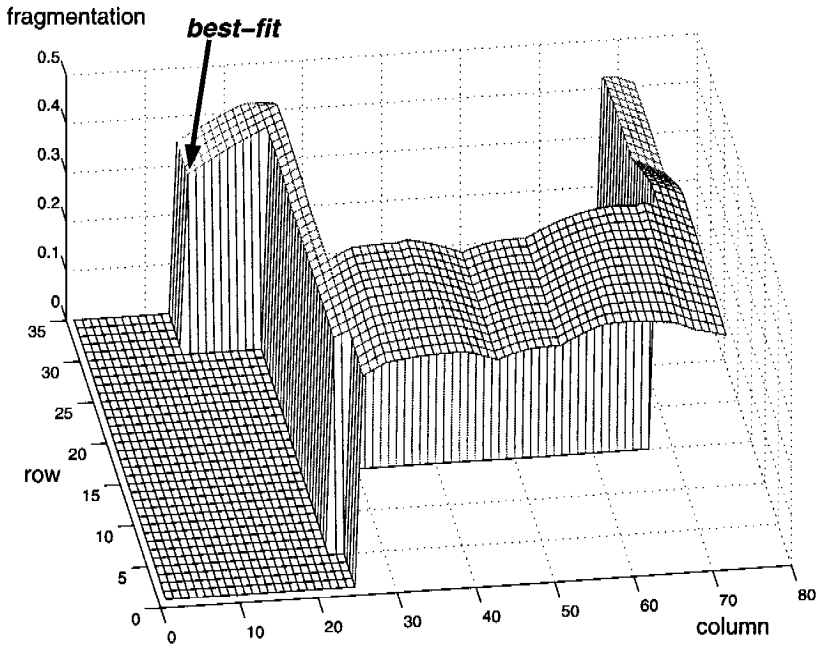


Fig. 30: Fragmentation grade $\mathcal{F}_A(\Lambda_D)$ for the example in Figure 28 as a function of all feasible placements of T_n in a XILINX VIRTEX XCV-1000 RLU-array.

minimizes $\mathcal{F}_A(\Lambda_D)$ to a value of 35%.

4.3.3 Footprint Transform

Due to the high area fragmentation caused by dynamically placed tasks $T_{1..n}$, the undesirable situation can occur, that a task T_i cannot be placed, although enough free RLUs are available on the device. Formally, this situation can be described as

$$A_i^m \leq A_D - \sum_{j=1}^{n} A_j^m \tag{4.30}$$

however, (according to Equation 4.16) no feasible location $P = (x_p, y_p)$ to place T_i can be determined that fulfills the placement constraint as defined in Equation 4.14.

We denote such a situation as *placement paradox*. In a preemptive system, such a situation could be used to initiate a compaction sequence as proposed in [DE98, DEM+00]. For a non-preemptive system, we propose a different approach.

Coarse-granular tasks often consist of a set m of subtasks $S_{1..m}$ (cf. Section 4.2.1). This motivates the translation of individual subtasks in order to fit the overall task into the currently available FPGA area. We have experimented with

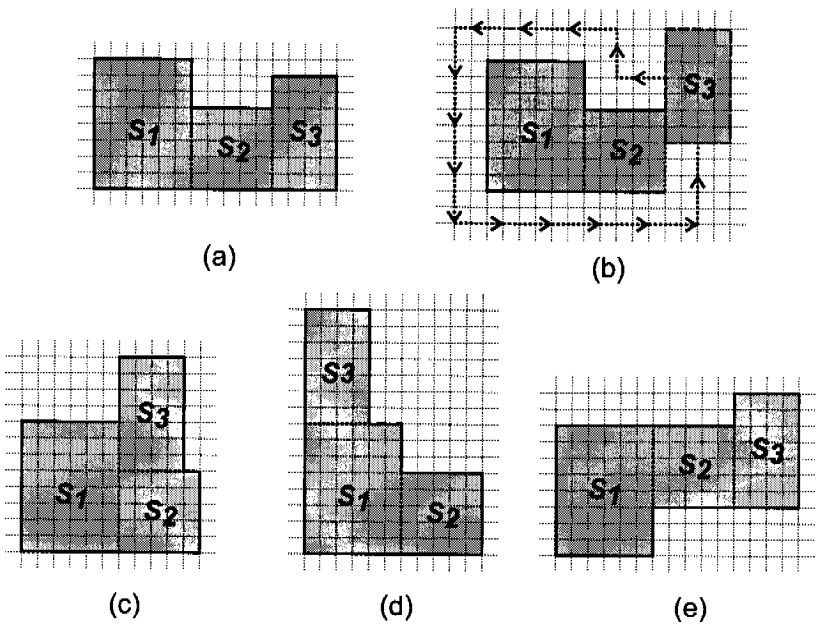


Fig. 31: (a) Footprint-transformable task T_i consisting of three Subtasks $S_{1..3}$, (b) possible transformation steps for subtask S_3, (c/d) Resulting task shapes T_i' and T_i'', respectively, after footprint-transformations of subtask S_3, (e) Resulting task shape T_i''' after footprint-transformation of subtask S_1.

a footprint-transform that extends the first-fit placement technique. This is a novel re-active strategy that does not require preemption.

When no direct fit for a task T_i is found, one of its subtasks is translated to different positions with the restriction that the overall task area must remain contiguous. Figure 31 shows an example. The task T_i in Figure 31(a) is transformable and consists of three subtasks $S_{1..3}$. Figure 31(b) indicates the possible positions to which subtask S_3 can be translated. When the transforms do not lead to a feasible placement, the next subtask, e.g. S_1 is considered. Figures 31(c) and 31(d) show different footprint-transformed tasks T_i' and T_i'' with translations of S_3; Figure 31(e) displays T_i''' after a translation of subtask S_1.

Footprint transforms at the RLU level are atomic. All other transforms, including rotation and flipping, as described in [CCKH01], can be expressed by a sequence of RLU translations. However, transforming many RLUs of a coarse-grained task does not seem to be realistic with current FPGA technology, because of the complex issues of re-routing and timing preservation. Lerjen et al. [LZ02] report on a successful implementation of this technique using a XILINX VIRTEX FPGA, but limited to RLU level and rather simple tasks.

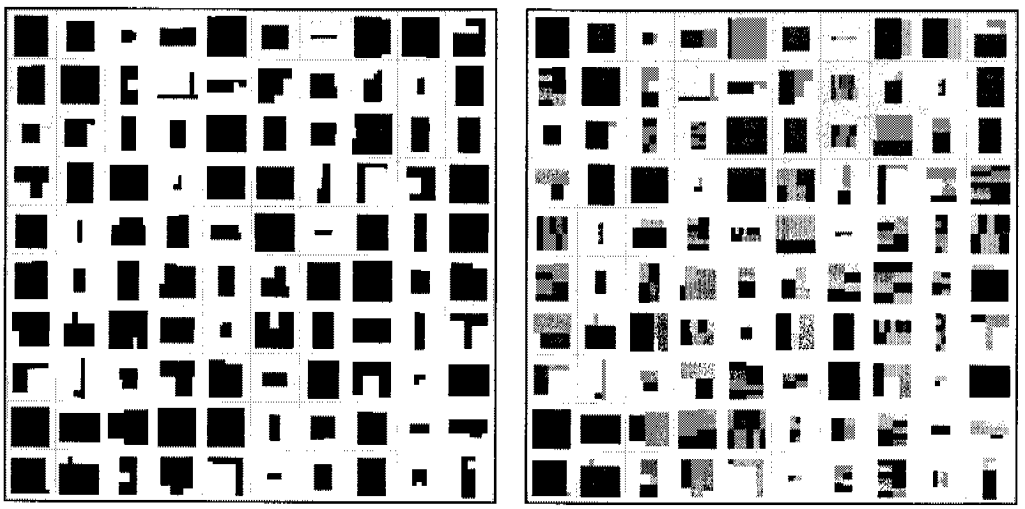


Fig. 32: Randomly generated task set consisting of 100 polyomino shaped tasks: (a) shows the outline only, whereas in (b) the different grey-levels denote the subtasks.

Combined First-Fit and Footprint-Transform Placement

The combination of first-fit and footprint-transforms is a possibility to solve the placement paradox in some cases: The placer first searches for a direct first-fit, i.e., a free area large enough to accommodate a newly arrived task T_n (in its original shape). In case of no possible direct placement, the placer tries footprint-transforms to modify the shape of T_n. When a feasible position for a transformed task T'_n is found, the task is placed at this location.

Many scheduling techniques are conceivable. In our current implementation, the scheduler searches the pending task queue for the first task that fits and places it. This process is repeated until no more tasks fit. The scheduler is invoked every time a new task arrives or an executing task terminates.

4.3.4 Evaluation and Conclusions

To evaluate the placement techniques, we have conducted a series of simulation experiments.

Simulation Settings

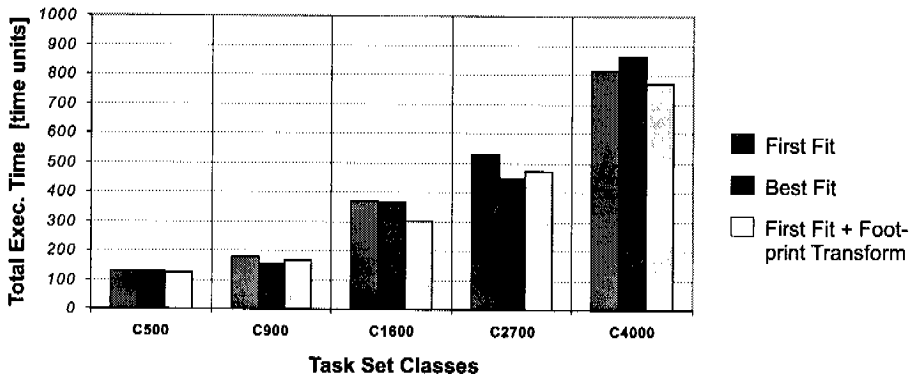
We simulated the system described in Section 4.2.3 with randomly generated task sets. Each task set contains $i = 100$ tasks $T_{1..100}$. The arrival times a_i are equally distributed in $[1, 100]$ time units, the execution times e_i in $[5, 25]$ time units. We have generated five classes of task sets differing in task size. The classes are denoted by C_j and contain tasks of equally distributed size in the interval $[100, j]$

RLUs. The classes are C_{500},C_{900},C_{1600},C_{2700}, and C_{4000}, respectively. Each task exhibits a polyomino shape and consists of a random number of subtasks. Figure 32 displays one randomly generated task set, whereas (a) shows only the outline, and (b) how the task is structured in subtasks, each denoted by different grey-levels.

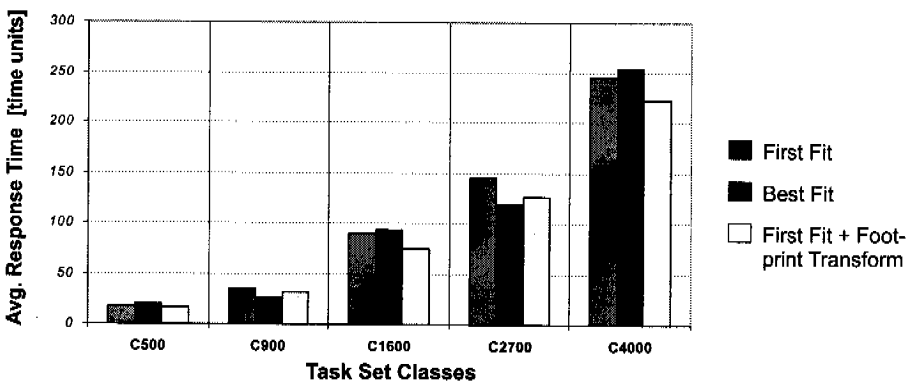
Simulation Results

Figure 33 presents the simulation results for the five task classes. Figure 33(a) shows the total execution time t_{tot}, and (b) the average response time $\overline{r}$, respectively, for the placement techniques first-fit, best-fit, and first-fit combined with footprint-transform. In each class, the data have been averaged over three runs with different task sets. For our specific task sets, the following observations can be made:

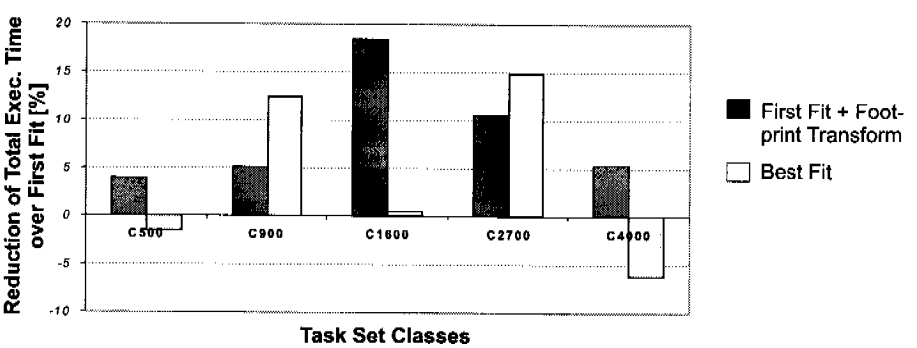
- Best-fit sometimes performs better than first-fit, sometimes worse. The differences can be significant; e.g., for C_{2700} best-fit outperforms first-fit by 14.9% in the total execution time. On average, best-fit improved the execution time by only 4%.
- Footprint transform turns out to be quite beneficial. On average, 25% of the tasks that could be placed were footprint transformed. If tasks are small compared to the overall FPGA area (C_{500}), footprint transform is not that often applied, as many tasks have direct fits. For large tasks, on the other hand, even footprint transform is often unsuccessful. The biggest improvement was achieved for task set C_{1600} with 18.4%. On average, footprint transform improved execution time by 8.7% over first-fit.
- Figure 33(c) displays the reduction in execution time using best-fit and footprint transform. Although we have not concentrated on efficient implementations of the placement techniques, best-fit and footprint transform are clearly more complex than first-fit. From the simulation result, it could be concluded that time spent for a complex placement technique is best invested in footprint transform, as this technique gives sound and substantial improvements.



(a)



(b)



(c)

Fig. 33: Simulation results: (a) Total execution times, (b) Average waiting times, and (c) Reductions in execution times for different placement techniques and task set classes.

4.4 Partitioning-based Free Area Management

Problem Statement

The demand for a *reconfigurable resource management function* is one of the major differences between an RTOS and an RHWOS. The interesting question is *how* the free reconfigurable area of the FPGA can be managed. An algorithm executing this management function is expected to (i) facilitate a high resource utilization, and (ii) meet the requirements of an on-line scenario in terms of both time-efficiency and space-efficiency. The basic problem in managing the free area is to prevent so-called *placement mistakes*. A placement mistake eventuates when a task cannot be placed even though there is a free area on the FPGA of sufficient size and appropriate dimensions. The reason for this incident can be found in the data-structures of the reconfigurable resource manager (RRM) if representing the free area in a disadvantageous way. The design of the RRM algorithm involves a trade-off between its quality and complexity: An optimal algorithm causes no placement mistakes but is costly. A more economical one leads to placement mistakes and, thus, lowers the utilization of the reconfigurable area.

Contributions and Results

A promising way to manage the free area is to partition it into free rectangles. The main reference for this approach is Bazargan et al. [BKS00]. They developed an efficient heuristic-based algorithm that partitions the free area into a set of non-overlapping free rectangles. Starting from this algorithm, we derive three new partitioner versions, which all manage larger free rectangles and, thus, feature a significantly higher placement quality than Bazargan's. The complexity of our partitioners is only marginally higher. In order to evaluate the developed algorithms, we used our time-discrete simulation framework.

Our partitioning method improves the placement quality by up to 70% compared to [BKS00].

4.4.1 Background and Related Work

We consider the 2D-variable area model and rectangular shaped tasks that should be placed in an on-line environment. In an on-line scenario, tasks may arrive and finish execution at any time, leading to complex allocation situations on the FPGA. In order to be able to decide where a newly arrived task can be placed, the state of the FPGA, i.e., the free area, must be managed.

A straightforward way of managing the free area is to mark each RLU as *free* or *used* and to check all possible locations for an arrived task T_n. In a device D, there may be no more than $((W_D - W_n) \cdot (H_D - H_n))$ possible placements to consider.

To reduce the potentially large number of possible locations and increase placement efficiency, Bazargan et al. [BKS00] proposed basing the placement on

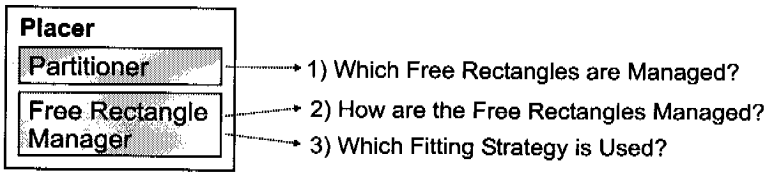


Fig. 34: Decomposition of the task placer module

keeping rectangular areas of free FPGA space. They presented two placement methods. The first method keeps all maximal free rectangles, i.e., rectangles that are not contained in other rectangles. Keeping all maximal free rectangles is optimal in the sense that a feasible location can be found, if one exists. On the other hand, the method has to manage $O(n^2)$ rectangles for n placed tasks $T_{1..n}$, and task insertion and deletion are difficult to implement. Bazargan's second method sacrifices optimality but is much more efficient as it only keeps $O(n)$ non-overlapping rectangles.

We have chosen the second version as a starting point for further investigations and optimizations. The goal is to achieve a higher placement quality, but unchanged complexity.

We can identify three main questions when developing rectangle-based placement algorithms:

1. *Which free rectangles are managed?*
   We have to decide which set of free rectangles is managed and how the operations *insert* and *delete* are implemented over this set.

2. *How are the free rectangles managed?*
   We have to choose a data structure that allows for efficient task operations.

3. *What kind of fitting strategy is used?*
   Generally, there will be more than one free rectangle fitting for a task. The *fitting strategy* decides which one to choose.

We divide the placer module into two submodules as shown in Figure 34. The *partitioner* deals with question 1) and is described in the remainder of this section. The *free rectangle manager* deals with questions 2) and 3) and is elaborated in Section 4.5.

Bazargan's Partitioner

Bazargan's efficient partitioner [BKS00] keeps a number of free rectangles linear in the number of placed tasks. Figure 35 shows the *insert* procedure of Bazargan's partitioner. The placer configures a task T_1 in the bottom-left corner of a free rectangle A. The free space splits into two smaller rectangles B

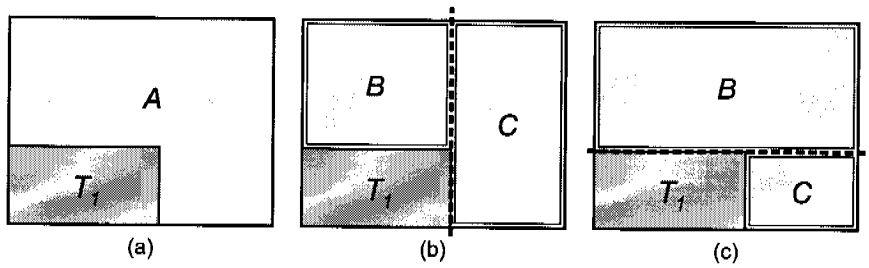


Fig. 35: Bazargan's heuristic-based splitting decisions: (a) A newly arrived task T_1 is placed in rectangle A, (b) Vertical split, and (c) Horizontal split, producing rectangles B, and C, respectively.

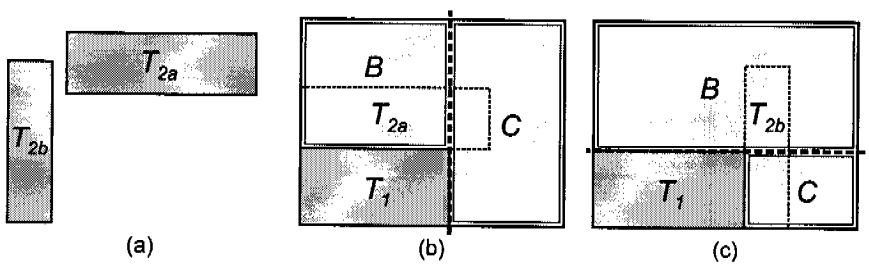


Fig. 36: Placement mistake: (a) Two possible tasks T_{2a} and T_{2b}, (b) Task T_{2a} cannot be placed, (c) Task T_{2b} cannot be placed, due to the wrong split decision.

and C, either vertically or horizontally, as shown in Figures 35(b) and 35(c), respectively.

To decide on which of the two splits should be performed, Bazargan et al. proposed several heuristics.

Because a free rectangle can split into two new rectangles at most, a binary tree is used to represent the FPGA state. The currently free rectangles are the leaves of the tree.

All data used in this algorithm are stored in a *rectangle tree*, as shown in Figure 37. The rectangle representing the whole surface of the FPGA is the root of this tree, Figure 37(a),whereas the leaves correspond to the free rectangles managed in the algorithm. Each split operation of a free rectangle creates two *child rectangles* and lets the tree grow by one level, Figure 37(b). In contrast, a merge operation combines two leaves and turns the superior node (parent) into a leaf.

The merge step after the *deletion* of a task basically consists of reverting to the state before the task was inserted. Figure 37(f) illustrates this. Task T_2 is inserted into rectangle C, Figure 37(c), which splits the residual free space of C into D and E. After the ensuing deletion of task T_2, rectangles D and E are deleted and rectangle C is marked as free again.

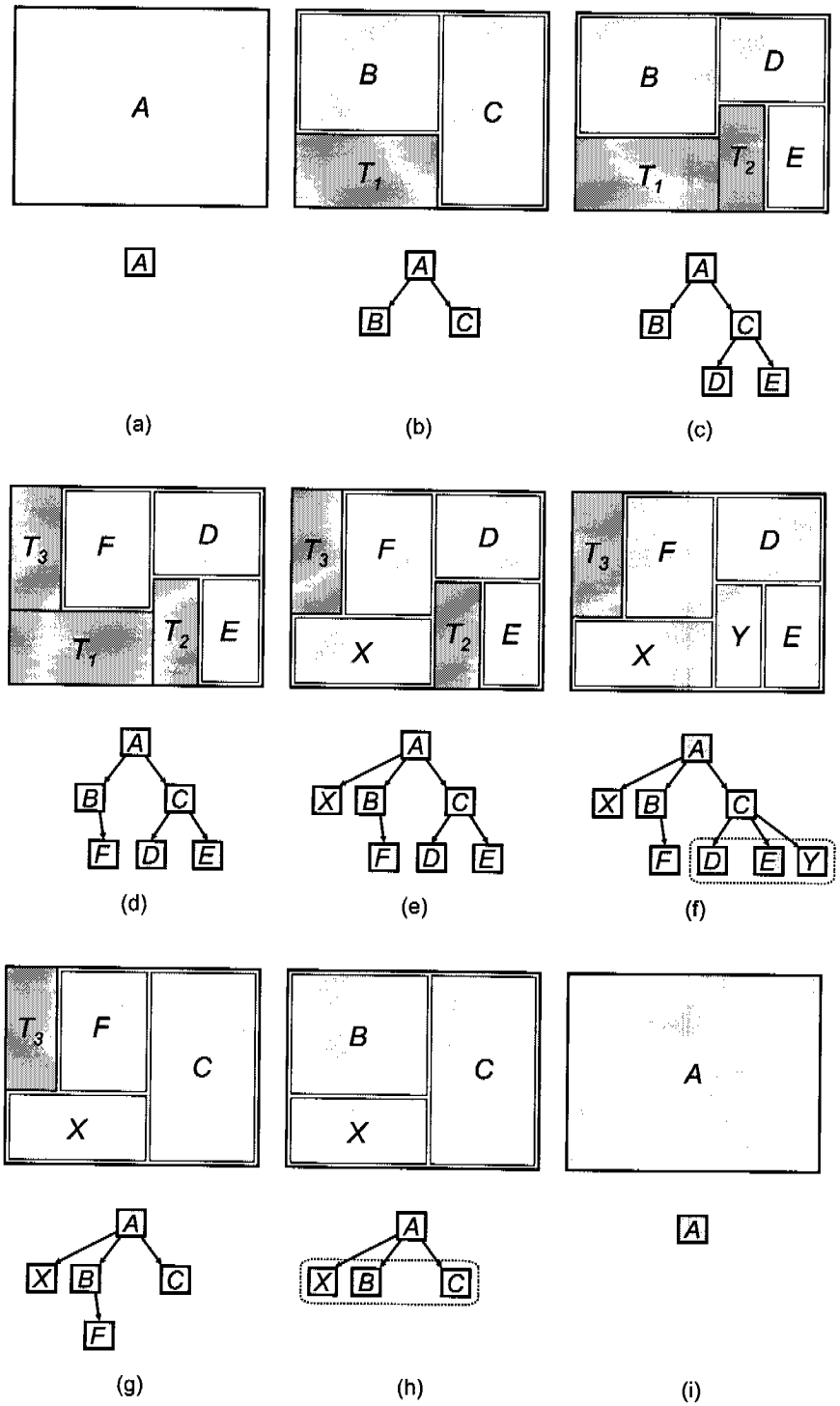


Fig. 37: Task insertion and merge steps, and corresponding tree data-structure.

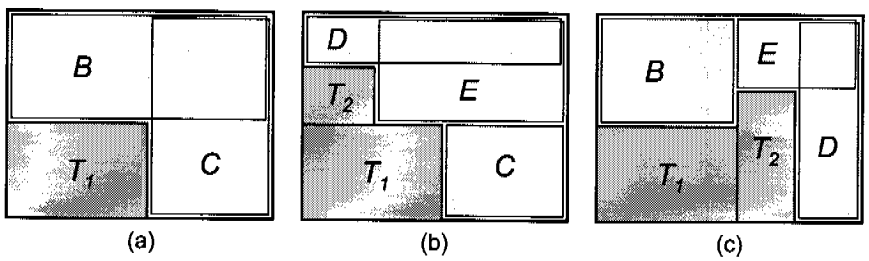


Fig. 38: Overlapping child rectangles, (a) B/C and, (b/c) D/E, as a consequence of delaying the split decision.

