# Real-Time Multitasking in Embedded Systems Based on Reconfigurable Hardware

**Dissertation**

A thesis submitted to the
**Faculty of Computer Science, Electrical Engineering and Mathematics**
of the
**University of Paderborn**
in partial fulfillment of the requirements for the
degree of *Dr. rer. nat.*

by

**Klaus Danne**

Paderborn, Germany
date of submission: 11.9.2006

*Supervisors:*

Prof. Dr. Franz J. Rammig
Prof. Dr. Marco Platzner

*Reviewers:*

Prof. Dr. Franz J. Rammig
Prof. Dr. Marco Platzner
Prof. Dr. Daniel D. Gajski

*Additional members of the oral examination committee:*

Prof. Dr. Ulrich Rückert
Dr. Ulf Lorenz

*Date of submission:*

September 11, 2006

*Date of public examination:*

November 22, 2006

# Acknowledgements

This PhD thesis was carried out during my work at the University of Paderborn in Germany as a member of the *Design of Parallel Systems* group under Prof. Dr. Franz. J. Rammig, from July 2002 to June 2005, and as a member of the *Computer Engineering* group under Prof. Dr. Marco Platzner, from July 2005 to November 2006. The work was partially supported by the German Research Foundation (Deutsche Forschungsgemeinschaft) in the research training group 776 *Automatic Configuration in Open Systems*.

I would like to thank Prof. Rammig for supervising my thesis, supporting my work as member of his group, and sharing in his great knowledge concerning embedded systems and computer science research in general. Likewise, I thank my co-supervisor Prof. Platzner. He employed me as a researcher just shortly after he became professor at Paderborn University and supported a great part of my publications as co-author. In addition, my thanks go to Prof. Dr. Daniel D. Gajski, who was a third reviewer as well as Prof. Dr. Ulrich Rückert and Dr. Ulf Lorenz, who were additional members of the oral examination commission.

Further, without mentioning everyone personally, I would like to thank many colleagues that helped me during my time at Paderborn University:

Prof. Dr. Christophe Bobda as a coauthor of several publications and for proofreading my thesis; my teammates Dr. Stefan Ihmor and Florian Dittmann for constructive and interesting discussions; Tales Heimfahrt and Marcelo Götz for teaching me the Brazilian thinking and fixing my Linux PC; my new teammates Paul Kaufmann, Enno Lübbers and Heiner Giefers for all the discussions. I am also grateful to Tarek Zeineldin and our working group visitors Prof. Dr. David Andrews and Jason Agron for proofreading the thesis. Further acknowledgments go to Dr. Heiko Kalte for co-authoring publications and to Roland Mühlenbernd and Martin Kreis for supporting my research as student assistants and by their bachelor's and master's theses respectively.

Finally, I would like to thank my parents for their great support during my education, my sister for being a good friend, and my lovely girlfriend Sonja for stepping into my life.

Klaus Danne, December 2006

# Contents

# List of Figures

# List of Tables

# Abstract

This thesis presents fundamental work in the new area of multi-tasking on reconfigurable hardware devices (RHDs) under real-time conditions.

RHDs can now execute several hardware-tasks (computations implemented as digital circuits) in parallel due to the increasing logic capacity, as well as sequentially due to runtime reconfiguration capabilities. To use RHDs for real-time workloads of embedded system applications, scheduling techniques and execution environments that create predictable task timings are required.

Specifically, we consider the problem of scheduling periodic real-time tasks for execution on shared RHD resources, which makes the problem different from single- and multiprocessor scheduling. In our first model, tasks are modeled by their resource requirements (area), their inter-arrival period and their computation time. Tasks can be executed in parallel as long as their accumulated area does not exceed the device area and can be preempted at any time. We develop three novel *earliest deadline first (EDF)* based scheduling methods and evaluate their performance.

1. The *global EDF* scheduler assigns device resources to all active tasks globally. For this algorithm we present a *linear-time scheduling test* which proves for a given task set at design time that it will be scheduled without missing any deadlines.
2. The *partitioned EDF* scheduler divides a task set at design-time into several subsets. Each subset is scheduled independently according to the EDF rule onto separate device resources. The feasibility of the schedule is achieved by considering the single processor EDF bound during partitioning. Integer linear programming (ILP) is used to compute the optimal partitioning while a next fit heuristic computes close to optimal results for even large task sets.
3. *MSDL* is a server based scheduling method which groups tasks into periodic servers for parallel execution. At runtime, servers are loaded onto the RHD and executed according to EDF. Only one server is running at a time, making this algorithm suitable for RHDs which do not support partial reconfiguration.

In our analytic performance comparison we show that there is no dominance among the three approaches. However, our simulation studies show that, on average, global

EDF outperforms partitioned EDF, which outperforms MSDL. Further we showed that the global EDF scheduling test is rather pessimistic. We analyze the reconfiguration overheads for all three approaches and include them within the schedulability tests. Simulation experiments show that *global EDF* suffers significantly more from this overhead than the other schedulers.

Two extensions of our task model are presented in order to consider the specific characteristics of reconfigurable applications: First we consider tasks for which not only one but several alternative implementations (circuits) exist. We present an ILP model to select the optimal implementations for the partitioned EDF scheduler, which considerably improves the scheduling performance. Second, we consider tasks memory usage under real-time constraints. We present a method for efficiently sharing RAM-banks without jeopardizing task deadlines, thus reducing the required memory resources.

Finally we describe our prototype FPGA based real-time kernel which, in contrast to others, is a hardware-only system with the operating system functions entirely implemented in hardware. Our prototype implements MSDL, uses full device reconfiguration, and is thus suitable for most of today's reconfigurable devices.

In summary, the thesis presents a foundation for real-time multitasking on RHDs, including models, algorithms, analysis and prototypical implementation.

# CHAPTER 1

# Introduction

The aim of this thesis is to enable the use of runtime reconfigurable hardware for computing time-critical tasks in embedded systems.

Besides circuit prototyping and glue-logic replacement there are two major application domains where reconfigurable hardware devices (RHDs) are most useful: *High Performance Computing* and *Embedded Systems*.

In high performance computing, RHDs can be used to speed up special applications. Example systems which have achieved enormous speedups by using pure *field programmable gate array* (FPGA) solutions or FPGAs as co-processors of host computers can be found in [31][95]. Cray's XD1 machine [34] is a commercial example for super-computing using XILINX FPGAs. Also not the application domain targeted by this work, several of the methods explored within this thesis may be equally applicable within high performance computing systems.

In the application domain of embedded systems, RHD based platforms are gaining in popularity and are sometimes preferred to microprocessor based systems because they can:

- respond to input values within very short latencies (shorter than software solutions),
- achieve high throughputs in data flow driven applications (parallelism within a task),
- process many computations in parallel without decrease in performance (multitasking in space),
- communicate to peripheral devices via high number of user definable I/O pins.

Applications within the embedded systems domain include *network-* and *communications systems* such as Ethernet switches, *data processing applications* and *industrial measurement systems* like digital logic analyzers, *military and aerospace systems* as for example satellites, *digital imaging and video systems* such as digital cameras, *consumer electronics* such as audio players, *automotive applications* such as Siemens car entertainment system VDO Dayton and many more [45][55][4].

The applications mentioned above share some common key characteristics:

- they contain computations which are mostly regular, data flow oriented and include digital signal processing, which can be efficiently computed on RHDs,
- they can be complex, i.e. they may consist of several different computational tasks,
- they have to react within precise time constraints to their environment, i.e., they are *real-time* applications.

Supporting the design of such characterized embedded applications for execution on RHD platforms is the aim of this thesis.

## 1.1 Motivation to Real-Time Multitasking

Multitasking is useful if the application is composed of several well separated computational tasks that should be processed quasi simultaneously on limited hardware resources. The concept of multitasking, in combination with the corresponding operating system (OS) support, enables efficient *sharing of resources* due to adequate task scheduling, allows *dynamic changes* of the application due to introduction of new tasks during runtime and greatly *simplifies application development* since tasks can be developed quite independent from each other.

As further motivation, we list several example tasks with real-time constraints which appear in real-world applications. Most of the mentioned applications come from the automotive domain. Some refer to only conceptual systems and some to existing implementations.

**Engine control system:** A task controlling the engine such as the gas injection and ignition process. Indeed this task may include several sub-controller tasks. The computation of these sub-tasks is based on the cyclic evaluation of the differential equation systems of the corresponding mechatronic controllers. Kopetz describes such example controller in [64] p. 22.

**Vehicle cruise controller:** In [8], Axelsson from Volvo Technological Development describes a case study of an advance cruise controller for cars. It keeps the car at a desired speed, but also uses a forward looking radar to avoid collisions with obstacles. The system is composed of several tasks, such as: *wheel-speed measuring*; *speed control* which adapts the speed to the desired value; *driver-interaction* which checks for break and throttle pedals; *obstacle-detection* which looks for obstacles ahead; *brake-control* and *throttle-control* which perform deceleration and acceleration.

**Radar system:** Several upper class cars use a radar system for measuring the distance to the cars driving ahead [32]. Such long distance control is used by the cruise control. The implementation of radar signal processing on FPGAs is issued in [6].

**Camera driving assistance:** The company MobilEye[1] develops several car driver assistance applications based on camera image processing, among them *pedestrian detection*, *headway monitoring and warning*, *forward collision warning*, *night vision* and many more [54]. For the detection of lane marks and vehicles, images are processed through a chain of various processes: *1. selecting candidate regions of interest*, *2. single frame classification*, *3. multi-frame approval* and *4. range measurement*. To achieve the required real-time image processing performance of 30 frames per second, the company developed a system on chip (SoC) which hosts two RISC processors and dedicated video processing units and runs at 120 MHz.

All of these tasks have a couple of common features. They are based on computations which can be efficiently implemented on RHDs, have to be evaluated periodically, and are subject to hard real-time constraints. Such systems are targeted within this thesis.

### Challenges

When designing embedded systems, a key challenge is to implement all required functionalities, meet the required performance and time constraints, but at modest cost. Hence, the system resources have to be used efficiently. For processor based systems, operating systems support resource sharing via multitasking, and appropriate scheduling algorithms ensure the efficient utilization of shared resources while meeting all real-time constraints.

The more reconfigurable architectures push into embedded systems and take over computations formerly implemented on processors or in application specific integrated circuits (ASICs), the more important it becomes to apply the same techniques to RHDs in order to use their resources efficiently. RHDs increase the existing challenges in the following three ways:

1. *Design of novel reconfigurable architectures:* Most of today's available commercial reconfigurable architectures support dynamic reconfiguration only in a limited fashion. The reconfiguration time of FPGAs ranges from some hundred microseconds to tens of milliseconds, which makes runtime reconfiguration infeasible for applications requiring very short response times. Also partial reconfiguration, which allows reconfiguring a part of the device resources for a new task while other parts keep executing other tasks, has only limited support. Moreover, the architectures do not provide native support for storing and restoring the state of a task before / after interruption, which is a requirement for preemptive multitasking.

Much promising research is currently being performed to overcome these drawbacks, including coarse-grain RHDs and other devices with multiple configuration-context memories on chip [96][31][50] (see Section 2.1).

2. *Runtime environments for reconfigurable devices:* Multitasking on RHDs requires some kind of support from a runtime or operating system, similar to an OS for processor based platforms. Such a system needs to manage the RHD resources, control the execution of hardware-tasks and control the device reconfiguration at runtime. Moreover, such an OS has to provide programming interfaces for the application developer to abstract details from the underlying hardware.

   Several prototypes of operating systems for FPGAs have been developed by research groups, but still are far behind their counterparts in the processor world [102][57][105][85].

3. *Task scheduling and placement:* Scheduling and placement methods are required, which allow the OS to efficiently utilize RHD resources for the application tasks and, if needed, deal with real-time requirements. Since the task and resource models of single- and multiprocessor systems are different from those of reconfigurable devices, their scheduling methods cannot be directly applied for RHDs. Moreover, tasks scheduling for RHDs under real-time constraints has been rarely addressed by researchers (e.g. in [80][90]). Scheduling problems for RHDs are much less understood and only a few proposed scheduling approaches exist compared to the classical scheduling problems for processor and multiprocessor platforms.

   Hence, there is a considerable need for research in this area in order to catch up with the state of the art of scheduling methods for processors.

## 1.2 Contribution of This Thesis

The main contributions of this thesis are related to the third challenge mentioned above, namely *(real-time) scheduling methods for reconfigurable architectures.* For the well known problem of scheduling periodic real-time tasks, we present one of the first work considering runtime reconfigurable architectures. Three basic scheduling methods are developed and evaluated. Two are driven by the basic (multi)-processor scheduling methods and the third is specially designed for devices without partial reconfiguration support.

A second contribution is given in the area of operating systems for reconfigurable architectures. As a proof of concept, a prototype of an *FPGA based real-time kernel (FBRK)* has been implemented. In contrast to other work, it is completely implemented in (re-configurable) hardware and designed for full- rather than partial reconfigurable devices. It implements the developed scheduling strategy for devices without partial reconfiguration support and hence is suitable for a wide range of today's devices and platforms.

The contributions in the field of real-time scheduling methods are described more precisely in the following list:

- A new task and resource model for reconfigurable architectures as a generalization of a simple multi-processor model, and definition of the periodic real-time tasks scheduling problem within this context. For this problem, we introduce three independent scheduling strategies based on *global scheduling*, *partitioned scheduling* and *server based scheduling*. We develop a *linear-time* scheduling test for the *global scheduling* strategy, and introduce and analyze optimal and heuristic algorithms for the *partitioned scheduling* and *server based scheduling* strategy.

- The performance of all three strategies is compared analytically and by simulation studies. We show that the three approaches do not dominate each other in performance. Simulation results show, that for many cases the *global scheduling* strategy achieves high device resource utilization of over 90% and outperforms the *partitioned scheduling* strategy. In contrast, when a scheduling test is required to predict that no deadline will be missed, significantly higher resource utilization is achieved by the *partitioned scheduling* strategy compared to the *global scheduling* strategy.

  As the reconfiguration time cannot be neglected, the associated overhead is included within the analysis of all three strategies. Main results state that the overhead is bounded and in many cases stays reasonably small, if the device configuration time is less than 10% of the runtime of the shortest task. Furthermore, the *partitioned scheduling* and *server based scheduling* methods suffer much less from this overhead than the *global scheduler*.

- In order to deal with the specific characteristics of hardware tasks and reconfigurable platforms, we generalize and extend our model. The first generalization allows considering tasks, for which not only one but several alternative circuit implementations exist. For such applications, the optimal variant selection is computed by integer linear programming, which improves considerably the performance of the *partitioned scheduler* compared to tasks without variants.

  Secondly, we extend our model to consider tasks' memory usage under real-time constraints. We present a method to efficiently share memory banks of the platform among tasks without jeopardizing task deadlines.

In summary this thesis presents a foundation for real-time multitasking on RHDs, presenting the models, algorithms, analysis and a prototypical implementation.

## 1.3 Chapter Outline

The thesis has the following organization:

Chapter 2 *Background and Related Work* provides a background to this thesis and summarizes the related work. It discusses the different kinds of reconfigurable hardware architectures that we consider as processing devices throughout this work. The concept of hardware-tasks is introduced and previously reported multitasking models for such tasks are reviewed and classified. Finally, the chapter presents related work in real-time scheduling, including single and multiprocessor scheduling algorithms for periodic tasks and the few research on real-time scheduling for RHDs.

Chapter 3 *Problem Modeling and Metrics* introduces the task and resource models and the basic notations used within this work. The general problem of scheduling periodic real-time tasks onto an RHD is defined.

Chapter 4 *Three Scheduling Algorithms* presents our main methods to schedule periodic task onto RHDs, namely *global EDF scheduling*, *partitioned EDF scheduling* and *server based scheduling*. The scheduling conditions and basic analysis are developed.

Chapter 5 *Comparison of Algorithm Scheduling Performance* evaluates the quality of our three proposed methods. Only the pure scheduling performance without considering any reconfiguration overhead is compared. The first part presents an analytic comparison and derives qualitative statements. The second part presents results from numerous simulation experiments, showing how the algorithms perform in average.

Chapter 6 *Execution Model and Overhead Analysis* goes away from the idealistic task model and accounts for execution time overheads. For each scheduling method a suitable reconfiguration model and its runtime system requirements are described. The time overhead resulting from device reconfigurations is included in the scheduling conditions of each algorithm. Finally, the results from simulation experiments on the algorithm performance including overheads are shown.

Chapter 7 *Model Extensions* takes special characteristics of hardware-tasks into account. A first part presents methods for a generalized task model that allows specifying not only one but several implementation variants per tasks - each with a different area and computation time. The second part captures the memory requirements of hardware-tasks. A method for sharing physical platform memories between parallel executed task is shown which has the potential to considerable reduce the memory requirements.

Chapter 8 *Prototype: FPGA Based Real-Time Kernel* describes our realization for the server based scheduler. The system architecture and an automated synthesis flow are described. Finally, results on the logic requirements and timings are reported.

Chapter 9 *Conclusion and Outlook* finalizes the thesis with a summary, draws conclusions and presents suggestions for future research.

# CHAPTER 2

# Background and Related Work

This chapter provides a background to this thesis and summarizes the related work.

The first section discusses the different kinds of reconfigurable hardware architectures that we consider as processing devices throughout this work. It gives a background to fine and coarse grain reconfigurable devices, the different reconfiguration modes and discusses the architecture of typical embedded systems based on such devices. Since reconfigurable devices and computing systems have been surveyed several times (e.g. in [31] p. 11 ff.), we keep this section therefore small.

In Section 2.2, we briefly introduce the concept of hardware-tasks, which refers to computational tasks implemented as digital circuits for execution on an RHD. By means of an example of a hardware task, several characteristics such as resource requirements and execution times are discussed.