Placement mistakes
Depending on the set of managed free rectangles, the placer might claim that a task cannot be placed even though there is a free area on the FPGA of sufficient size and appropriate shape. Such an incident is called a *placement mistake*. Figure 36 shows situations in which placement mistakes can occur as a result of a wrong split decision. Task T_1 was placed and the residual area was split vertically, as displayed in Figure 36(b). The newly arrived Task T_{2a} can not be placed due to the wrong split decision. On the other hand, if the split was done horizontally, Figure 36(c), the placement of a task T_{2b} will fail.

4.4.2 Enhancements of Bazargan's Partitioners

In the following, we present our three enhanced versions based on Bazargan's partitioner.

Enhancement I: Delaying the Split Decision
We have developed an enhanced version of Bazargan's partitioner with the same efficiency but improved placement quality. Our enhanced method delays the basic vertical/horizontal split decision and manages overlapping rectangles in a restricted form.

Bazargan et al. use heuristics to decide whether a free rectangle is split vertically or horizontally on a task insertion. No matter how good such a heuristic is, there is always the possibility of conducting the wrong split. That is, the next task cannot be placed in one of the resulting rectangles due to the wrong split decision.

The decisive observation is that the split decision can be *delayed*: whenever a task is inserted into a rectangle, two *overlapping* children rectangles are created as shown in Figure 38(a). The split decision for a rectangle A is not made until the next task for one of the two children, B or C, arrives. If the next task is inserted into rectangle B, the height of rectangle C is resized such that B and C do not overlap any more. Vice versa, an insertion into C leads to the correction

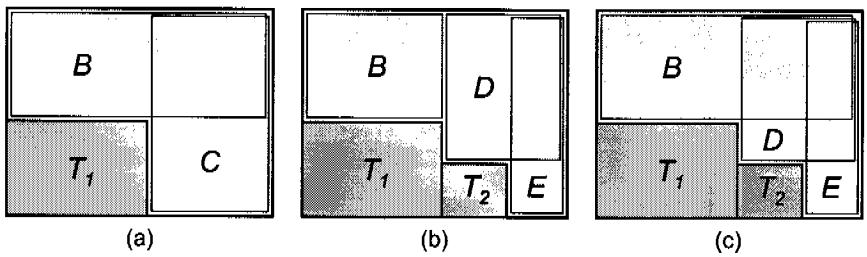


Fig. 39: Motivation for OTF partitioning

of B's width, Figure 38(c).

Delaying the split decision corresponds to a perfect heuristics: the split decision is taken at a point in time when it is known into which one of the two child rectangles a task is inserted.

Our enhancement of Bazargan's method requires only minor changes in the algorithm. If a task is inserted into rectangle R, we have to check whether R overlaps with its brother rectangle in the binary tree. In such a case we have to resize the width or height of R's brother, respectively. The task deletion procedure is identical to Bazargan's method.

The newly created child rectangles of R now have to overlap each other. In contrast to Bazargan's method, there is no split decision taken at this point in time. Delaying the split decision yields a considerable performance gain (see Section 4.4.4).

Enhancement II: On-the-fly (OTF) Partitioning

Our *On-the-fly (OTF)* partitioner defers the split decision even further. Consider the example shown in Figure 39. Task T_1 has been inserted and two overlapping child rectangles B and C have been created according to our enhanced version of Bazargan's partitioner, see Figure 39(a). Now, task T_2 arrives and is to be placed in rectangle C. The enhanced Bazargan partitioner resizes rectangle B, which is shown in Figure 39(b). However, task T_2 does *not* overlap with rectangle B. Therefore, one can leave rectangle B at its original size, getting a better partition of the free space, Figure 39(c).

The price to be paid is that it might be necessary to resize several rectangles after inserting a new task. Figure 40 illustrates this by extending the example of Figure 39(c).

Starting from an allocation depicted in Figure 40(a), 40(b) shows the result of inserting a task T_3 into rectangle B which overlaps C. The whole subtree of rectangles rooted at C has to be resized; i.e., the height of the rectangles in the subtree (in this case E and D) needs to be corrected. The implementation of the OTF partitioner differs from the enhanced Bazargan partitioner only in that all rectangles of a subtree might be resized at a later point in time. That is, rectangles

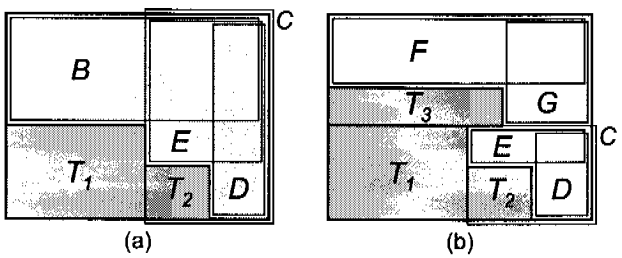


Fig. 40: Resizing of brother rectangles

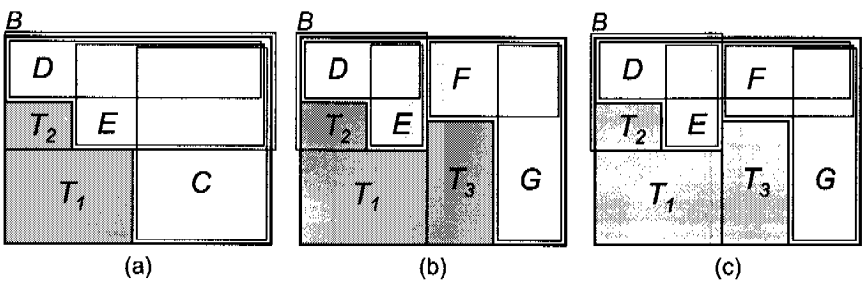


Fig. 41: Enhanced OTF: selective resizing

that have been resized upon the insertion of a task in a different subtree are not restored to their initial size when the task is deleted.

Enhancement III: Enhanced OTF-Partitioning
The OTF partitioner can further be improved, which results in the *Enhanced OTF* method. We have implemented two enhancements:

1. *Resizing rectangles only if necessary*
   In the OTF algorithm, all rectangles in a subtree are resized if a newly placed task overlaps with the root rectangle of the subtree. It might happen that rectangles that do not overlap with the task are resized.

   An example is shown in Figure 41. Figure 41(a) displays the initial situation occurring when the OTF partitioner inserts the first two tasks T_1 and T_2. Now a new task T_3 is inserted into rectangle C, which leads to the resizing of all rectangles rooted in B. The result of the resizing process is shown in Figure 41(b). Note that rectangle D was resized even though it did not overlap with T_3. Figure 41(c) shows the resulting partition of the free space if D was not resized.

2. *Resizing rectangles upon task deletion*
   The OTF partitioner resizes rectangles when new tasks are inserted. If the task which triggered such a resizing is deleted, the resized rectangles do not regain their original size. The enhanced OTF partitioner re-resizes

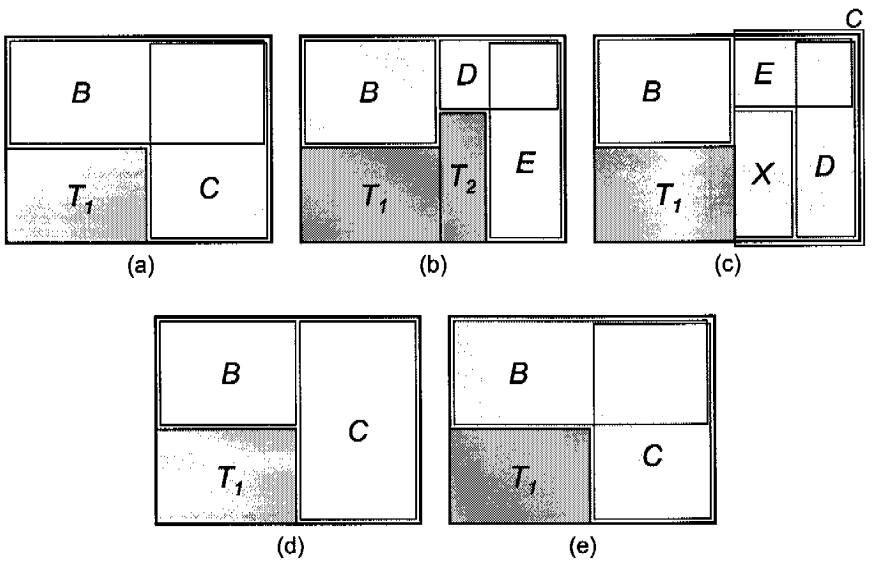


Fig. 42: Enhanced OTF: resizing upon deletion

rectangles back to their original size if tasks are deleted.

See Figure 42 for an example. Figure 42(a) shows the situation after task T_1 has arrived. Then, T_2 is inserted in C, Figure 42(b). After T_2's completion, all child rectangles of C (E, D, and X) are free again, thus a merge step is carried out, Figure 42(d). Afterwards, rectangle B is resized to its original size, see Figure 42(e).

4.4.3 Complexity

Space Complexity

The number of nodes in the rectangle tree is linear in the number of tasks running on the FPGA. Whenever a task is inserted, at most 2 new rectangles are generated. Upon the deletion of a task, one new rectangle is inserted in the tree if no merge step is possible. Hence, the number of nodes in the tree (i.e., the number of managed rectangles) can be bounded by $1 + 3 \cdot n$, with n being the maximum number of tasks executing simultaneously on the FPGA. Assuming a minimal task size $A_i^m = 50$ RLUs and an FPGA size $W_D = 96$, $H_D = 64$ RLUs[4], the maximum number of nodes may not be bigger than $3 \cdot \lfloor \frac{W_D \cdot H_D}{A_i^m} \rfloor + 1 = 367$ entries. Assuming the size for one node of 50 bytes, an upper bound for the space requirement is $367 \cdot 50$ bytes, roughly $18kB$.

Practically, this upper bound is not reached by far: Figure 45 shows the maximum and average number of managed rectangles for the OTF partitioning algo-

[4]Corresponds with a XILINX VIRTEX XCV-1000 device.

rithm. The reason for the significantly lower number of managed rectangles is because (i) tasks can be larger than the assumed size of 50 RLUs, and (ii) placement paradoxa (cf. Section 4.3.3) may occur due to the dynamic placement of tasks (Placement paradoxa reduce the number of simultaneously running tasks).

Run-Time Complexity

The insert and delete operations of the partitioner module operate on the rectangle tree (cf. Figure 37). Depending on the partitioner, some parts of this tree need to be traversed. Since the number of nodes in the tree is linear in the number of tasks executing simultaneously on the FPGA, the complexity of the insert and delete operations of all four partitioners is at most linear.

More specifically, we measured the complexity of the partitioning algorithms by looking at the average number of nodes visited in the rectangle tree per task insertion. Figure 46 shows those numbers for the task class C_{500}. As expected, the newly developed partitioning algorithms visit more nodes in the rectangle tree.

4.4.4 Evaluation

Simulation Settings

In order to evaluate the developed algorithms, we have constructed a time discrete simulation framework which allows the system parameters for randomly generated task sets to be measured. All the simulations were conducted for an FPGA with a RLU-array of size 96×64, corresponding to Xilinx's Virtex XCV-1000.

We have generated task sets in six different classes, varying in the task size. The classes are C_{100}, C_{300}, C_{500}, C_{900}, C_{1600} and C_{2700} (cf. Section 4.2.4). For every C_i, 50 task sets have been generated with 100 (200 for C_{100}) randomly generated tasks each. Simulation results have been averaged over these 50 task sets.

For all task classes, task computation times are equally distributed in [5,25] time units. Arrival times have been chosen to be equally distributed in the ranges [1,15] to [1,800], depending on the task class. Different arrival time ranges for different C_i make sure that the waiting times of tasks and the system's task load stay within limits that allow a proper analysis of the characteristics and effects of the different placement techniques.

Results

We have simulated all partitioners, combined with all fitting strategies described in previous sections. For each partitioner and C_i, we have selected the fitting strategy yielding the best performance.

Figure 43 presents the performance of our three new partitioners compared to

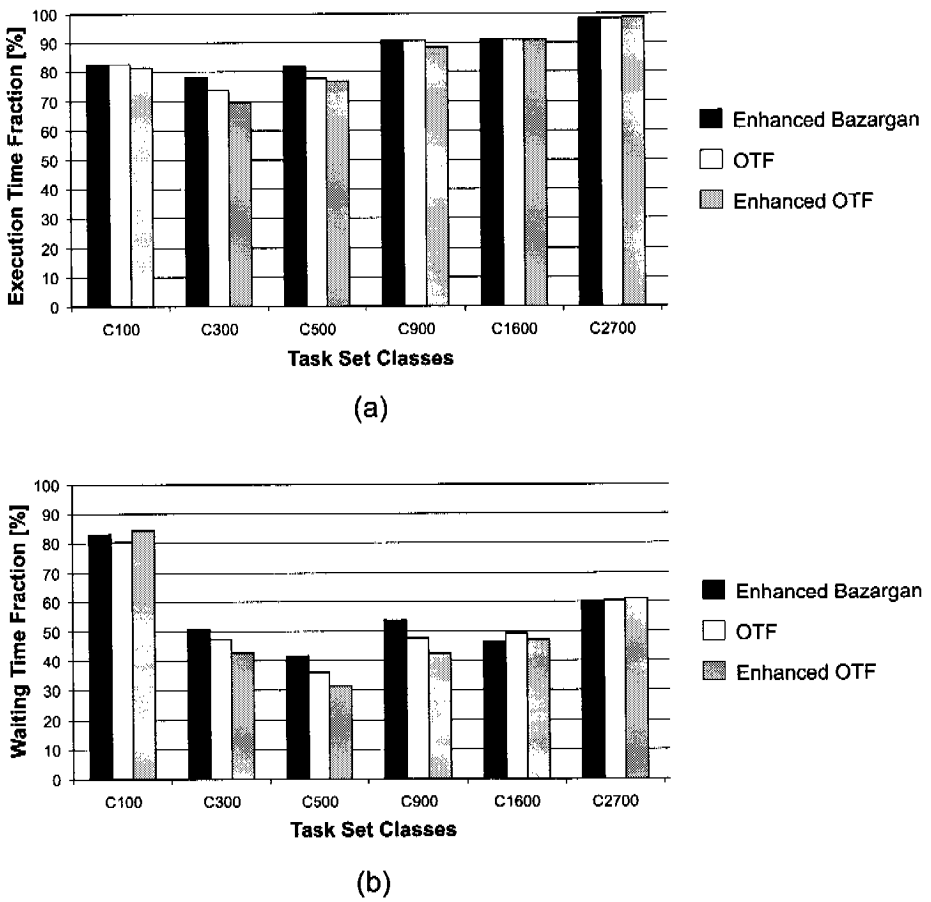


Fig. 43: Partitioner performance: Fraction of average waiting time over Bazargan's partitioner.

the partitioner of Bazargan et al. Figure 43(a) shows the percentage of the *total execution time* t_{tot} and Figure 43(b) the *average waiting time* $\overline{w}$ our methods achieve compared to Bazargan's method. The following observations can be made:

- Using our new partitioners, the *total execution time* t_{tot} and the *average waiting time* $\overline{w}$ can be reduced by up to 30% and 70%, respectively. The improvement in t_{tot} is smaller because this metric depends strongly on the arrival times a_i of the tasks. Tasks with late arrival times diminish any placement improvements achieved before.
- The performance differences are biggest for medium-sized tasks. For small tasks, Bazargan's partitioner splits the free area in a very large and in a very small rectangle. Improved partitioning methods are less effective here, as the large rectangle is likely to accommodate any upcoming task. The placement of big tasks leaves only very small rectangles. Placing the next task is then

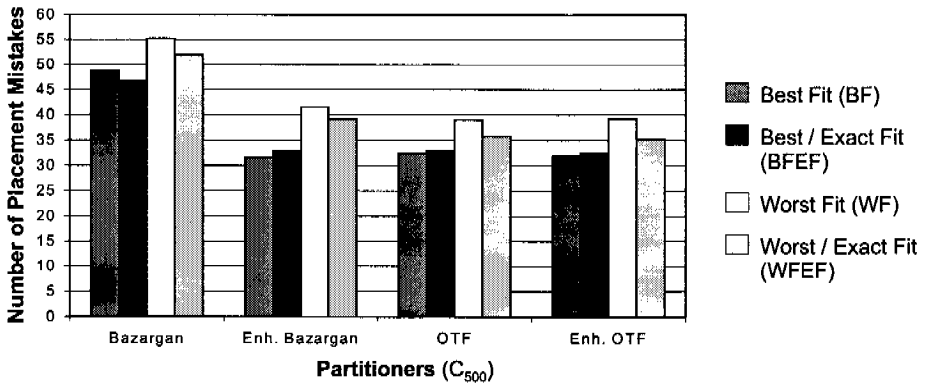


Fig. 44: Number of placement mistakes vs. partitioner and fitting strategy (for C_{500}).

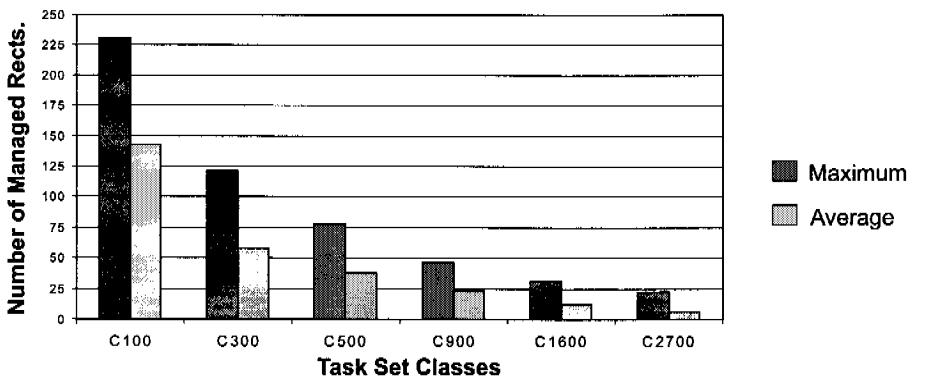


Fig. 45: Partitioner performance: Maximum and average number of managed rectangles for the OTF partitioner.

often unsuccessful. Again, the difference between different partitioners is smaller.

4.4.5 Conclusion

The simulations showed that the newly developed partitioning algorithms substantially improved Bazargan's partitioner in terms of total execution times t_{tot} and especially in terms of average waiting time $\overline{w}$. Comparing the newly developed partitioners, the OTF partitioner outperforms the enhanced Bazargan in all scenarios. A small performance difference can be observed between the OTF partitioner and the enhanced OTF partitioner.

The new partitioners were shown to be only slightly more complex than

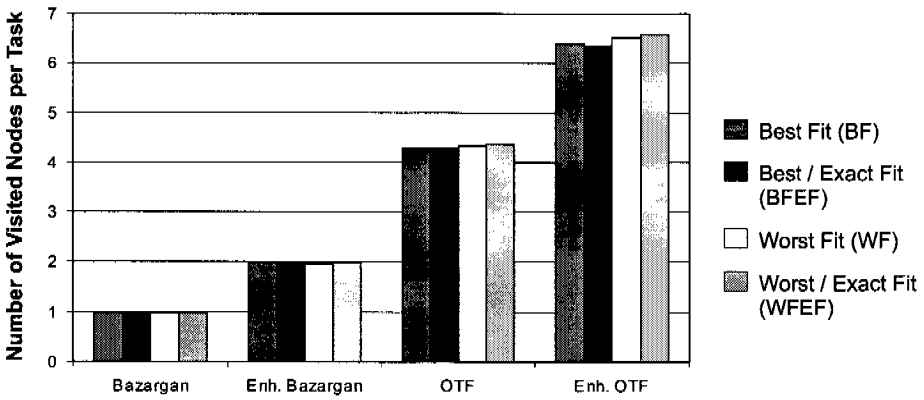


Fig. 46: Average Number of Visited Nodes per Task for every Partitioner.

Bazargan's partitioner. Considering the impressive performance improvements, the additional complexity is acceptable.

4.5 Run-Time Hardware Task Placement

Problem Statement

In an RTOS, the executable code representing a task resides in the memory at every point in time. Starting a task mainly means to set the CPU's *program counter* to the memory address of the task's entry point. This process lasts a few clock cycles. In an RHWOS, a hardware task activation process is more elaborate and involves several steps, essentially *finding a placement*, *updating data-structures* (carried out by the *task placer module*), and *loading the partial bitstream* which represents the hardware task (performed by the *FPGA driver*). Nevertheless, short and predictable task latencies form stringent preconditions for scheduling tasks in a real-time environment. Hence, the task management functions of an RHWOS also need to ensure a short hardware task latency.

Since the maximal bandwidth of the configuration port is a given FPGA device parameter, the potential for shortening the loading time is limited. Consequently, the time to find a placement and to update the internal data-structures needs to be optimized. In this section, we focus on this issue.

Contributions and Results

Our approach to keep the hardware task latency low is based on two steps: (i) by means of reordering parts of the task activation sequence, we enable parallelization which shortens the over-all task activation process, and (ii) we employ a two-dimensional hashing approach for quickly finding a task placement.

We explicate the algorithms needed to create the hash-matrix. Further, we analyze the influence of different task fitting strategies in the complexity of the hash-matrix updates. With the help of our time-discrete simulation framework, we evaluated the performance and the dynamic behaviour of our solution.

Bazargan et al. [BKS00] presented a method to find a placement in a linear time, depending on the number of already running tasks. Our novel data-structures allow for determining a task placement in constant time $O(1)$.

4.5.1 Background and Related Work

A placer was described by Bazargan et al. [BKS00] that finds a feasible location for a newly arrived task in $O(n)$ time, where n is the number of already placed tasks $T_{1..n}$ on an FPGA. They scanned a linear list holding all free rectangles to find one which fits the task. After a free rectangle was found, the task was inserted in this rectangle and the data-structures were updated prior to configuration of the task.

To compare this method with our approach, we first review the entire task placement and loading process: Figure 47(a) shows the steps that need to be done when a new task T_i arrives at a point in time a_i. First, the *placer* has to find a location on the FPGA at which the task fits. Second, the *placer* has to update the data structure representing the FPGA state. Finally, the *loader* starts

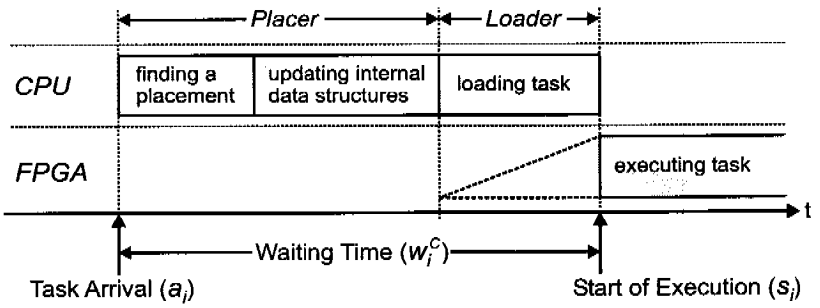


Fig. 47: Runtime task placement: Conventional sequence (waiting time w_i^C).

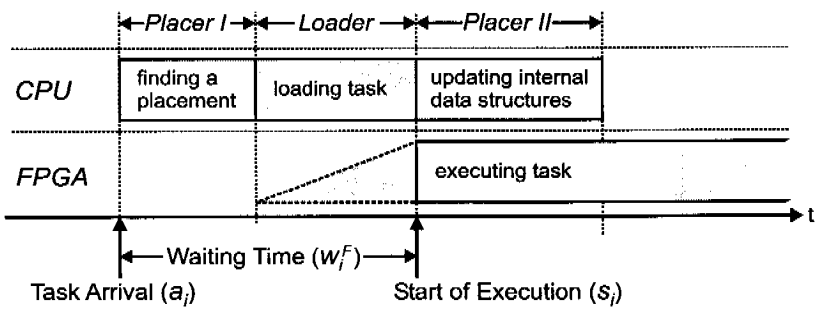


Fig. 48: Runtime task placement: Reordered sequence with reduced waiting time w_i^F.

to configure the task onto the FPGA. Only after the configuration process has been completed, task T_i can start executing (at point in time s_i).

Since we consider an on-line scenario, we want to minimize, $(s_i - a_i)$, i.e., the time elapsed from a task arrival to the beginning of its execution.

The loading time depends on the size of the task and the bandwidth of the FPGA's configuration port. The placement time, however, is largely dependent on the placer's data structures and algorithms.

In Bazargan's approach [BKS00] the free rectangles are located at the leaves of the binary tree that represents the FPGA surface. Given a new task, the placer has to search this list for a suitable free rectangle. The updating step involves operations on the binary tree, as described in the previous Section 4.4.

An important observation, as shown in Figure 48, is that we can start loading a task (see phase *Loader*) immediately after finding a feasible placement (right after phase *Placer I*). The update of the placer's data structure (*Placer II*) can be delayed and done in parallel to the task execution on the FPGA. In this context, two questions need to be answered:

1. How can a suitable free rectangle be quickly found?
2. Which rectangle should be selected if there is more than one suitable?

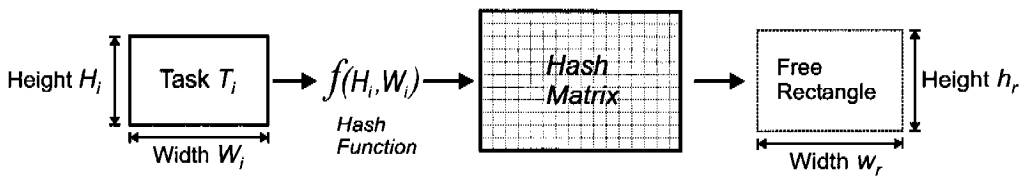


Fig. 49: 2D-hashing approach to quickly find a free rectangle for task T_i with dimensions $W_i \times H_i$.

We have developed a placer that maintains a hash-matrix in addition to the binary tree. The hash-matrix allows a suitable rectangle to be found in constant time. Compared to Bazargan et al., we provide the fastest possible method to find a location. The price paid is that we have to spend time for the update of this data structure.

Our Approach: 2D-Hashing

Hashing is a process during which data items are stored in a data structure called a *hash-table*. A hash-table supports the dictionary operations *search*, *insert*, and *delete*.

In hashing, items are retrieved by means of *keys*. A *hash-function* maps a key to the entry in the hash-table that holds the data item referenced to by the key. For a detailed description of the hashing principle, we refer to [NH93].

Figure 49 shows how we adopted the hashing approach to be used for managing free rectangles: an item in the hash-table represents a free rectangle. The key with which the hash-table is accessed is built of the dimensions of the task. The hash-function maps the dimensions of the task to a free rectangle of sufficient size.

Given an FPGA of size $H_D \times W_D$ RLUs, we define a *hash-matrix* as an array ar of size $H_D \times W_D$ elements (see Figure 50). A free rectangle of size $a \times b$ is associated with the entry $ar\,[a, b]$ of this array. Every entry consists of a pointer to a list of free rectangles of the corresponding size and a so-called *free pointer*. All free rectangles are stored in the hash-matrix. The following invariant for the free pointer of every entry must hold:

Invariant: The free pointer of entry $ar\,[a, b]$ points to a free rectangle R with $R.height \geq a$ and $R.width \geq b$ according to a given fitting strategy.

If the invariant is enforced, finding a free rectangle for a new task is very efficient. Assuming that a newly arrived task has width b and height a, retrieving the suitable free rectangle takes one line of code:

```
return hash_matrix[a][b].free_pointer;
```

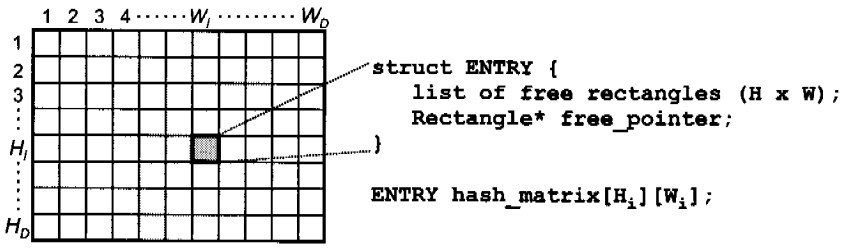


Fig. 50: Hash-matrix.

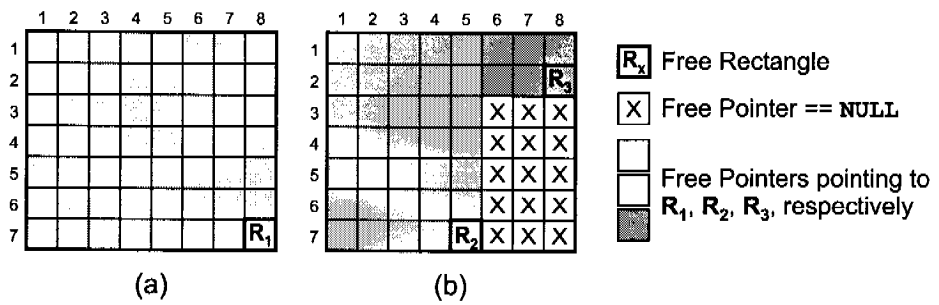


Fig. 51: Updating the free pointers

This code is executed by the placer module whenever the scheduler asks for a placement. The hashing approach therefore clearly fulfills the timing requirement for an on-line scenario.

This efficiency comes at the price that all free pointers in the matrix must be kept consistent. Whenever a new task is inserted or deleted, free pointers of some entries need to be updated. Figure 51 shows an example to illustrate this. Initially, 51(a), the hash-matrix holds one empty rectangle R_1 reflecting an empty FPGA surface. If a task of size 5×3 is placed, the previously empty rectangle R_1 is deleted and two new free rectangles R_2 and R_3 are inserted into the hash-matrix. During this process, all free pointers need to be updated, such that entries belonging to rectangles with sizes up to 7×5 point to R_2 and those with heights $1 \leq H \leq 2$ and widths $6 \leq W \leq 8$ point to R_3. All other entries point to NULL as displayed in Figure 51(b). After task completion, R_2 and R_3 are deleted and R_1 inserted again. At this point of time, the initial situation is re-established, see Figure 51(a).

4.5.2 Fitting Strategies for the Hashing Approach

If a newly arrived task fits into more than one free rectangle, a fitting strategy is used to choose a rectangle.

```
insert( Rectangle R, HashMatrix hm ) {
   int STOP := 0;
   insert R into the free rectangle list at entry hm[R.h,R.w];
   IF( free rectangle list at the entry was empty ) THEN
      FOR(i=R.h; i>0; i--) DO
         FOR(j=R.w; j>STOP; j--) DO
            IF( hm[i,j].free_pointer is smaller than rectangle R ) THEN
               STOP := j;
            ELSE
               change hm[i,j].free_pointer to point to R;
            END
         END
      END
   END
}
```

Fig. 52: Pseudo-code for the insertion with Best Fit (BF)

I) Best Fit (BF)
BF chooses the free rectangle with the smallest size that can accommodate the task. Intuitively, this strategy tries to keep 'big' rectangles, placing a task in the smallest possible rectangle.

- **Insertion into the hash-matrix**
  Using BF, only free pointers of smaller entries which do not already point to a smaller rectangle need to be updated. Figure 53(a) shows a hash-matrix containing two rectangles R_1 and R_2 of size 5×3, and 3×6, respectively. Note that the free pointers of entries with height ≤ 3 and width ≤ 3 point to rectangle R_1 because of its smaller size. Now, a rectangle of size 6×7 is inserted, see Figure 53(b). The free pointers already pointing to rectangles R_1 or R_2 do not need to be updated at all, leading to considerably less entries to be scanned.

  Figure 52 shows the pseudo-code for the insertion of a rectangle into the hash-matrix. The update of free pointers only takes place if another rectangle was not present in the list at the entry. The smaller entries are scanned row by row, and the column index at which the scan stops (variable `STOP`) is corrected if a smaller rectangle is found. In the case of best fit, the free pointer of an entry containing a free rectangle always points to this rectangle.
- **Deletion from the hash-matrix**
  Free pointers of smaller entries pointing to the deleted rectangle have to be redirected to point to bigger rectangles if possible. For different entries, different rectangles may be the next bigger ones. In a first step, these bigger rectangles have to be found. Once this is accomplished, the smaller entries are scanned as in the insert method.

  Figure 53(c) shows an example (initial situation after rectangles $R_{1..4}$ were placed). If rectangle R_2 is deleted, the free pointers pointing to it have to be

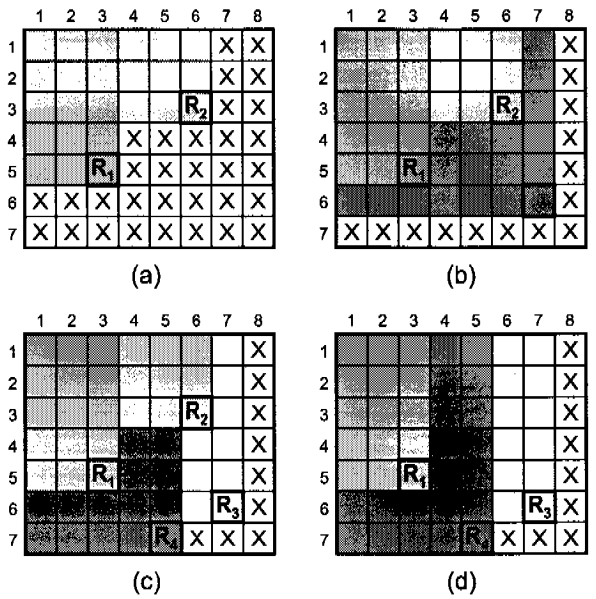


Fig. 53: Updating the free pointers using the Best Fit (BF) strategy.

redirected to point to the next smallest rectangle. For some entries, rectangle R_3 is the next smallest; for other entries it is R_4, as indicated in Figure 53(c).

II) Worst Fit (WF)

WF chooses the free rectangle with the biggest size that can accommodate the task.

- **Insertion into the hash matrix**
  Two cases can be differentiated:
  Case 1: A rectangle exists in the hash matrix which has bigger width and bigger height than the rectangle to be inserted. In this case, no free pointers have to be changed: the free pointer invariant states that all the free pointers of smaller entries already point to the bigger rectangle.
  Case 2: A bigger rectangle does not exist. In this case, the free pointers of all smaller entries pointing to a smaller rectangle have to be changed to point to the newly inserted rectangle.
- **Deletion from the hash-matrix**
  The delete method is similar to the delete in the case of best fit: free pointers pointing to the deleted rectangle have to be redirected to appropriate bigger rectangles.

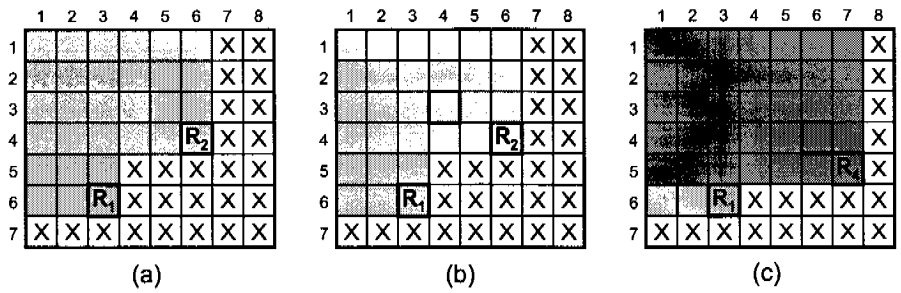


Fig. 54: Updating the free pointers using the Worst Fit (WF) strategy.

Figure 54 gives an example. 54(a) shows the initial situation. 54(b) illustrates *case 1* by inserting a rectangle R_3 of size 3×4 (which has no effect to the hash-matrix), and 54(c) illustrates *case 2* by inserting a rectangle R_4 of size 5×7. In this case, all free pointers previously pointed to R_2 now point to R_4, because R_4 is a worse fit than R_2.

III) Best Fit with Exact Fit (BFEF)

A potential disadvantage of best fit lies in the fact that after the insertion of a task into the smallest possible rectangle R, the remaining free space on R's area might be too small (e.g. too narrow) to accommodate new tasks, making this free space temporarily unusable. BFEF tries to tackle this disadvantage.

Among all rectangles which can accommodate the task, BFEF chooses the *smallest* rectangle which has exactly the same width or exactly the same height as the task. If no such rectangle is found, the task is placed according to BF.

Keeping the free pointers consistent works the same as for BF. Additionally, it is necessary to modify free pointers in the column and the row in which the free rectangle was inserted or deleted to make sure that 'exact fits' are taken into account.

IV) Worst Fit with Exact Fit (WFEF)

This fitting strategy is similar to BFEF. The difference is that it is tried to find a biggest free rectangle with the same width or height as the task.

Among all rectangles which can accommodate the task, WFEF chooses the *biggest* rectangle which has exactly the same width or exactly the same height as the task. If no such rectangle is found, the task is placed according to WF.

Keeping the free pointers consistent works the same as for WF. Additionally, it is necessary to modify free pointers in the column and the row in which the free rectangle was inserted or deleted to make sure that 'exact fits' are taken into account.

Figure 55 shows the result of applying the different fitting strategies to an exam-

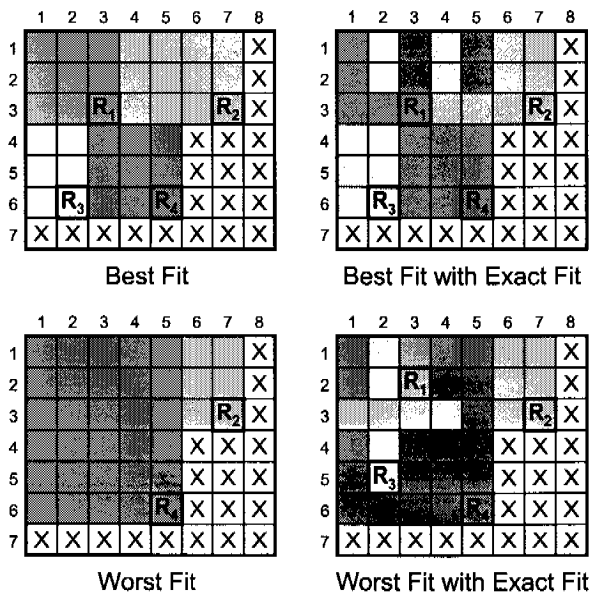


Fig. 55: Free pointers according to fitting strategies

ple with rectangles of sizes 3×3, 3×7, 6×2, and 6×5. Entries with the same gray value hold identical free pointers.

4.5.3 Complexity Estimations

Space Complexity

The size of the hash-matrix corresponds to the size of the FPGA $W_D \times H_D$. In case of a XILINX VIRTEX XCV-1000 holding an array of $96 \times 64 (= 6144)$ RLUs, and assuming that every entry has a size of 12 bytes, this yields a size of $6144 \cdot 12$ bytes $= 72kB$. The storage size of the free rectangles in the hash-matrix does not have to be incorporated into this calculation, since the same rectangles are managed in the partitioner modules of the placer (as described in the previous Section 4.4).

$72kB$ is still bearable. But if the size of the FPGAs keeps growing, the space overhead of the hash-matrix might become a critical issue. However, the hash-matrix approach is scalable: free rectangles could be managed at a coarser granularity, such that for example one unit of rectangle size corresponds to two RLUs.

Run-Time Complexity

A benefit of the hash-matrix approach is the time-efficiency of getting a placement for a newly arrived task T_n.

In the worst case, *all* of the 6144 hash-matrix entries need to be updated. However, this situation only occurs when a task T_i of size $W_i = W_D = 96$,

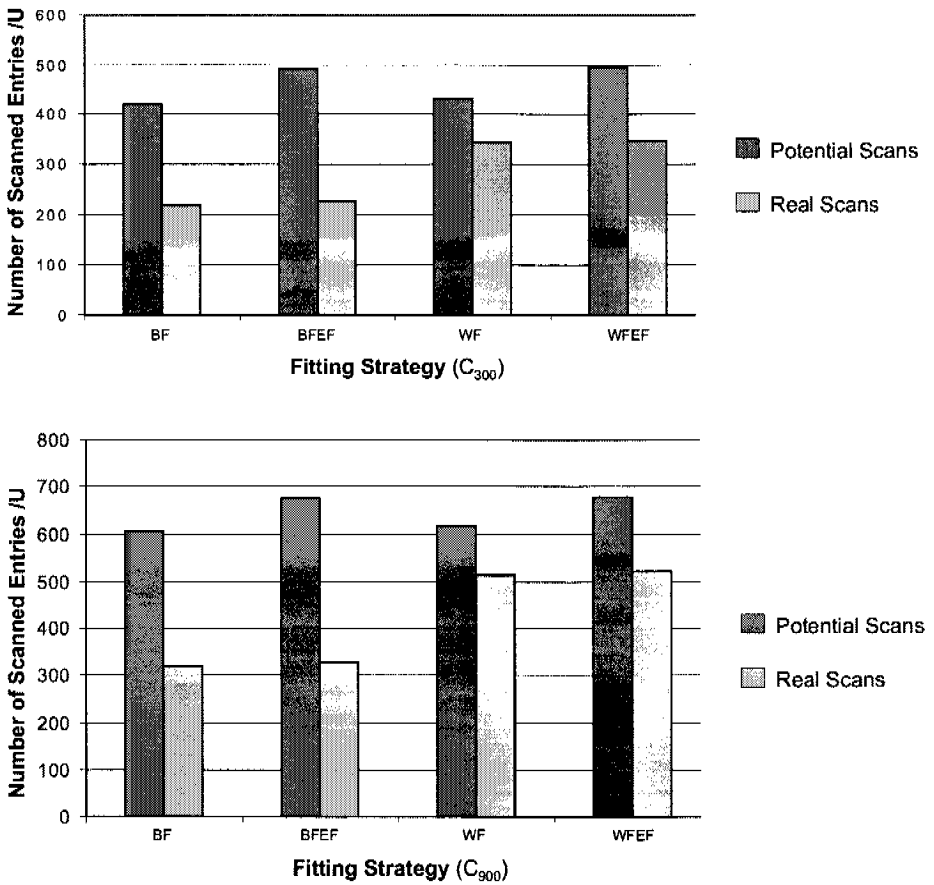


Fig. 56: Number of *potential* and *real* matrix entries scanned, depending on the fitting-strategy for task class C_{300}, and C_{900}, respectively.

$H_i = H_D = 64$ RLUs is inserted or deleted.

4.5.4 Evaluation

To evaluate the runtime complexity of the hash-matrix, we have investigated the overhead produced by updating the free pointers. We have applied the same simulation framework and settings as described in the previous Section 4.4.

Simulation Results and Discussion

We simulated the whole placer module (partitioner and hash-matrix) on a XILINX VIRTEX XCV-1000 like FPGA. Figure 56 displays the average number of matrix entries scanned per update operation (insertion or deletion of a task) for all fitting strategies (task classes C_{300}, and C_{900}, respectively). The figure also compares the number of *potential scans* with the number of *real scans*.

If a free rectangle of size $a \times b$ is inserted in or deleted from the hash-matrix,

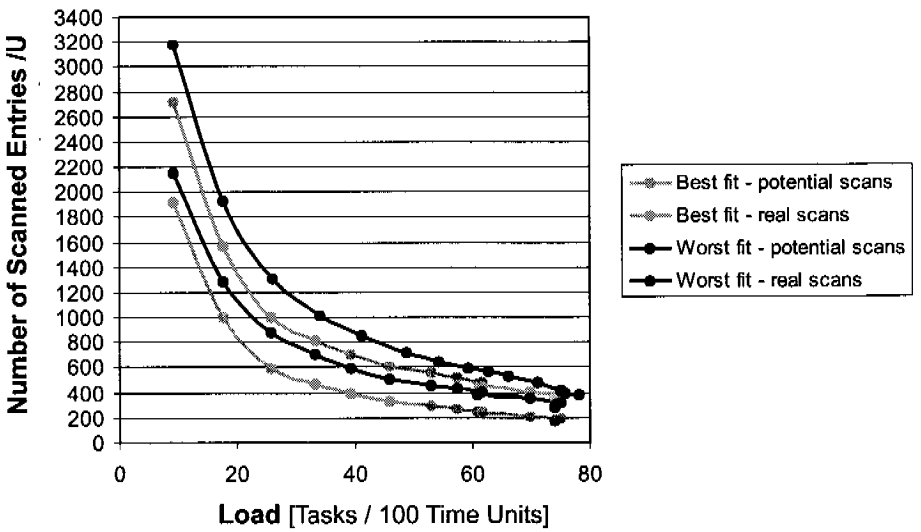
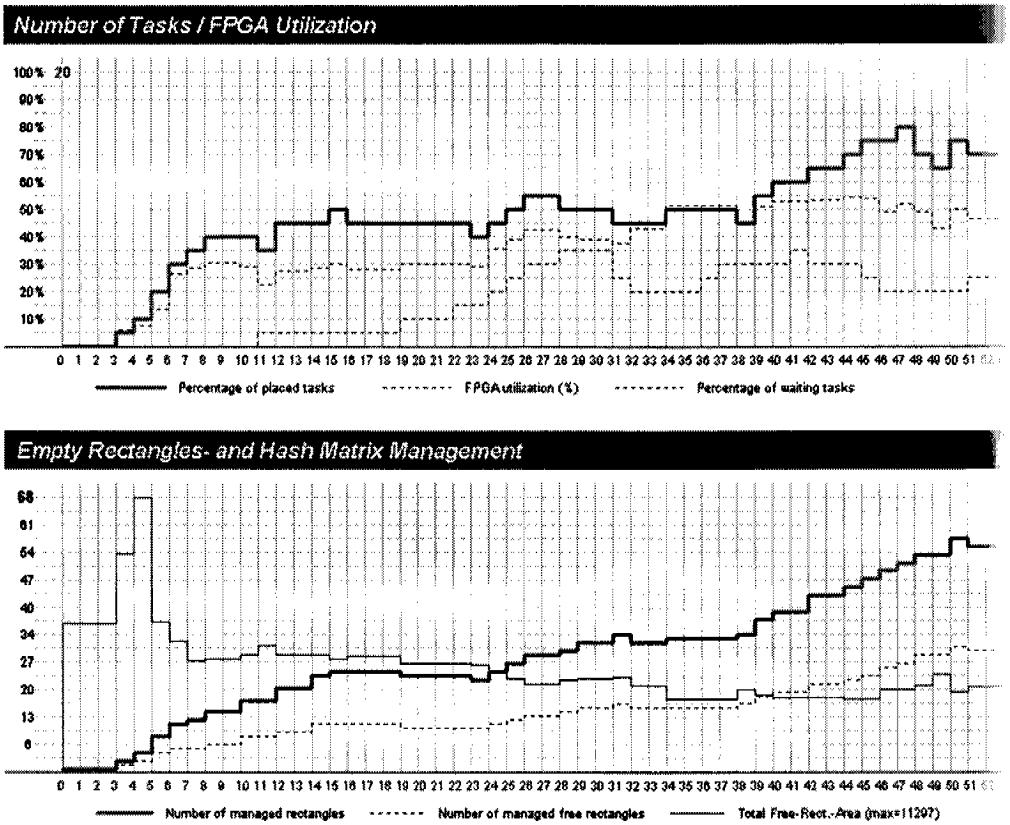


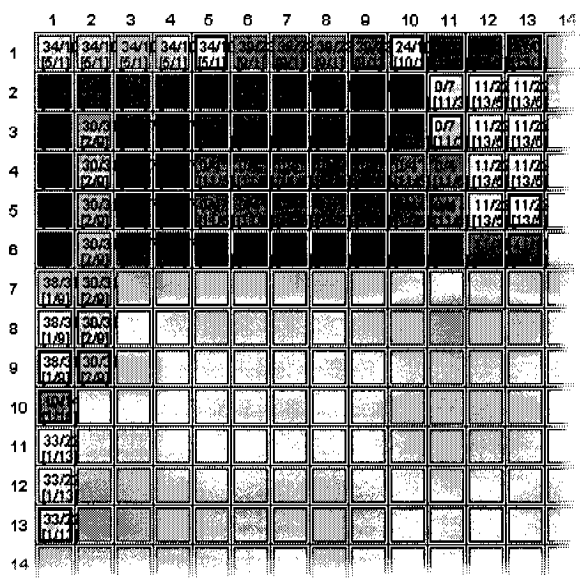
Fig. 57: Average number of entries scanned for different loads

the number of potential scans is $a \cdot b$. As displayed in Figure 56, even the potential scans are one order of magnitude smaller than the worst case. The fitters BF and BFEF are less expensive than WF and WFEF.

We can observe a remarkable trade-off between the load (number of tasks arriving per time unit) and the number of entries changed in the hash-matrix. The higher the load is, the more tasks are on the FPGA and the smaller is the number of matrix entries that have to be scanned. Figure 57 shows the average number of scanned entries per update for different loads, measured for task class C_{500}. The figure displays a significant decrease in the number of scanned entries for higher loads. Both, potential and real scans decrease the same amount. This means that for highly loaded systems where the partitioner is busy, the number of scanned entries decreases and thus, updating the hash-matrix is rather cheap. If the load is low on the other hand, the placer module has more time to conduct a more expensive update of the hash-matrix.



(a)



(b)

Fig. 58: Screen-shot of our *Task Placement and Scheduling Simulation System (TPS^3)* [TPS]. (a) Example-visualization of device utilization and number of managed rectangles over time, (b) Run-Time example of a hash-matrix.

4.6 Scheduling and Placement in Slotted Area Models

Problem Statement
A variety of policies and algorithms are used in RTOS for scheduling software tasks in on-line and/or real-time scenarios [Pin95, SSRC98, But00]. Compared to an RTOS, the scheduler in an RHWOS additionally has to manage *hardware tasks*. Since many characteristics of hardware tasks are fundamentally different to software tasks, the question arises whether the already known scheduling policies and algorithms can also be applied in an RHWOS. The three major differences that influence scheduling are the

- *hardware task placement and scheduling nexus*, i.e., there is no guarantee that the placer finds a feasible placement for every task that has been selected for execution
- *reconfiguration overhead* and the *serialization* of task management functions caused by the FPGA's configuration/readback port[5]
- *execution parallelism* of several hardware tasks that are running simultaneously on an FPGA. In an RTOS, software tasks are only executing in a *pseudo multitasking* manner.

The scheduler and placer in an RHWOS have to cope with these differences to ensure the efficient operation of the system. As discussed in Section 4.2.2, partitioning of the reconfigurable area into a number of fixed *reconfigurable slots* simplifies the placement and scheduling problem to some extent.

Contributions and Results
We have developed a number of new preemptive and non-preemptive on-line schedulers suitable for a 1D-slotted area model. The schedulers are based on known scheduling policies, such as *first come first serve (FCFS)*, *earliest deadline first (EDF)*, etc., but adapted to the special environment of an RHWOS. We explain the internal structure and functions of the different scheduler variants and evaluate their performance depending on the number and size of reconfigurable slots.

In order to investigate the influence of the reconfiguration overheads, we extended our simulation framework [TPS] to model the characteristics of the configuration/readback port, according to [XAPa, XAPb]. The modeling can be enabled or disabled for individual simulation runs. A Gantt-chart viewer allows to be visualized and verified the sequence of the scheduling process.

Our experiments show that (i) our novel on-line schedulers are feasible for operating in an RHWOS environment, and (ii) the overheads of the reconfiguration/readback port let increase the execution time of a task set between 1.2% and 7.3%.

[5] We consider the XILINX VIRTEX configuration architecture as described in [XVI, XV2a].