The third section concentrates on multi-tasking on reconfigurable hardware. In order to be able to develop a reasonable task and resource model for our work (Chapter 3), six different device resource sharing models, that have been used in the literature, are discussed and classified (Section 2.3). We clear out why only two of the six models will be considered in our scheduling approaches. Afterwards, we present related work that has specifically targeted the problem of scheduling tasks on RHD architectures. While this documents the attraction of the research field, it shows that most work does not consider tasks with real-time constraints. Finally the section closes with a brief description of projects that have implemented reconfigurable systems supporting hardware multitasking. It shows that various prototypes exist, which support execution environments as they will be assumed in our task and resource model of Chapter 3.

The last part of this chapter presents related work in real-time scheduling (Section 2.4). We introduce the periodic task model and clarify its importance, since it is used as a basis for our task model. Furthermore, we present the basics of single processor scheduling algorithms, since they are also fundamental to all scheduling methods for parallel architectures. Scheduling methods for multiprocessors are discussed in more detail, since the multiprocessor model is closely related to our RHD resource model and, furthermore, the basic techniques of multiprocessor scheduling have been inspired two of our three scheduling methods which will be introduced in Chapter 4. Finally, we close with the few work considering real-time scheduling for RHDs or similar architectures. We conclude, that the problem considered in this work is actually novel, but as well a straight forward extension of the basic single- and multiprocessor scheduling problems. Furthermore, the approaches we use are based on scheduling methods with strong foundation in literature.

## 2.1 Reconfigurable Hardware Devices

This section gives a background on different types of *reconfigurable hardware devices (RHD)*, their inner architecture and how they are used to implement computations.

The most prominent type of RHD is the field-programmable gate array (FPGA). It was invented in the eighties and is a further development of early programmable logic devices. Even if the FPGAs where mainly invented for prototyping of digital circuits and for implementing glue logic, researchers got attracted by the feature of runtime reconfiguration (or runtime reprogramming) of these devices, which allows general FPGA based computations. Since then, the research field *Reconfigurable Computing* [31][46] was born, which focuses on new RHD based computing paradigms beside the Von Neumann paradigm as well as on the invention of new RHD architectures to overcome the drawbacks of commercial available FPGAs.

First we will take a closer look to the architecture of RHDs which can be classified into *fine-grained* architectures, which are basically FPGAs and *coarse-grained* architectures, which are mostly academically proposed devices or prototypes from startup companies.

### 2.1.1 Fine-Grain RHDs (FPGAs)

The term *fine-grain* is used to denote devices, which can be configured on bit-level. We use *fine-grain RHD* as a synonym for FPGA.

The basic architecture of today's FPGAs is shown in Figure 2.1 and consists mainly of an array of programmable *logic blocks*, programmable *interconnect* as well as general purpose *input output cells*. The logic blocks can be configured to implement small logic functions and registers, which can be composed to larger digital circuits by the programmable interconnect. A typical structure of a logic block is shown in Figure 2.2. Its main components are *look-up tables* (LUTs) to implement combinatorial logical and flip flops to implement storage. An $n$ input LUT is an $n$-address memory, which can store the $2^n$ possible values of a Boolean function with $n$ input variables. To implement a specific Boolean function, the LUT is programmed with the function's truth table. The input values are used to address the LUT and the data output represents the value of the function. The output of the LUT is either fed into a flip flop of the logic block or directly

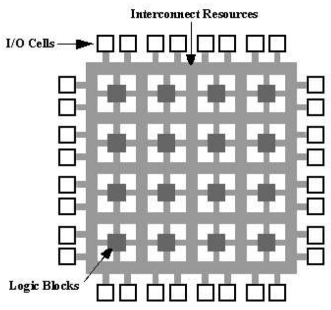


Figure 2.1: Basic structure of an FPGA

connected to the logic block's output. Typically, 4-input LUTs are used, since many studies show that they present a good tradeoff for most digital designs [2]. Recently, Xilinx switched from 4- to 6-input LUTs in their latest Virtex-V architecture [33]. Altera uses in their Stratex-II architecture adaptive logic modules with a configurable LUT size, which can be used e.g. as one 7-input LUT or as two 4-input LUTs [3].

The flip flops of the logic blocks are used to implement the storage registers inside a digital design. They are connected to a global clock network and support features like synchronous or asynchronous reset.

The programmable interconnect consists of configurable crossbar switches, which allow establishment of connections between outputs and inputs of the logic blocks as well as to the I/O cells.

Also these elements are all what is needed to implement a digital design, modern FPGAs include many more types of resources to achieve better performance and efficiency. Typical resources of the market leader FPGA vendors are briefly described below.

**Block RAMs** are dedicated blocks of static RAM of medium size (typical with a size of 18 kBit), which can be used to store larger amounts of data onto the device. In contrast to flip flops which can be accessed independent of each other, the data

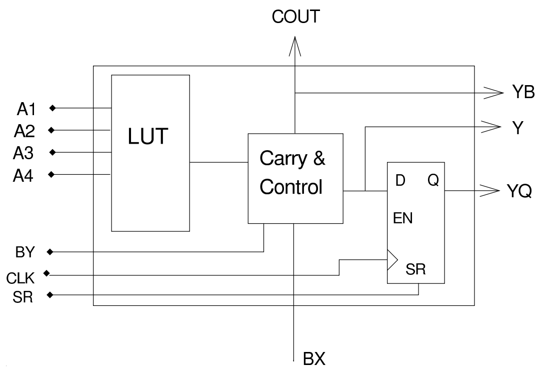


Figure 2.2: Structure of FPGA logic block (source: [106])

of each block RAM memory may only be read or written sequential on a word by word basis. (Some devices provide dual ported block RAM.)

**Carry logic** is a special logic inside the logic blocks to forward carry bits to improve the performance of arithmetic functions such as adders.

**DSP cores** are dedicated functional units like multiplier and multiply-accumulate structures to improve the performance of digital signal processing functions.

**Processor cores** are dedicated microprocessor cores hosted on the FPGA chip. They are used for complex system-on-chip designs which include software and hardware parts.

**High speed I/O transceivers** are dedicated transceiver circuits which implement several high speed I/O standards to enable serial I/O of several GBit per second.

**Clock managers** are units, which allow generating and distributing clock signals with minimal skew throughout the FPGA chip. Modern FPGAs support multiple global clock networks enabling multi clock designs.

**Tri state buffers** are components, to support on chip communication bus systems with multiple clients.

FPGAs are typically programmed using an ASIC-like design flow. Design entry is usually provided in a hardware description language like VHDL or VeriLog. The design flow includes the synthesis of the design into a net-list, mapping the net-list to the components provided by the FPGA and a place and route step, in which components are allocated to specific FPGA resources and connected via the configurable interconnect. Based on the result a programming file for the FPGA, also called a bit-stream, is generated which can be directly downloaded into the FPGA configuration memory.

### 2.1.2 Coarse-Grain RHDs

Like FPGAs, coarse-grain RHDs are based on an array of programmable logic blocks called *processing elements* and *programmable interconnect*. In contrast to the fine-grain FPGAs, coarse grain RHDs provide programmability on word-width level of e.g. 8, 16, 24 or 32 bits [50][35][56][13]. The processing elements are not based on LUTs anymore, but provide arithmetic operations like addition, multiplication and shift on word level operands. Therefore, sometimes these devices are referred to as ALU arrays. The LUT approach, which allows the implementation of general functions, is not feasible on word width level: Imagine a processing element with two 16-bit inputs and one 16-bit output. Enabling programmability of a general logic function would require 16 times a 32-bit input LUT which results in $16 \times 2^{32}$ bit = 8 GB of memory for each processing element.

The operation on word-width level gives the coarse-grain RHDs several advantages compared to FPGAs: The arithmetic functions are implemented as dedicated circuits, which makes them much smaller in terms of chip-size and faster than functions implemented on FPGA logic blocks. Also the crossbar switches become simpler, as not single signals but entire buses of signals are connected at once. Since a coarse-grain RHD consists of much fewer processing elements than an FPGA of comparable chip size, the configuration memory is dramatically reduced. E.g. to program the device to implement a 16 bit multiply-accumulate unit, only two processing elements and the crossbar in between have to be configured, compared to tens or hundreds of logic blocks and crossbars that have to be configured when the same function is implemented on an FPGA. This reduction of configuration memory is directly translated to higher reconfiguration speed. Typically, coarse-grain devices can be reprogrammed several times as fast as FPGAs.

The coarse-grain RHDs gain these benefits at the cost of sacrificing some flexibility, which makes them inefficient when the problem does not mainly consists of word-width arithmetic operations. For example an *add* operation of 3-bits operands could occupy an entire 32 bit processing element.

### 2.1.3 Configuration Memory and Reconfiguration

The current function and behavior of the RHD is determined by the content of its configuration memory. This memory consists of static RAM memory cells, which are distributed on the device array. For each logic block of an FPGA, there are configuration memory cells storing the content of the LUT, the information about how the flip flop is connected, its initial state and its reset mode. The configuration memory of the processing elements of coarse-grain devices basically determines the function of the ALU. The configuration memory of the crossbar switches determines the connections of the logic blocks or processing elements inputs and outputs to routing channels as well as which horizontal and vertical routing lines are connected.

Since the configuration memory consist of SRAM cells, it can be programmed arbitrarily often, which does not only allows the definition of the function of the RHD after the chip production, but also changing it online during runtime. Therefore, we distinguish between *static* configuration and *dynamic* reconfiguration. Moreover, some RHDs allow reprogramming some parts of the configuration memory and therefore to change the functionality of only some logic blocks or processing elements respectively, while others keep operating. Therefore, we distinguish between *full*- and *partial* device reconfiguration.

#### Architecture of Configuration Memory

For physical reasons, the SRAM cells of the configuration memory have to be local to the elements they are configuring.[1] A simple and area efficient method to make the configuration memory accessible from outside the device is to connect all its cells to a long shift register. The RHD can be reprogrammed by sequentially shifting new configuration bits into the configuration memory until the entire device is reconfigured. Obviously this addressing schema allows only full configuration and the device cannot be used during the time of the reconfiguration process. Also the reconfiguration time is

[1]It is inefficient in terms of path delay and chip area to rout long connection lines between configuration memory cells and the elements to configure.

very long, since only 1-bit per clock cycle can be written.

These drawbacks can be overcome by a more sophisticated configuration memory architecture. The width of the configuration interface can be enhanced, which allows to reprogram several bits in parallel. The configuration memory can be made partial accessible by allowing addressing certain portions of it at the cost of extra chip area for address decoding. For example the FPGAs of the Xilinx Virtex and VirtexII families allow reprogramming entire columns of logic blocks, which allows a partial reconfiguration of the FPGA at a rather coarse level.

Reconfiguration times of coarse-grain reconfigurable architectures are typically much shorter than that of FPGAs, since they have less cells and hence a much smaller configuration memory. A further approach to overcome the long reconfiguration time are *multi-context* devices [31][35][96]. These devices have not only one but some few configuration memories for all device resources on chip. While this adds considerable cost in terms of chip area, it allows changing from one configuration to another within few clock cycles.

Within our resource model of Chapter 3 we consider only the homogeneous cells of an RHD and neglect additional resources such as I/O pins, DSP functions and embedded microprocessors. How such additional resource can be included within our model and our scheduling methods will be discussed in the future work section of Chapter 9. Dedicated memory blocks of the RHD and external SRAM banks are considered in our extended model of Section 7.2.

### 2.1.4 Architecture of RHD Based Computer Platforms

By the term RCP *(RHD based computer platform)* we denote an embedded computer system, whose main computational resources consist of RHDs (see Figure 2.3). The main computational tasks of the applications are executed on the RHD, e.g. by programming the device with digital circuits compiled as FPGA configuration bit-streams. Microprocessors are optional and their usage is not given special consideration within this thesis. However, microprocessors may be included within the system as separate chips external to the RHD, or integrated as dedicated circuits within the RHD (e.g. Xilinx VirtexII Pro [108]) or as soft core CPUs[2], which are implemented out of RHD logic cells (e.g. the Xilinx Microblaze or the Altera NIOS).

Beside the RHDs, the system includes memory to store the RHD programming data as well as application data and peripheral I/O devices to communicate with the environment. Additionally, a device controlling the reconfiguration process of the RHD is included. Concrete examples for FPGA based reconfigurable computing platforms are the RC100, RC200, RC300 from CELOXICA [27], the XF-board [98] from ETH Zurich, the Erlangen Slot Machine [20], or development boards distributed by XILINX. Some platforms are module based and can include several FPGA such as the RAPTOR 2000 system [62]. An example of a system including a coarse-grain RHD is the DAPDNA-EB4

[2]A soft core CPU is a CPU which is implemented as FPGA design.

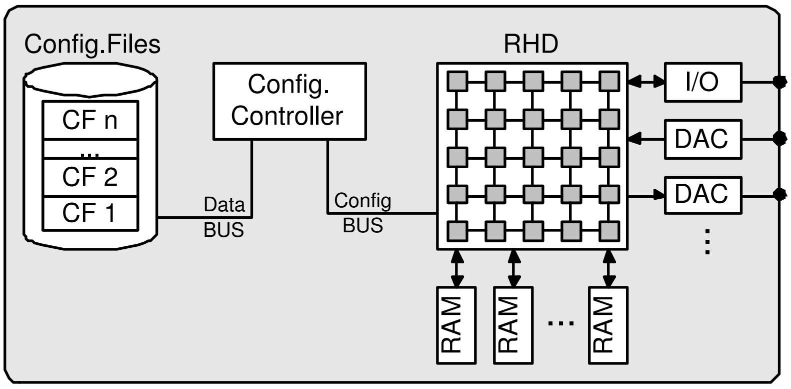


Figure 2.3: Architecture of an RHD based platform

board of the IPFlex company, which includes two DAPDNA-2 reconfigurable processors consisting of 376 processing elements each [56].

As for the memories, the platforms typically host several SRAM (static RAM) banks which are directly connected to the RHD. They are used to store processing data to which relatively fast access with predictable timing is required, but which is too large to be stored in on-chip memories. Additionally non-volatile memory such as flash memory is included to store the configuration data (programming files) and other persistent data. Larger but slower DRAMs (dynamic RAM) are included to store large blocks of application data, which is accessed with subject to weaker timing constraints.

RCPs often have a wide set of peripheral I/O. This includes for example digital-analog and analog-digital converters, video and audio interfaces such as S-VIDEO or VGA, standard interfaces like PS2, USB, fire-wire, network interfaces such as Ethernet or CAN, and user definable digital I/O.

In the models and methods used in this thesis we abstract from platform details and consider as platform resources only the logic blocks or processing elements of the RHD, the internal and external SRAM memory banks, the memory hosting the RHD configuration data as well as the bus connections between these resources. We neglect several additional resources and details for the following reasons:

- Details have to be abstracted, to keep the model valid for wide class of RCPs from different vendors as well as to keep the model complexity low.
- Microprocessors are not included within the model, since we assume they are used only for controlling the execution and reconfiguration of the RHD, not for executing the application tasks themselves. Such hybrid systems, which couple microprocessors with RHDs and use both for computation (e.g. [56]) open the issue of hardware–software co-design[92][75]. That is another independent research field and its topics are not addressed within this thesis.
- An RCP according to our model has only one RHD. From a practical point of view, multiple RHDs appear mostly in high-performance computing systems such

as Cray's XD1 [34] with the aim to get highest computational power at any price. In contrast, in embedded system design the challenge is rather to choose from the wide range of different size RHDs the smallest one that satisfies the application requirements.

From a theoretical point of view, multiple RHDs open issues which are stronger related to parallel processor systems than to the special characteristics of RHDs. Such issues are for example clustering and partitioning of applications and localization of communication, which have been exhaustively studied for multiprocessors but are not addressed within this thesis.

## 2.2 Hardware Tasks

In our application model of Chapter 3 and later on in the enhanced application models of Chapter 7, we will make several assumptions for the characteristics of application tasks. To give a better idea of what kind of tasks we consider and to show that the assumptions we make are reasonable, this section presents a short case study of an image processing task based on *Discrete Cosine Transformation (DCT)* computations.

**Example 1.** *We assume an image processing system which includes a task that computes the DCT coefficients of the rows of an image. The data to be processed has a fix size of N samples, each coded by 16 bits on which an 8 point one dimensional DCT should be performed.*

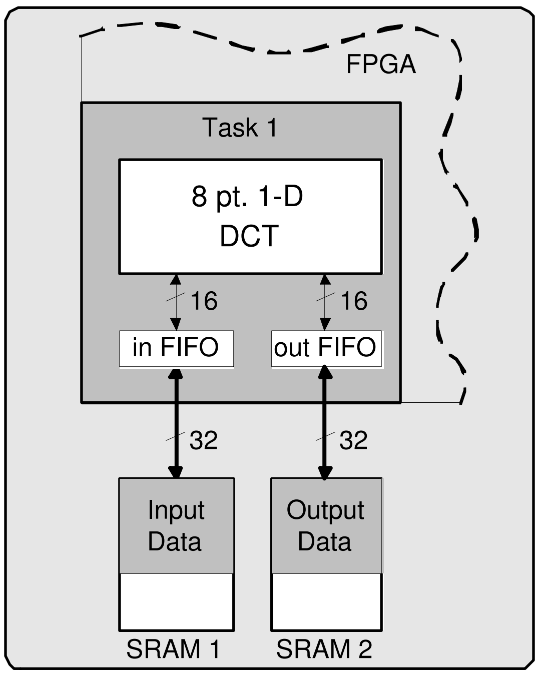


Figure 2.4: Example of a DCT task on platform

*Figure 2.4 illustrates how this task is realized on an FPGA based platform. Since the image data may be too large for FPGA internal block RAMs, the N input samples as well as the output coefficients to be computed are stored in the external SRAM banks 1 and 2 respectively. For the actual DCT computation a XILINX 1-D DCT IP-core [107] is used. Since the external RAM banks have a word width of 32 bits but the samples are 16 bit values, input and output FIFOs*[3] *are introduced to buffer the input samples and the produced output coefficients respectively and to convert between the 32 and 16 bit words.*

| variant | FPGA area | throughput | ext. RAM port access | | | exec. time |
|---|---|---|---|---|---|---|
| | (slices) | (sample/clk) | period | amount | utiliz. | $N$=1024 |
| 1 | 982 | 1 | 2 clks | 1 ×32 bit word | 50% | 1028 clks |
| 2 | 767 | 8 / 9 | 9 clks | 4 ×32 bit words | 45% | 1185 clks |
| 3 | 589 | 8 / 17 | 17 clks | 4 ×32 bit words | 24% | 2232 clks |

Table 2.1: Size, throughput and memory access pattern of several variants of a XILINX 8 point 1D-DCT IP core [107].