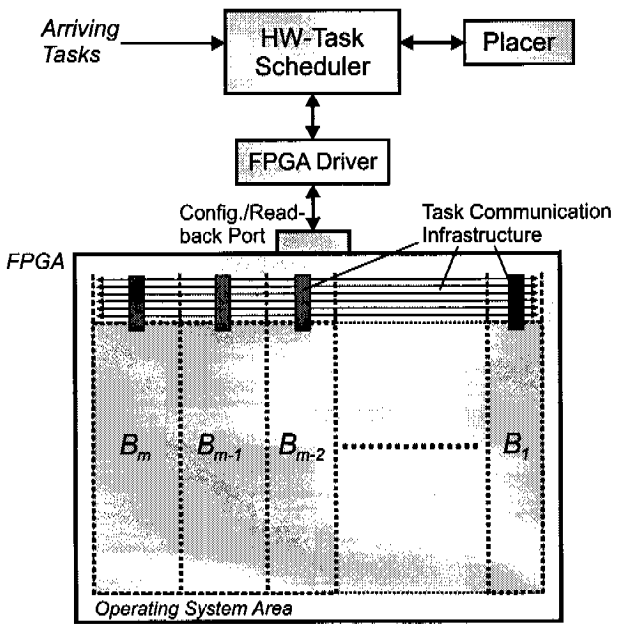


Fig. 59: Architecture model using the *1D-Slotted area model.*

4.6.1 Background

In this section, we consider hardware task scheduling in an on-line scenario in which the task properties (such as the task's widths W_x, arrival times a_x, execution times e_x, and deadline sd_x) are not known in advance. We assume that the task widths W_x are equally distributed in the interval $[W^{min}, W^{max}]$. Tasks are grouped into *task sets*, consisting of n unrelated tasks $T_{1..n}$.

We focus on the architectural model shown in Figure 59. According to our definition of the 1D-slotted area model (cf. Section 4.2.2), the reconfigurable device is split into a number of blocks B with fixed horizontal size, all having the same vertical dimension. We allow for different block widths, i.e., there are m blocks $B_{1..m}$ with $l \leq m$ different widths $W_1^B, \ldots, W_l^B$. We arrange the blocks such that their widths decrease monotonically from left to right, i.e., $W_j^B \geq W_i^B$ for $j > i$. Basically, the layout is static during the execution of task sets. Each block B_i can accommodate exactly one task T_x, if the task's width $W_x \leq W_i^B$.

The motivation of having different block sizes (as depicted in Figure 59) is to achieve a better match between the resources $B_{1..n}$ and the tasks $T_{1..n}$. Adapting block widths to task widths decreases block-internal fragmentation $\mathcal{F}_B(B, T_x)$ (as defined in Section 4.2.4) and leads to a higher average resource utilization. However, since we consider the layout as static, the adaption of the block widths can only be done during compile time.

An important question is *how* the block widths $W_{1..m}^B$ are assigned. We will discuss this issue in Section 4.6.3.

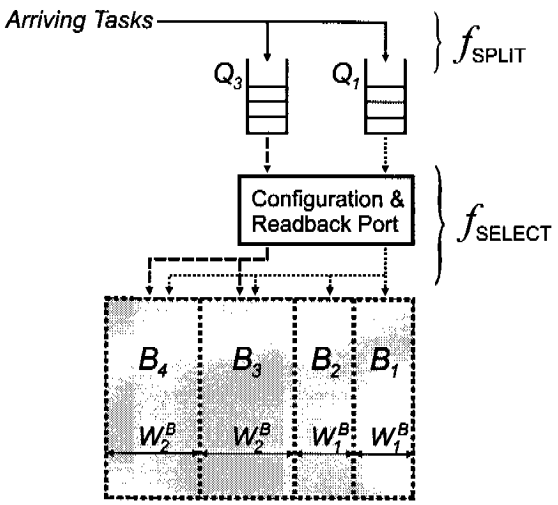


Fig. 60: Structure of the online scheduler for an exemplary partitioning, consisting of four blocks $B_{1..4}$. The width of block $B_{1..2} = W_1^B$, and $B_{3..4} = W_2^B$, respectively. Since there are two different block widths $W_{1..2}^B$, two queues Q_1 and Q_3 are available.

In the following section we present the structure of our on-line scheduler and several non-preemptive and preemptive scheduling methods tailored to RHWOS.

4.6.2 Scheduling Algorithms and Methods

All scheduling methods described in the following use the structure shown in Figure 60. The scheduler consists of a number of *queues* and the two functions f_{SPLIT} and f_{SELECT}. The number and positions of the queues depend on the device's block partitioning. A queue Q_j is created and assigned to block B_j if the next block to the right, B_i, has a smaller width, $W_j^B > W_i^B$. The right-most block B_1 always gets a queue Q_1 assigned.

Function f_{SPLIT} works in two steps. First, it assigns an arriving task T_x with width W_x in the right-most queue corresponding to a block wide enough to accommodate T_x. Second, f_{SPLIT} inserts the task into that queue according to some *sorting rule*. The sorting rule depends on the scheduling policy.

The function f_{SELECT} actually selects and places the task that is to be executed next. f_{SELECT} is invoked every time an executing tasks terminates, a configuration or readback process ends, or a new task arrives at the head of one of the queues Q. Among all queue heads, f_{SELECT} selects a task that can be allocated and configures it onto the smallest idle block able to accommodate the task. The selection is based on some *selection rule* which depends on the scheduling policy.

The *placer* function that is a part of f_{SELECT} determines the block in which tasks (only those at the queue heads) are placed. It can operate in two different

modes:

- *Restrict Mode*
  In the restrict mode, tasks in queue Q_i can only be placed into blocks that correspond to Q_i.
- *Prefer Mode*
  In the prefer mode, the placer can allocate a task to any block that is able to accommodate it. Consequently, tasks waiting in queue Q_j can be allocated to blocks $B_j, \ldots, B_m$, but not to blocks $B_1, \ldots, B_i$.

The example in Figure 60 indicates the prefer mode. Tasks from Q_3 can be placed in B_4 and B_3, whereas tasks in Q_1 can be placed in any block $B_{1..4}$.

The implementations of f_{SPLIT} and f_{SELECT} depend on the scheduling policy and are discussed in the next paragraphs.

Non-preemptive Methods
The non-preemptive schedulers neither preempt tasks running on the reconfigurable device nor the configuration process itself. Once f_{SELECT} selects a task, the task is loaded and run to termination. Figure 61 shows the resulting task state diagram. We have implemented the following non-preemptive schemes:

- *First Come First Serve (FCFS)*
  On single processors, FCFS executes tasks strictly in the order of their arrival. Tasks cannot be blocked from execution by later arriving tasks. FCFS is a very simple scheme and does not require knowledge about task execution times.

  f_{SPLIT}(FCFS) assigns a time-stamp to each arriving task and inserts it into the appropriate queue. The sorting rule is first-in first-out (FIFO). The selection rule of f_{SELECT}(FCFS) is to pick the task with the earliest arrival time-stamp.
- *Shortest Job First (SJF)*
  SJF is a scheduling policy that prefers shorter jobs over longer ones and minimizes the average response time in case all tasks arrive simultaneously on a single processor.

  f_{SPLIT}(SJF) sorts the queues according to the execution times of the tasks. In each queue, the head entry identifies the task with the smallest execution time. The selection rule for f_{SELECT}(SJF) is to pick the task with the smallest execution time, considering only the tasks at the queue heads.

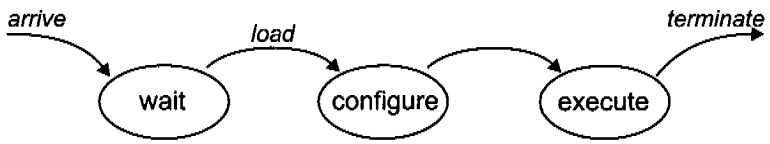


Fig. 61: Task states for non-preemptive scheduling.

Preemptive Methods

The preemptive schedulers preempt tasks running on the reconfigurable device to allocate a task with higher priority. Moreover, the configuration and readback processes can also be preempted (load abort and unload abort). Figure 62 shows the resulting task state diagram. Configuration processes are always aborted by higher-priority tasks. A readback process that unloads a block B_l is aborted only when the higher-priority task is to be loaded onto a block different from B_l. Otherwise, the readback is continued as B_l must be unloaded anyway. We have implemented the following preemptive schemes:

- *Shortest Remaining Processing Time (SRPT)*
  SRPT is scheduling scheme that executes at any time the task with the shortest remaining processing time. On single processor this scheme is known to minimize the average response time.

  $f_{\text{SPLIT}}(\text{SRPT})$ sorts the queues according to the remaining execution times of the tasks. The selection rule for $f_{\text{SELECT}}(\text{SRPT})$ is to pick the task with the smallest remaining execution time of all queue heads.
- *Earliest Deadline First (EDF)*
  EDF executes at any time the task with the earliest deadline.

  $f_{\text{SPLIT}}(\text{EDF})$ sorts the queues according to the task deadlines. In each queue, the head entry identifies the task with the earliest deadline. The selection rule of $f_{\text{SELECT}}(\text{EDF})$ picks the task with the earliest deadline of all queue heads.

A preempted task is treated like a newly arrived task; that is, function $f_{\text{SPLIT}}(\text{EDF})$ assigns the task to a queue Q (note that in case of FCFS, the previously assigned time-stamp is not changed). Then, the queue is sorted according to the respective rule.

When all blocks are of equal width, there is only one queue assigned to the right-most block B_1. In this case, FCFS, SJF, SRPT, and EDF behave exactly as their single processor counterparts. Differently-sized blocks pose an additional resource constraint that might break the known policy. For example, FCFS might schedule a later arrived smaller task before an earlier arrived bigger task.

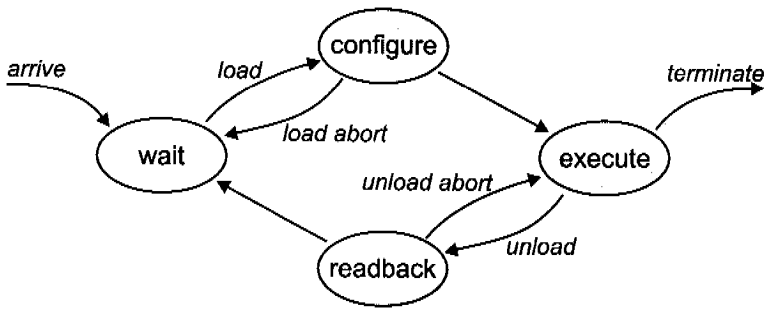


Fig. 62: Task states for preemptive scheduling.

Example FCFS and EDF Schedules

Figure 63 displays a screen-shot of the Gantt chart viewer [TPS]. The reconfigurable surface is partitioned into two blocks (B_1, B_2) of widths $(W_1^B, W_2^B) = (5, 10)$. The scheduler runs in FCFS mode; the placer in the prefer mode. The examples in Figure 63(a,b) detail the scheduling of the same sample task set with following four tasks $T_x(a_x, W_x, e_x, d_x)$:
$T_1(1, 5, 8, 26)$, $T_2(2, 5, 8, 24)$, $T_3(8, 5, 4, 23)$, $T_4(9, 8, 4, 20)$

The configuration and readback times for T_1, T_2 and T_3 are 2, for task T_4 3 time units.

Figure 63(a) displays the Gantt chart of an FCFS schedule (prefer mode):
At time 1, T_1 arrives, starts to configure onto B_1. At time 2, T_2 arrives. Since the configuration port is still busy with configuring T_1, T_2 has to wait. At time 2, the configuration process for T_1 is completed, and T_1 is now executing in B_1. The placer allocates T_2 to B_2 (prefer mode) and starts its configuration process. Both blocks B_1 and B_2 are now occupied. At time 8, T_3 arrives but cannot be configured because both blocks are still executing T_1, and T_2, respectively. At time 9, one more task, T_4, arrives but also cannot start. Both blocks $B_{1..2}$ are still busy and $T_{3..4}$ have to wait.

At time 11, T_1 terminates and frees B_1, which causes the scheduler to configure T_3 onto this block. At time 13, T_2 terminates and T_4 can be loaded. Both tasks $T_{3..4}$ now run to completion.

Figure 63(b) depicts the EDF schedule (prefer mode):
At time 1, T_1 arrives and starts to configure onto B_1. At time 2, T_2 arrives and gets a higher priority than T_1. The placer allocates T_2 to B_2 (prefer mode). The configuration of T_1 is preempted (configuration abort) and T_2 is configured onto B_2.

At time 4, the configuration/readback port is released by T_2. T_1 starts again configuration onto B_1. At time 8, T_3 arrives with a higher priority than T_1 and T_2. As both blocks are already in use, the scheduler selects T_1 to be preempted due

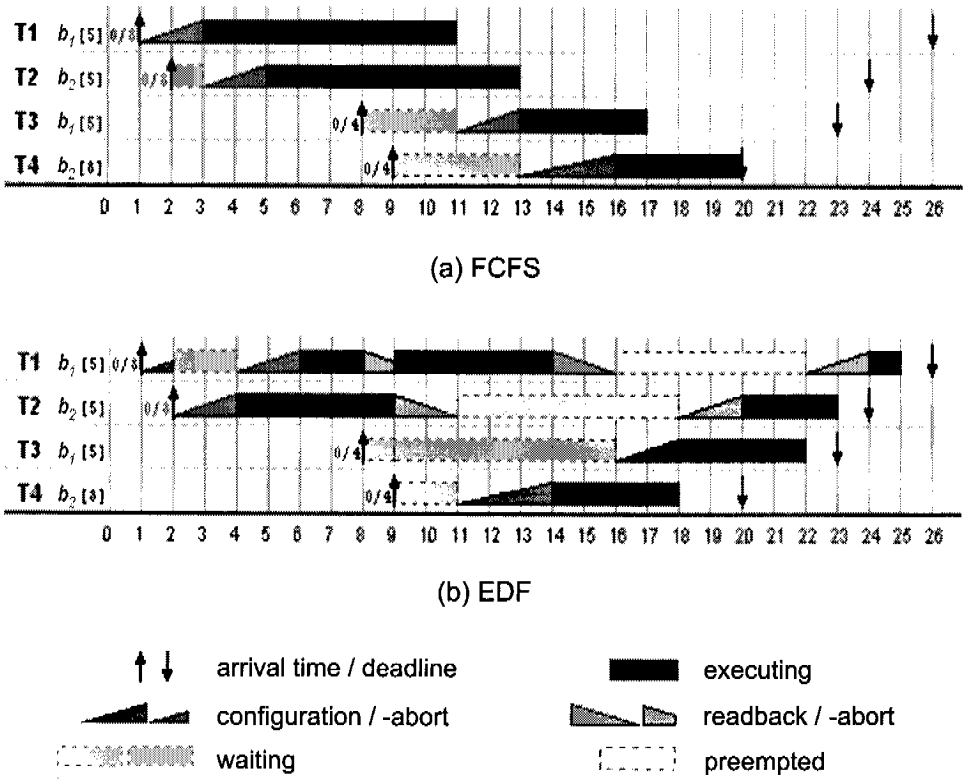


Fig. 63: Gantt chart of (a) *first come first serve (FCFS)* and (b) *earliest deadline first (EDF)* schedule.

to its long deadline. Readback of T_1 from B_1 starts. At time 9, T_4 arrives with the shortest deadline, and therefore the highest priority among all tasks. T_4 can only be placed in B_2. Consequently, T_2 must be preempted. The readback of T_1 which is currently in progress is stopped (unload abort) and the readback of T_2 is started instead, whereas T_1 resumes execution on B_1. At time 11, readback of T_2 has finished and T_4 is loaded onto B_1. At time 14, the configuration/readback port is free again, which causes the scheduler to select T_3 to be allocated to B_1. For this, T_1 has to be preempted and read back. At time 18, T_4 terminates. At this point in time, T_1 is still preempted, but since T_2 has a shorter deadline than T_1, T_2 is configured onto B_2. Only at time 22, T_3 terminates and frees block B_1. T_1 is loaded onto B_1 and runs to completion. In this example, all tasks $T_{1..4}$ have met their deadlines.

4.6.3 Finding Good Partitionings

The *partitioning* determines the number and widths of the reconfigurable blocks. Finding a partitioning that yields good scheduling results, measured for example, by the total execution time t_{tot} for a task set, is non non-trivial. In this section,

we will present a metric for determining the optimal partitioning P_{opt} for a given device width W_D and the interval $[W_{min}, W_{max}]$ of the task's widths.

As defined in Section 4.6.1, we assume that the widths of the arriving tasks are uniformly distributed in $[W_{min}, W_{max}]$. The device is partitioned into m blocks of l different widths $W_1^B, \ldots, W_l^B$, with a number of $w_1, \ldots, w_l$ blocks of each width. The placer is assumed to operate in the restrict mode.

The restrict mode groups the tasks into classes. A class is defined by an interval of widths, e.g., $(W_{i-1}^B, W_i^B]$. A task falling into this interval finds w_i blocks to execute on.

We define $\delta_i = W_i^B - W_{i-1}^B$ and $\Delta = W_{max} - W_{min} + 1$. The percentage of tasks scheduled to $(W_{i-1}^B, W_i^B]$ is δ_i/Δ. This expression can be taken as a measure for the execution time requirement (load) of the tasks with widths in $(W_{i-1}^B, W_i^B]$ for one block. Since there are w_i blocks available to accommodate tasks of this class, we can define a measure for the execution time requirement for this class as

ε_i : *Execution Time Requirement* of class i

$$\varepsilon_i = \frac{\delta_i}{\Delta \cdot w_i} \tag{4.31}$$

The task set's overall execution time is the maximum over all class execution requirements. We define

$\mathcal{E}(P)$: the *Collective Execution Time Requirement* which depends on the underlying partitioning P as

$$\mathcal{E}(P) = \max_{s=1\ldots l} \{\varepsilon_s\} \tag{4.32}$$

The scheduling goal is to minimize t_{tot} of a task set. Thus, a good partitioning P should select the parameters W_i^B and w_i in order to minimize $\mathcal{E}(P)$.

As an example, we consider a device with $W_D = 80$, and a task set with $W_{min} = 4$ and $W_{max} = 20$. The partitioning $P_a = [\, 3 \times 20,\ 2 \times 10\,]$ leads to $\mathcal{E}(P_a) = \max\left\{\frac{7}{17 \cdot 2}, \frac{10}{17 \cdot 3}\right\} = 0.206$. The partitioning $P_b = [\, 2 \times 20,\ 2 \times 15,\ 1 \times 10\,]$ leads to $\mathcal{E}(P_b) = \max\left\{\frac{7}{17 \cdot 1}, \frac{5}{17 \cdot 2}, \frac{5}{17 \cdot 2}\right\} = 0.41$. Partitioning P_a can thus be expected to produce better results for the described scenario.

The validity of such an analysis depends on the knowledge about the task set's distribution. We did not take into account the effects of different arrival times (queue loads), scheduler policies, and the configuration port that serializes task loading.

4.6.4 Simulation and Evaluation

We have implemented a simulation framework to experimentally investigate the behavior and the performance of all the on-line schedulers described above. The parameters of the simulator include the dimension of the reconfigurable device W_D, the partitioning P, and the configuration and readback times for one device column[6]. In the current state, the simulation neglects CPU run-times required

- for the bitstream manipulations to relocate tasks to different blocks,
- for the task context extraction and -insertion, and
- to execute the on-line schedulers.

The simulation framework comprises the simulator module, a task generator, a module for data collection and statistical analysis including a Gantt chart viewer, and a graphical display of the allocation situation and queue loads [TPS].

We present two experiments: (i) we analyze the influence of the partitioning on the performance of the FCFS scheduler for both the restrict and the prefer mode. The scheduler performance is measured by the total execution time for a task set; (ii) we determine the influence of the configuration and readback overheads on the total execution time t_{tot} of a task set.

Simulation Settings

The simulation models a XILINX VIRTEX XCV-1000 where the configuration or readback of one column takes $159\mu s$. We assume that only 80 columns (out of 96 columns available) are usable for blocks; the remaining columns are occupied by the operating system. Since there are no benchmarks or statistical data available from real-world applications so far, we have to resort to randomly generated tasks. We have generated task sets with 100 tasks each. The tasks widths are uniformly distributed in $[4, 20]$ columns, the execution times in $[2, 200]ms$, and the arrival times in $[0.5, 500]ms$. The resolution of the simulator, the duration of one clock tick, has been set to $500\mu s$.

Simulation Results and Discussion

I) Influence of Partitioning to an FCFS-Scheduler

We have selected six different partitionings $P_{1..6}$ for conducting the simulations. Figure 64(left) lists these partitionings, and Table 11 indicates the collective execution time requirement $\mathcal{E}(P)$ per partitioning. P_1 and P_6 are *extremal partitions*. P_1 contains $w_1 = 4$ blocks of the maximal width $W_1^B = W_{max}^B = 20$. This partitioning prefers large task sizes but involves high block internal fragmentation

[6] According to the reconfiguration model of XILINX VIRTEX devices.

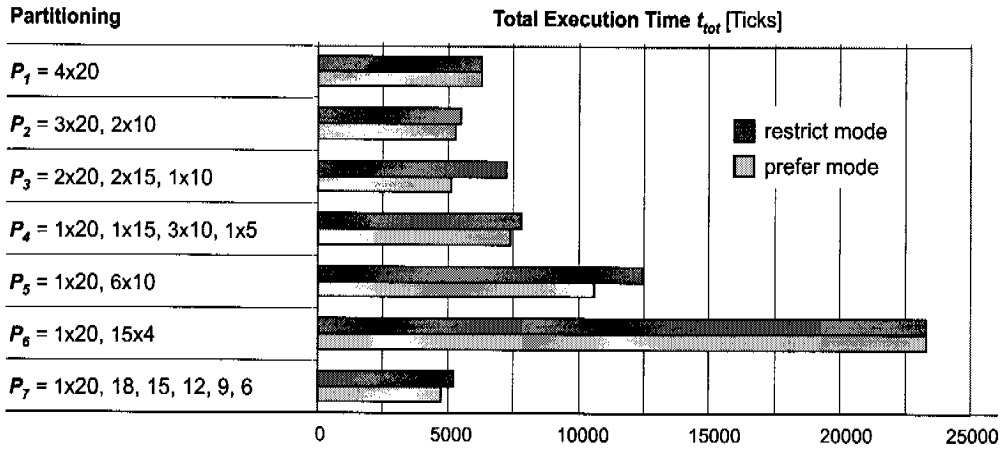


Fig. 64: Total execution time t_{tot} of randomly generated task-sets for different partitionings $P_{1..7}$ using FCFS in the restrict, and prefer mode, respectively.

for small tasks. Therefore, P_1 is expected not to show a very high performance ($\mathcal{E}(P_1)$ yields to 0.25). The partitioning P_6 is a pathologic case because the 15 blocks with width = 4 are quite ineffective. In this case, 94.1% of the total load is scheduled to the one large block (with width = 20). Consequently, the total execution time is expected to increase dramatically (note that $\mathcal{E}(P_6) = 0.9412$).

Partitioning	$\mathcal{E}(P_{1..7})$
P_1	0.2500
P_2	0.2059
P_3	0.4118
P_4	0.2941
P_5	0.5882
P_6	0.9412
P_7	0.1765

Tab. 11: Selected partitionings $P_{1..7}$ for simulation.

Figure 64(right) shows the results for both the restrict and prefer modes for seven different partitionings $P_{1..7}$. From the arbitrarily selected partitionings $P_{1..6}$, P_2 performs best in the restrict mode, whereas P_3 shows the best performance in prefer mode. In the restrict mode, P_3 clearly shows a bottleneck for small tasks; the prefer mode achieves a 29.4% better result, as the smaller tasks can switch to the larger blocks.

For both partitionings P_1 and P_6, the restrict and prefer mode have no influence on t_{tot}. In case of P_1, the prefer mode operates equal to the restrict mode,

because there is only *one* block width (i.e., $l = 1, w_1 = 4, W_1^B = 20$). In P_6, the 15 blocks of width $= 4$ are unallocated most of the time, since there are only a few tasks of width $= 4$. None of the tasks can take benefit from the prefer mode, i.e., switch to a larger block. Therefore, both modes lead to the same t_{tot}.

Generally, the prefer mode sometimes allocates smaller tasks to larger blocks. This can turn out to be useful when the task set contains alternating bursts of smaller tasks and larger tasks. A good choice for such a case is a partitioning that contains many large blocks and few smaller ones together with a placer in prefer mode. Bursts of large tasks will apparently match the resources; bursts of smaller tasks are supported by the prefer mode.

In order to derive the optimal partitioning P_{opt} (in the sense of minimizing $\mathcal{E}$), we have enumerated *all* partitionings for $W_D = 80$ and $W_{min}^B = 4$, $W_{max}^B = 20$. We only considered partitionings for which $\sum_m W_m^B \equiv W_D$, since partitionings with $\sum_m W_m^B < W_D$ are worse than P_{opt}. We determined 7 (out of a total of 7652) partitionings which minimize $\mathcal{E} = 0.17647$, as listed in Table 12.

Optimal Partitionings
$P_{opt1} = [\,2 \times 20,\ 1 \times 14,\ 1 \times 11,\ 1 \times 9,\ 1 \times 6\,]$
$P_{opt2} = [\,1 \times 20,\ 1 \times 18,\ 1 \times 15,\ 1 \times 12,\ 1 \times 9,\ \times 6\,]$
$P_{opt3} = [\,1 \times 20,\ 2 \times 17,\ 1 \times 11,\ 1 \times 9,\ 1 \times 6\,]$
$P_{opt4} = [\,1 \times 20,\ 1 \times 17,\ 2 \times 14,\ 1 \times 9,\ 2 \times 6\,]$
$P_{opt5} = [\,1 \times 20,\ 1 \times 17,\ 1 \times 14,\ 1 \times 11,\ 2 \times 9\,]$
$P_{opt6} = [\,1 \times 20,\ 1 \times 17,\ 1 \times 14,\ 1 \times 11,\ 1 \times 8,\ 1 \times 6,\ 1 \times 4\,]$
$P_{opt7} = [\,1 \times 20,\ 1 \times 17,\ 1 \times 14,\ 1 \times 11,\ 1 \times 8,\ 2 \times 5\,]$

Tab. 12: Optimal partitionings $P_{opt1..7}$, according to the collective execution time requirement metric $\mathcal{E}$. All above listed partitionings yield a $\mathcal{E}(P) = 0.176471$.

Partitioning P_7 of our simulation experiment (see Figure 64) equals to P_{opt2}. P_7 outperforms all other partitionings, which confirms our analytical approach to determine optimal partitionings.