*As reported in the DCT IP-core data-sheet [107], the designer can choose between several implementation variants with a trade-off between FPGA area and throughput as shown in Table 2.1. To guaranty that the DCT core can achieve these throughput values – which is mandatory for real-time processing – the FIFOs have to be chosen of appropriate size and a certain access rate to the external RAMs has to be guaranteed. The required access rate to external memory is reported in the middle column of Table 2.1. E.g. variant 1 processes one 16-bit sample per clock cycle. Therefore, within a period of 2 clock cycles the task reads one 32 bit word from SRAM1 into the input FIFO. In contrast variant 3 has a throughput of* $8/17$. *Hence it reads within a period of 17 clock cycles 4 data words from the external RAM. Note that since a FIFO buffers the data it does not matter in which exact clock cycles the data is read from the RAM - as long as 4 words within a period of 17 clocks are read.*[4] *Thus, the RAM access pattern of the task has the nature of periodic real-time access: At the beginning of a period it requests to read/write a certain amount of data from/to the external RAM bank into/out of the FIFO. The deadline for this transaction is the beginning of the next period to prevent an underflow or overflow of the FIFO buffers.*

*The total worst case execution time of the DCT example task is computed using Equation 2.1. Beside the internal latency of the DCT core, two times the RAM access period is added to account the initial filling of the input FIFO and the final deflating of the output FIFO.*

$$exectime = \frac{N}{throughput} + DCTlatency + 2 \times RAMaccessperiod \qquad (2.1)$$

*For a data size of* $N = 1024$ *samples, the execution time according to Equation 2.1 for*

[3]first in first out buffers

[4]The same rates apply for the write access to the SRAM2.

*each of the three variants of the task is reported in Table 2.1 as well.*

This case study of a DCT task showed that it is reasonable to assume application models where tasks have the following characteristics:

- Tasks may be realized as various implementation variants with different performance vs. RHD-resource tradeoffs. Such various implementations can be achieved by considering different circuit architectures. For example, a multiplication can be realized by a bit-serial or a bit-parallel architecture, hence resulting in a small but slow and in a fast but large circuit.
- A task variant requires a specific amount of RHD resource. For FPGA devices, the synthesis tool typically provides information on how many FPGA slices are required by the design. Often, slice resources are further distinguished to the number of required LUTs and flip flops. Additional resources are reported as well, such as the required block RAMs, or multiplier cores. Hence, the basic resources can be captured by a scalar value (e.g. number of FPGA slices) or by a resource vector.
- A task variant may have a known execution time which is usually given by the number of required clock-cycles divided by the clock-frequency at which the task is executed. Determining the worst case execution time of a hardware task is discussed in the next paragraph.
- A task may require external memory to store larger amounts of data and may access it periodically within hard real-time constraints.

These assumptions are the motivation for our application models used within this thesis, namely the basic periodic real-time tasks model, introduced in Chapter 3, and the model extensions by task variants and periodic memory access, introduced in Chapter 7.

### Worst Case Task Execution Time

Determining the execution time is often not as easy as it has been for the DCT task of the example above and a deeper analysis is required. Without knowledge on the tasks' execution times we cannot develop a scheduling strategy, which allows us to guaranty that a task will finish before some given deadline. Moreover, in order to guaranty hard deadlines of tasks we have to design the system for peak load situations; even if in average the load of the system is low. Hence, we would like to know an upper bound on the execution time for each task. This is done by a so called *worst case execution time (WCET) analysis*, which is another research field within real-time systems and out of the scope of this thesis (see e.g. [64] p.86ff. and [24]p.411ff.). Rather we briefly discuss the issues of WCET analysis of hardware tasks executed on reconfigurable architectures compared to software tasks running on CPUs.

The WCET is mainly influenced by the *program flow* of the task itself and the *micro architecture* of the machine executing the task.

**Program flow:** To determine the WCET, the first step is the analysis of the program flow of the considered task. Usually, it is described by directed graph models capturing the sequential basic blocks and control decisions. Once the computation times of the basic blocks are known, the directed graph can be collapsed. For example a branch is collapsed to one node by assigning the WCET of the longer path. Loops are replaced by accounting $n$ times the WCET of the body, where $n$ is the bound on the number of loop iterations (loops must be bounded to have a bounded WCET of the task).

In general, the program flow analysis is independent of whether the task is implemented as hardware or software, but the high amount of concurrency in hardware tasks might make their analysis more challenging. However, many applications considered for the execution on reconfigurable hardware are data flow dominated, such as signal processing tasks, filters, coders and decoders. Due to few control decisions, such tasks are often simpler to analyze than arbitrary tasks.

**Micro architecture:** The WCET of the task's basic blocks can be derived by accounting the time required by the clock cycles in case of hardware tasks or by the instructions in case of software tasks.

Unfortunately, determining the time for an instruction on modern CPUs can be quite difficult. The data and instruction cache of processors provide enormous speedups when fetching instructions and accessing data, but this is transparent to the programming model. To be able to account these speedups when computing the WCET requires a detailed analysis of the program code together with the cache model, in order to know which items can be guaranteed to be inside the cache and which not. This becomes even harder, when a task can be interrupted by another task which might overwrite some cache values. Also the pipelined execution of instructions does not provide guaranteed performance, since we have to consider pipeline stalls.

In case of a hardware task, the programming model (i.e. VHDL) does not abstract from platform details that influence the number of clock cycles required by a basic block. Often this number can be directly achieved from the register transfer level (RTL) description, or results from the scheduling phase of high level architecture synthesis. In other cases, the vendor of an IP-core reports on the required clock cycles in the according data-sheets. The maximal clock frequency is reported by the place and route tools for a specific target device.

We can conclude that for many hardware tasks of our interest, obtaining the WCET is not a major issue. Analyzing the WCET for arbitrary hardware tasks might be yet another research topic.

## 2.3 Multi-Tasking on Reconfigurable Hardware

This section presents an overview of work in the field of *RHD based multi tasking*, which is directly related to this thesis. Additional work, which includes models and methods that are related to- or applied within the methods of this thesis, is introduced in the following chapters.

We categorize different models of RHD based multitasking that have been proposed in related work. The related systems can be classified according to the following properties:

**RCP Architecture:** Existing or proposed systems are either based on single RHDs, based on multiple RHDs or are hybrid systems coupling microprocessors with RHDs.

**RHD Resource Sharing Model:** Many approaches exist on how systems allow multiple tasks to share the RHD resources. The most relevant are discussed in detail in Section 2.3.1.

**Task execution model:** The system may or may not allow preemptive multitasking.

**Application model:** It can be distinguished between *off-line* and *online*-problems. In the first case, the entire application parameters such as the tasks and their arrival and execution times are known at design time. For example the application to be scheduled for execution is modeled in the form of a directed acyclic task-graph. Typical optimization goals are to minimize the overall execution time or to minimize the required resources, i.e., the size of the RHD. In *online*-problems, application parameters such as the tasks and their arrival times are determined at runtime. Therefore the system has to determine at runtime, which resources are assigned to the tasks without knowing which tasks have to be executed in the future. A typical optimization goal in this scenario is to minimize the average response time of tasks.

### 2.3.1 Resource Sharing Models

There exist numerous approaches how multiple tasks share the resources of an RHD. These are sometimes closely related to the supported reconfiguration features of the devices, i.e. if the RHD supports unrestricted partial, column-wise partial or only full reconfiguration. We describe six models; models (a) to (c) are based on full device reconfiguration and models (d) to (f) are based on partial reconfiguration.

**(a) Static Partitioning:** If multiple tasks have to be processed by one RHD, today's practice would be, using the static assignment partitioning approach, which is illustrated in Figure 2.5-(a). All system tasks are integrated within one RHD configuration, which is loaded to the RHD while the system is booting and stay active until the system shuts down. In this approach, the RHD behaves similar to an ASIC. The required size of the RHD is determined by the sum of the resource

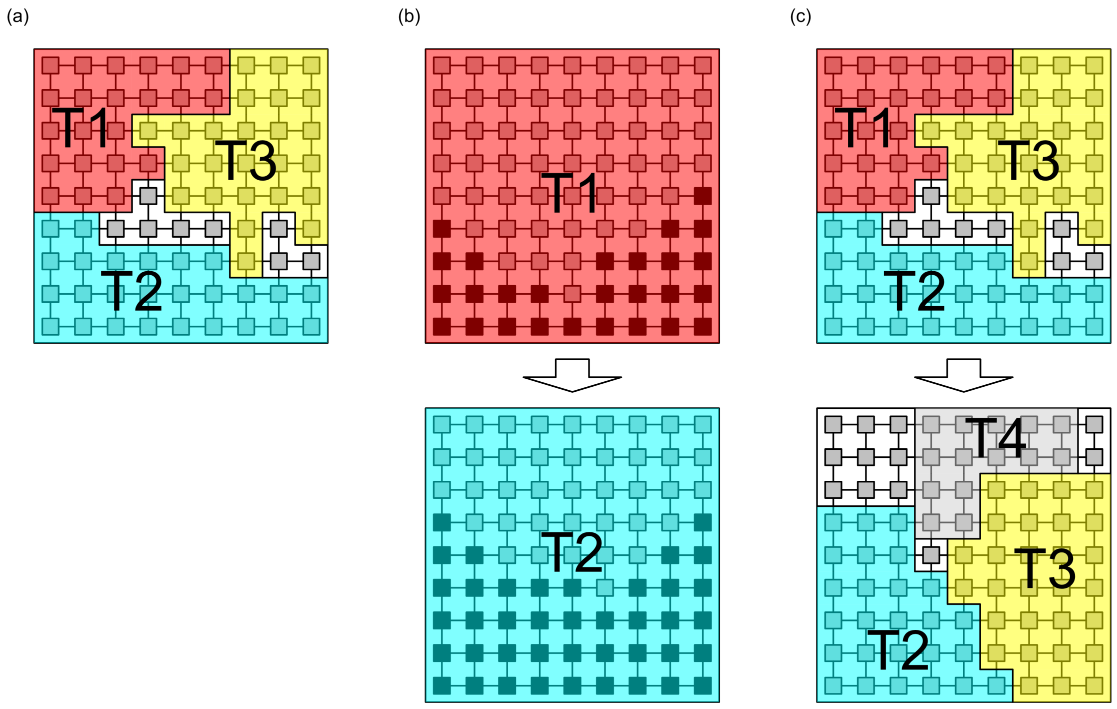


Figure 2.5: RHD Resource sharing approaches based on full RHD configuration.

requirements of all tasks. Since no timesharing of RHD area takes place, task scheduling and sophisticated resource management methods are obsolete. This greatly simplifies the system implementation. This approach is efficient when most of the tasks need to be running most of the time. It becomes ineffective when some tasks have to be executed just once or only sporadically from time to time, since all existing tasks allocate resources no matter whether they are running or not.

**(b) Non-Space Partitioning:** The simplest approach to realize multitasking with resource reuse is a time sharing approach without partitioning the area of the RHD (see Figure 2.5-(b)). The entire RHD resources are assigned to only one task at a time. To switch from one task to another, the entire device is fully reconfigured. Either each task runs until completion before the RHD is configured for execution of another task (non-preemptive multitasking), or tasks can get interrupted and resumed later to free the RHD for other tasks (preemptive multitasking). Concerning the resource management, this approach is equivalent to a single processor model and all existing scheduling strategies of that field can be applied. Systems following this approach are e.g. [58, 83, 69, 84, 85].

This model has the drawback, that only one task is executed at a time. Since no parallelism between tasks can be exploited, the design options are limited. I.e., we cannot speed up the application by choosing a larger RHD with more resources. Also the device utilization becomes inefficient if the tasks vary much in their area requirement. The required RHD size is determined by the area requirement of the largest task. We call this *internal fragmentation*, since each task is enlarged to the full device size, but leaving resources inside the task boundaries idle, which is illustrated by black RHD cells in the figures. Furthermore, for the realization of a quasi simultaneous task execution the RHD has to be reconfigured frequently, which adds a great time overhead and hence presents another drawback.

**(c) Free Space Partitioning - Time Sharing:** A combination of the two approaches above is the *free space partitioning - time sharing* model, which is illustrated in Figure 2.5-(c). In a certain RHD configuration, the RHD resources are freely distributed among a set of running tasks. In praxis, several task source files are synthesized into the same device configuration [58][5][104][49]. The only placement constraint is that the sum over the area of all tasks from this running set is less or below the area of the RHD. Timesharing of the RHD is realized by full device configuration. The example in Figure 2.5-(c) shows task 1, 2 and 3 running. If task 1 is preempted by task 4, the entire device is reconfigured by a configuration containing tasks 2,3 and 4.

Using this approach, whenever a new task starts, the device gets reconfigured and all running tasks have to be interrupted. Hence, the system needs to support functionality to rescue the inner state of the interrupted task during the reconfiguration process, no matter if preemptive or non-preemptive multitasking is considered.

The main drawback of this approach is that one configuration file has to be created off-line for every combination of running tasks that can occur during the execution of the application. This can result in large memory requirements for storing the configuration files. In general, the number of configuration files grows exponentially with the number of tasks, but can be reduced and bounded by a proper scheduling strategy.

We can overcome this drawback, if the system is able to construct the configuration data for the RHD out of a set of partial RHD configuration bit-streams during runtime. This way, only the programming data of each task is stored only once as a partial RHD configuration bit-stream. Composing a full configuration file out of a set of partial configuration data requires address relocation such as it has been implemented in systems described in [52][53] [60].

Another drawback is the high time overhead resulting from the full device reconfiguration and the interruption of *all* running tasks whenever the RHD is reconfigured.

The advantage of this model is the practicability: It is realizable on any full reconfigurable device omitting the problems involved with partial reconfiguration. Moreover the model has the potential of high device utilization. No task internal fragmentation occurs, since the area allocated to a task is variable. Also no external fragmentation occurs, in the sense that an additional task can always be added to the executing tasks, if the sum of the area of all running tasks does not exceed the device size. The reason is, that the task locations can be rearranged whenever the device is fully reconfigured.

[5] uses full configuration, but system consists of $n$ parallel FPGAs and each task requires $a \leq n$ FPGAs

In Section 4.3 we present a scheduling algorithm, which uses this resource model and keeps the number of configuration files bounded by the number of tasks. Hence, it overcomes the main drawback of this model. Further, we will use this resource model and the designed scheduling algorithm for our prototype of an RHD based real time system in Section 8.

**(d) Equal Slot Partitioning:** Figure 2.6-(d) shows a widely used resource sharing approach, where the RHD area is partitioned into a fixed number of $m$ slots of equal size [101][100][97]. Every running task gets exactly the RHD resources of one slot assigned, which means that at most $m$ tasks can be in execution in parallel. If a new task is selected for execution due to termination or preemption of a running task, the RHD area of the according slot is reconfigured by the configuration data of the new task. This is done via partial reconfiguration, which keeps the executing tasks of the other slots unaffected during this process. Concerning the resource management, this approach is equivalent to a homogeneous multi-processor model.

The major disadvantage of this model is the internal task fragmentation, which results generally in low device utilization. Since each task has to fit into a slot, the slot size has to be chosen according to the largest task of the application which also determines the number of slots for a given device. All other tasks are enlarged to this slot size.

There are several advantages of this model: Among the models using partial re-

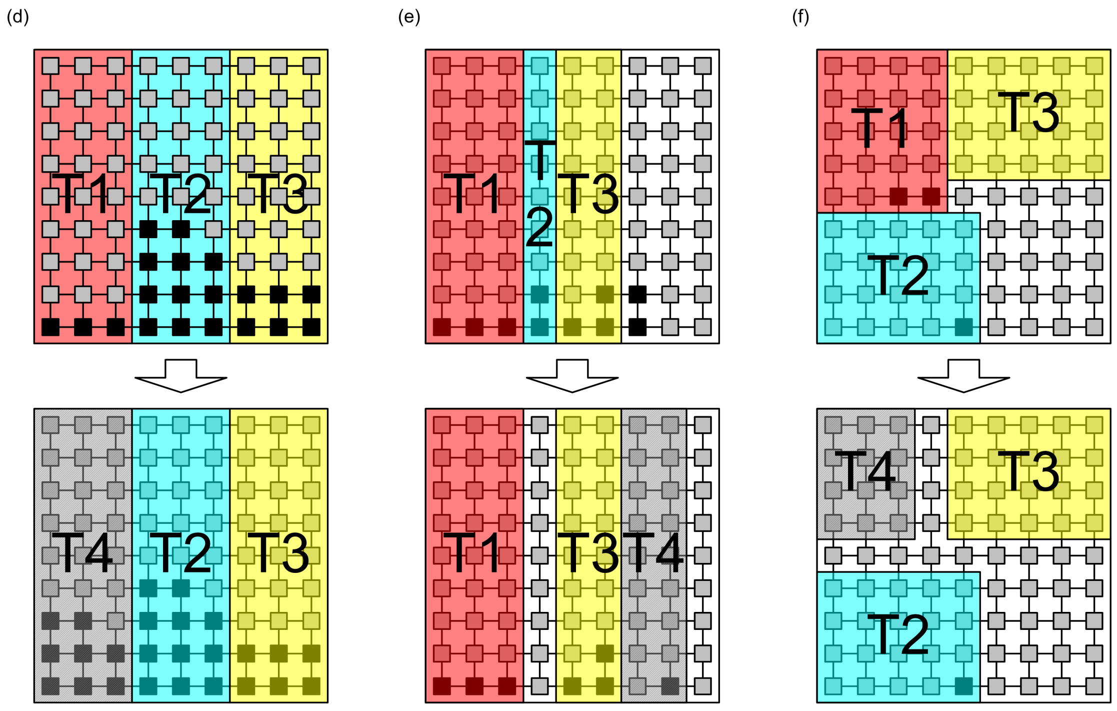


Figure 2.6: RHD Resource sharing approaches based on partial RHD configuration

configuration, it is the one with the easiest technical implementation. The model is consistent with the constraints of most of todays partial reconfigurable FPGAs like XILINX Virtex and VirtexII devices which require the reconfiguration of entire columns of the logic blocks. Systems using this approach are introduced in [101][100][97]. Also, the management of allocated and free resources has low complexity, since the slots are equivalent and only the number of allocated or free slots has to be considered. Hence, no placement strategy is necessary. The equivalence to a homogeneous multi-processor enables the reuse of many scheduling strategies developed in the research field of parallel systems.

**(e) One Dimensional Partitioning:** An extension of the equal slot partitioning is the one dimensional partitioning model, referred to as 1D model from now on. As shown in Figure 2.6-(e), the tasks are stripes of full RHD array height, but with variable width and are placed side by side on the device. If a running task terminates or gets preempted, its occupied resources are deallocated. A new task selected for execution is placed into a free continuous area of adequate width. This model is called one dimensional partitioning, since each task has a footprint with one fixed and one variable dimension. Proposed systems using this approach are for example [59, 22, 43, 99].

Compared to the slot approach **(d)**, this model has the advantage of high potential device utilization, since the size of tasks can scale according to their resource requirements (almost no internal task fragmentation). This comes at the cost of external fragmentation. Situations can occur, where there is enough free space on the RHD for a new task but not as a continuous stripe. To minimize such situations and to get good device utilization would require sophisticated placement strategies as well as management of free continuous stripes. This resource model is no longer equivalent to a homogeneous multiprocessor. The appearing scheduling and optimization problems rather have relations to orthogonal packing problems. Furthermore, the model still fits to the capabilities of today's partial reconfigurable FPGAs and design tools: The task footprints are constrained to be stripes of the device array. However, if the actual task position onto the device is unknown at design time, the system needs to support *task relocation*, which brings new realization challenges but has been successfully implemented by the authors of [60].

We use the 1D model for our global scheduling method in Section 4.1, which requires task relocation, and for our partitioned scheduling method in Section 4.2, which works without task relocation.

**(f) Two Dimensional Partitioning:** A further extension is the two dimensional partitioning approach, where tasks are modeled as rectangles of variable width and length which can be freely placed onto the RHD, as shown in Figure 2.6-(f). Running tasks are exchanged by partial device reconfiguration of the affected area [38, 40, 14, 41, 93, 94, 42].

This approach enables more flexibility concerning the shape of a task's footprint. In the one dimensional approach of model **(e)**, the resource requirements of a task have to be distributed within a rectangle of full device height - which generally results in an unbalanced ratio of rectangle height to width. This can affect the ability to place and route the task logic and the task's performance (e.g. maximal clock frequency) since the path length increases. Also the internal fragmentation of model **(f)** is less than that of model **(e)**, since in the latter the smallest unit of resources is one entire RHD column. However, studies have shown that these negative effects are relatively small [59].

One major practical drawback of this approach is that most of today's commercial available devices do not support this kind of free partial reconfiguration. Some devices allowing free addressing of the element to reconfigure are the coarse grain XPP architecture [13] and – to some extend – Xilinx Virtex 4 FPGAs [74]. Also I/O communication issues occur, when tasks are placed in the middle of the RHD area with no direct connection to I/O cells. Possible solutions are network on chip approaches like [19]. Also the algorithmic problems involved to optimize the task placement and scheduling become more complex. The dimension of the packing problems increases and in scenarios where tasks arrive online, the management of free and occupied device resources becomes much more complex than in the 1D model [14].

The characteristics of the six introduced approaches of resource sharing are summarized in Table 2.2.

| issue \ model | a) static | b) no space | c) free space | d) slot | e) 1D | f) 2D |
|---|---|---|---|---|---|---|
| reconfiguration | static | full | full | partial | partial | partial |
| supported RHDs | all SRAM based | | | column-wise reconf. | | free reconf. |
| model similar to: | ASIC | uniprocessor | - | multi-processor | memory allocation | - |
| utilization | low | low | high | medium | high | high |
| area fragmentation | non | high intern | non | medium intern | extern; low intern | extern |
| reconfiguration overhead | non | high | high | medium | low | low |
| complexity of placement | non | non | non | low | medium | high |
| complexity of scheduling | non | low | high | medium | high | high |
| task interruption | non | for preemptive MT | always | for preemptive MT; for de-fragmentation | | |
| implementation complexity | low | low | medium | medium | medium | high |
| references | [7] | [58, 83, 69, 84, 85] | [58, 104, 49] | [101, 100, 97] | [59, 22, 43, 99, 60] | [38, 40, 14, 41, 93, 94, 42] |

Table 2.2: Comparison of resource sharing models

### 2.3.2 Task Scheduling and Placement Methods for RHDs

Scheduling and placement of tasks on RHD based systems has recently attracted much attention in research. The work differs in the considered resource models (e.g. slotted, 1D, 2D, etc.) and the considered scheduling problems (e.g. off-line vs. online task activation, preemptive vs. non-preemptive, independent vs. precedence constraints). Moreover, all approaches described here do not consider hard real-time tasks but rather minimize cost functions like total make-span or average respond time. Scheduling approaches for real-time tasks on processor machines as well as on RHD devices are described in more detail in Section 2.4.

Fekete, Teich et al. studied off-line scheduling and placement of synchronous activated task with precedence constraints onto an RHD, assuming the 1D and the 2D resource model. For the problems of minimizing the total make-span for a given device size, and for the problem of finding the smallest device for a given time-limit, they presented an optimal method [44] [94]. The central idea is, to represent feasible solutions using so called *packing classes*, which can be efficiently represented as interval graphs. A packing class captures not the absolute task positions in the x, y and time dimension on the device, but rather captures whether the placement of two tasks overlap in a certain dimension or not. Hence, the search space is considerably reduced and a branch and bound method allows finding optimal solutions in reasonable time.

Danne et al. kept up with an extended problem where task variants are considered [DS05]. The off-line problem of temporal partitioning and temporal placement of data flow graphs for reconfigurable systems was studied by Bobda in [18].

Bazargan et al. considered the placement of online arriving, independent and non-preemptable tasks onto an RHD under the 2D model [14]. When a task arrives it is either directly placed or rejected. The goal is to maximize the fraction of accepted tasks. For the efficient management of the free space onto the device, two methods have been proposed which have been evaluated together with several different placement heuristics such as bottom-left, first-fit and best-fit.

Walder et al. kept up with a similar online problem and improved placement qualities by allowing task footprint transformations [99]. Köster et al. considered placement of tasks onto inhomogeneous RHDs, where not all resources are arranged in an array [63].

### 2.3.3 Dynamic Reconfigurable Systems

Many research projects involve multitasking on RHD based platforms and several prototypes have been implemented. Often, the systems are specialized for a certain application domain and use individual models, making an exact classification and comparison of these systems difficult. Therefore, we select some representative projects which have a rather high relation to our work for a brief description. The first three systems follow a fully FPGA reconfiguration approach, whereas the last three systems have been designed for partial reconfiguration.

**Simmler's OS:** Simmler at el. developed an operating system for preemptive multitasking on FPGA co-processors [83][84][69]. Their system couples a host CPU with an FPGA co-processor board. Considered applications run on the host CPU and use the FPGA to accelerate computation intensive tasks. A task running on the FPGA may be preempted by a higher priority task. Restoring the state of a preempted task before its resumption is realized by a *configuration read-back, extract and write-back* approach via the FPGAs configuration interface.

An interesting feature of the system is the set of several SRAM banks, which are connected via a switch to the FPGA as well as to the host CPU. RAM-banks are assigned to tasks exclusively, such that their contents do not need to be swapped when a task switch occurs. While the system presents a feasible prototype of preemptive hardware-multitasking, its main drawback is that only one hardware-task runs at a time. I.e., it implements the resource sharing model *(b) Non-Space Partitioning*, which has the disadvantages described above.

**SONIC:** This project aims to map real-time video applications to a specifically designed reconfigurable computing platform called SONIC, (later called UltraSONIC) [104]. The platform consists of multiple modules each combining an FPGA with one local external RAM bank. A bus system connects the modules with a host computer.

The application is specified as a directed acyclic graph (DAG) of tasks where edges represent data dependencies. The tasks are partitioned into hardware and software tasks, therefore the system is a truly *hybrid* system, where the host processor is not only used for control issues but to execute parts of the application. The hardware tasks are clustered to FPGA configurations, such that the device size is taken into consideration. Tasks in the DAG are processed from top to bottom and hardware tasks are loaded on demand via full FPGA configuration. The aim of the schedule is to minimize the make span. Although multiple tasks reside on a programmed FPGA, only one is active at a time due to the limitation of the single port external memory. (Our work explicitly solves this problem by scheduling RAM accesses of tasks to RAM banks under real-time constraints as introduced in Section 7.2.) However, in the SONIC system multiple tasks can run in parallel when mapped to separate hardware modules. Task communication occurs only at the beginning and end of a task execution and is done via the local memory. The SONIC system shares FPGA area using model *(c) free space – time sharing*. Hence, it is related to our prototype of Chapter 8, where we use the same resource model. In contrast to our work, no parallel execution of tasks onto an RHD takes place and the RAM banks cannot be shared by simultaneous executing task. Furthermore, task preemption is not allowed and deadlines of tasks are not considered.

**SPARCS:** SPARCS [49] stands for *Synthesis and Partitioning for Adaptive Reconfigurable Computers*. The aim of the project is to map a high level specification application onto a reconfigurable architecture. In contrast to our RCP architecture, the authors assume a platform hosting multiple FPGAs, each coupled with only one external memory bank. Throughout an interconnection network, the FPGAs are connected to memory storing bit-streams, to one (large) shared memory on board, and to I/O devices. The application is specified as a directed graph, where nodes can be either tasks or logical memory buffers. The edges represent either channels or dependencies. Dependencies among the task represent control flow, whereas channels represent inter-task or task-to-memory communication.

The high level synthesis involves two basic steps: In the *temporal partitioning* phase, the application is partitioned into temporal segments such that these segments can be executed sequentially considering the task dependencies. Moreover, it is ensured that the logic cell and memory requirements of each segment do not exceed the platform resources. Afterwards, in the *spatial partitioning phase* the tasks of each temporal segment are divided into partitions, such that each fits within one FPGA and the logical memories are mapped to the physical board memories. Using standard synthesis tools, one FPGA programming file for each partition is created. At runtime, a controller reconfigures the FPGAs according to a global schedule, using full device configuration.

In contrast to our work, in SPARCS each task is contained within one and only one partition (configuration). Thus, a non-preemptive multitasking model is considered and each task finishes its processing within one partition. Also, no real-time constraints are associated with the tasks. Interestingly enough, the SPARCS system is one of the few that considers several implementation variants for each task that allow to explore various area-latency tradeoffs of a task. The variants are used to balance the latency of tasks within a partition. Similarly we consider task variants in our extended task model of Section 7.1, but in contrast to the SPARC system we consider periodic real-time task rather than static task graphs.

**ESM:** The *Erlangen Slot Machine (ESM)* [20][21] is an FPGA based reconfigurable platform. It has been specially designed, targeting applications which use partial FPGA reconfiguration. One main advantage of the system is, that it allows each hardware-task to access its peripheral resources independent of its location via a crossbar. Also several types of inter task communication are supported by special communication lines. The system allows placing tasks relocatable according to the 1D resource approach, and hence provides an implementation platform for multi-tasking applications.

**RAPTOR 2000, REPLICA** Kalte et al. developed a rapid prototyping system called RAPTOR 2000 [62], which is able to host several FPGA modules and supports partial FPGA reconfiguration. Based on the RAPTOR platform, they developed a framework for dynamically reconfigurable systems focusing on the 1D resource model. The relocation of tasks onto the FPGA is supported by the REPLICA [60] filter. RECPLICA is a hardware circuit, which takes a partial FPGA bit-stream of a task and a target FPGA location as inputs. The column addresses within the bit-stream are manipulated on the fly, such that the output bit-stream can be used to configure the task at the desired position onto the FPGA. Moreover, they implemented task context saving and restoring by using the FPGA configuration interface [61]. A *state extraction* filter allows storing the register content of a task that has just been stopped, by reading the current FPGA configuration and extracting the state of the flip flops. Later on, when the task has to be resumed, even at a different position, a *state inclusion* filter allows merging the state information into the configuration bit-stream of the task before it is loaded onto the FPGA. Hence, the REPTOR 2000 system together with its tools presents a working prototype and proof of concept for preemptive multitasking on FPGAs under the 1D resource model. It supports all requirements for the execution model assumed in our global- and partitioned scheduling methods of Chapter 4.

**XFORCES:** The *Executives for Reconfigurable Embedded Systems (XFORCES)* project aims to investigate and develop an operating system kernel for reconfigurable hardware devices [100][102][103]. The runtime system loads tasks on the FPGAs. For more sophisticated dynamically reconfigurable systems, the runtime system also schedules tasks at runtime and saves the context of preempted tasks. Furthermore, it establishes communication interfaces between the tasks, no matter whether they are mapped to the same or different FPGAs.

The XF-board [98] was developed as a platform for the XFORCES OS prototype. It hosts two FPGAs where on the first one a soft core CPU is implemented that controls the overall system. The second FPGA is used as a reconfigurable resource for the execution of hardware-tasks. All I/O devices and memory banks are connected to the left and right device edges of the second FPGA. Hence, many routing problems are avoided and the design matches the requirements of a 1D partial reconfiguration model.

To avoid the manual creation of the bus infrastructure for the OS kernel and the tasks, which is a hard an error prone process, the *XFOSgen* tool was developed. It generates the OS logic and communication structure in conjunction with task templates of different width. Hence, it considerably simplifies the design of a reconfigurable multitasking application.

This section summarized related work on multitasking of hardware tasks. We introduced several resource sharing models used in literature, presented work on different scheduling problems and briefly described prototype systems implementing some kind of hardware multitasking. In summary, hardware multitasking has obtained much attention by researches, numerous related scheduling problems have been studied and several system prototypes have been built, proving feasibility of the technique. However, real-time constraints considerations have been omitted in most of the presented work.

## 2.4 Real-Time Scheduling

This section gives a background on those parts of real-time scheduling, which are directly related to our work.

### 2.4.1 A Background on RT Systems

Real-time systems are embedded computer systems, that must react within precise time constraints to events from their environment [25]. The correctness of the system depends not only on the logical correct computation of results but also on temporal correctness, i.e. the time at which the results are delivered. A computation is temporally correct if it finishes within a specified time frame [23].

In a multitasking system, a task $T_i$ is typically modeled by its arrival or release time $r_i$, its worst case computation time $C_i$ and its timing constraint is modeled by a *deadline* $d_i$, which represents the time until the task has to complete its execution. Depending on the consequences of missing a deadline, the literature distinguishes at least between two classes of real-time tasks:

If a *hard real-time task* misses its deadline, this can cause catastrophic consequences on the entire system. For example, a task controlling the airbag system of a car. If it decides to fire the airbag – and this result is produced too late – it is worth nothing. This can be modeled by a value function of the task, which equals one before the deadline of the task and is minus infinity after the deadline of the task. Hence, if only one task misses its deadline there is no value of the entire system.

If a *soft real-time task* misses its deadline, the performance of the system decreases but it can still work correctly. The computed result is not useless, but it is less valuable than if it would be produced before the tasks deadline. Examples are video systems, where a frame displayed a little too late may reduce the system quality. In this case the value function of the task is one before its deadline and decreases monotonically after its deadline until it reaches a value of zero.

Furthermore, a real-time system can have a *static* or a *dynamic* task set. In a static real-time system, the task set is fixed and all task parameters as arrival times, computation times and deadlines are known a priori at design time.[6] Therefore, we design the system for the worse case scenario and a task schedule can be computed off-line and validated, whether every deadline is met or not. In a dynamic real-time system, a new task can be (unpredictable) activated at runtime. Therefore, without further assumptions we cannot guarantee that all released tasks can be processed within there deadlines. A common method to deal with unpredictable task activations is to introduce an acceptance test. When a task is requested at runtime, a procedure checks whether the new task can be processed within its deadline without jeopardizing the timing constraints of the other tasks. When this test is negative the task activation is rejected, otherwise it is accepted and planned for execution. Obviously, the application has to be designed to deal with

[6]At least worst case assumptions have to be known

task rejections for example by specifying an alternative system behavior.