II) Influence of Configuration and Readback Overheads on t_{tot}

This section discusses an experiment to analyze the effects of configuration and readback on the system performance. For this, we have simulated the SRPT scheduler in the restrict placement mode with and without modeling the configuration port. The scheduler performance is measured by the total execution time t_{tot} for a task set. We have generated task sets with 25 tasks each. The arrival times are distributed in $[0.1, 10]ms$ to eliminate the influence of late arriving tasks on the total execution time. All other parameters are identical to the previous section, which results in configuration and readback times of $[0.6, 3.1]ms$.

The results show 1.2% to 7.3% increase in the total execution time when configuration and readback times are included.

We believe that in the targeted application domains, such as wearable computing [PEW+02], typical task execution times will be higher than the times assumed in this experiment. Consequently, the presented partially reconfigurable system will not suffer from a bottleneck formed by the single configuration port.

5

RHWOS Prototype

Overview

In the previous chapters, we described the functions of an RHWOS on a conceptual and algorithmic level, disregarding detailed technology related issues. In contrast, this chapter is devoted to the analysis of problems arising when it comes to a practical realization of an RHWOS.

Problem Statement

We aim at realizing a prototypical implementation of the main parts of an RHWOS in order to (i) prove the feasibility of an RHWOS, based on current FPGA technology, (ii) verify the practicability of the conceptual claims (stated in the previous chapters) by means of a functioning run-time reconfiguring case study application, and (iii) experimentally measure and evaluate run-time system parameters.

In the first instance, we concentrate on getting a *running complete system*, instead of optimizing performance issues of single functions.

Contributions and Results

We have chosen XILINX VIRTEX-II FPGA devices for our prototype. These devices offer column-wise fast partial reconfiguration and readback capabilities, which ideally match the 1D area model. First, we review the requirements of both the reconfigurable device and the specific architecture of a platform running an RHWOS. We present the XF-BOARD, our self-developed prototyping board tailored to the requirements of an RHWOS. Based on this platform, we implemented our *run-time environment* featuring a shared bus for high-speed inter-task communication and the novel *granular 1D-variable area model*, which allows

for run-time placing of hardware tasks of different widths. This characteristic is the novelty which distinguishes our work from the current state in the research community.

As final result, we present a running case study application controlled by our RHWOS prototype. This demonstrator executes an application from the audio domain, which generates waveforms under real-time constraints by performing more than 10 partial run-time reconfigurations per second.

5.1 Background and Related Work

In contrast to conceptual and algorithmic RHWOS issues, there are only a few reports on practical implementations executing run-time reconfiguring applications.

Dyer et al. [DPP02] presented a prototype which allows for dynamically configuring *coprocessor cores* into a static environment. The environment includes a *LEON* soft CPU core [Gai, LEOa] and special fixed wires to connect to the coprocessor. Since the design tools did not yet allow for defining routing constraints, they used so-called *feed-though macros* to get rid of *disturbing lines* interfering with the coprocessors's circuit. As a demonstrator, they implemented an networked audio streaming application which dynamically loads different audio decoders as coprocessor core. However, this environment only provides *one* reconfiguration slot.

In [NCV+03, MNC+03], the reconfigurable research group at *Imec* [IME] presents the *T-ReCS Gecko* [Gec] a reconfigurable multimedia demonstrator, which supports partial run-time reconfiguration.

In our previous work [WP03b] we have demonstrated a first prototype using a 1D-slotted area model. The prototype was done on the commercial XESS XSV-800 rapid prototyping board [XES, XSV]. However, the suitability of this board to implement an RHWOS is limited.

5.2 RHWOS Platform Design Requirements

An RHWOS makes certain demands on both the reconfigurable device to be used as a dynamically allocatable logic resource, and in the way this device is connected to its environment, e.g. the CPU. In the following sections, we review these requirements.

5.2.1 Reconfigurable Device Requirements

A reconfigurable device needs to fulfill the following criteria, in order to be suitable to be employed in an RHWOS platform.

- *Reconfiguration/readback capabilities*
  The device must support partial (re-)configuration to make loading of independent hardware tasks during run-time possible. Full or partial readback is required, if the RHWOS should be able to preempt tasks. A high-bandwidth of the (re-)configuration/readback port is needed to keep the reconfiguration times low, which is essential for a high dynamic device utilization.
- *RLU-array size*
  The device must offer an RLU-array large enough to accommodate a number of coarse-grained hardware tasks (as listed in Table 10) and RHWOS elements at the same time.
- *Several independent clock nets*
  Since several hardware tasks that are concurrently executing on the device may have different clock requirements, the device should offer multiple independent clock nets which can be individually driven or controlled from the device's outside.
- *Design tool capabilities*
  Design tools must support the creation of partial bitstreams and shall allow for applying area and routing constraints to the design. For run-time manipulation of the bitstream, the internal structure need to be well-described and disclosed by the device supplier.

We have chosen XILINX VIRTEX-II FPGAs [XV2a]. This is the only commercially available device family, which offers a sufficiently large RLU array and supports fast partial reconfiguration and readback[1].

XILINX VIRTEX FPGAs can be partially configured in vertical chip-spanning columns. The configuration port, the so-called *SelectMap*[2] port, supports 8-bit parallel bidirectional data transfers at a maximal synchronous data rate of 400Mbit/s. VIRTEX-II devices are available with RLU-array sizes of up to 112×104 (=11'648 RLUs), which is sufficient to accommodate a number of large hardware tasks in parallel. A total of 16 clock input pins is provided, which can drive up to eight clock nets per device quadrant. Moreover, XILINX's design tools allow for generation and modification of partial bitstreams [XIF, XMD, JBi]. Based on XILINX's specification of the VIRTEX-bitstream-structure, third party tools are available for bitstream manipulation, such as the *PARBIT* tool developed by [HLK02].

[1] This holds for the point in time of writing this thesis in August 2004

[2] Naming defined by XILINX for all VIRTEX, VIRTEX-II, and VIRTEX-II-PRO devices.

Devices from other FPGA suppliers, such as ALTERA [Alt], overweight aspects like design security (bitstream encryption) rather than (re-)configuration speed, or do not support partial reconfiguration [Str, Cyc]. LATTICE's ORCA 4 FPGA devices [ORC] support partial reconfiguration but are not available in large RLU-array sizes.

An interesting alternative would have been XILINX's VIRTEX-II PRO [XV2b]. This device family features a IBM POWERPC embedded CPU core [PPC] coupled with a reconfigurable array as a *Reconfigurable System on a Chip (RSoC)*. However, at the time of this writing, this device family was in early commercialization stage and only available in small RLUs array sizes.

5.2.2 Platform Architecture Requirements

As defined in Section 1.1, we consider an embedded system architecture mainly consisting of a CPU, an FPGA, and a number of external I/O devices (cf. Figure 1 on page 3).

The implementation of an RHWOS imposes a number of specific requirements to the underlying hardware platform architecture. The FPGA-internal architectural properties and its reconfiguration capabilities additionally govern the design of the platform.

The requirements for an RHWOS platform based on XILINX VIRTEX-II can be defined as follows:

- *Fast bidirectional access to reconfiguration and readback port*
  To allow for maximum-speed full and partial (re-)configuration and readback, a fast bidirectional access to the FPGA's configuration port is needed. The system's memory should be fast enough to allow for high-speed reconfiguration/readback.
- *Fast CPU/FPGA inter-communication*
  A sufficiently high number of general purpose wires are required running directly between the CPU and the FPGA. The resulting high communication bandwidth enables efficient partitioning of operating system functions and user tasks between software and hardware.
- *Sophisticated clock generation and control*
  The user tasks residing in the FPGA might run at different clock speeds, depending on the circuit they implement. Therefore, several parallel clock signals should be provided to feed into the FPGA. Full control over the clock signals enables efficient debugging functions, such as halting or single-stepping of tasks or RHWOS elements.
- *External device connection*
  The most important characteristic of XILINX VIRTEX-II is its column-wise chip-spanning partial reconfiguration characteristic. Consequently, FPGA

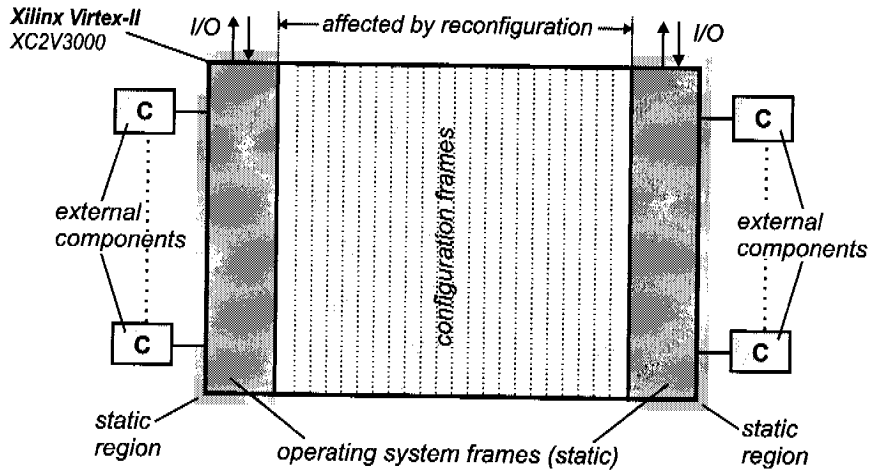


Fig. 65: Basic verticalized FPGA Layout: Partitioning of the reconfigurable area into vertical oriented *static* and *dynamic* regions. Location of external components connection.

internal I/O blocks at the upper and lower edges are potentially affected by reconfiguration processes, whereas those at the left and right edges are not. From this, it follows that external devices must be connected exclusively to the left and right FPGA edges in order to ensure proper function.

None of the commercially available FPGA-boards known to us, complies with these special requirements, e.g. [XSV, Tre, B5X, Sun, PP03] etc. Due to this fact, we decided to develop our own RHWOS platform (see Section 5.4).

5.3 Implementation of the Runtime Environment

A runtime environment is a set of circuits and structures implemented in an FPGA that allows for executing dynamic hardware tasks using partial reconfiguration. There is a strong nexus between the reconfiguration properties of the FPGA, the area model operated by the RHWOS, and the structure of the runtime environment.

The column-wise chip-spanning reconfiguration model of XILINX VIRTEX-II devices, ask for a 1D area model and imposes a verticalized structure on the runtime environment.

In Section 3.5, we proposed to divide the reconfigurable area into static and dynamic regions, in which the static region accommodates RHWOS elements and the dynamic region is dedicated to be dynamically used by user hardware tasks. In respect to the predefined locations of the external device connection, we devise a basic layout of the runtime environment as depicted in Figure 65.

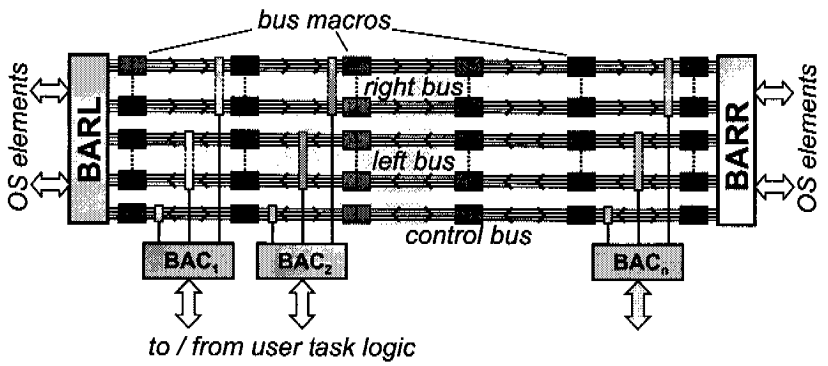


Fig. 66: Task Communication Bus (TCB) consisting of *bus arbiter left / right (BARL/BARR)*, n *bus access controllers* ($BAC_{1..n}$), and *right-*, *left-*, and *control bus*; *Bus macros* serving as position-invariant reconfiguration boundary.

We denote the static regions, which contain the RHWOS modules, as *OS-frames*. The OS-frames remain unchanged during whole system operation, but nevertheless, it can still be adapted to fit the needs of different application domains on a longer time scale.

In the following, we emphasize the communication infrastructure and a special partitioning of the reconfigurable surface, which we denote as *granular 1D-variable area model.*

5.3.1 Task Communication Bus

An RHWOS runtime environment has to implement a communication infrastructure for the exchange of data between user hardware tasks and OS objects. There are several ways to implement such a communication infrastructure, with trade-offs between the achieved data throughput and the needed resources. Assigning each task a set of dedicated wires and turning the communication infrastructure into a crossbar would result in the highest-possible throughput. At the same time, a crossbar also leads to enormous area consumption and is not scalable.

We rely on a diametrical solution and implement a shared bus structure as communication network. A bus requires less resources but, on the other hand, delivers a lower communication bandwidth. We use two methods to scale the throughput, varying the bus width and using several independent busses. However, the bus approach scales poorly with the FPGA size. Future devices with strongly increased densities will allow to accommodate dozens of user tasks to be accommodated at the same time. Then, the bus would become a severe bottleneck. For such architectures, a two-dimensional partitioning into slots and a 2D communication infrastructure will be better choices [MNC+03, BMK+04]. We have decided for a bus structure in this work, as currently neither a 2D parti-

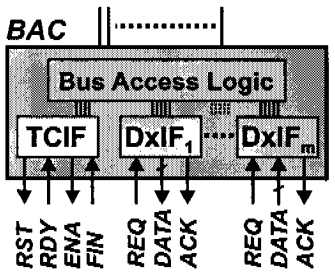


Fig. 67: Bus Access Controller (BAC) providing a *task control interface (TCIF)* and m *data transfer interfaces* ($DxIF_{1..m}$).

tioning nor a 2D communication network is supported by the available FPGAs.

Figure 66 shows the bus structure and its elements. The bus system consists of *bus wires*, *bus macros*, two *bus arbiters (BARL,BARR)* located in the right and left OS frames, and n *bus access controllers* ($BAC_{1..n}$), one in each user task. The bus is split into a left bus, a right bus, and a control bus. The left bus serves read/write accesses to OS elements located in the left OS frame and the right bus accesses OS elements in the right frame, respectively. The left bus is arbitrated by the BARL, and the right bus by the BARR. The control bus comprises request/grant signals connecting each BAC to both arbiters. BARx run a round-robin protocol to grant the bus.

When the RHWOS configures a user task into a task slot, the read/write ports of the task and the OS elements it connects to are known. Depending on the current mapping of OS elements to the left and right OS frames, the *bus access controllers (BAC)* of the user tasks are parameterized such that bus requests can be sent to the responsible bus arbiter. The bus access controller provides an interface that consists of the bus access logic and, at the user tasks side, of a *task control interface (TCIF)* and m *data exchange interfaces* ($DxIF_{1..m}$). The *bus access logic* handles the bus reservation and implements the data transaction protocol. Thus, the actual bus protocol is hidden from the user task.

The $DxIF_{1..m}$ interfaces allow user tasks to send and receive data. Depending on the characteristics of the storage services offered by the RHWOS, the DxIFs can show different interface implementations. For this prototype, we concentrate on streaming-oriented applications and have thus implemented interface logic for FIFOs.

5.3.2 Granular 1D-Variable Area Model

We present an implementation of the *granular 1D-variable area model*. Compared to the *1D-variable area model*, it restricts placement of tasks to a granular grid, but other than in a *slotted area model* there are no fixed blocks defined. In the granular 1D-variable area model, tasks can have several widths in the gra-

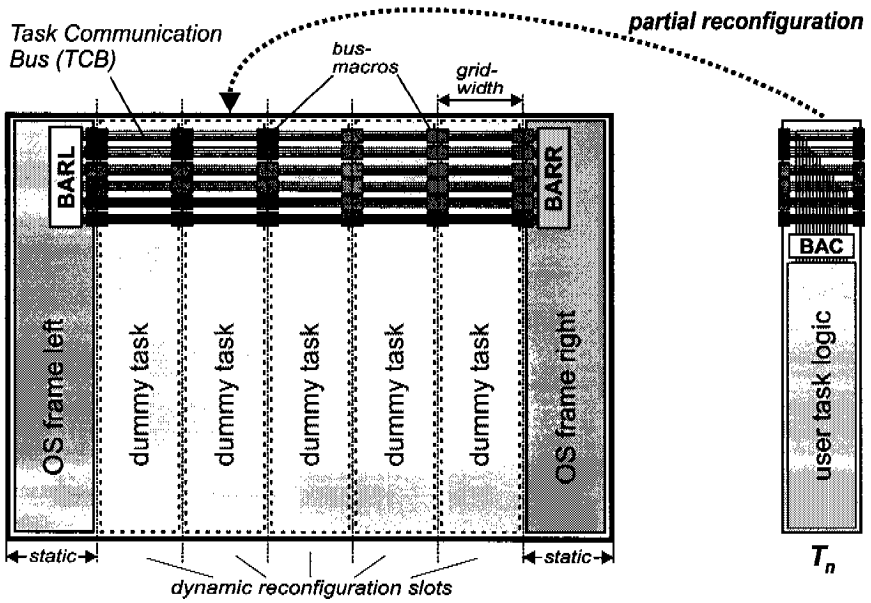


Fig. 68: Runtime Environment (initial state) preset with 5 dummy tasks. User task T_n of width $w = 1$ to be partially configured into reconfiguration slot 2.

dation of the grid-with. The grid width is a multiple of one RLU width. This concept eliminates *block internal fragmentation* and leads to better utilization of the device.

Figure 68 visualizes the overall structure of our runtime environment implementing two OS-frames (left / right) and a total of five dynamic reconfiguration slots.

I) Initial Situation

Whenever the system is powered up or the RHWOS is adapted, the FPGA undergoes a full configuration. This initial configuration contains the OS frames with the RHWOS elements and the dynamic area organized into a number of dummy tasks, as illustrated in Figure 68 left hand side. Dummy tasks are place-holders for user tasks. They do not implement any functionality but establish the task communication bus of the runtime environment. Each dummy task implements a part of the overall communication infrastructure consisting of bus macros and bus wires. The width of a dummy task defines a static grid of reconfigurable slots on the reconfigurable area.

Bus macros play an important role regarding the partial reconfiguration process. They are position invariant and located at the boundary of two reconfigurable slots, or between a slot and an OS-frame, respectively. Each bus macro implements four wires that connect to the adjacent slot or OS-frame. While one half of a bus macro is involved in a reconfiguration process, the other half re-

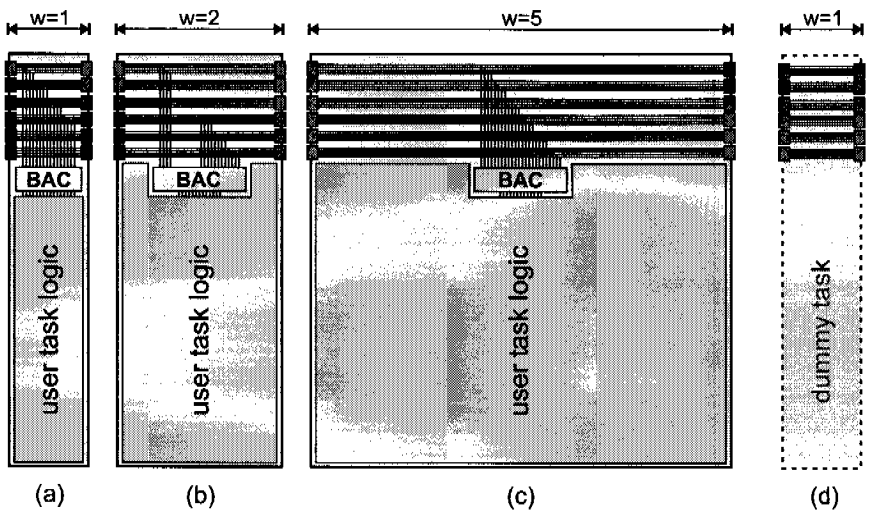


Fig. 69: User Tasks with different widths (a) $w = 1$, (b) $w = 2$, and (c) $w = 5$; (d) shows a dummy task ($w = 1$).

mains unaffected. Bus macros are anchor points in the dynamically reconfigured area and join the isolated parts (OS-frames and reconfiguration slots) of the run-time environment together.

II) Dynamic Hardware Task Insertion

When a user task is inserted into the static grid, it occupies an integer multiple of the grid width, i.e., the width of one dummy task. Figure 69 shows user tasks with different widths w, i.e., with $1\times$, $2\times$, and $5\times$ the dummy task's width. Tasks with widths $w \geq 2$ do not necessarily need to implement the bus macros within their area. Implementing the bus macros at the user task borders suffices. After a task with width $w \geq 2$ has terminated, the occupied area is again filled with dummy tasks to re-establish the communication infrastructure for subsequent user tasks. Some potential for optimizations exists at this point: For example, if a wide task T_A is followed by two smaller ones T_B and T_C that reside in the same area, the RHWOS does not need to insert dummy tasks first but can directly load T_B and T_C, as long as the communication infrastructure remains intact. In the long run, the overhead for inserting dummy tasks could also be used as a parameter that drives task scheduling and placement. Dummy tasks as well as user tasks are available in the raw task repository as partial bitstreams.

The widths of the OS-frame and the grid can be adapted to the area requirements of the OS elements, and the diversity of the task complexities. An obvious approach concerning gird width would be to choose it as small as possible, e.g.

one CLB[3] width, in order to achieve high placement flexibility and, thus, a higher utilization. However, this leads to some effects which need to be considered: (i) the minimal width of a bus macro amounts to four CLBs. If bus macros are placed too close together, the routing between bus wires and BAC can become un-realizable due to the lack of routing resources; (ii) smaller grid width leads to a higher task management load for the RHWOS; e.g. if a large task T_A finishes execution, the RHWOS needs to fill the freed area with a high number of dummy tasks to re-establish the task communication bus. Configuring one larger dummy task is less time consuming than loading several smaller ones.

Consequently, the dimensioning of the runtime environment layout includes a trade-off between device utilization and runtime efficiency.

III) Bitstream Generation
XILINX's design tools [XIF] allow for generating both full and partial bitstreams for the VIRTEX-II family. We used the MODULARDESIGN flow which additionally supports a module (=task) oriented design, and the use of *location-, area- and routing constraints* [XMD, XAPc]. These constraints are essential for establishing the grid based structure of our runtime environment.

5.4 The XF-Board

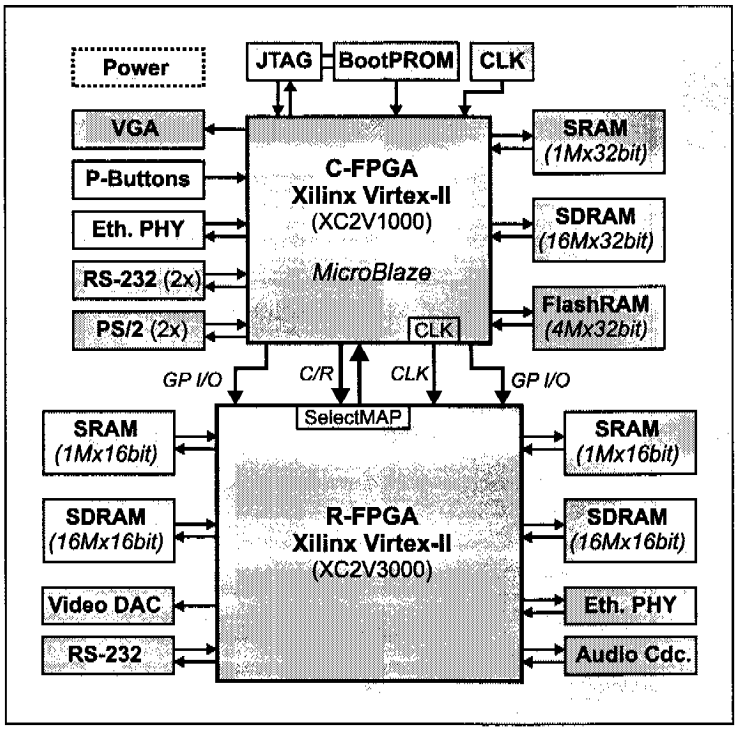
We present the XF-BOARD, our self-developed RHWOS platform, which matches well all requirements for an RHWOS platform as we have specified in Section 5.2.2.

The board features two tightly coupled FPGAs. One FPGA, the *C-FPGA*, implements a soft CPU core that controls the overall system; the second FPGA, the *R-FPGA*, is used as a dynamically allocatable reconfigurable resource. The main features of the platform are fast partial reconfiguration and readback, advanced clock control, a multitude of memory and I/O devices, and support for the 1D area model by connecting all memory and I/O devices to the left and right device edges of the R-FPGA. Figure 70 displays the block schematics and a photograph of the XF-BOARD.

5.4.1 C-FPGA (CPU Equivalent) / XILINX XC2V-1000

Instead of using a dedicated standard processor (e.g. ARM, PIC, MCORE), we decided to use a soft core for the implementation of the system's CPU. Although inferior in performance, a soft CPU offers a much higher flexibility than a dedicated CPU. We selected the 32-bit RISC processor core XILINX MICROBLAZE [XMB].

[3]CLB = *C*onfigurable *L*ogic *B*lock; CLB is XILINX's naming for RLU.



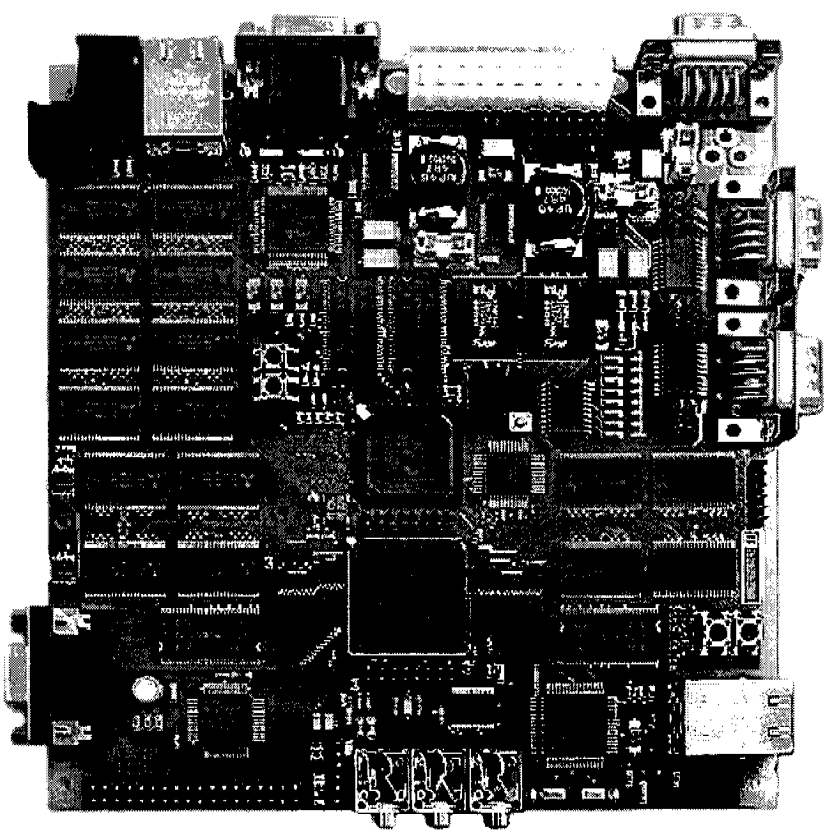


Fig. 70: Block schematics and photograph of the XF-BOARD.