It is further distinguished between *aperiodically* released and *periodically* released tasks. In the later case, a task is released multiple times in a periodic manner, i.e. the time between two releases of the task is constant and denoted as the *period* of the task. This is a reasonable assumption for many real-time applications. Examples are control processes of mechatronic systems, where control outputs are computed based on periodically sampled sensor values. Often the assumption is made, that the relative deadline of a periodic task is equal to its period, i.e. the task has to finish its computation before its next release. Some relaxation of periodic tasks are sporadic tasks, where the period presents the minimum inter-arrival time of two task instances. Most results for periodic tasks do also apply to sporadic tasks.

### 2.4.2 RT Scheduling Problems and Algorithms for Uniprocessor Systems

A task schedule defines at any point in time, the allocation of system resources to the application tasks. In the simplest model, only the processor of a single-processor system is considered. In this case, a schedule is a step-function over time, where the value of the function defines the ID of the task that is executed by the processor at a given point in time. A schedule is said to be feasible, if all (hard real-time) tasks meet their deadlines.

The construction of a schedule is done by a scheduling algorithm. For static task sets, this could be done off-line and the schedule could be stored in a dispatcher table. However, this is only useful in some cases. In most cases, the parameters of a static task set present only worst-case parameters, such as the worst case computation time of a task. In average, the computation time may be much smaller. Hence to benefit from early terminating tasks, most systems use scheduling algorithms online and at runtime, but verify the feasibility of the scheduling strategy off-line at design time against the worst case parameters of a task set. Also the memory requirements by the dispatcher table might be too high.

Another aspect of a scheduling algorithm is, whether it constructs a preemptive or a non-preemptive schedule. A preemptive schedule is in general more flexible and a feasible schedule can be computed with less effort than in the non-preemptive case. However, the system has to support task preemption at a reasonable cost.

#### Aperiodic Task Sets on Uniprocessors

For aperiodic task sets the *earliest deadline first (EDF)* algorithm has been proposed by Horn [51].[7] It follows a simple rule:

*Among all released tasks always the task with the earliest absolute deadline is assigned to the processor.*

Therefore, whenever a task gets activated, it is inserted into a deadline sorted ready-queue. The task at the head of the queue is selected for execution if currently no task is

[7]The results of the original reference have been summarized in [25], p. 50 ff.

in execution or if the deadline of the head task is less than that of the task in execution. In the latter case, the task in execution is preempted by the head task – therefore EDF constructs a preemptive task schedule. The EDF schedule can be computed within $O(n^2)$ steps, where $n$ is the number of tasks. EDF have been shown to be optimal in the sense, that it finds a feasible schedule to a given task set if one exists [36][7]. Moreover, it can be modified to optimally schedule task sets with precedence constraints [28][7].

Finding a feasibly non-preemptive task schedule for tasks with arbitrary arrival times has been proven to be NP-hard even for the single processor case [68]. Therefore, an optimal algorithm for the non-preemptive case is computational complex such as the tree search method of Bartley (see [25] p.58 ff.) which runs in $O(n \cdot n!)$ time. For large task sets, such runtime is unacceptable and we have to fall back on heuristic approaches, such as the Spring algorithm [87].

### Periodic Task Sets on Uniprocessors

The periodic task model introduced by Liu and Layland in [72] is widely-accepted (e.g. as a standard model in textbooks [25]p. 71 ff.,[24]p. 399 ff.) and has been probably the most studied subject in real-time scheduling [82]. In this model, each task has a known worst case execution time $C_i$ and is requested periodically with a rate $P_i$. For the problem of scheduling a task set to a uniprocessor system, two particular algorithms have received the major attention.

**Rate Monotonic (RM):** RM is a fixed priority assignment schema. It assigns priorities to the tasks according to increasing request rates and this assignment stays fix during runtime. I.e. the task with the smallest period gets the highest priority. At runtime, always the active task with the highest priority is selected for execution.

It has been shown, that RM is optimal among all fixed priority assignment schemas [72]. However, that is only for fixed priority assignment schemas and there are cases, where RM does not find a feasibly schedule, even if one exists. In general, RM is not able to utilize the processor up to 100%. Liu and Layland [72] provided a *sufficient but not necessary* scheduling test by providing an upper bound to the processor utilization:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n \cdot \left(2^{1/n} - 1\right), \quad \text{and for arbitrary large } n: \quad \lim_{n\to\infty} \sum_{i=1}^{n} \frac{C_i}{P_i} \leq \ln 2 \tag{2.2}$$

According to Equation 2.2, all task sets that accumulate the processor by less than about 69% will be feasibly scheduled by RM. The test is pessimistic, e.g. some task sets do not satisfy Equation 2.2 but are still schedulable by RM. However, the test is tight in the sense that there exist still cases where the task set utilizes the processor only by $\ln 2 + \epsilon$ which cannot be scheduled by RM.

Better scheduling tests for RM, that are less pessimistic have been proposed. In particular the so called *hyperbolic bound* of Equation 2.3 was shown in [77][17] (see [25] for details).[8]

$$\prod_{i=1}^{n}\left(\frac{C_i}{P_i}+1\right) \leq 2 \tag{2.3}$$

**Earliest Deadline First (EDF):** The EDF rule as it has been introduced for aperiodic tasks above can be applied to periodic tasks as well. It assigns the highest priority always to the active task with the closest *absolute* deadline. Since the priority of a task may change at runtime when another task is released or terminates, EDF is called a dynamic priority assignment schema.

EDF has been shown to be optimal for periodic task sets scheduled on a uniprocessor system. It is able to schedule all task sets that utilize the processor by not more than 100%. Hence, the exact scheduling condition is given by:

$$\sum_{i=1}^{n}\frac{C_i}{P_i} \leq 1 \tag{2.4}$$

Obviously the performance of EDF is much better than that of RM and it is easier to check, if a particular task set will be feasibly scheduled. However, due to the fixed priorities, an RM dispatcher is very easy to implement and hence, RM is still widely used in industrial systems.

A detailed comparison study between EDF and RM for all important aspects has been presented in [26]. Clearly, EDF is favored over RM, as it has a better performance, is easier to analyze and usually produces less context switches than RM. For the same reasons, all of our proposed three scheduling approaches presented in Section 4 use EDF as basis. Only when we schedule the memory read- and write accesses for the RAM buses in Section 7.2, where scheduling decisions have to be made within one clock cycle, we fall back to the RM schema. An implementation of an EDF scheduler, which requires list data-structures, seems not to be adequate for that case.

### 2.4.3 RT Scheduling Problems and Algorithms for Multiprocessor Systems

The scheduling of tasks on parallel or distributed systems is much more challenging than scheduling tasks on a single resource. As reconfigurable architectures belong to the class of parallel systems, we present approaches on the multiprocessor scheduling problem which are closely related to our work.

At this point, we are not able to survey the large amount of work that has been done on this topic. We rather introduce the basic methods used for multiprocessor scheduling

[8] It appears that the bound has been first proven in [77] and successfully used for multiprocessor scheduling in [78]. Independently, Buttazzo et. al. provided the hyperbolic bound of Equation 2.3 in [17] and presented further analysis.

and go into detail, whenever a method is closely related to our proposed methods or has been used within this thesis.

**Definition 1** (periodic multiprocessor scheduling problem)**.** *Given a set of periodic tasks $T_1$ to $T_n$ each with a worst case computation time $C_i$ and a period $P_i$ and a machine with $m$ processors of equal speed. Find a schedule that assigns the $n$ tasks to the $m$ processors over time such that each task executes for $C_i$ units within each period. Assumptions:*

- *Task preemption is permitted. A task can be preempted and later on resumed at no cost.*
- *Task migration is permitted. A preempted task can be resumed at a different processor at no cost.*
- *Task parallelism is forbidden. A task can only execute on one processor at a time.*

The scheduling approaches to this problem can be categorized into two main classes, namely *partitioned* and *global* (non-partitioned) scheduling [5].

## Partitioned Scheduling

In a partitioned schedule, all instances of a task are executed on the same processor. Hence, the set of $n$ tasks is partitioned into $m$ subsets, each associated with one processor. At runtime, each processor executes only its associated task set using some standard uniprocessor scheduling algorithm such as EDF or RM.

The *partitioned scheduling* method has attracted much attention for two reasons. The first is a practical issue: In a partitioned schedule no task migration occurs, which might be difficult to implement and might introduce much more time overhead than a normal context switch. Actually, the system can be simply composed out of several uniprocessor systems, which are loosely coupled and all implement their own independent uniprocessor scheduler.

The second reason concerns the theoretical analysis: The multiprocessor scheduling problem is reduced to a partitioning problem, where each subset of tasks has to be feasibly scheduled by a uniprocessor algorithm. Hence, all results achieved for uniprocessor scheduling algorithms, such as the scheduling conditions for EDF or RM, can be reused. We will first discuss the dynamic priority assignment under EDF and afterwards the fixed priority assignment under RM, which is more complex to analyze.

**Dynamic Priority Partitioned Schedulers:** Consider that an independent EDF scheduler is used for each processor. Since EDF schedules all task sets up to a processor utilization of 100% (see Equation 2.4), the problem evolves to a classical bin-packing problem. Each task is an item of size equal to its processor utilization $U_i = C_i/P_i$ and each processor is a bin of size 1. For solving the NP-hard bin-packing problem, we can use one of the several bin-packing approximation algorithms [29]. Typical choices for real-time schedulers have been *first fit (FF)*, *best fit (BF)* and *first fit decreasing (FFD)* and *best fit decreasing (BFD)* [88]. In the later two algorithms the tasks are sorted according to decreasing utilization values before they are packed to the bins (processors). It has been shown (see [29]), that the BFD and FFD assignment schemes have asymptotic upper bounds of 11/9, meaning that the number of processors used by BFD or FFD is only 1.222... times the number required by an optimal assignment in the asymptotic case. Hence, these heuristics offer a quite good performance.

We pick up this basic approach of partitioning the task sets and then scheduling each subset individually by EDF and extend the approach to our task and resource model in Section 4.2. We show that in that case our scheduling problem is no longer equivalent to simple bin-packing, but equivalent to *two-dimensional level strip packing* [73].

**Fixed Priority Partitioned Schedulers:** The multiprocessor scheduling problem using the partitioning approach becomes more challenging when the *rate monotonic* algorithm is used for each processor. More complex schedulability conditions have to be used to decide whether a processor (bin) is full or not. For example, the condition of Equation 2.2 can be used, when a first fit decreasing heuristic is used to assign tasks to processors and each processor uses the RM scheduler. However, the quality of such assignment can be quite bad: For the RM scheduler in conjunction with an FFD task assignment an asymptotic bound of 2 has been shown (see [23]), meaning that it may happen that an assignment requires twice the number of processors compared to the optimal assignment.

In [70][23], the authors proposed an improved assignment schema based on the fact, that RM achieves better processor utilization than that of Equation 2.2, if the task periods show some special relation (e.g. harmonic periods). Hence they group tasks with *nicely related* periods together - with the result that for light-weight tasks all processors could be almost fully utilized. Hence, surprisingly the drawback of the low RM performance in the uniprocessor case can be overcome in the multiprocessor case.

Moreover, in [78] the authors considered the RM algorithm with an FFD assignment, but used the improved RM scheduling condition of Equation 2.3. They showed, that due to this improved RM scheduling condition the assignment uses in worst case 1.66... times the number of processors used by an optimal assignment.

### Global (non-partitioned) Scheduling

In a global or non-partitioned schedule, a task is allowed to be preempted on one processor and to be resumed on another processor. For architectures where the penalty of task migration is low, such as multiprocessors with a shared memory, this can be a reasonable alternative to the partitioned schedules. Due to the higher freedom of the task to processor assignment, there exist task sets that can be scheduled by a global scheduler but cannot be scheduled by a partitioned scheduler. The challenge is, to design good global schedulers that achieve a high scheduling performance and that can be proven to schedule a particular task set, by means of an efficient computable scheduling test. Again, we can distinguish between fixed and dynamic assigned task priorities.

**Fixed Priority Global Schedulers:** The rate monotonic priority assignment scheme can be also used for a global scheduler. In *global rate monotonic scheduling (GRMS)* the tasks get fixed priorities assigned in order of decreasing task periods. At runtime the dispatcher selects the $m$ active tasks with the highest priority for execution on the $m$ processors.

In [66] an efficient computable scheduling condition for GRMS has been developed and the performance has been compared to several partitioning schemes based on the RM scheduler. It turned out that the scheduling condition is rather pessimistic and allows only about 50% utilization of the processors in average. However, an exact scheduling test based on simulation over the task sets hyper-period shows, that GRMS feasibly schedules task sets up to 80% processor utilization. This performance is comparable to – or only slightly worse than – that of the best partitioning schemes. However, it stays more robust against variation of task set parameters. The authors showed that GRMS can outperform partitioning schemes for soft real-time systems, where some deadline misses can be afforded. Especially when the worst-case computation times of the tasks are inaccurate, the GRMS profits from the global processor assignment since overloaded processors take advantage from spare processing power of under-loaded processors.

In [5], the authors studied a different fixed priority assignment schema for global multiprocessor scheduling. Whereas RM assigns priorities only according to task periods, their proposed algorithm $TkC$ assigns priorities according to the difference between task period and task computation time. More specific, the priority of a task $T_i$ is set to $P_i - k \cdot C_i$, where higher values mean higher priorities. Based on simulation experiments, they found out that for a value of $k = 1.1$, their algorithm achieved the best performance in average and significant improvements over the GRMS. Moreover, $T1.1C$ commonly outperforms the best RM partitioning algorithms. However, the authors failed to present an efficient scheduling test for $T1.1C$. Hence, the main fields of application are still soft real-time systems.

As mentioned earlier, we use dynamic-priority assignment schemes (especially EDF) for our scheduling methods, hence the results of the fixed-priority global schedulers are of minor interest for our work. However, in principle we could replace the dynamic EDF priority assignment in our global scheduler of Section 4.1 by a fixed assignment such as RM. In that case, the results of the work mentioned above would become most relevant. We leave this option for further work.

**Dynamic Priority Global Schedulers:** The dynamic-priority EDF assignment scheme, which is optimal for uniprocessors (see Section 2.4.2) and has been successfully applied in partitioned multiprocessor scheduling, can also be used for global multiprocessor scheduling.

A global EDF scheduler always executes on the $m$ processors the $m$ active tasks with the shortest absolute deadlines. Unfortunately, global EDF is no longer optimal. Several sufficient polynomial time scheduling tests for global EDF have been developed [48][47][16][11][10] and surveyed and compared in [16] and [11]. There exist three basic scheduling tests for global multiprocessor EDF:

1. Goossens et al. [48] developed a utilization based test, using the resource augmentation approach introduced in [81]. It is based on the following idea: A uniform multiprocessor (each with individual speed) is considered, on which the task set can be feasibly scheduled by an optimal scheduling algorithm. Then a homogeneous multiprocessor is considered (all processors have the same speed) and the number of processors is increased until it can be guaranteed that the global EDF scheduler does, at any time, at least the same amount of work on the task set, than the optimal algorithm on the uniform multiprocessor. It follows that also EDF feasibly schedules the task set and the following scheduling condition is derived:

   **Theorem 1** ([48]). *A periodic task set $\Gamma = \{T_1, \dots, T_n\}$ can be feasibly scheduled by* EDF *onto an identical multiprocessor of $m$ unit capacity processors, if:*

   $$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq U_{max}^T + m \cdot (1 - U_{max}^T) \tag{2.5}$$

   *where $U_{max}^T$ is the highest time utilization of any task appearing in $\Gamma$.*

   In Section 4.1, we define global EDF schedulers for our RHD execution model and develop a scheduling test based on the Goossens et al. [48] multiprocessor scheduling test. Hence, we skip the details of the test of Theorem 1 at this place and go into more detail in Section 4.1.2, where we discuss the relation between the multiprocessor scheduling test and our work.

2. Baker presented a sufficient scheduling condition in [10] and later on improved it in [11]. It is based on the idea that if a task $T_i$ misses its deadline, there must be a certain amount of other tasks that have been executed between the release time and deadline of $T_i$. If it is possible to show that the task set cannot generate this certain amount of load, $T_i$ will never miss its deadline.

3. Bertogna et al. kept up with a similar idea, that a task must suffer certain interference by other tasks between its release time and its deadline. Since computing this interference turns out to be difficult, they present an upper bound to the interference using the workload generated by the task set. Based on that, they present a linear time scheduling condition.

The three scheduling tests are surveyed and compared in [16] and in [11]. It turns out, that they do not dominate each other. I.e. each of the scheduling tests accepts some task sets that the others do not accept. Hence the best known scheduling test is to combine all three of them [11] – and each subtest is an important contribution to the state of the art. Further, it turned out in simulation studies that the scheduling tests are still very pessimistic – meaning that they reject many task sets even if they would be successfully scheduled by global EDF [16]. Moreover, the performance achieved by the global EDF scheduling tests is still short compared to the best partitioned scheduling approaches [11]. As it has been the case for the fixed priority RM schedulers, it seems that the global EDF schedulers have their main advantages in soft real time systems, where they have to deal with inexact worst case task execution times.

**PFAIR Scheduling:** The *proportionate fairness (PFAIR)* scheduling is a global scheduling method proposed by Baruah et al. [12] [86]. Theoretical, PFAIR scheduling is optimal and has a linear-time exact scheduling test. The basic idea is, that tasks should be executed at a steady rate. To achieve this, the tasks are divided into quantum-length subtasks. In order to be optimal, the task execution times have to be multiples of the quantum-length. This results in a unrealistic small quantum-length and unrealistic many subtasks for general task systems. An option is to round the execution times to defined quantum-length but that will bring a drawback of decreasing scheduling performance. The major problem is that, due to the many subtasks, a PFAIR schedule generally introduces much more context switches than other schedulers. As in our execution model a context switch corresponds to a reconfiguration of the RHD, and the associated time-overhead is generally higher than that of microprocessors, PFAIR scheduling seems not to be adequate for reconfigurable architectures. However, studying PFAIR scheduling in our context might be a subject for future work.

### 2.4.4 Real-Time Scheduling on RHDs and Similar Architectures

Even if there has been a large amount of research on hardware-multitasking on RHDs and associated scheduling problems, up to now there has been only few research on real-time scheduling for RHD architectures.

**Steigers et al. Online Schedulers:** In [90][89], the authors first describe the design of an operating system for reconfigurable architectures that is able to execute hardware tasks according to the 1D model. E.g. each task footprint spans an arbitrary width but the entire device array height can be placed at an arbitrary horizontal array position. Further, they assume non-preemptive execution of tasks. The authors consider the problem of scheduling (and placing) online arriving real-time tasks onto the device. Each task is characterized by its arrival time $a_i$, its computation time $C_i$, its deadline $d_i$ and its required width $w_i$ of device area. Since tasks arrive online and arrival times are not known in advance, the system implements an acceptance test which either accepts a task – if and only if the scheduler could find a feasible schedule for this task, or rejects it. The challenge is to design a scheduling and placement method, such that only a minimum fraction of tasks get rejected.

Beside a reference method, they propose two scheduling algorithms for the described problem: The *reference method* simply checks whenever a task arrives, if there is free device area to place and to start it immediately. If not, the task is rejected. The other two methods increase the acceptance ratio by planning the task execution into future. Starting a task $T_i$ may be delayed until time $d_i - C_i$ and it still will meet its deadline. The *horizon* technique keeps a list of free available intervals of device area, which is an area interval that is currently free or becomes free at some future time and will not be occupied by any task already planned for future execution. This scheduling horizon, defined by the currently executing tasks and the tasks already planned for future execution, is used for searching a place for a newly arriving task. If one is found, the task is planned and the area is marked as occupied by updating the scheduling horizon.

The computational more complex *stuffing* method is an improvement, by considering all free device areas, even if they will be occupied by a planned task at some time in future. In simulation studies, the authors show that the *horizon* technique rejects considerable less tasks than the reference model, but is outperformed by the *stuffing* method. Moreover the methods have been extended to a 2D task model, where task footprints are rectangles that can be arbitrary placed. Up to now, the authors did not present bounds on the worst case behavior of their scheduling methods. Further studies considering sophisticated methods to manage the free device area have been presented in [91].

The work is strongly related to the topic of this thesis. However, it differs from our work in the following aspects. In [90], online scheduling of aperiodic tasks with unknown arrival times has been considered. Hence the scheduler has to reject arriving tasks for which the required deadline cannot be guaranteed. In contrast, we consider the execution of static task sets, where task have periodic or sporadic arrival times. Our main goal is to decide at design-time, if a task set can be scheduled or not. Some of our methods can deal with dynamic task sets as well, but in that case it is decided if a new *periodic* task can be added to the system. Another difference is, that we consider preemptive multitasking while in [90] non-preemptive multitasking has been considered.

**RT Schedulers for Hypercubes:** Hypercubes are similar to our RHD model in the aspect, that the amount of occupied computational resources may be different for each task. Where RHD tasks require a certain amount of device area, hypercube tasks require a sub-cube of certain dimension. I.e. a task occupies $1, 2, 4, \ldots$ processors of the total machine, which have to be organized in a hypercube network.

Babbar et al. studied the problem of scheduling online arriving, independent, preemptable tasks onto a hypercube and proposed the *Buddy/RT* algorithm, which is a straight forward extension of the Buddy method, and a *Stacking* algorithm, which is shown to perform better [9]. Mohaparta kept up with the same problem and improved the performance with his proposed *Deferred Earliest Deadline First* algorithm [76]. The problem of synchronous, independent, non-preemptable hypercube task scheduling has been studied by Kwon et al. [65]. While the hypercube resource model is quite similar to our RHD models, the considered scheduling problems are different since we study periodic real-time tasks within this work.

A real-time scheduling problem, considering tasks with implementation variants as we do in Section 7.1.1, has been recently studied by Lee et al. [67]. In their model, each scalable real-time task comes in certain implementations, each for a different number of processors and hence, has a different computation time. While they have studied the problem of online arriving aperiodic task scheduling, their resource model is different from our extended resource model of Section 7.1.1 as well. We require a task to allocate a *continuous* amount of RHD resources. In contrast, they assume that any free processors of the system can be used to execute a parallel task.

**Pellizzonis et al. Allocation of Real-Time Tasks to Hybrid Systems:** In [80, 79] Pellizzoni et al. consider the execution of periodic real-time tasks by a system composed of a CPU and an FPGA. Each task exists in a hardware and a software implementation and can be migrated between the two devices. An allocation assigns some tasks to the CPU, where they are scheduled according to EDF, and the other tasks to the FPGA, where they run (occupy resources) during their entire period. At runtime, new arriving tasks have to pass an acceptance test. If a task is accepted, it is always assigned to the CPU. Hence, a reconfiguration of the FPGA, which might add a considerable time overhead, is not necessary. Later on, the system tries to migrate as much load to the FPGA as possible, in order to keep the CPU utilization low. During migration, a task allocates resources on both devices. After a task bits-stream has been configured on the FPGA, the task computation is handed over from CPU to the FPGA between two task periods. Since the CPU utilization is always kept at a minimum, the probability that a new task can be accepted is maximized. In their analysis, the authors present an acceptance test which not only guaranties that the new task can be executed without jeopardizing any deadline, but also that the system can later on perform task migrations which will lead to the new optimal allocation. Further, they present a method to select the tasks to migrate from the CPU to the FPGA and vice versa.

This work is strongly related to our work since it considers periodic hardware tasks. However, it differs greatly due to the following aspects: Each task exist in a hardware and software configuration, but only the software tasks are scheduled at runtime. All tasks assigned to the FPGA run permanently and no device area is shared over time. Hence, hardware tasks are non-preemptive and they allocate resources during their entire period, even if their computation time is much less than their period. This may lead to a low utilization of the FPGA. Rather than considering the scheduling of hardware tasks onto a reconfigurable device, their work concentrates on the dynamic allocation from tasks to a CPU and FPGA.

## 2.5 Chapter Conclusion

This chapter has provided the background and related work for this thesis. We have discussed the architecture and reconfiguration features of today's RHDs and how they are employed to process hardware-tasks. Related work on multi-tasking of hardware-tasks on RHDs has been introduced in Section 2.3. We classified six resource sharing models used in literature, three based on full- and three based on partial device reconfiguration. Our resource model of Chapter 3 will cover only two of them, the most advanced model based on full device reconfiguration and the 1D partial reconfiguration model. The 2D model will not be considered directly, since it has the discussed disadvantages in complexity and practicability. Further, we have summarized research on scheduling and placement of hardware-tasks and introduced several projects, which have implemented dynamically reconfigurable systems providing multitasking environments. Hence, we conclude that multitasking of hardware-tasks on reconfigurable devices is reasonable and feasible.

Finally, in Section 2.4 we provided the background in real-time scheduling. We discussed related work in multiprocessor scheduling and showed that the basic results and methods known for processor based systems have not been developed for reconfigurable devices so far. Closing this gap is a major aim of this thesis.

Based on the gathered background, the next chapter will define our basic task and resource model and the problem of scheduling tasks with real-time constraints to reconfigurable devices.

CHAPTER 3

# Problem Modeling and Metrics

This chapter introduces the task and resource models and the basic notations used within this work. The general scheduling problem is defined. Utilization metrics are introduced that allow us to capture the *load* generated by a task set. Furthermore, some basic necessary conditions for the schedulability of task sets are derived.

## 3.1 Task and Resource Models

We consider real-time applications that consist of a set of independent aperiodic or periodic tasks which have to be scheduled for execution such that each task meets its deadline. The processing device is an RHD which provides resources to execute several tasks in parallel. Moreover, we consider preemptive multitasking, i.e. the execution of a task may be preempted by a higher priority task and resumed later on.

Our particular interest is in the widely-accepted and well-studied periodic task model introduced by Liu and Layland in [72]. This simple task model is not just one among many task models in real-time systems. It is used as a basic model in well-established textbooks such as [25](p. 71 ff.) and [24](p. 399 ff.). Moreover, the model has been used widely by researchers, such that [72] became one of the most cited research papers in computer science.[1]

However, before we come to the periodic tasks, we need to introduce the notations of aperiodic tasks, called *jobs* from now on, first.

[1]With over 1500 citations, [72] is rated as second most cited document by the CiteSeer scientific literature digital library of 2005.

**Aperiodic Real-Time Tasks:** An aperiodic real-time task is called *job* and denoted by $J_i$. Each job $J_i$ is characterized by its release time $r_i$, its deadline $d_i$, its (worst case) computation time $C_i$, and by the amount of required reconfigurable logic resources - referred to as the job's area - $A_i$. These parameters are illustrated in the scheduling diagram of Figure 3.1, where an upward arrowhead presents the task's release time and a downward arrowhead presents its deadline.

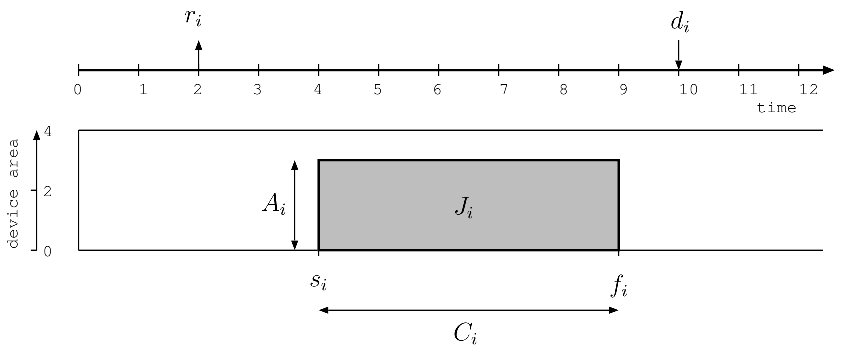


Figure 3.1: Parameters of a single aperiodic job

A job can be in several states. We call a job *active* when it has been released but not yet terminated. An active job is in one of two states, *running* or *ready*. A job is running after it has been dispatched by the scheduler and executes on the RHD. A ready job is waiting in a ready queue to be selected for execution (Figure 3.2). A job $J_i$ terminates at the latest after it has been running for $C_i$ time units. It may terminate earlier, since $C_i$ presents an upper bound on its execution time. The finishing time of a job $J_i$ is denoted by $f_i$. A set of jobs $I$ is said to be *feasibly scheduled* by some algorithm onto some execution platform, if and only if $f_i \leq d_i$ holds for all jobs.

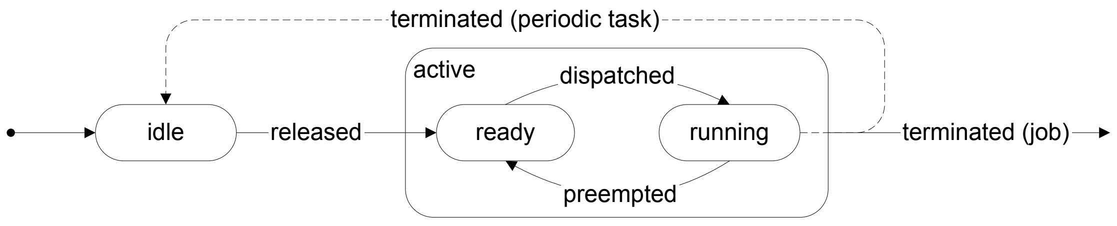


Figure 3.2: States of a job and a periodic task respectively

The notations for arbitrary (aperiodic) real-time tasks are summarized in the following list:

| | |
|---|---|
| $I$ | denotes a set of jobs |
| $J_i$ | denotes a job and refers to some concerned computation to be done |
| $C_i$ | denotes the worst case computation time of $J_i$ |
| $A_i$ | denotes the area requirement of $J_i$ |
| $r_i$ | denotes the absolute release or arrival time of $J_i$ |

$d_i$ denotes the absolute deadline of $J_i$ until which the computation has to be done
$s_i$ denotes the start time, when job $J_i$ begins its execution
$f_i$ denotes the finishing time of $J_i$

**RHD Resources:** In this work, we consider an RHD $H$ as execution platform. $H$ offers a certain amount of computational resources $A(H)$ which is also referred to as the *area* of the device. $A(H)$ may correspond to the number of configurable logic blocks of an FPGA or to the processing elements of a coarse-grain reconfigurable architecture. At any point in time the RHD $H$ can be reconfigured to execute an arbitrary set of jobs $R \subseteq I$, if all jobs in $R$ are active and fit together on the device:

$$\sum_{J_i \in R} A_i \leq A(H) \tag{3.1}$$

As we consider preemptive multitasking, an executing job $J_i$ may be preempted by a higher priority job and later on resumed. The time overhead associated with preemption will be neglected in our first scheduling analysis but will be analyzed in Chapter 6.

This job and resource model presents a generalization of the (simple) homogeneous multiprocessor model. Each instance of a set of real-time jobs to be scheduled onto an $m$ processor machine can easily be expressed using our model: The area of each job is set to 1 and the area of the device $H$ is set to $A(H) = m$.

**Periodic Real-Time Tasks:** As mentioned before, our main interest is in applications with period real-time tasks. A periodic task, often called just *task* from now on, is denoted by $T_i$ and refers to some computation which has to be performed periodically. As shown in Figure 3.2, a periodic task does not leave the system after termination but goes in idle state and waits for its next release which is triggered by a periodic timer.

Considering a periodic task set $\Gamma$, each periodic task $T_i \in \Gamma$ represents a sequence of jobs, alternatively called task instances, which are the actual objects to be scheduled for execution on the considered RHD. $T_{i,j}$ denotes the $j$th instance of the task $T_i$ and has the same form as an aperiodic job:

$$T_{i,j}(C_{i,j}, A_{i,j}, r_{i,j}, d_{i,j}) \tag{3.2}$$

The according periodic task $T_i$ is described by four parameters which are used to define the parameters of all of its task instances $T_{i,j}$:

$$T_i(P_i, D_i, C_i, A_i) \tag{3.3}$$

The relation between task and task instance parameters is illustrated in Figure 3.3, which shows an example scheduling diagram of a task $T_i(P_i = 5, D_i = 4, C_i = 3, A_i = 3)$.

The computation time $C_i$ and area $A_i$ of the periodic task directly define the computation time and area of all task instances. Hence, $C_{i,j} = C_i$ and $A_{i,j} = A_i$.

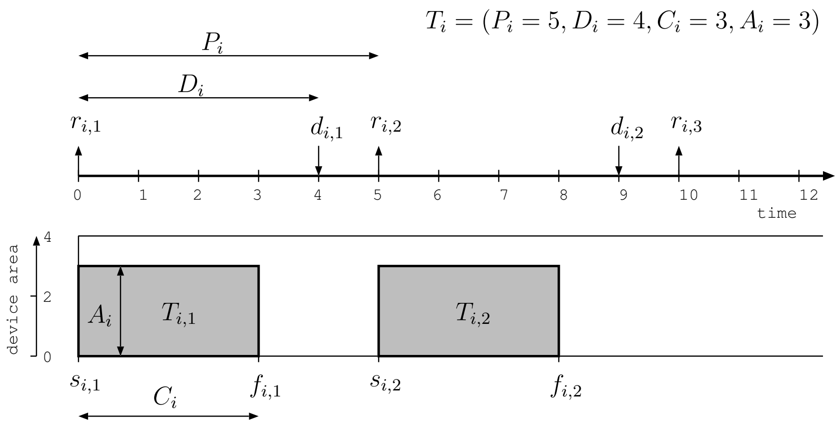


Figure 3.3: Parameters of a periodic task

The task period $P_i$ defines the release time of the task instances $T_{i,j}$. For simplification, we assume that the first instance of each task is released at $t = 0$, i.e. $r_{i,1} = 0$. We differentiate between periodic and sporadic task activations.

**Periodic task activation:** The task instances $T_{i,j}$ are requested at a fixed periodic rate. I.e. the interval between two releases of the task is fixed and given by its period $P_i$. Therefore, the release times are given by:

$$r_{i,j} = (j-1) \cdot P_i : j \geq 2 \tag{3.4}$$

**Sporadic task activation:** For a sporadic task, the parameter $P_i$ presents a minimum inter-arrival time. The distance between two task requests may be as short as $P_i$, but can be longer as well. Therefore, the release time $r_{i,j}$ of $T_{i,j}$ is at least:

$$r_{i,j} \geq r_{i,j-1} + P_i : j \geq 2 \tag{3.5}$$

The relative deadline $D_i$ of a periodic task defines the absolute deadlines of the task instances $T_{i,j}$. The deadline of a task instance is generally given by its release time plus the relative deadline:

$$d_{i,j} = r_{i,j} + D_i \tag{3.6}$$

In our scheduling analysis, we restrict this general form and distinguish two cases:

**Implicit deadline:** ($D_i = P_i$) Most often, we assume that the relative deadline is equal to the period of the task. I.e. the deadline of one task instance is equal to the release time of the next instance. This simplifies our scheduling analysis. Whenever we do not explicitly specify a relative deadline $D_i$ for a task, we assume that it is equal to the task's period.

**Constrained deadline:** $(D_i \leq P_i)$ In this more general form, the relative deadline is less or equal to the task's period. I.e. the deadline of one task instance can be at some time before, but not later than the release time of the next task. Some of our results can be easily extended to consider this more general form of task sets.