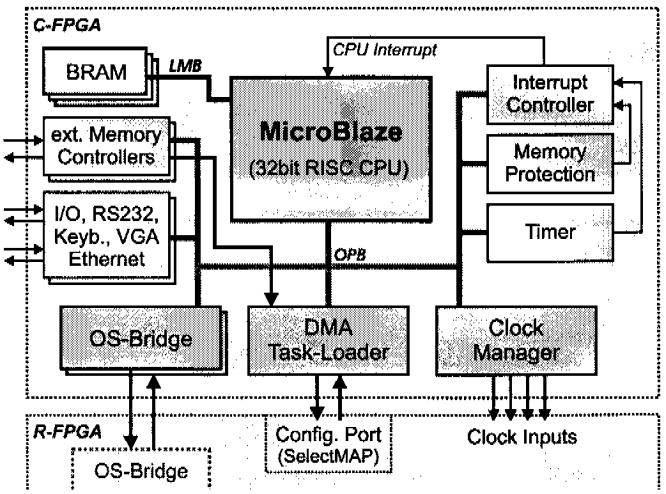


Fig. 71: RHWOS modules located in C-FPGA

Figure 71 shows the internals of the C-FPGA, which implements a configurable system on chip (CSoC). The MICROBLAZE system comprises the CPU core and a number of peripheral cores, connected to system bus [OPB]. We focus on the three most relevant in terms of RHWOS:

- *DMA task loader core*
  To download the configuration data (full and partial bitstreams) to the R-FPGA without bothering the CPU, we implemented a direct memory access (DMA) controller attached to the system bus, and to the external memory, respectively. The DMA controller allows for configuring the R-FPGA at maximum speed, i.e., 26.2ms for a full configuration, including 1'311'796 configuration bytes[4] (@50MHz configuration clock [XV2a]).
- *OS bridge core*
  The operating system bridge is a simple yet powerful peripheral core used for data transfer from and to the R-FPGA using a set of customized functions. These functions include communication with and control of operating system functions residing in reconfigurable hardware, checking the systems state (e.g., fill levels of FIFOs), setting run-time parameters of operating system objects (e.g., FIFO sizes), control of user hardware tasks, and debugging functions.
- *Clock manager core*
  During system runtime, the clock manager generates various clock signals according to the needs of the user tasks and OS-elements in the R-FPGA.

[4]The length of a configuration bitstream may vary, since a VIRTEX-II bitstream may contain so-called *multi-frame write* commands which can lead to compression effects [XV2a, XAPa, XAPb].

For debugging purposes, single clock pulses (for single stepping) or a defined number of clock pluses can be generated. The clock manager is implemented as an autonomous subsystem, controlled by the CPU.

5.4.2 R-FPGA / XILINX XC2V-3000

For the reconfigurable FPGA (R-FPGA) we have chosen a XILINX VIRTEX-II 3000 which offers an RLU array of 64×56 (=3584) RLUs.

The manual entry of the bus infrastructure is an extremely tedious and error-prone process. To facilitate the rapid construction of different runtime environments differing in the width of the OS-frames and number of reconfiguration slots, we have devised the interactive application *XFOSGen*, a runtime environment generator.

The input for XFOSGen are (i) the widths of the left and right communication bus, (ii) the list of required RHWOS elements, (iii) the dimension of the OS-frames (left and right), and (iv) the grid size of the reconfigurable slots. XFOSGen outputs the initial full configuration with the OS frames and the dummy tasks including the communication infrastructure (TCB, BARR, BARL). Moreover, XFOSGen generates task templates of different widths including parameterizable bus access controllers (BAC). These templates are then the starting point for hardware task development.

For an in-depth description of the XF-BOARD platform, we refer to [WNP04a, WNP04b, XFB, Nob03, Nob04, Ste04]. The following work reports on modules and applications developed for the XF-BOARD platform: [Weg04, Jon04].

5.5 Case Study Application

We have successfully implemented and tested several versions of the runtime environment, differing in the widths of the OS-frames and slot-grids.

At the time of this writing we have completed the implementation of a number of OS elements and user tasks. The OS elements include device drives such as a video driver, Ethernet MAC, IP protocol stack, audio driver, and a UART. Furthermore, we have developed a memory management unit (MMU) that utilizes internal BRAM and external SRAM and SDRAM memory. The size of the MMU amounts to 1262 slices (9% of the FPGA capacity). The user tasks include audio decoders, a MIDI controller, decryption cores, digital filters, a fractal generator, and allow applications from the multimedia, networking, and real-time signal processing domains to be tested. For a detailed description of this work, we refer to [WW04, Ste04, Jon04, Weg04].

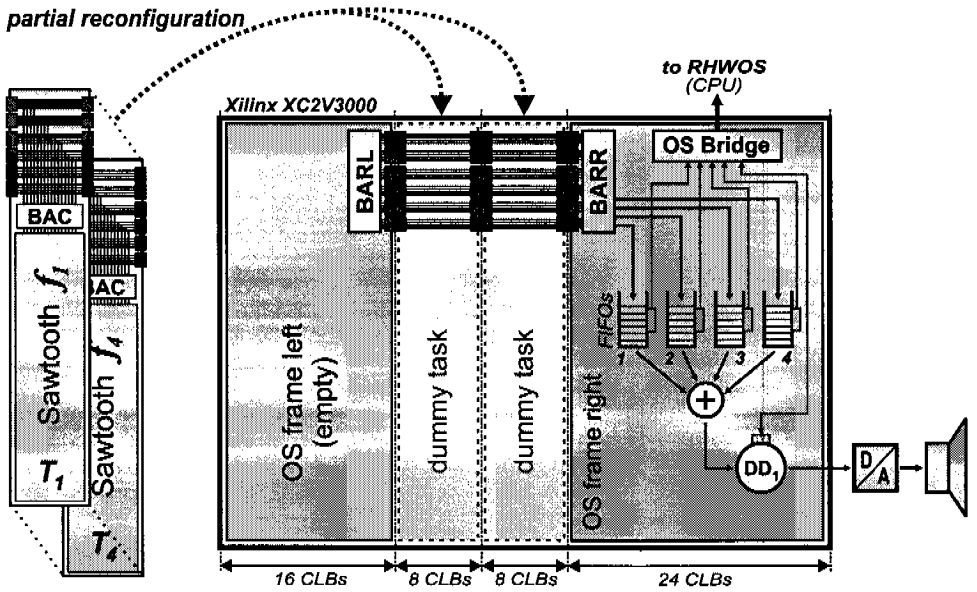


Fig. 72: Layout of the OS frame for the case study application.

In the remainder of this section we present two versions of a case study application, in order to discuss the runtime environment in more detail, particularly to investigate dynamic effects occurring as a result of partial reconfiguration. The application targets the audio domain and includes hard real-time requirements.

The goal of the application is to generate a *continuous* chord consisting of *four* sounds in real-time.

5.5.1 Application Scenario

Figure 72 illustrates the elements involved in the application: Four tasks $T_{1..4}$ are available that generate a sawtooth wave-form, each with a different frequency $f_{1..4}$. Once configured into a task slot, each task T_i writes its data across the TCB and BARR into the associated FIFO buffer $FIFO_i$. Each task T_i runs to completion, that is, until $FIFO_i$ is filled. An adder unit sums up the output of each $FIFO_{1..4}$. An empty *FIFO* contributes the value 0 to the summation. The 18-bit sum is trimmed to a 16-bit value and is forwarded to the audio device driver which feeds the external audio codec with the value received from the adder.

Each *FIFO* has a depth of 8192 16-bit words; the audio driver runs at a sample rate of 22'050 Hz. Hence, the play out of one entirely filled buffer $FIFO_i$ produces no more than $\frac{8192}{20'050} = 371$ ms of sound at frequency f_i.

Table 13 lists the bitstreams of the OS-frame and tasks $T_{1..4}$, their lengths and required configuration times.

The runtime environment, as depicted in Figure 72, offers two reconfigurable

Bitstream	*Size* [byte]	*Config. Time* [ms]
OS-frame (initial)	1'199'402	23.98
Task T_1 ($f_1 = 220$ Hz)	168'214	3.36
Task T_2 ($f_2 = 261$ Hz)	162'214	3.24
Task T_4 ($f_3 = 329$ Hz)	171'390	3.43
Task T_4 ($f_4 = 392$ Hz)	162'214	3.24

Tab. 13: Bitstream length and configuration times of OS-frame and tasks $T_{1..4}$.

slots, each capable of accommodating one of the four tasks $T_{1..4}$. Obviously, to obtain a continuously played sound, none of FIFO buffers $FIFO_{1..4}$ must underrun. Since there are four tasks but only two slots, the four tasks $T_{1..4}$ need to be dynamically scheduled by the RHWOS to the two slots, such that the FIFOs keep a sufficient number of samples at every point in time. To this end, the fill-level of each FIFO is monitored by the scheduler. The scheduler itself is running on the CPU and periodically sends commands to the OS-bridge to retrieve the fill-level of each FIFO. As soon as the scheduler detects that the fill-level of $FIFO_i$ falls below the lower threshold, task T_i is partially configured into a reconfigurable slot using the DMA loader. The scheduler runs a non-preemptive policy.

We are aware that this is a quite simple application, but nevertheless best suited to analyze the dynamics of a runtime reconfiguring application with real-time constraints.

5.5.2 Runtime Observations

Executing an RHWOS, e.g. conducting partial reconfigurations, evokes a number of effects that deserve special consideration.

1) Transient Effects

A communication bus assumes that its wires remain stable at all times of bus operation. In an FPGA, wires are composed of a number of wire-segments. The design tool decides during the map/place/route step from which parts a wire is formed. The decision can be influenced by optimization settings; moreover, routing is often based on non-deterministic algorithms. Consequently, routes in identical designs may have different courses after implementation. In static designs, this has no negative effect, since the design tool ensures the compliance with all relevant constraints (e.g. timing).

In a dynamic reconfiguration environment, this fact leads to some problems in connection with the realization of bus structures. If an existing bus-wire is reconfigured, the course of a wire may have changed even though its end-points

are identical. The result of different courses are *transient effects*, e.g. glitches on bus-wire signals.

II) Our Solution: Freeze Signal
In an earlier work, Erni and Reichmuth [ER03] already pointed at the transient effects during partial reconfiguration processes. Their solution was to *freeze* all critical signals (Reset, Read-Enable and WriteEnable) during reconfiguration. Thereby, the affected signals were stored in a flip-flop and conserved until configuration was complete and the freeze signal released. The freeze signal was implemented using an external wire that connects the OS frames.

Our application also adheres the idea of freezing signals in order to suppress influences of transient effects. The difference lies in the fact that there is no longer any need for external wiring. Instead the operating system can initiate a freeze phase, whose duration is proportional to the size of the task to be reconfigured. During this freeze time, every task neither reads nor writes any signal outside its own area, but it can still continue with intermediate computations. The duration of the freezing is submitted to the tasks via a dedicated signal line.

The minimal time for a full configuration of a XILINX VIRTEX-II 3000 is $t_{fc} = 26.2ms$; this time was reached and measured in [Jon04]. Configuration time degrades linear with the area size to be reconfigured. Therefore one minimal-sized task of width $w_{min} = 8$ CLBs should take about $t_{pc} = 3.74$ms for reconfiguration. We define the freeze signal to be active for one clock cycle if a minimal-sized task is to be configured. For each additional w_{min} in task size, the signal needs to be kept HIGH for an additional period, to keep the freeze signal active long enough. So if the operating system needs to reconfigure a task with a width of $3 \cdot w_{min}$, the freeze signal will be pulled up for 3 clock periods. As soon as the freeze signal is released, a counter is started in each bus access controller to freeze all access to the bus for the following $3 \cdot t_{pc}$.

We have successfully implemented and tested this method in our current runtime environment.

III) Alternative Approach: Hard-Macro
Another approach would be to make sure that the bus-wires follow the same course. Design tools allow for placing pre-defined structural elements, so called *hard macros*, into a design. In this way, the non-deterministic routing effect can be suppressed for parts of the circuits, i.e., for bus-wires.

We regard this approach as promising, since it leads to optimized bus-structures and eliminates all transient effects.

5.5.3 Measurements and Discussion

The case study application as previously described (according to Figure 72) works very well. We have successfully verified that none of the four FIFO buffers $FIFO_{1..4}$ ever under-run.

However, to *artificially create* a more constraining situation for the RHWOS, we derived one more version of the runtime environment that only offers *one* reconfigurable slot of 8 CLB width, instead of two. As a result, at most one task can be present in the system at any point in time. Thus, the four tasks $T_{1..4}$ need to be consecutively loaded into a single slot in order to properly execute the application. Compared to the first version, the frequency of hardware task configuration processes is doubled.

The application was still running well and produced continuous waveforms. In this new environment, we have measured the following attributes:

I) Area and Time Measurements

Table 14 indicates the area measurements regarding the tasks $T_{1..4}$ (all values are identical for all four tasks) and the runtime environment. The sources of the data are device specifications [XV2a] and design tool reports [XIF]. All measurements were made according to the definitions in Section 4.2.

Area Attribute	*Def.*	*Value*
(a) Width and Height of device D	$W_D \times H_D$	56 × 64 CLBs
(b) Area of the reconfigurable device D	A_D	3584 CLBs
(c) Area of the static part of device D	A_{D_s}	3072 CLBs
(d) Area of the dynamic part of device D	A_{D_d}	512 CLBs
(e) Number of effectively involved RLUs in task T_i	E_i	52 CLBs
(f) Physical area requirement of task T_i	A_i^p	61 CLBs
(g) Modeled area requirement of task T_i	A_i^m	512 CLBs
(h) Width and Height of task T_i	$W_i \times H_i$	8 × 64 CLBs

Tab. 14: Case study application: Area measurements.

In Table 15, timing data based on the executing case study application are displayed. The data were acquired with standard hardware measurement equipment. We have focussed on the following criteria:

- *Task Activation Latency (t_{ta})*
  Time difference between the detection of a fill-level shortfall of FIFO$_i$ and after enabling the appropriate task T_i.
- *Task Run-Time (e_i)*
  Time span in which a task T_i stays enabled (effectively running from an

RHWOS point of view).

- *Task Reset / Enable Time (t_r / t_e)*
  Time needed by the RHWOS to assert/de-assert the reset/enable signals for task T_i. This delay is caused by the transmission across the OS bridge.

Timing Attribute	*Def.*	*Typ. value*	*Max. value*
(a) Task activation latency	t_{ta}	3.94ms	13.02ms
(b) Task execution time	e_i	0.875ms	9.74ms
(c) Task reset/enable time	t_r/t_e	1.56us / 2.56us	n/a

Tab. 15: Case study application: Timing measurements.

II) Resulting Metrics (calculated)
Based on the measured data and according to the definitions in Section 4.2, we have derived the metrics as stated in Table 16.

Calculated Metric	*Def.*	*Value*
(a) Task internal fragmentation of task T_i	$\mathcal{F}_T(T_i)$	88.1%
(b) Block internal fragmentation of block B	$\mathcal{F}_B(B, T_i)$	88.1%
(c) Static area utilization of device D	$\mathcal{U}_A^s(D)$	5.94%
(d) Average area utilization of device D	$\overline{\mathcal{U}}_A^s(D)$	5.13%
(e) Dynamic area utilization of device D	$\overline{\mathcal{U}}_A^p(D)$	0.89%
(f) Dynamic area utilization of the dynamic part D_d	$\overline{\mathcal{U}}_A^p(D_d)$	6.23%
(e) Area Loss Ratio of device D	$\mathcal{L}_A(D)$	85.7%

Tab. 16: Case study application: Calculated fragmentation and utilization values.

III) Discussion
In the following, we will discuss the obtained results and metrics for the case study application:

- *Task/Block Internal Fragmentation*
  Since we use the granular 1D-variable area model in a degenerated form with only one reconfigurable slot, the values for task internal and block internal fragmentation are identical. The value of 88.1% is very high since the tasks

$T_{1..4}$ only implement simple functions and, thus, use a small amount of reconfigurable resources.

- *Static and Dynamic Area Utilization*
  The complete design, including RHWOS elements and one sawtooth task occupies 213 CLBs, which yields to a static utilization $\mathcal{U}_A^s(D)$ of 5.94% (reported by the design tool [XIF]). One task T_i uses 52 RLUs, whereof 41 RLUs constitute the *TCB* and *BAC* infrastructure.

  The task activation latency varies between 3.94 and 13.02 ms. Long activation times come from situations in which another task is already running when a new one is scheduled. Since it's a non-preemptive system, the next task has to wait until the previous one has completed.

  The measured execution time e_i of each task T_i is typically 0.875 ms, which is close to the theoretical value of 0.825 ms, that can be calculated based on the TCB bandwidth[5] and FIFO depth. However, sometimes we measured an e_i of up to 9.74 ms. This value can result, when a task remains in its slot because all FIFO fill-levels are above their lower threshold, and thus, no other task needs to be scheduled.

  The resulting average area utilization $\overline{\mathcal{U}}_A^s(D) = 5.13\%$. Based on the theoretical value of e_i, the dynamic utilization $\overline{\mathcal{U}}_A^p(D)$ reduces to 0.89%. Focusing on a single slot of size $D_d = 512$ CLBs, the utilization $\overline{\mathcal{U}}_A^p(D_d)$ rises to 6.23%. These utilization values are below reasonable limits, which is caused by the fact, that the tasks of our case study application only implement very small circuits.

- *Area Loss Ratio*
  Since we artificially decreased the dynamically reconfigured area, the area loss $\mathcal{L}_A(D)$ is more than 85% (for the version as depicted in Figure 72 $\mathcal{L}_A(D)$ amounts to 71.4%). Both values are too high. To achieve better utilizations, the loss must be significantly lower.

Obviously, as the area loss and utilization metrics reveal, this case study application is not adequate to prove the benefit of using an RHWOS. The current implementation only serves as a demonstrator to show that an RHWOS can be realized with the help of currently available FPGA devices. The major difficulties we experienced were caused by the design tools, which often failed to place/route designs with tight area constraints.

However, we were able to experimentally verify the metrics we defined. Moreover, we are convinced that both our XF-BOARD platform, and our runtime environment are suitable for implementing more sophisticated applications which are able to prove the profitability of an RHWOS.

[5] One 16-bit write access on the task communication bus (TCB) requires 5 clock cycles. The system runs at 50 MHz, thus, the resulting bandwidth of the TCB amounts to 160Mbits^{-1}. A FIFO buffer of depth 8192 can be filled within 819.2 μs.

6

Conclusions

The goal of this work is to gain insight into the design of a *Reconfigurable Hardware Operating System (RHWOS)*. Essentially, an RHWOS can be considered as an RTOS that additionally manages an *SRAM-based partially reconfigurable FPGA* as a dynamically allocatable system resource. An RHWOS is indispensable to ensure the efficient utilization of FPGAs when employing its resources in a dynamic way.

This thesis has focused on three main aspects of an RHWOS, namely on

- models and architectures on a conceptual and device independent level,
- typical algorithms for hardware task- and resource management, and
- the realization of an RHWOS based on current FPGAs and design tools.

6.1 Results

The following section presents the main achievements of this thesis structured along the three previously mentioned principal topics.

6.1.1 RHWOS Concepts and Architectures

The purpose of the conceptual analysis was to describe an RHWOS on a device and technology independent level, in order to identify the fundamental problems and to understand the basic mechanisms of an RHWOS.

- We present an *integrated design concept* of an RHWOS, covering the compile-time and the run-time aspects of the complete system.
- The *compile-time system* offers a programming model similar to that of an RTOS, which allows for decomposing an application into a set of cooperating tasks and OS objects. We extend this concept by introducing a new layer of abstraction: the *hardware task*. A hardware task represents a logic circuit which (i) executes a well-defined function, (ii) provides a well-defined interface, and (iii) can be dynamically and partially configured into an FPGA. We postulate that in an RHWOS driven system, hardware and software tasks are simultaneously executing and interacting.
- We discuss the characteristic properties of software and hardware tasks, define their interface structures, and describe the mechanism of the communication between hardware and software tasks, and OS-objects, respectively.
- We present a complete design-flow to construct RHWOS driven applications, which provides automatically generated *task templates* as starting points for application development.
- In our approach for the architecture of a *run-time system*, we identify a set of modules, each executing well-defined run-time functions, and propose an exemplary partitioning of these modules to run in software, or hardware, respectively.
- The run-time system includes dedicated elements on the FPGA that enable the partial reconfiguration of hardware tasks and allow communication between modules implemented in software and hardware.

Although the presented concepts are technology and device independent, we have developed these concepts in consideration if the properties and possibilities of current FPGAs, and design tools, respectively.

6.1.2 RHWOS Task and Resource Management

Most of the task and resource management algorithms in an RHWOS are fundamentally different from those employed in an RTOS. We have identified several problems typical for RHWOS and have developed various highly specialized algorithms to solve these problems, e.g. for placement and scheduling of hardware task in on-line scenarios.

- We present a novel *area fragmentation metric* that allows hardware tasks to be places in a given allocation, such that the fragmentation of the residual free area is pro-actively kept low. We denote this placement strategy as *best-fit placement*. In our simulations, best-fit placement for small and mid-sized hardware task sets achieved a reduction in the total execution time of up to 14.9% compared to first-fit.
- As a second method for improving the device utilization, we introduce a

hardware task transformation rule, called *foot-print transform*. This transformation rule changes the shape of a hardware task in order to fit it into the shape of an unallocated area. Foot-print transform in combination with first-fit turned out to be beneficial, as it showed performance gains of up to 18.4% over first-fit.

- Based on an already known *free area partitioning* algorithm, presented by Bazargan et al. [BKS00], we developed three enhanced versions that show performance improvements of up to 70%. The time and space complexity of the enhanced algorithms, however, are only marginally higher compared to Bazargan's version.
- To allow for fast run-time placement of hardware tasks, we propose a novel approach for determining feasible placements using a *2-dimensional hash-matrix*. Employing this technique, placements for hardware tasks of any dimensions can be retrieved in constant time $O(1)$, whereas the hash-table update is performed independently of the hardware task's execution.
- For slotted reconfigurable area models, we present the internal structure and functions of a *hardware task scheduler* able to perform various preemptive and non-preemptive scheduling policies. The policies base on those known from RTOS, such as FCFS, SJF, SRPT, and EDF[1], respectively, but have been adapted to RHWOS. In addition, the scheduler models the characteristics of the FPGA's configuration port. The experimental evaluation of the different schedulers shows that the reconfiguration overheads increase the total execution time of a task set between 1.2% and 7.3%.

In order to experimentally evaluate the above mentioned algorithms, we have constructed the *Task Placement and Scheduling Simulation System (TPS^3)*, an integrated time discrete simulation framework. It allows for measuring a wide range of parameters when simulating placement and scheduling randomly generated task sets.

6.1.3 RHWOS Prototype Implementation

The practical realization of an RHWOS, based on currently available FPGA's, rises a number of problems on a very detailed technical level. To investigate this class of problems, we have implemented the essential parts of an RHWOS and evaluated their performance by means of a case study application.

- We present the XF-BOARD, a tailored prototyping platform for RHWOS implementing XILINX VIRTEX-II FPGAs. The XF-BOARD meets the main requirements that an RHWOS poses on the underlying platform, i.e., fast partial reconfiguration and read-back operations to/from the FPGA device,

[1]FCFS = First Come First Serve; SJF = Shortest Job First; SRPT = Shortest Remaining Processing Time First; EDF= Earliest Deadline First.

and high-speed communication between the CPU and the FPGA devices.

- Considering the specific reconfiguration/readback characteristics of the XILINX VIRTEX-II family, we have realized an *RHWOS run-time environment* that follows the novel *granular 1D-variable area model.* It allows the partial reconfiguration of hardware tasks of different widths, and implements a *shared bus* which enables high-bandwidth inter-task communication. The complexity of the bus protocol is entirely hidden from the hardware tasks by a bus access controller. Furthermore, we have developed and successfully tested a number of RHWOS elements, such as clock manager, OS-bridge, memory management unit, and device drivers.
- To investigate the run-time aspects of both the RHWOS and the user tasks, we have implemented a case study application. The application targets the audio signal processing domain, holds hard real-time constraints and performs more than 10 partial reconfiguration processes per second.

The case study application running on the XF-BOARD acts as a *proof of concept*, and demonstrates the feasibility of an RHWOS, using currently available FPGA devices and design tools.

6.2 Further Research Issues

Promising starting points for further research in the field of *RHWOS* may include:

6.2.1 RHWOS Task and Resource Management

- In this work, we have merely considered system architectures consisting of a CPU and a *single* FPGA (cf. Figure 1). A possible extension would be to *increase the number of FPGAs* connected to the CPU. Thus, the RHWOS would be in charge of simultaneously managing several FPGAs. For such a system, our hardware task placement and scheduling functions need to be enhanced. For example, if more than one FPGA could accommodate a newly arrived hardware task, the placer would have to choose a device, according to a well-defined strategy. Scheduling would be influenced, since more than one hardware task could be in a configuration/readback state at the same time (assuming that the target architecture supports parallel configuration/readback processes).
- The presented on-line-placement and -scheduling algorithms deal with task sets consisting of *unrelated* and *independent* hardware tasks. In reality, tasks often show *execution dependencies* and applications may consist of a mix between *periodic* and *a-periodic* tasks. Furthermore, the *execution time* of tasks may be calculated or at least estimated during compilation time, either as an exact value or a time-range (e.g. by best/worst-case execution time

analysis). This additional information could be included in order to run *future oriented* scheduling and placement strategies. In this way, better results in terms of total execution time could be achieved, which would further increase the FPGA utilization.

- Our scheduling algorithms are tailored to merely execute hardware tasks. However, real systems may consist of both hardware *and* software tasks. The scheduler's algorithms need to be extended to deal with both kinds of tasks.

  Optionally, several tasks executing the same functionality could be implemented in hardware *as well as* in software, or with different timing and/or quality properties. During run-time, the scheduler could decide to activate either the software or the hardware version. This decision could be taken based on (i) the current resource allocation situation in the FPGA, (ii) timing constraints, and (iii) quality requirements of the application.

6.2.2 RHWOS Implementation

- While our prototypical case study application proves the feasibility of an RHWOS, its complexity is limited. More complex applications and OS elements should be implemented in order to further explore practical and technology related issues of an RHWOS.
- In our prototype, we have realized a highly speed-optimized DMA-based hardware task loader. Other hardware task management functions, such as *relocation*, *preemption*, *context extraction/insertion*, and *restoration*, are also crucial for the performance of the RHWOS, and the over-all system, respectively. These functions also need to be implemented in a time and space efficient way.
- We have presented a shared-bus solution in our run-time environment to allow for inter-task communication. A bus requires less resources than e.g. a crossbar but delivers lower communication bandwidth. The bandwidth can be varied by the bus width or the number of independent busses. However, the bus approach scales poorly with the FPGA size. Future devices with strongly increased densities will allow dozens of user tasks to be accommodated at the same time. In this case, the bus would become a severe communication bottleneck. Therefore, other more flexible communication structures need to be developed which are suitable for a dynamically reconfiguring environment.
- Generally, current FPGA architectures (such as XILINX VIRTEX(-II)) prevent to a large extent the realization of an efficient RHWOS. These devices exhibit a number of disadvantageous characteristics, e.g. inhomogeneous structures of the reconfigurable surface, coarse-grained access to the configuration memory, and large overheads in the configuration bitstreams. More-

over, the structure of the bitstreams are non-disclosed by the FPGA suppliers.

To allow for high-performance RHWOS implementations, we envision an FPGA architecture to be compliant with the following needs[2]

(i) *Reconfigurable Surface Homogeneity and Symmetry*
In XILINX VIRTEX(-II) devices, several types of special function blocks, such as multipliers or block RAMs, are irregularly scattered over the reconfigurable surface. Thus, hardware tasks using such blocks are position-invariant; i.e. they cannot be relocated.

In contrast, a homogeneous reconfigurable surface consists of *repetitively identical structures*. This ideally supports *position independent* hardware tasks. Task relocation can be done by simply changing the column- and row-offsets of all reconfigurable resources belonging to this task.

If the structure of the reconfigurable surface additionally exhibits rotation symmetry, then hardware tasks could be rotated before being placed. In a given allocation situation, this makes a successful placement of a task more likely, since its orientation can be adapted to fit into a free area.

Both the relocation and rotation methods, potentially lead to an increased placement quality and device utilization.

(ii) *Reconfiguration Granularity*
The VIRTEX(-II) device family only supports partial reconfiguration in the granularity of *frames*, whereas a frame includes several hundred configuration bits. The frames include configuration bits that control logic functions, interconnect/routing *and* I/O elements.

We propose a configuration architecture that (i) is fully disclosed by the supplier, and (ii) allows for accessing (reading/writing) the configuration memory in the granularity of *single bits*.

This would enable independent and efficient reconfiguration processes. Functions such as *context extraction* and *-insertion*, *dynamic re-routing* of inter-task communication wires could be realized much more efficiently.

[2]The needs are derived from the properties of the XILINX VIRTEX device family.

Bibliography

General References

[ABBT04] Ali Ahmadinia, Christophe Bobda, Marcus Bednara, and Jürgen Teich. A New Approach for On-line Placement on Reconfigurable Devices. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04) / Reconfigurable Architectures Workshop (RAW'04)*, page 134. IEEE Computer Society, April 2004.

[ABF+04] Ali Ahmadinia, Christophe Bobda, Sandor P. Fekete, Jürgen Teich, and Jan C. van der Veen. Optimal Routing-Conscious Dynamic Placement for Reconfigurable Devices. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'04)*, pages 847–851. Springer, LNCS 3203, August 2004.

[ABK+04] Ali Ahmadinia, Christophe Bobda, Dirk Koch, Mateusz Majer, and Jürgen Teich. Task Scheduling for Heterogeneous Reconfigurable Computers. In *Proceedings of the 17th International Symposium on Integrated Circuits and Systems Design (SBCCI'04)*, pages 22–27, Pernambucco, Brazil, September 2004. ACM Press.

[ABT03] Ali Ahmadinia, Christophe Bobda, and Jürgen Teich. Temporal Task Clustering for Online Placement on Reconfigurable Hardware. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT'03)*, pages 359–362, Tokyo, Japan, December 2003. IEEE.

[ABT04] Ali Ahmadinia, Christophe Bobda, and Jürgen Teich. A Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware. In *Proceedings of International Conference on Architecture of Computing Systems (ARCS'04)*, pages 125–139. Springer Verlag Heidelberg, February 2004.

[BA04] Faycal Bensaali and Abbes Amira. Design and Efficient FPGA Implementation of an RGB to YCrCb Color Space Converter Using Distributed Arithmetic. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applica-*

tions (FPL'04), pages 991–995. Springer, LNCS 3203, August 2004.

[BD01] Gordon Brebner and Oliver Diessel. Chip-Based Reconfigurable Task Management. In *Proc. 11th Int'l Workshop on Field Programmable Gate Arrays (FPL'01)*, pages 182–191. Springer, LNCS, 2001.

[BDH+97] Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. A Dynamic Reconfiguration Run-Time System. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 66–75. IEEE CS Press, 1997.

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge (UK), 1998.

[BFRV92] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[BKS00] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. *IEEE Design and Test of Computers*, 17(1):68–83, 2000.

[BMK+04] Christophe Bobda, Mateusz Majer, Dirk Koch, Ali Ahmadinia, and Jürgen Teich. A Dynamic NoC Approach for Communication in Reconfigurable Devices. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'04)*, pages 1032–1036. Springer, LNCS 3203, August 2004.

[BR96] Stephen Brown and Jonathan Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

[Bre96] Gordon Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proc. 6th Int'l Workshop on Field-Programmable Logic and Applications (FPL'96)*, pages 327–336. Springer, LNCS, 1996.

[Bre97] Gordon Brebner. The Swappable Logic Unit: a Paradigm for Virtual Hardware. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, 1997.

[BRM99] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Boston (USA), February 1999.

[BS99] Kiarash Bazargan and Majid Sarrafzadeh. Fast Online Placement for Reconfigurable Computing Systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'99)*, pages 300–302. IEEE Computer Socitey Press, April 1999.

[But00] G.C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston (USA), 1st edition, 2000.

[Car97] John W. Carter. *Digital Designing with Programmable Logic Devices*. Prentice-Hall, Inc., New Jersey, 1997.

[Car01] John D. Carpinelli. *Computer Systems Organization and Architecture*. Addison-Wesley, Boston (USA), October 2001.

[CCKH01] Katherine Compton, James Cooley, Stephen Knol, and Scott Hauck. Configuration Relocation and Defragmentation for Reconfigurable Computing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'01)*. IEEE CS Press, April 2001.

[CH99] Katherine Compton and Scott Hauck. Configurable Computing: A Survey of Systems and Software. Technical report, Northwestern University, Dept. of Electrical and Computer Engineering, 1999.

[CLC+02] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol, and Scott Hauck. Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):209–220, 2002.

[DE98] Oliver Diessel and Hossam ElGindy. On Scheduling Dynamic FPGA Reconfigurations. In *Proc. 5th Australasian Conference on Parallel and Real-Time Systems (PART'98)*, pages 191–200, 1998.

[DEM+00] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs. *IEE Proceedings – Computers and Digital Techniques*, 147(3):181–188, May 2000.

[DJR01] Santanu Dutta, Rune Jensen, and Alf Rieckmann. Viper: A Multiprocessor SoC for Advanced Set-Top Box and Digital TV Systems. *IEEE Design and Test of Computers*, September-October:21–31, 2001.

[DM94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., New York (USA), 1994.

[DPP02] Matthias Dyer, Christian Plessl, and Marco Platzner. Partially Reconfigurable Cores for Xilinx Virtex. In *Proc. 12th Int'l Conference on Field-Programmable Logic and Applications (FPL'02)*, pages 292–301. Springer, LNCS, September 2002.

[DW99] Oliver Diessel and Grant Wigley. Opportunities for Operating Systems in Reconfigurable Computing. Technical report, Advanced Computing Research Centre, School of Computer and Information Science, University of South Autstralia, August 1999.

[EGJ99] D. Eatmon and C.S. Gloster Jr. Evaluating Placement Algorithms for Run-Time Reconfigurable Systems. In *Proceedings of the Military and Aerospace Programmable Logic Device (MAPLD'99) International Conference*, pages 110–115, September 1999.

[EH96] James G. Eldredge and Brad L. Hutchings. Run-Time Reconfiguration: A Method for Enhancing the Functional Density. *Journal of VLSI Signal Processing*, 12:67–86, 1996.

[Eka04] Paul Ekas. FPGA Co-Processors Optimize Automotive Infotainment and Telematic Systems. *Electronic Engineering Times (Asia)*, 11:3, 2004.

[EMSS00] Hossam ElGindy, Martin Middendorf, Hartmut Schmeck, and Bernd Schmidt. Task Rearrangement on Partially Reconfigurable FPGAs with Restricted Buffer. In *Proceedings of the 10th International Conference on Field Programmable Logic and Applications (FPL'00)*, pages 379–388. Springer, LNCS, 2000.

[EP00] Michael Eisenring and Marco Platzner. An Implementation Framework for Run-time Reconfigurable Systems. In *In Proceedings of the 2nd International Workshop of Engineering of Reconfigurable Hardware/Software Objects (ENREGLE'00)*, pages 151–157, June 2000.

[FKT01] Sandor Fekete, Ekkehard Köhler, and Jürgen Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Proc. Design Automation and Test in Europe (DATE'01)*, pages 658–665, Los Alamitos, USA, 2001. IEEE CS Press.

[FP98] William Fornaciari and Vincenzo Piuri. Virtual FPGAs: Some steps behind the physical barriers. In *IPPS/SPDP Workshops*, pages 7–12, 1998.

[GASF02] Manuel G. Gericota, Gustavo R. Alves, Miguel L. Silva, and José M. Ferreira. On-line Defragmentation for Run-Time Partially Reconfigurable FPGAs. In *Proc. 12th Int'l Conf. on Field Programmable Logic and Applications (FPL'02)*, pages 302–311. Springer, LNCS, 2002.

[GNVV04] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs over Processors. In *Proceedings of the 2004 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA'04)*, pages 162–170. ACM Press, New York, USA, 2004.

[Hau98b] Scott Hauck. The Future of Reconfigurable Systems. In *Proceedings of the 5th Canadian Conference on Field Programmable Devices*. Keynote Address, June 1998.

[Hau98c] Scott Hauck. The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–639, April 1998.

[HLK02] Edson L. Horta, John W. Lockwood, and Sérgio Kofuji. Using PARBIT to Implement partial Run-Time Reconfigurable Systems. In *Proc. 12th Int'l Conf. on Field Programmable Logic and Applications (FPL'02)*, pages 182–191. Springer, LNCS, 2002.

[Hsi00] Hsieh, Harry and Balarin, Felice and Lavagno, Luciano and Sangiovanni-Vincentelli, Alberto. Efficient Methods for Embedded System Design Space Exploration. In *In Proceedings of the 37th Conference on Design Automation (DAC'00)*, pages 607–612. ACM Press, 2000.

[HUBB04] M. Huebner, M. Ullmann, L. Braun, and J. Becker. Scalable Application-Dependent Networks on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'04)*, pages 1037–1041. Springer, LNCS 3203, August 2004.

[HV04a] Manish Handa and Ranga Vemuri. A Fast Algorithm for Finding Maximal Empty Rectangles for Dynamic FPGA Placement. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE'04)*, pages 744–745. IEEE Computer Society, March 2004.

[HV04b] Manish Handa and Ranga Vemuri. An Integrated Online Scheduling and Placement Methodology. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'04)*, pages 444–453. Springer, LNCS 3203, August 2004.

[HV04c] Manish Handa and Ranga Vemuri. Area Fragmentation in Reconfigurable Operating Systems. In *Proceedings of the 4th International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'04)*, pages 77–83. CSREA Press, June 2004.

[JTY+99] Jack S.N. Jean, Karen Tomko, Vikram Yavagal, Jignesh Shah, and Robert Cook. Dynamic Reconfiguration to Support Concurrent Applications. *IEEE Transactions on Computers*, 48(6):591–602, June 1999.

[Kal01] Deb Kalyanmoy. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley, Chichester, 1st edition, Juni 2001.

[KKKR04] H. Kalte, M. Koester, B. Kettelhoit, and U. Rückert. A Comparative Study on System Approaches for Partially Reconfigurable Architectures. In *Proceedings of the 4th International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'04)*, pages 70–76. CSREA Press, June 2004.

[KLV+02a] H. Kalte, D. Langen, E. Vonnahme, A. Brinkmann, and U. Rückert. Dynamically Reconfigurable System-on-Programmable-Chip. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP'02)*, page 235, Gran Canaria Island, Spain, January 2002. IEEE Press.

[KSM03] Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. III. Mooney. A Configurable Hardware Schedulder for Real-Time Systems. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'03)*, pages 95–101. CSREA Press, June 2003.

[MBV+02] Théodore Marescaux, Andrei Bartic, Dideriek Verkest, Serge Vernalde, and Rudy Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs. In *Proc. 12th Int'l Conf. on Field-Programmable Logic and Applications (FPL'02)*, pages 795–805. Springer, LNCS, 2002.

[MJL98] Pedro Merino, Margarida Jacome, and Juan Carlos Lopez. A Methodology for Task Based Partitioning and Scheduling of Dynamically Reconfigurable Systems. In *Proc. IEEE Symopsium on FPGAs for Custom Computing Machines (FCCM'98)*, pages 324–325, 1998.

[ML01] John MacBeth and Lysaght. Dynamically Reconfigurable Cores. In *Proceedings of the 11th International Conference on Field Programmable Gate Arrays (FPL'03)*, pages 451–460. Springer, LNCS, September 2001.

[MLJ98] Pedro Merino, Juan Carlos Lopez, and Margarida Jacome. A Hardware Operating System for Dynamic Reconfiguration of FPGAs. In *Proc. 8th Int'l Workshop on Field Programmable Gate Arrays (FPL'98)*, pages 431–435. Springer, LNCS, 1998.

[MMB+03] T. Marescaux, J.-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, R. Vernalde, and R. Lauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable Systems. In *Proceedings of the 13th International Conference on Field Programmable Gate Arrays (FPL'03)*, volume LNCS 2778, pages 595–605. Springer, September 2003.

[MNC+03] J-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, Vernalde S., and R. Lauwereins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In *Proceedings of Design, Automation and Test in Europe (DATE'03)*, pages 986–991. IEEE Computer Society, March 2003.

[MSV00] Bingfeng Mei, Patrick Schaumont, and Serge Vernalde. A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems. In *Proceedings of the 11th ProRISC Workshop on Circuits, Systems and Signal Processing*, pages 405–411, 2000.

[MVVL02] J-Y. Mignolet, Serge Vernalde, Diederick Verkest, and Rudy Lauwereins. Enabling Hardware-Software Multitasking on a Reconfigurable Computing Platform for Networked Portable Multimedia Appliances. In *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'02)*, pages 116–122. CSREA Press, June 2002.

[NCV+03] V. Nollet, P. Coene, D. Verkest, Vernalde S., and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *Proceedings of the 17th International Parallel and*

Distributed Processing Symposium (IPDPS'03) / Reconfigurable Architectures Workshop (RAW'03), page 174. IEEE Computer Society, April 2003.

[NH93] Juerg Nievergelt and Klaus H. Hinrichs. *Algorithms and Data Structures*. Prentice Hall, Inc., Upper Saddle River (USA), 1993.

[NMB+03] Vincent Nollet, J-Y. Mignolet, T.A. Bartic, Diederik Verkest, Serge Vernalde, and R. Lauwreins. Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'03)*, pages 81–87. CSREA Press, June 2003.

[NMV90] Vincent Nollet, Theodore Marescaux, and Diederik Verkest. Operating-System Controlled Network on Chip. In *Proceedings of the 41st Design Automation Conference (DAC'04)*, pages 256–259. ACM / IEEE, June 1990.

[OD95] John V. Oldfield and Richard C. Dorf. *Field-Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. Wiley-Interscience, 1995.

[PB99] K. Purna and D. Bhatia. Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. *IEEE Transactions on Computers*, 48(6):579–590, June 1999.

[Pin95] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall International, Englewood Cliffs, NJ (USA), 1995.

[Pla99] Marco Platzner. Executives for Reconfigurable Embedded Systems (X-FORCES), Research Proposal, September 1999. Swiss Federal Institute of Technology (ETH), Zurich / Computer Engineering and Networks Laboratory.

[PLP+03] Kiran Puttegowda, David I. Lehn, Jae H. Park, Peter Athanas, and Mark Jones. Context Switching in a Run-Time Reconfigurable System. *Journal of Supercomputing*, 26(3):239–257, 2003.

[PMW04] Ju Hwa Pan, Tulika Mitra, and Weng Fai Wong. Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs. In *Proceedings of theInternational Conference on Computer Aided Design 2004 (ICCAD)*, pages 31–38. IEEE Press, November 2004.

[PP02] Christian Plessl and Marco Platzner. Custom Computing Machines for the Set Covering Problem. In *Proceedings of 10th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'02)*, pages 163–172, Napa, USA, April 2002. IEEE CS.

[PP03] Christian Plessl and Marco Platzner. TKDM - a reconfigurable co-processor in a PC's memory slot. In *Proceedings IEEE International Conference on Field-Programmable Technology (FPT'03)*, pages 252–259, Tokyo, Japan, December 2003. IEEE Press.

[Sag98] Vivek Sagdeo. *The Complete VERILOG Book*. Kluwer Academic Paublishers, Boston (USA), 1st edition, June 1998.

[SBJ+96] John A. Stankovic, Alan Burns, Kevin Jeffay, Mike Jones, and Gary Koob. Strategic Directions in Real-Time and Embedded-Systems. *ACM Computing Suerveys (CSUR)*, 28(4):751–763, December 1996.

[SG98] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison Wesley Longman, Inc., Reading, Massachsetts (USA), 5th edition, 1998.

[Sha98] Ashok K. Sharma. *Programmable Logic Handbook - PLDs, CPLDs, and FPGAs*. McGraw-Hill, New York (USA), 1st edition, 1998.

[SLM00] H. Simmler, L. Levinson, and R. Männer. Multitasking on FPGA Coprocessors. In *Proc. 10th Int'l Workshop on Field Programmable Gate Arrays (FPL'00)*, pages 121–130. Springer, LNCS, 2000.

[SSRC98] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Buttazzo Giorgio C. *Deadline Scheduling for Real-Time Systems*. Kluwer Academics Publishers, Norwell, Massachusetts (USA), 1998.

[Tan87] Andrew S. Tannenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Inc., Englewood Cliffs, NJ (USA), 1987.

[TCGK03] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, and Simon Künzli. Design Space Exploratin of Network Processor Architectures. *Network Processor Design: Issues and Practices*, pages 55–89, 2003.

[TDC97] Steve Trimberger, Khue Duong, and Bob Conn. Architecture Issues and Solutions for a High-Capacity FPGA. In *Proceedings of the 5th International Symposium on Field Programmable Gate Arrays (FPGA'97)*, pages 3–9. ACM Press, New York (USA), 1997.

[Tri94] Stephen M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Norwell, MA (USA), 1994.

[TSMN04] Jesus Tabero, Julio Septien, Hortensia Mecha, and Daniel Nozos. A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'04)*, pages 241–250. Springer, LNCS 3203, August 2004.

[Ver01] Eric Verhulst. *RTOS for DSPs: What makes it tick?* Eonic Systems BV, `http://www.eonic.com`, Delft, NL, 2001.

[VMS96] John Villasenor and William H. Mangione-Smith. Configurable Computing. *Scientific American (International Edition)*, 276:54–59, June 1996.

[Wal66] R.J. Walker. *CUPL - The Cornell University Programming Language*. Cornell University Press, Inthaca, NY, USA, 1966.

[Wan98] Markus Wannemacher. *Das FPGA-Kochbuch*. International Thomson Publishing, Bonn, Germany, 1 edition, 1998.

[WH97] Michael J. Wirthlin and Brad L. Hutchings. Improving Functional Density through Run-Time Constant Propagation. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'97)*, pages 86–92. ACM Press, New York, USA, 1997.

[Wir97] Michael J. Wirthlin. *Improving Functional Density Through Run-Time Circuit Reconfiguration*. PhD thesis, Brigham Young University, Department of Electrical and Computer Enigeering, November 1997.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management*, pages 1–116. Springer Verlag LNCS, September 1995.

[WK01a] Grant Wigley and David Kearney. The Development of an Operating System for Reconfigurable Computing. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'01)*, 2001.

[WK01b] Grant Wigley and David Kearney. The First Real Operating System for Reconfigurable Computers. In *Proceedings of the 6th Australasian Computer Science Week (ACSAC'01)*, pages 130–137, Gold Coast, 2001. IEEE Press.

[WK02a] Grant Wigley and David Kearney. Research Issues in Operating Systems for Reconfigurable Computing. In *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'02)*, pages 10–16. CSREA Press, June 2002.

[WK02b] Grant Wigley and David Kearney. The Management of Applications for Reconfigurable Computing using an Operating System. In *Proceedings of 7th Asia Pacific Computer Systems Architecture Conference*, pages 32–39, 2002.

[YSC02] Candice C. Yui, Gary M. Swift, and Carl Carmichael. Single Event Upset Susceptibility Testing of the Xilinx Virtex-II FPGA. Technical report, California Institute of Technology, Jet Propulsion Laboratory, Pasadena, CA., 2002.

[Zai93] Navabi Zainalabedin. *VHDL Analysis and Modelling of Digital Systems*. McGraw-Hill, Inc., New York (USA), 1993.

[Zit99] Eckart Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Swiss Federal Institute of Technology ETH, Zurich, December 1999.

Authors's Publications (selection)

[PEW+02] Christian Plessl, Rolf Enzler, Herbert Walder, Jan Beutel, Marco Platzner, and Lothar Thiele. Reconfigurable Hardware in Wearable Computing Nodes. In *Proceedings of the 6th International Symposium on Wearable Computers (ISWC'02)*, pages 215–222. IEEE Computer Society, October 2002.

[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.

[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, LNCS, September 2003.

[SWP04] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transaction on Computers*, 53(11):1392–1407, November 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 International Real-Time Systems Symposium (RTSS'03)*, pages 224–235. IEEE Computer Society, December 2003.

[WNP04a] Herbert Walder, Samuel Nobs, and Marco Platzner. XF-Board: A Prototyping Platform for Reconfigurable Hardware Operating Systems. In *Proceedings (Posters) of the 4rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'04)*, page 6. CSREA Press, June 2004.

[WNP04b] Herbert Walder, Samuel Nobs, and Marco Platzner. XF-Board: Prototype Platform for Reconfigurable Hardware Operating System. Technical Report TIK Nr. 193, Swiss Federal Institute of Technology (ETH), Zurich, March 2004.

[WP02] Herbert Walder and Marco Platzner. Non-preemptive Multitasking on FPGA: Task Placement and Footprint Transform. In *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'02)*, pages 24–30. CSREA Press, June 2002.

[WP03a] 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'03)*, pages 290–295. IEEE Computer Society, March 2003.

[WP03b] Herbert Walder and Marco Platzner. Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'03)*, pages 284–287. CSREA Press, June 2003.

[WP03c] Herbert Walder and Marco Platzner. Reconfigurable Hardware OS Prototype. Technical Report TIK Nr. 168, Swiss Federal Institute of Technology (ETH), Zurich, April 2003.

[WP04a] 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, LNCS, August 2004.

[WP04b] Herbert Walder and Marco Platzner. Implementation of a Runtime Environment for Reconfigurable Hardware Operating Systems. Technical Report TIK Nr. 195, Swiss Federal Institute of Technology (ETH), Zurich, March 2004.

[WSP03] Herbert Walder, Christoph Steiger, and Marco Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS) / Reconfigurable Architectures Workshop (RAW'03)*, page 178. IEEE Computer Society, April 2003.

[WW04] Herbert Walder and Silvan Wegmann. Case Study Applications for Reconfigurable Hardware Operating System Platform (XF-Board). Technical Report TIK Nr. 200, Swiss Federal Institute of Technology (ETH), Zurich, July 2004.

PhD-, Master- and Term-Thesis

[DH96] Andre De Hon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, October 1996.

[DW02] Matthias Dyer and Marco Wirz. Reconfigurable System on FPGA. Master's thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2002.

[Eis02] Michael H. Eisenring. *Communication Channel Synthesis for Heterogenous Embedded Systems*. PhD thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2002.

[ER03] Andres Erni and Stefan Reichmuth. Inter-Task-Communication in Reconfigurable Hardware OS. Master's thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2003.

[HL04] Dani Hobi and Pascal Lüdi. Audio Playback Tasks for RHWOS. Term Thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2004.

[Jon04] Kristofer Jonsson. Memory Management Unit for RHWOS. Master's thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2004.

[LZ02] Michael Lerjen and Chris Zbinden. Reconfigurable BlueTooth-Ethernet Bridge. Master's thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2002.

[Nob03] Samuel Nobs. Prototype Board for Reconfigurable Hardware Operating Systems, 2003.