The notations used for periodic real-time tasks and their instances are summarized in the following list:

| | |
|---|---|
| $\Gamma$ | denotes a set of periodic or sporadic tasks |
| $T_i$ | denotes a generic periodic task of $\Gamma$ |
| $T_{i,j}$ | denotes the $j$the instance of task $T_i$. |
| $P_i$ | denotes the period of $T_i$, |
| $D_i$ | denotes the relative deadline of $T_i$ |
| $C_i$ | denotes the computation time of $T_i$ and all its task instances |
| $A_i$ | denotes the area requirement of $T_i$ and all its task instances |
| $r_{i,j}$ | denotes the absolute release or arrival time of $T_{i,j}$ |
| $d_{i,j}$ | denotes the absolute deadline of $T_{i,j}$ |
| $s_{i,j}$ | denotes the start time, when $T_{i,j}$ begins its execution |
| $f_{i,j}$ | denotes the finishing time of $T_{i,j}$ |

## 3.2 Feasible Schedule

Generally speaking, we would like to schedule and execute a set of jobs $I$, such that each job meets its deadline. In the case that we consider a periodic task set $\Gamma$ the jobs of $I$ are the task instances $T_{i,j}$ of the periodic tasks.

Formally a schedule can be described by a function[2] $R : \mathbb{R}^+ \rightarrow \mathbf{P}(I)$. $R(t)$ denotes the set of jobs from $I$ running at time $t$. A schedule $R$ is called *feasible* if each job $J_i$ is scheduled for execution for at least $C_i$ time units within the interval given by its release time $r_i$ and its deadline $d_i$. In order to be *feasible* on a particular device $H$, the schedule $R(t)$ must keep at any time $t \geq 0$ the bound on device resources given by Equation 3.1.

An example task set $\Gamma^*$ of four tasks and a feasible schedule are shown in Figure 3.4. The table specifies the period, computation time and area of each task. The $U_i$ values will be explained in the following section. The scheduling diagram presents a feasible schedule of $\Gamma^*$ on a device with an area of $A(H) = 8$. While the time-line shows the release times and deadlines of the tasks, the lower part of the figure illustrates which tasks instances are in execution at a certain moment. The vertical dashed lines denote, that the task instance $T_{3,1}$ is preempted at $t = 4$ and resumed at $t = 6$. The feasibility of the schedule can be easily proven: It is obvious that for the time interval $0 \leq t \leq 12$ each released task instance terminates before its deadline and that the device resources are not exceeded. This interval of length 12 has a special meaning, since it presents the *hyper-period* of the task set $\Gamma^*$. The hyper-period is the least common multiplier of the periods of a task set. The pattern of task releases repeats with the hyper period

[2] $P(I)$ denotes the power set of $I$

of a task set. Hence, a feasible schedule defined over the hyper-period can be repeated infinity times without any missed deadline.

| $T_i$ | $P_i$ | $C_i$ | $A_i$ | $U_i^T$ | $U_i^S$ |
|---|---|---|---|---|---|
| $T_1$ | 4 | 2 | 4 | 1/2 | 2 |
| $T_2$ | 6 | 5 | 2 | 5/6 | 5/3 |
| $T_3$ | 12 | 3 | 6 | 1/4 | 3/2 |
| $T_4$ | 12 | 2 | 2 | 1/6 | 1/3 |
| $\sum$ | | | | 1.75 | 5.5 |

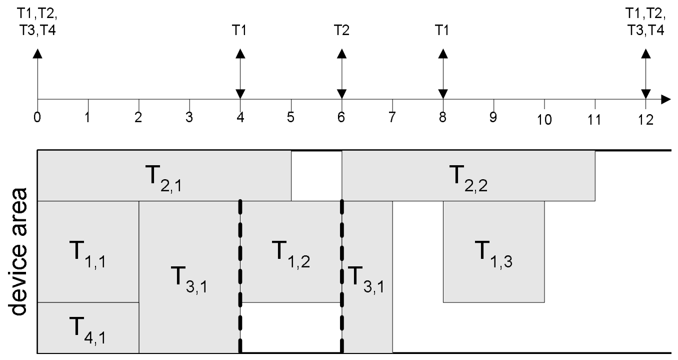


Figure 3.4: Example task set $\Gamma^*$ and a feasible schedule.

**Scheduling challenges:** We are interested in developing various scheduling algorithms and finding answers for each algorithm to the following questions:

1. Given a task set $\Gamma$, can we execute it on a reconfigurable device $H$ with area $A(H)$, such that all deadlines are met?
2. What is the size of the smallest device on which $\Gamma$ can be feasibly executed?
3. How does this feasible schedule $R$ look like?

## 3.3 Utilization Metrics

We define two utilization metrics to measure the computational load generated by a periodic task $T_i$ or a set of tasks $\Gamma$, respectively. Similar to the processor utilization factor defined in single processor real-time scheduling, we define the *time-utilization factor* of a task $T_i$ to be:

$$U^T(T_i) = \frac{C_i}{P_i} \tag{3.7}$$

It presents the fraction of time a certain task occupies the device in order to complete its execution. Respectively we define the *time-utilization factor* for the complete task set $\Gamma$ to be:

$$U^T(\Gamma) = \sum_{T_i \in \Gamma} \frac{C_i}{P_i} \tag{3.8}$$

For the special case that all tasks are executed sequentially, $U^T(\Gamma)$ is the fraction of time the RHD spends executing tasks whereas $1 - U^T(\Gamma)$ is the idle time.

Since a running task usually does not occupy the entire device, we define a *system-utilization* metric that captures the degree by which the device is utilized by $T_i$:

$$U^S(T_i) = U^T(T_i) \cdot A_i \tag{3.9}$$

In order to measure the load generated by a task set $\Gamma$, we now define the total *system-utilization* of a task set by summing up the system utilization factors of all tasks:

$$U^S(\Gamma) = \sum_{T_i \in \Gamma} U^S(T_i) \cdot A_i \tag{3.10}$$

Since in general we consider devices $H$ with an arbitrary area $A(H)$, we define the *relative system-utilization* as the fraction a certain task set $\Gamma$ utilizes a particular device $H$:

$$U^{RS}(\Gamma, H) = \frac{U^S(\Gamma)}{A(H)} \tag{3.11}$$

Often we simply consider devices $H$ of size $A(H) = 1$. In this case, there will be no difference between $U^S$ and $U^{RS}$. Whenever it is clear from the context that we consider the utilization of a task $T_i$, we may use the notation $U_i^T$ instead of $U^T(T_i)$ and $U_i^S$ instead of $U^S(T_i)$ respectively. The time and system utilization factors of the example task set $\Gamma^*$ are shown in the table of Figure 3.4.

**Schedulability Conditions:** The question whether a given task set $\Gamma$ can be feasibly scheduled onto a device $H$ by any algorithm, or if it will be feasibly scheduled by a particular algorithm, is not trivial to answer. Obviously some necessary conditions are, that:

- Each task must have a time-utilization not greater than 100%:
  $U^T(T_i) \leq 1 : \forall T_i \in \Gamma$
- Each task must fit onto the device:
  $A_i \leq A(H) : \forall T_i \in \Gamma$
- The relative system utilization must not be greater than 100%:
  $U^{SR}(\Gamma, H) \leq 1$

However, these conditions are only necessary and not sufficient. In particular there exist task sets with an infinitesimal small system-utilization, which cannot be scheduled by *any* algorithm. As example consider a task set $\Gamma = \{T_1, T_2\}$ with $P_1 = P_2 = 10$ where the first task has a computation time of $C_1 = \epsilon$ and an area of $A_1 = A(H)$ while the parameters of the second task are $C_2 = 10$ and $A_2 = \epsilon$. The system utilization of $\Gamma$ computes to $\lim_{\epsilon \to 0} U^S(\Gamma) = \frac{\epsilon}{10} A(H) + \epsilon = 0$. However, since both tasks cannot be executed in parallel the task set cannot be feasibly scheduled, as illustrated in Figure 3.5.

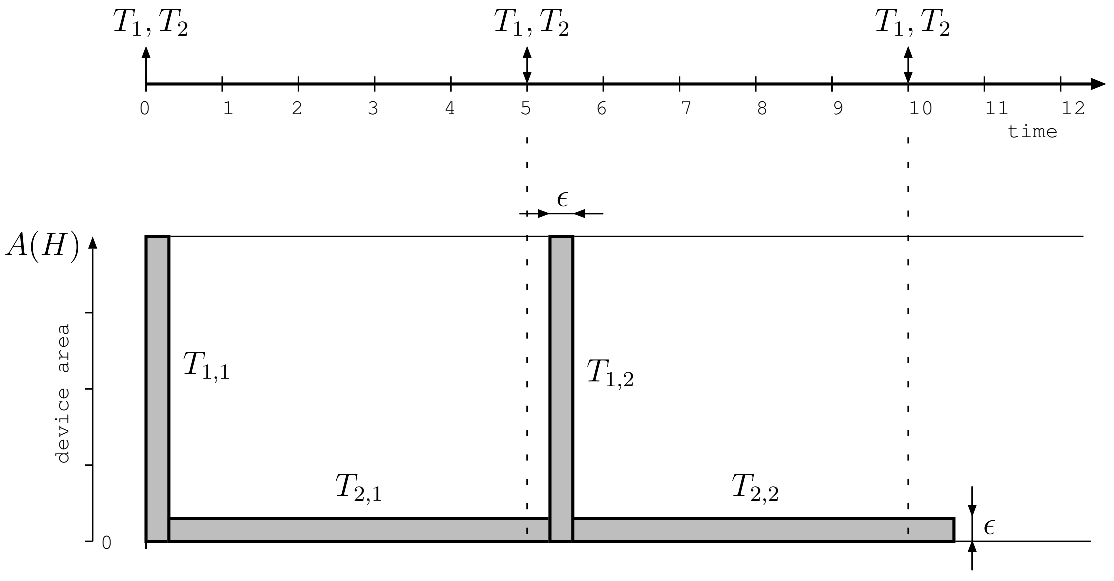


Figure 3.5: Scheduling Anomaly: An infeasible task set with infinitesimal system utilization

## 3.4 Chapter Conclusion

This chapter has defined our model of real-time applications for execution on reconfigurable devices. In particular we defined the notations for aperiodic jobs and periodic task sets. The execution model allows task preemption. The resource model has been kept rather simple: A task requires a certain (scalar) amount of logic resources and a device can execute tasks in parallel as long as its resources are not exceeded. This simple model generalizes the common multi-processor models, where each task requires exactly one processor. As another simplification we neglected the time overhead associated with the device reconfiguration. However, it will be included into our analysis in Chapter 6.

For our simple task model, we have defined the properties of a feasible task schedule. Moreover, utilization metrics that capture the load of a periodic task set have been defined. On this basis, the next chapter will propose various scheduling strategies. The goal is to guaranty at design time, that a particular task set will be feasibly scheduled by a certain algorithm.

CHAPTER 4

# Three Scheduling Algorithms

Within this chapter, we introduce three main methods to schedule periodic task sets to RHDs according to the model described in the preceding chapter, namely *global EDF scheduling*, *partitioned EDF scheduling* and *server based scheduling*.

1. The idea of global scheduling is, to manage the resources globally, and dynamically assign ready tasks to free resources whenever they become available. As a consequence, a task might get assigned different resources over runtime. For example in case of a multiprocessor machine a task may be started on one processor and be resumed on another. While this method leads to very good resource utilization in average, it is rather difficult to predict the schedulability of a task set.

   In Section 4.1 the multiprocessor global EDF scheduling method is adapted to our task model in a straight forward way and yields two global scheduling algorithms, namely *EDF-NextFit* and *EDF-First-k-Fit*. We develop a scheduling test based on the task set parameters that can be evaluated in linear-time. In our analysis we prove, that both algorithms produce a feasible schedule when the test conditions are satisfied.

   We presented these algorithms in [DP05a, DP05c] and the scheduling analysis in [DP06a].

2. The idea of partitioned scheduling is to partition a task set and to assign each partition a fraction of the platform resources off-line. In case of a multiprocessor machine this means that a particular task is always executed on the same processor.

   In Section 4.2 we adapt the multiprocessor partitioned EDF scheduling method to our resource model. We formulate the problem of finding a feasible partitioning of a given task set. We discuss the relation to two-dimensional packing problems and apply two methods to solve it. The first is called *optimal-partitioned-EDF* and uses integer linear programming (ILP) to solve the problem to optimality. Since solving the ILP takes exponential time, the second method is a simple and fast heuristic called *Next-Fit-Deceasing-Area*. For the heuristic we develop and prove a scheduling condition which can be evaluated in linear time.

Most of our results on partitioned scheduling have been presented in [DP06b].

3. The idea of server based scheduling is to execute application tasks by (virtual) server tasks. While periodic server tasks are usually used to serve the requests of aperiodic application tasks, we present a totally different and novel approach.

   In Section 4.3 we present a method to schedule *periodic* tasks onto an RHD by means of periodic servers. Tasks are grouped to servers for parallel execution but the computation of tasks may be distributed among many servers. We present a heuristic called *Merge-Servers-Distribute-Load (MSDL)* that creates a set of servers in such a way, that all application tasks are guaranteed to meet their deadlines. Furthermore, the number of servers is kept small. As a result, each server can be compiled to a separate RHD programming file (bit-stream) which makes the approach feasible for all reconfigurable devices and not only for partial reconfigurable RHDs.

   We presented our results on server bases scheduling in [DP05a, DP05c].

Hence the chapter extends the two popular multiprocessor scheduling approaches to our RHD model and presents a novel approach especially designed to avoid the restrictions of partial device reconfiguration.

## 4.1 Global Scheduling

In a global EDF schedule for a multiprocessor platform, all active tasks are sorted according to non-decreasing deadlines in a global ready-queue. If $m$ is the number of processors, the first $m$ tasks of the ready-queue are selected for execution. As a consequence, a task might be started on one processor, becomes preempted by a higher priority task and, later on, be resumed on another processor. While this method can lead to very good resource utilization, it is rather difficult to predict the scheduleability of a task set.

However, within this section we adapted this method to our task and resource model and define two global scheduling algorithms. We develop a scheduling test based on task set utilization parameters that can be evaluated in linear-time. In our analysis we prove, that both algorithms produce a feasible schedule when the test conditions are satisfied.

### 4.1.1 Earliest Deadline First on RHDs

The *Earliest Deadline First* rule has been proven optimal for single processor systems and has also been successfully used in global (non-partitioning) multiprocessor scheduling [81, 48]. In multiprocessor systems, the number of active jobs that can be executed in parallel depends only on a platform parameter, i.e., the number of processors. Applying EDF to our RHD execution model, we find that the number of active jobs that can be executed in parallel shows a more complex relation between the platform parameter $A(H)$ and parameters of the job set, $A_i$. Therefore, we see at least two possibilities to define EDF for our RHD model, which we call *EDF-First-k-Fit* and *EDF-Next-Fit* respectively. *EDF-First-k-Fit* executes jobs according to the following definition:

**Definition 2** (EDF-First-k-Fit)**.** *Let $Q = (J_1, \ldots, J_m)$ be the list of all active jobs, w.l.g. sorted after non-decreasing deadlines.* EDF-First-k-Fit *selects at any time the first k jobs $R = \{J_1 \ldots J_k\}$ of $Q$ for execution, with the largest $k$ for which $\sum_{J_i \in R} A_i \leq A(H)$ holds.*

In addition to the formal definition, we present a pseudo code implementation of the *EDF-FkF* dispatching procedure in Algorithm 1. It is called by the scheduler each time a new job is activated or a running job terminates.

Note, that according to this definition *EDF-First-k-Fit (EDF-FkF)* creates a preemptive schedule, since a newly activated job $J_i$ may cause a preemption if it has a shorter deadline than any of the running jobs. Moreover, *EDF-FkF* executes tasks strictly in deadline order in the sense that for any pair of active jobs, $J_i \in R, J_j \notin R \Rightarrow d_i \leq d_j$. It follows that *EDF-FkF* is *priority driven* according to the definition given in [48].

The *EDF-FkF* algorithm may leave some RHD resources idle, even if there are ready jobs that would fit onto the device in addition to the first $k$ jobs selected for execution. In order to exploit these resources, we define an enhanced algorithm called *EDF-Next-Fit (EDF-NF)*:

**Algorithm 1** Earliest Deadline First - First k Fit

```
Require: list $Q$ of active tasks, sorted by non-decreasing absolute deadlines
 1: procedure EDF-NF($Q, H$)
 2:     $R \leftarrow \emptyset$
 3:     $A^{running} = 0$
 4:     $k = 1$
 5:     while $A^{running} + A_k \leq A(H)$ and $|Q| \geq k$ do
 6:         $R \leftarrow R \cup J_k$
 7:         $A^{running} \leftarrow A^{running} + A_k$
 8:         $k \leftarrow k + 1$
 9:     end while
10:     return $R$
11: end procedure
```

**Definition 3** (EDF-Next-Fit)**.** *At any time, the set of running tasks $R$ is determined according to the following algorithm: Start with $R \leftarrow \emptyset$ and scan all active jobs $J_i \in Q$ in order of non-decreasing deadlines. $J_i$ is added to $R$ if and only if $\sum_{J_k \in R \cup J_i} A_k \leq A(H)$.*

Again, we present a pseudo code implementation of the *EDF-NF* dispatching procedure in Algorithm 2.

**Algorithm 2** Earliest Deadline First - Next Fit

```
Require: list $Q$ of active tasks, sorted by non-decreasing absolute deadlines
 1: procedure EDF-NF($Q, H$)
 2:     $R \leftarrow \emptyset$
 3:     $A^{running} = 0$
 4:     for $i \leftarrow 1, |Q|$ do
 5:         if $A^{running} + A_i \leq A(H)$ then
 6:             $R \leftarrow R \cup J_i$
 7:             $A^{running} = A^{running} + A_i$
 8:         end if
 9:     end for
10:     return $R$
11: end procedure
```

A part of the *EDF-FkF* and *EDF-NF* schedules for an example task set is shown in Figure 4.1. The difference between the two schedules lies in the execution of $T_{4,1}$. *EDF-FkF* schedules $T_{4,1}$ not before $T_{3,1}$ has terminated (hatched task), whereas *EDF-NF* schedules $T_{4,1}$ as soon as there are idle resources (speckled task).