[Nob04] Samuel Nobs. Reconfigurable Hardware Operating System Prototpye, Part C-FPGA. Master's thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2004.

[Rup03] Michael Ruppen. Reconfigurable OS Prototype. Master's thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2003.

[Ste04] Simon Steinegger. Reconfigurable Hardware Operating System Prototpye, Part R-FPGA. Master's thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2004.

[Weg04] Silvan Wegmann. Video Playback Tasks for RHWOS. Master's thesis, Swiss Federal Institute of Techology (ETH), Zurich (Switzerland), 2004.

Products, Manuals, Data Sheets, and Links

[A7S] Triscent Corp., Triscent A7S Field Configurable System on Chip, `http://www.triscent.com/products/a7.htm`.

[Act] Actel Corporation, , 2061 Stierlin Court, Mountain View, CA 94043, USA, `http://www.actel.com`.

[Alt] Altera Corporation, 101 Innovation Drive, San Jose, CA 95134, USA, `http://www.altera.com`.

[AMB] AMBA Specification (Rev 2.0), 1990, ARM Ltd. .

[AMD] Advanced Micro Devices Inc., P.O. Box 3453, Sunnyvale, CA 94088, USA, `http://www.amd.com`.

[Amp] Amphion Semiconductor Ltd., http://www.amphion.com.

[ARM] ARM Holdings Ltd., 110 Fulbourn Road Cambridge, CB1 9NJ, UK, http://www.arm.com.

[Atm] Atmel Corporation, 2325 Orchad Parkway, San Jose, CA 95131, USA, http://www.atmel.com.

[B5X] B5-X300 FPGA Board, Burch Electronic Designs (BurchED), North Ryde NSW 2113, Australia, http://www.burched.biz.

[Cha] Chameleon Systems, Inc., CS2000 Reconfigurable Processor, CS2000 Advanced Product Information, 2000.

[Cyc] Altera Corporation, Cyclone II Device Handbook, Volume 1, July 2004, http://www.altera.com.

[Dat] ABEL (Advanced Boolean Equation Language), Data I/O Corporation, 10525 Willows Road NE, Redmond, WA 98073, USA, http://www.dataio.com.

[ECo] eCos / sCosPro Realtime Operating System, eCosCentric Ltd., West Wickham, Cambridge, UK, http://www.ecoscentric.com.

[ECP] Lattice Semiconductor Corporation, Lattice ispXPGA Family Data Sheet, July 2004, http://www.latticesemi.com.

[EDK] Xilinx Embedded Development Kit and Plaform Studio, Xilinx Inc.

[ELi] Embedded Linux, Embedded Linux Consortium (ELC), http://www.embedded-linux.org.

[EPA] Atmel Corporation, Atmel High Performance EE PAL Datasheet ATF22V10C.

[Exc] Altera Corporation, Excalibur Device Overview, Version 2.0, May 2002, http://www.altera.com.

[Gai] Gaisler Research, http://www.gaisler.com.

[Gar04] Gartner Inc., Gartner Dataquest Market Research, http://www.gartner.com, 2004.

[Gec] IMEC Interuniversity Micro Electronic Center, T-ReCS Gecko: Hardware/Software Multitasking on a Reconfigurable Platform.

[IME] IMEC Interuniversity Micro Electronic Center, Kapeldreef 75, 3001 Leuven, Belgium, http://www.imec.be.

[isp] Lattice Semiconductor Corporation, LatticeECP/EC Family Data Sheet, August 2004, http://www.latticesemi.com.

[iSu] iSuppli Corporation, 1700 East Walnut Avenue, CA 90245, USA, http://www.isuppli.com.

[JBi] JBits 3.0 SDK for Virtex-II, Xilinx Inc.

[JED] Joint Electron Device Engineering Council, 2500 Wilson Blvd., Arlington, VA 22201, USA, http://www.jedec.org.

[JTA] JTAG (Joint Test Action Group) Boundary-Scan, IEEE 1149.1-4, http://www.jtag.com.

[Lat] Lattice Semiconductor Corporation, 5555 North East Moore Court, Hillsboro, USA, http://www.latticesemi.com.

[LEOa] The LEON Processor User's Manual, Version 2.3.7, August 2001, Gaisler Research, http://www.gaisler.com.

[LEOb] The LEON2 Processor User's Manual, Version 1.0.23, May 2004, Gaisler Research, http://www.gaisler.com.

[Lnx] Linux Operating System http://www.linux.org.

[Mac] Mac OS X, Apple Computer Inc., 1 Infinite Loop, Cupertino, CA 95014, USA, http://www.apple.com/macosx.

[Men] Mentor Graphics Corporation, 8005 S.W. Boeckmann Road, Wilsonville, OR 97070, USA, http://www.mentor.com.

[Mon] Monolithic Memories Inc., acquired by AMD in 1987.

[NIO] Altera Corporation, NIOS CPU, NIOS CPU 3.0 Data Sheet, March 2003, http://www.altera.com.

[OPB] IBM Inc., On-Chip Peripheral Bus: Architecture Specifications Version 2.1, SA-14-2528-02, April 2001.

[ORC] Lattice Semiconductor Corporation, Lattice ORCA Series 4 FPGAs Data Sheet, November 2003, http://www.latticesemi.com.

[PAL] PALASM (Programmable Array Logic Assmbler), Advanced Micro Devices Inc., P.O. Box 3453, Sunnyvale, CA 94088, USA, http://www.amd.com.

[PPC] IBM Inc., PowerPC Embedded Processor Core User's Manual, Fifth Edition (December 2001.

[QNX] QNX Software Systems, QNX Neutrino Realtime Operating System, http://www.qnx.com.

[Qui] QuickLogic, 1277 Orleans Drive, Sunnyvale, CA 94089, USA, http://www.quicklogic.com.

[Sha04] Richard C. Shannon. The PLD Industry: Strong Growth and Profitability Lie Ahead. Piper Jaffray Equity Research, http://www.piperfaffray.com, July 2004.

[Sig] Signetics Corporation, http://www.signetics.com.

[Str] Altera Corporation, Stratix 2 Device Handbook , Version 1.0, July 2004, http://www.altera.com.

[Sun] Sundance Multiprocessor Technology Ltd., Chesham, UK, http://www.sundance.com.

[Syn] Synplicity Inc., 600 W. California Avenue, Sunnyvale, CA 94086, USA, http://www.synplicity.com.

[Syo] Synopsys Inc., 700 East Middlefield Road, Mountain View, CA 94043, USA, http://www.synopsys.com.

[TAS01] Rich Templeton, Steve Appleton, and George Scalise. SIA Semiconductor Forecast 2001-2004, SIA Semiconductor Industry Association, November 2001.

[TDB] Texas Instruments Inc., DSP/BIOS Real-Time Operating System for DSPs, http://www.ti.com.

[TPS] Task Placement and Scheduling Simulation System (TPS^3), http://www.tik.ee.ethz.ch/walder/TPSSS.htm.

[Tre] AML91S100E ARM+FPGA Embedded CPU Module, Trenz Electronic GmbH, Brendel 20, 32257 Bünde, Germany, http://www.trenz-electronic.de.

[Unx] UNIX Operating System, http://www.unix.org.

[Vir] Virtuoso V.4 Reference Manual, Eonic Systems BV.

[VxW] VxWorks, Wind River Systems Inc., 500 Wind River Way, Alameda, CA 94501, USA, http://www.windriver.com.

[WEm] Windows Embedded (Windows CE / Windows XP Embedded), Microsoft Corporation, Redmond (WA), USA
http://www.microsoft.com/embedded.

[Win] Microsoft Windows, Microsoft Corporation, Redmond (WA), USA, http://www.microsoft.com/windows.

[XAPa] Xilinx XAPP138: Virtex FPGA Series Configuration and Readback, Xilinx Inc.

[XAPb] Xilinx XAPP151: Virtex Series Configuration Architecture User Guide, Xilinx Inc.

[XAPc] Xilinx XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based, Xilinx Inc.

[XC6] Xilinx Inc., Xilinx XC6200 Platform FPGA: User Guide.

[XCG] Xilinx Virtex Core Generator, Xilinx Inc.

[XCR] Xilinx CoolRunner CPLD Familiy, Product Specification, Xilinx Inc.

[XES] X Engineering Software Systems Corporation (XESS), Raleigh, NC 27636, USA, http://www.xess.com.

[XFB] XF-Board, Integrated Platform for Reconfigurable Hardware Operating System Prototyping, Swiss Federal Institute of Technology (ETH), Computer Engieering and Networks Laboratory (TIK), Switzerland http://www.tik.ee.ethz.ch/ xfboard.

[XFE] Xilinx FPGA Editor, Version 6.2, Xilinx Inc.

[XIF] Xilinx ISE Foundation Dsign Tools, Version 6.2, Xilinx Inc.

[Xil] Xilinx Corporation, 2100 Logic Drive, San Jose, CA 95124, USA, http://www.xilinx.com.

[XMB] Xilinx MicroBlaze 32-bit RISC Soft Processor Core, Xilinx Inc.

[XMD] Xilinx Modular Design: Advanced Design Techniques, Xilinx Inc.

[XS3] Xilinx Inc., Xilinx Spartan-3 FPGA Family: Complete Data Sheet, July 2004.

[XSV] XESS Corporation, XSV-800 Xilinx Virtex Rapid Prototyping Board, http://www.xess.com.

[XV2a] Xilinx Inc., Xilinx Virtex-II Platform FPGA: User Guide.

[XV2b] Xilinx Inc., Xilinx Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet.

[XVI] Xilinx Inc., Xilinx Virtex Platform FPGA: Data Sheet.

Appendix

Seite Leer /
Blank leaf

List of Abbreviations and Acronyms

ABEL	**A**dvanced **B**oolean **E**quation **L**anguage
AES	**A**dvanced **E**ncryption **S**tandard
AIM	**A**dvanced **I**nterconnect **M**odule [XCR]
ALU	**A**rithmetic **L**ogic **U**nit
API	**A**pplication **P**rogramming **I**nterface
ARP	**A**ddress **R**esolution **P**rotocol (IEEE...)
AST	**A**pplication **S**pecific **T**ask
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **RAM**, **R**andom **A**ccess **M**emory [XVI]
BSP	**B**oard **S**upport **P**ackage
CLB	**C**onfigurable **L**ogic **B**lock [XVI]
CoDec	**Co**der / **Dec**oder
CPLD	**C**omplex **P**rogrammable **L**ogic **D**evice
CPU	**C**entral **P**rocessing **U**nit
CRC	**C**yclic **R**edundancy **C**heck
DCI	**D**igitally **C**ontrolled **I**mpedance [XV2a]
DCM	**D**igital **C**lock **M**anager [XV2a]
DSP	**D**igital **S**ignal **P**rocessor
DES	**D**igital **E**ncryption **S**tandard
EST	**E**xecution **S**upport **T**ask
FFT	**F**ast **F**ourier **T**ransform
FIFO	**F**irst **I**n **F**irst **O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GAL	**G**eneric **A**rray **L**ogic
HEX	**Hex**adecimal
HT	**H**ardware **T**ask
HW	**H**ardware
I/O	**I**nput/**O**utput
IC	**I**ntegrated **C**ircuit
ICMP	**I**nternet **C**ontrol **M**essage **P**rotocol
IP	**I**ntellectual **P**roperty / **I**nternet **P**rotocol (IEEE...)
IDCT	**I**nverse **D**iscrete **C**osine **T**ransform
IOB	**I**nput-/**O**utput-**B**lock [XVI]
JEDEC	**J**oint **E**lectron **D**evice **E**ngineering **C**ouncil
JTAG	**J**oint **T**est **A**ction **G**roup [JTA]
LUT	**L**ook-**u**p **T**able [XVI]
LVTTL	**L**ow **L**evel **T**ransistor/**T**ransistor **L**ogic
μC	**Micro** **C**ontroller

MUX	**Mu**ltiplexor
OS	**O**perating **S**ystem
OSO	**O**perating **S**ystem **O**bject
PAL	**P**rogrammable **A**rray **L**ogic
PCB	**P**rinted **C**ircuit **B**oard
PHY	**Phy**sical Transceiver
PIP	**P**rogrammable **I**nterconnection **P**oint [XVI]
PLD	**P**rogrammable **L**ogic **D**evice
PLA	**P**rogrammable **L**ogic **A**rray
RAM	**R**andom **A**ccess **M**emory
RGB	**R**ed **G**reen **B**lue
RHWOS	**R**econfigurable **H**ardware **O**perating **S**ystem [WP03a]
RLU	**R**econfigurable **L**ogic **U**unit
RTOS	**R**eal **T**ime **O**perating **S**ystem
SPLD	**S**imple **P**rogrammable **L**ogic **D**evice
ST	**S**oftware **T**ask
SW	**S**oftware
TBUF	**T**riState **Buf**fer [XVI]
TCP	**T**ransmission **C**ontrol **P**rotocol (IEEE...)
TTL	**T**ransistor/**T**ransistor **L**ogic
UDP	**U**ser **D**atagram **P**rotocol (IEEE...)
UT	**U**ser **T**ask
VHDL	**V**LSI **H**ardware **D**escription **L**anguage
VLSI	**V**ery **L**arge **S**cale **I**ntegration
XF-Board	**XF**ORCES RHWOS Prototyping **Board** [WP04a, Nob03, Nob04]
XFORCES	Executives **for** **R**econfigurable **E**mbedded **S**ystems [Pla99]
YUV	Color-Model: **Y**=luminance, **U/V**=chrominance (also YCbCr, YPbPr)

Paper Summary

This section lists all the papers, ordered in categories *Journal Papers*, *Conference Papers*, and *Technical Reports* which we have published as a part of our research activity.

Journal Papers (reviewed)

Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-time Tasks [SWP04]

IEEE Transactions on Computers, November 2004

Todays reconfigurable hardware devices have huge densities and are partially reconfigurable, allowing for the configuration and execution of hardware tasks in a true multitasking manner. This makes reconfigurable platforms an ideal target for many modern embedded systems that combine high computation demands with dynamic task sets. A rather new line of research is engaged with the construction of operating systems for reconfigurable embedded platforms. Such an operating system provides a minimal programming model and a runtime system. The runtime system performs online task and resource management.

In this paper, we first discuss design issues for reconfigurable hardware operating systems. Then, we focus on a runtime system for guarantee-based scheduling of hard real-time tasks. We formulate the scheduling problem for the 1D and 2D resource models and present two heuristics, the *horizon* and the *stuffing* technique, to tackle it. Simulation experiments conducted with synthetic workloads evaluate the performance and the runtime efficiency of the proposed schedulers. The scheduling performance for the 1D resource model is strongly dependent on the aspect ratios of the tasks. Compared to the 1D model, the 2D resource model is clearly superior. Finally, the runtime overhead of the scheduling algorithms is shown to be acceptably low.

The Case for Reconfigurable Hardware in Wearable Computing [PEW+03]

Personal and Ubiquitous Computing, Special Issue, 2003

Wearable computers are embedded into the mobile environment of their users. A design challenge for wearable systems is to combine the high performance required for tasks such as video decoding with low energy consumption required to maximize battery run-times and the flexibility demanded by the dynamics of the environment and the applications.

In this paper, we demonstrate that reconfigurable hardware technology is able to answer this challenge. We present the concept and the prototype implementation of an autonomous wearable unit with reconfigurable modules (WURM). We discuss experiments that show the uses of reconfigurable hardware in WURM: ASICs-on-demand and adaptive interfaces. Finally, we present an experiment towards an operating system layer for WURM.

Conference Papers (reviewed)

Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform [WP02]

2nd International Conference of Engineering of Reconfigurable Systems and Architectures (ERSA'02)
Las Vegas (USA), June 2002

Partial reconfiguration allows for mapping and executing several tasks on an FPGA during runtime. Multitasking on FPGAs rises a number of questions on the management of the reconfigurable resource, which leads to concepts of reconfigurable operating systems.

This paper focuses on a major aspect of a reconfigurable operating system: task placement and transformation. We first discuss task characteristics and system models, and then concentrate on the execution of independent task sets on non-preemptive reconfigurable systems. We investigate placement techniques for non-rectangular, coarse-grained tasks and propose footprint transforms that change task shapes in order to find possible mappings. Finally, we discuss simulation experiments to evaluate these techniques.

Reconfigurable Hardware in Wearable Computing Nodes [PEW+02]

6th International Symposium on Wearable Computers (ISWC'02)
Seattle (USA), October 2002

Wearable computers are embedded into the mobile environment of the human body. A design challenge for wearable systems is to combine the high performance required for tasks such as video decoding with low energy consumption required to maximize battery run-times and the flexibility demanded by the dynamics of the environment and the applications.

In this paper, we demonstrate that reconfigurable hardware technology is able to answer this challenge. We present the concept and the prototype implementation of an autonomous wearable unit with reconfigurable modules (WURM). We discuss two experiments that show the uses of reconfigurable hardware in WURM: ASIC-on-demand and adaptive interfaces. Finally, we develop and evaluate task placement techniques used in the operating system layer of the WURM.

Online Scheduling for Block-partitioned Reconfigurable Devices [WP03a]

International Conference on Design, Automation and Test in Europe (DATE'03)
Munich (Germany), March 2003

This paper presents our work toward an operating system that manages the resources of a reconfigurable device in a multi- tasking manner. We propose an online scheduling system that allocates tasks to block-partitioned reconfigurable device. The blocks are statically-fixed but can have different widths, which allows to match the computational resources with the task requirements. We implement several non-preemptive schedulers

as well as different placement strategies. Finally, we present a simulation environment that allows to experimentally investigate the effects of specific partitioning, placement and scheduling methods.

Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing [WSP03]

17th International Parallel and Distributed Processing Symposium (IPDPS'03) / Reconfigurable Architectures Workshop (RAW'03)
Nice (France), April 2003

Partial reconfiguration allows for mapping and executing several tasks on an FPGA during runtime. Multitasking on FPGAs raises a number of questions on the management of the reconfigurable resources, which leads to concepts of reconfigurable operating systems. A major aspect of such an operating system is task placement. Online placement methods are required that achieve a high placement quality and lead to efficient implementations.

This paper presents placement methods that rely on efficient partitioning algorithms and a hash matrix as a data structure to maintain the free space. Given n as the number of placed tasks, Bazargan et al. presented a placer that finds a feasible location in O(n) time. Our approach is able to find a feasible location in constant time. Additionally, simulations show that our methods improve the placement quality by up to 70%.

Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations [WP03b]

3rd International Conference of Engineering of Reconfigurable Systems and Architectures (ERSA'03)
Las Vegas (USA), June 2003

In this paper, we approach the rather new area of reconfigurable hardware operating systems in a top-down manner. First, we describe a design concept that defines basic abstractions and operating system services in a device-independent way. Then, we refine this model to an implementation concept on the Xilinx Virtex XCV-800 technology. The Implementation concept proposes a multitasking environment that executes relocatable hardware tasks, uses a memory management unit translating task requests to internal and external memory accesses, and relies on device drivers and triggers to connect to external I/O. Finally, we present a detailed prototypical implementation of and an application case study. The application consists of a set of dynamically loaded and executed networking and multimedia tasks such as IP packet processing, AES decryption, and audio stream decoding.

Heuristics for Online Scheduling Real-time Tasks to Partially Reconfigurable Devices [SWP03]

13th International Conference on Field Programmable Logic (FPL'03)
Lisbon (Portugal), September 2003
Nominated for the Best Paper Award

Partially reconfigurable devices allow to configure and execute tasks in a true multitasking manner. The main characteristics of mapping tasks to such devices is the strong nexus between scheduling and placement.

In this paper, we formulate a new online real-time scheduling problem and present two heuristics, the horizon and the stuffing technique, to tackle it. Simulation experiments evaluate the performance and the runtime efficiency of the schedulers. Finally, we discuss our prototyping work toward an integration of scheduling and placement into an operating system for reconfigurable devices.

Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices [SWPT03]

24th International Real-Time Systems Symposium (RTSS'03)
Cancun (Mexico), December 2003

This paper deals with online scheduling of tasks to partially reconfigurable devices. Such devices are able to execute several tasks in parallel. All tasks share the reconfigurable surface as a single resource which leads to highly dynamic allocation situations. To manage such devices at runtime, we propose a reconfigurable operating system that splits into three main modules: scheduler, placer and loader. The main characteristics of the resulting online scheduling problem is the strong nexus between scheduling and placement.

We discuss a fast online placement technique and then focus on scheduling real-time tasks. We devise guarantee-based schedulers for two scenarios, namely tasks with arbitrary and synchronous arrival times. The schedulers exploit the knowledge about task properties to improve the system's performance. The experiments show that the developed schedulers lead to substantial performance gains at an acceptable runtime overhead.

XF-Board: A Prototyping Platform for Reconfigurable Hardware Operating Systems [WNP04a]

4th International Conference of Engineering of Reconfigurable Systems and Architectures (ERSA'04)
Las Vegas (USA), June 2004

We present the XF-BOARD, a prototyping platform for reconfigurable hardware operating system research. The platform is based on two tightly coupled FPGAs. The first (C-FPGA) implements a soft CPU core that controls the overall system; the second (R-FPGA) is used as dynamically reconfigurable hardware resource. Furthermore, a rich set of I/O and memory devices are available for implementing multimedia and networking applications.

A Runtime Environment for Reconfigurable Hardware Operating Systems [WP04a]

14th International Conference on Field Programmable Logic (FPL'04)
Antwerp (Belgium), August 2004

We present a runtime environment that partially reconfigures and executes hardware tasks on Xilinx Virtex-II devices. To that end, the FPGAs reconfigurable surface is split into a varying number of variable-sized vertical task slots that can accommodate the hardware tasks. A bus-based communication infrastructure allows for task communication and I/O. We discuss the design of the runtime system and its prototype implementation on an reconfigurable board architecture that was specifically tailored to reconfigurable hardware operating system research.

Technical Reports (not reviewed)

Reconfigurable Hardware OS Prototype [WP03c]

Swiss Federal Institute of Technology (ETH), TIK Report Nr. 168

In this paper, we approach the rather new area of reconfigurable hardware operating systems in a top-down manner. First, we describe a design concept that defines basic abstractions and operating system services in a device-independent way. Then, we refine this model to an implementation concept on the Xilinx Virtex XCV-800 technology. The Implementation concept proposes a multitasking environment that executes relocatable hardware tasks, uses a memory management unit translating task requests to internal and external memory accesses, and relies on device drivers and triggers to connect to external I/O. Finally, we present a detailed prototypical implementation of and an application case study. The application consists of a set of dynamically loaded and executed networking and multimedia tasks such as IP packet processing, AES decryption, and audio stream decoding.

XF-Board: A Prototype Platform for Reconfigurable Hardware Operating Systems [WNP04b]

Swiss Federal Institute of Technology (ETH), TIK Report Nr. 193

We present the XF-BOARD, a prototyping platform for reconfigurable hardware operating system research. The platform is based on two tightly coupled FPGAs. The first (C-FPGA) implements a soft CPU core that controls the overall system; the second (R-FPGA) is used as dynamically reconfigurable hardware resource. Furthermore, a rich set of I/O and memory devices are available for implementing multimedia and networking applications.

Implementation of a Runtime Environment for Reconfigurable Hardware Operating Systems [WP04b]

Swiss Federal Institute of Technology (ETH), TIK Report Nr. 195

We present a runtime environment that partially reconfigures and executes hardware

tasks on Xilinx Virtex. To that end, the FPGAs reconfigurable surface is split into a varying number of variable-sized vertical task slots that can accommodate the hardware tasks. A bus-based communication infrastructure allows for task communication and I/O. We discuss the design of the runtime system and its prototype implementation on an reconfigurable board architecture that was specifically tailored to reconfigurable hardware operating system research.

Case Study Applications for Reconfigurable Hardware Operating System Platform (XF-Board) [WW04]

Swiss Federal Institute of Technology (ETH), TIK Report Nr. 200

To test the basic infrastructure of the reconfigurable hardware operating system (RHWOS), we developed a set of demo applications using sawtooth wave generators and audio filters.

In this document we describe the structure, the implementation details and some pitfalls that will help you to understand and maybe even to extend these applications. For details about the OS part C-FPGA including the shell interface and the underlying hardware parts consult the documentation of the master thesis Reconfigurable Hardware OS Prototype - Part CPU written by Samuel Nobs and the master thesis Components and Services for RHWOS written by Kristofer Jonsson.

Author's Curriculum Vitae

Name, Prename:

Walder, Herbert H.

Date and Place of Birth:

April 3rd, 1969, in Zurich (Switzerland)

Education:

1990-1995	Studies in Electrical Engineering at Swiss Federal Institute of Technology (ETH), Zurich; Graduation as *Dipl. El.-Ing. ETH* in 1995
1984-1988	Mathematisch Naturwissenschaftliches Gymnasium (MNG), Zurich Matura Typus C
1976-1984	Primary and secondary school, Zurich

Current Position:

since 2005	Business Engineer at Swisscom Fixnet AG, Zurich (Switzerland)

Professional Experience:

2004	Technical Consultant for Oerlikon Contraves AG (Switzerland) and Rheinmetall Defence Electronics (Germany)
2000-2004	Research Assistant / PhD Student at Swiss Federal Institute of Technology (ETH), Member of the Computer Engineering and Networks Laboratory (TIK)
1998	Software Developer (Freelancer) for Ardani AG, Rümlang
since 1996	Several positions at Swisscom Fixnet AG, Zurich (Switzerland) Project Leader, Team-Leader in Network Management and Business Process Optimization
since 1995	Hardware/Software Developer and Technical Consultant (Freelancer) for F.E.M. AG, Spreitenbach (Switzerland)
1994	Lecturer for *Signal Analysis and Theory of Electrical Measurement Equipment* at TEKO Engineering School, Olten (Switzerland)
1993	Software Developer (Freelancer) for Samen-Mauser AG, Zurich
1989-1995	Hardware/Software Developer at I.L.E.E. AG, Industrial Laser and Electronic Engineering, Urdorf (Switzerland)

Swiss Army:

Rank	First Lieutenant (Oblt)
Functions	Telecommunication Officer (TC Of), Stab Ter Div Stabsbat 4 Instruction Officer, Uem UoS / RS II/262, Kloten (Switzerland)