**Lemma 1.** EDF-NF *is superior to* EDF-FkF *in the sense that if a set of jobs $I$ is feasibly scheduled by* EDF-FkF*, the same is true for* EDF-NF*.*

*Proof Sketch.* Let $R(t)$ and $Q(t)$ be the set of running and active jobs at time $t$ in the *EDF-FkF* schedule and $\tilde{R}(t)$ $\tilde{Q}(t)$ in the *EDF-NF* schedule, respectively. We can

| $T_i$ | $P_i$ | $C_i$ | $A_i$ | $U_i^T$ | $U_i^S$ |
|---|---|---|---|---|---|
| $T_1$ | 4 | 2 | 4 | 1/2 | 2 |
| $T_2$ | 6 | 5 | 2 | 5/6 | 5/3 |
| $T_3$ | 12 | 3 | 6 | 1/4 | 3/2 |
| $T_4$ | 12 | 2 | 2 | 1/6 | 1/3 |
| $\sum$ | | | | 1.75 | 5.5 |

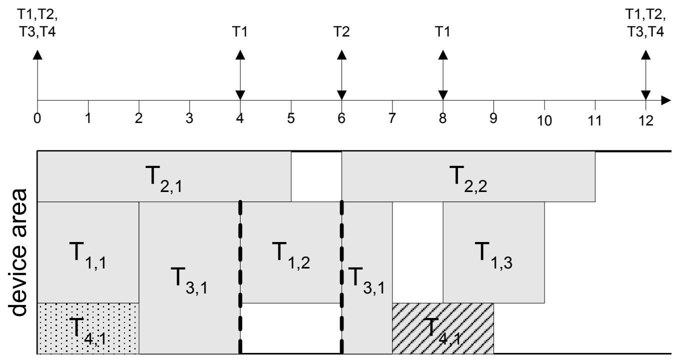


Figure 4.1: A schedule for $\Gamma^*$ by *EDF-FkF* vs. *EDF-NF*

observe that for an equal set of active jobs, *EDF-NF* will execute at least the same jobs as *EDF-FkF*, i.e., if $Q = \tilde{Q}$ then $R \subseteq \tilde{R}$. Further, if the active jobs in the *EDF-NF* schedule become a subset of those in the *EDF-FkF* schedule due to earlier terminations, each job that is active in both schedules and executed by *EDF-FkF* will also be executed by *EDF-NF*, i.e., if $J_k \in \tilde{Q} \subseteq Q$ then $J_k \in R \Rightarrow J_k \in \tilde{R}$.

Since both schedulers start with identical sets of active jobs, $Q(0) = \tilde{Q}(0)$, and experience the same job activations over time, it follows that $\tilde{Q}(t) \subseteq Q(t)$ for all $t > 0$ and every job $J_k$ scheduled by *EDF-NF* will have a finishing time smaller or equal to the finishing time when scheduled by *EDF-FkF*.

□

### 4.1.2 Relation to Multiprocessor EDF

The scheduling test we will present for *EDF-NF* is related to an earlier reported scheduling test for EDF upon multiprocessors. In fact, when all RHD tasks have equal area $A_i = A_{const}$, our execution model merges into a multiprocessor with $\lfloor \frac{A(H)}{A_{const}} \rfloor$ processors. Therefore, we first introduce the utilization-based scheduling test for identical multiprocessors presented by Goossens, Funk and Baruah [48]:

**Theorem 2** ([48])**.** *A periodic task set $\Gamma$ can be feasibly scheduled by* EDF *onto a multiprocessor of $m$ identical unit capacity processors, if:*

$$U^T(\Gamma) \leq U^T_{max} + m \cdot (1 - U^T_{max}) \tag{4.1}$$

*where $U^T_{max}$ is the task with the highest time utilization appearing in $\Gamma$.*

The proofs for this theorem and its tightness can also be found in [48]. We provide an example here that illustrates the scheduling test using a geometrical explanation and in addition proves the tightness of the bound of (Equation 4.1). Assume five tasks to be scheduled on an $m = 4$ processor machine. $T_1$ to $T_4$ have equal periods of $P_i = 6$ and equal computation times of $C_i = 2$, respectively. Task $T_5$ has also a period of $P_5 = 6$ and a computation time of $C_5 = 4$. As EDF may schedule tasks with equal deadlines in arbitrary order, Figure 4.2 presents a valid EDF schedule. From time 2 on, $T_5$ is the only active task which results in idle times for three processors. An infinitesimal increase of the computation times of tasks $T_1$ to $T_4$ to $C_i = 2 + \frac{6}{4}\varepsilon$ would cause task $T_5$ to miss its deadline. The theorem condition is tight, since:

$$\forall \varepsilon > 0 : U^T(\Gamma) = 2 + \varepsilon > U^T_{max} + m(1 - U^T_{max}) = 2 \tag{4.2}$$

Theorem 2 limits the maximum utilization a feasibly schedulable task set can have in dependence of the maximal utilization $U^T_{max}$ and the number of processors $m$. This limit can be derived by accumulating the utilization-rectangles shown in the utilization diagram on the right side of Figure 4.2. The task with the highest time utilization occupies a $1 \times U^T_{max}$ rectangle, while all other tasks together occupy a $m \times (1 - U^T_{max})$ rectangle. Adding both terms derives the upper bound on $U(\Gamma)$ shown in Theorem 2.

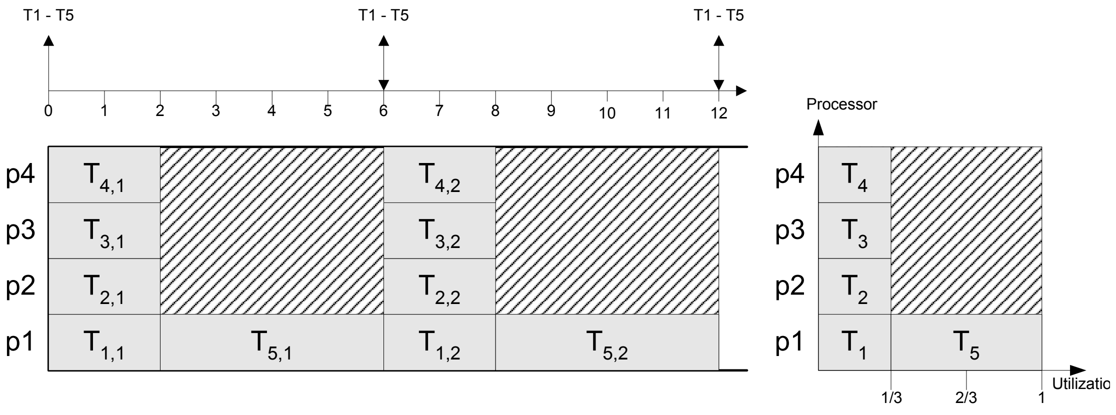


Figure 4.2: EDF multiprocessor schedule

### 4.1.3 Schedulability Analysis for EDF-FkF

This section presents a scheduling test for the *EDF-FkF* algorithm applied to a periodic task set executing on an RHD. It is a *sufficient* but not *necessary* condition, i.e., it will reject any task set that *EDF-FkF* fails to schedule on the considered RHD, but may also reject task sets that could indeed be feasibly scheduled by *EDF-FkF*. Since *EDF-NF* is superior to *EDF-FkF*, the theorem holds for both algorithms.

**Theorem 3** (EDF-FkF scheduling condition)**.** *Any periodic task set* $\Gamma$ *can be feasibly scheduled by* EDF-FkF *onto an RHD* $H$ *with area* $A(H) \geq A_{max}$*, if:*

$$\begin{aligned}&\forall T_k \in \Gamma : \\ &U^S(\Gamma) \leq (A(H) - A_{max}) \cdot \left(1 - U^T(T_k)\right) + U^S(T_k)\end{aligned} \tag{4.3}$$

*where $A_{max}$ is the largest area of all tasks in $\Gamma$.*

The proof of Theorem 3 follows a resource augmentation approach [81] and is closely related to the multiprocessor scheduling test of [48] which we have just discussed in Section 4.1.2. We will proceed in three steps:

1. We construct a theoretical multi-RHD machine $\pi$, on which a given task set $\Gamma$ can be feasibly scheduled by some algorithm $OPT$ (Section 4.1.3.1).
2. We calculate the required size of a (unit-speed) single-RHD machine $H$ such that any algorithm with a property called $\alpha$-*work-conserving*, applied to the task set $\Gamma$ does at any time at least as much *work* on the task set $\Gamma$ as the algorithm $OPT$ on the multi-RHD machine (Section 4.1.3.2).
3. We prove that *EDF-FkF* fulfills the $\alpha$-*work-conserving* property and show, that it produces a feasible schedule for $\Gamma$ on $H$ (Section 4.1.3.3).

#### 4.1.3.1 Machine Definitions

First, we refine the RHD model by defining a speed factor for an RHD. While real devices do have different speed grades, we use speed here purely as a means to construct the proof for Theorem 3.

**Definition 4** (RHD)**.** *An RHD $H$ is a processing device with area $A(H)$ and speed $S(H)$. It can execute a set of jobs $R$ simultaneously, iff $\sum_{J_i \in R} A(J_i) \leq A(H)$. If a job $J_i$ is in execution on $H$ for $t$ units of time, it completes $S(H) \cdot t$ units of its execution requirement, i.e., of its computation time $C_i$. We define the computing capacity of $H$ as $Cap(H) = A(H) \cdot S(H)$.*

**Definition 5** (multi-RHD)**.** *A multi-RHD $\pi$ is a set of RHDs $\{H_1, H_2, \dots\}$, each with its own area $A(H_j)$ and speed $S(H_j)$. At each point of time, each RHD $H_j$ can execute its individual set $R_j$ of jobs, iff $\sum_{J_i \in R_j} A(J_i) \leq A(H_j)$ for all $H_j \in \pi$. The capacity of $\pi$ is defined as $Cap(\pi) = \sum_{H_j \in \pi} Cap(H_j)$.*

Now we are able to define a specific multi-RHD machine where for each task an RHD is reserved that matches exactly the task's area and has a speed which fully utilizes the device:

**Definition 6** (feasible-multi-RHD)**.** *For a given periodic task set $\Gamma$, we define a multi-RHD $\pi$ with capacity $Cap(\pi) = U^S(\Gamma)$ such that for any task $T_i \in \Gamma$ there is an RHD $H_j \in \pi$ with $A(H_j) = A_i$ and $S(H_j) = U^T(T_i)$. Algorithm $OPT$ assigns each task $T_i$ to its corresponding RHD $H_j$.*

By construction, all task instances will meet their deadlines under $OPT$ on this multi-RHD machine.

#### 4.1.3.2 Work Done by Algorithms

A uniprocessor scheduling algorithm is *work-conserving*, if it never idles the processor while there are any active jobs left in the ready queue. In the same way, [48] extended the definition to identical multiprocessors: a scheduling algorithm for identical multiprocessors is work-conserving, if it never idles any processor while there are any active jobs in the ready queue. By these definitions, EDF is work-conserving for both the uniprocessor and the identical multiprocessor model.

We now apply the concept of *work-conservation* to RHD machines. In an RHD machine the utilization of the device depends on the areas of the jobs. Therefore, we base our definition of work-conserving on the minimal utilized RHD area:

**Definition 7** ($\alpha$-work-conserving)**.** *An algorithm for scheduling jobs onto an RHD $H$ is said to be $\alpha$-work-conserving, iff it utilizes at least a fraction $\alpha$ of the device area $A(H)$ whenever there are active jobs waiting in the ready queue. In an* overload *situation there are tasks in the ready queue waiting for execution. In contrast, in an* underload *situation all active jobs are running.*

As a next step, we define a function $W$ that captures the amount of work done on a given job or job set by some algorithm on some machine.

**Definition 8** (Work-done function)**.** *A job with computation time $C_i$ and area $A_i$ represents $C_i \cdot A_i$ work. If the job has been executed for $t$ time units on an $S(H)$ speed RHD, the work that has been done on this job is $t \cdot S(H) \cdot A_i$. Let $I$ denote any set of jobs and $\pi$ any multi-RHD platform. For any algorithm alg and time instance $t \geq 0$, let $W(alg, \pi, I, t)$ denote the amount of work done on jobs of $I$ over the interval $[0, t)$, when $I$ is scheduled by alg on $\pi$.*

The following lemma states that on a specific single RHD $H$, an *$\alpha$-work-conserving* algorithm will never do less work than the algorithm $OPT$ on the reference platform $\pi$ of Definition 6.

**Lemma 2.** *Let $\Gamma$ be a periodic task set with at least two tasks and $U^T_{max} < 1$. Let $I$ be the related set of jobs produced by $\Gamma$. Let $\pi$ and $OPT$ be the multi-RHD machine and scheduling algorithm according to Definition 6. Further, let $H$ be a single RHD with speed $S(H) = 1$. If*

$$\forall H_k \in \pi : \quad \alpha \cdot A(H) \geq \frac{Cap(\pi) - Cap(H_k)}{1 - S(H_k)} \tag{4.4}$$

*the work done on $I$ until any point in time $t \geq 0$ by any $\alpha$-work-conserving algorithm $\alpha WC$ on RHD $H$ is never less than the work done on $I$ by algorithm $OPT$ on the multi-RHD $\pi$:*

$$W(\alpha WC, H, I, t) \geq W(OPT, \pi, I, t) \tag{4.5}$$

*Proof.* Suppose that while Equation 4.4 holds, in contradiction to Lemma 2 there exists a time instance $t_0$ such that

$$W(\alpha WC, H, I, t_0) < W(OPT, \pi, I, t_0), \tag{4.6}$$

i.e., algorithm $\alpha WC$ does less work on $H$ than algorithm $OPT$ on $\pi$. Then there must exist at least one job $T_{k,j}$ with release time $r_{k,j} < t_0$ that has not finished execution at time $t_0$, and during the interval $[r_{k,j}, t_0)$ the algorithm $\alpha WC$ did strictly less work than algorithm $OPT$.

According to Definition 7 the single RHD machine is either in *overload* or *underload* situation. Note that a fully utilized RHD with an empty ready queue denotes an underload situation. Let $x$ denote the portion of time in interval $[r_{k,j}, t_0)$ where the system is *overloaded*, and $y$ the portion where it is *underloaded*.

Considering job $T_{k,j}$ during time interval $[r_{k,j}, t_0)$, we can observe that:

- Algorithm $\alpha WC$ executes $T_{k,j}$ for at least $y$ time units, i.e., it does at least $y \cdot A_k$ work on $T_{k,j}$.
- Algorithm $OPT$ executes $T_{k,j}$ for at most $x + y$ time units, i.e., $OPT$ does at most $(x + y) \cdot S(H_k) \cdot A_k$ work on $T_{k,j}$.
- By our contradiction assumption, $OPT$ performs more work on $T_{k,j}$ than $\alpha WC$:

$$(x + y) \cdot S(H_k) \cdot A_k > y \cdot A_k \tag{4.7}$$

Now we consider the work done on the entire set of jobs $I$ during interval $[r_{k,j}, t_0)$. We can observe that:

- The overall work done by algorithm $\alpha WC$ on $H$ is at least $x \cdot \alpha \cdot A(H) + y \cdot A_k$. During overload the RHD is utilized by at least a fraction of $\alpha$ and during underload at least $T_{k,j}$ must be executing since it has not finished until $t_0$.
- The overall work done by algorithm $OPT$ on $\pi$ is at most $(x + y) \cdot Cap(\pi)$, which corresponds to a full utilization of all RHDs.
- By our contradiction assumption, $OPT$ performs more work on $I$ than $\alpha WC$:

$$(x + y) \cdot Cap(\pi) > x \cdot \alpha \cdot A(H) + y \cdot A_k \tag{4.8}$$

Now we show that Equations 4.7 and 4.8 cannot be true, if Equation 4.4 in Lemma 2 holds. To that end, we multiply Equation 4.7 by $(\frac{\alpha \cdot A(H)}{A_k} - 1)$ and add it to Equation 4.8:

$$\begin{multline}(\frac{\alpha \cdot A(H)}{A_k} - 1) \cdot (x + y) \cdot S(H_k) \cdot A_k + (x + y) \cdot Cap(\pi) > \\ (\frac{\alpha \cdot A(H)}{A_k} - 1) \cdot y \cdot A_k + x \cdot \alpha \cdot A(H) + y \cdot A_k \end{multline} \tag{4.9}$$

$$[(\frac{\alpha \cdot A(H)}{A_k} - 1) \cdot S(H_k) \cdot A_k + Cap(\pi)](x + y) > \alpha \cdot A(H)(x + y) \tag{4.10}$$

$$Cap(\pi) - Cap(H_k) > \alpha \cdot A(H) \cdot (1 - S(H_k)) \tag{4.11}$$

Equation 4.11 contradicts the condition of Equation 4.4. Hence, the proof assumption must be wrong and Equation 4.5 holds which proves the lemma. □

#### 4.1.3.3 EDF Feasibility

As the last step, we will show that *EDF-FkF* is an $\alpha WC$ algorihtm.

**Lemma 3.** EDF-FkF *is an $\alpha$-work-conserving algorithm, with*

$$\alpha = 1 - \frac{A_{max}}{A(H)}, \tag{4.12}$$

*where $A_{max}$ is the largest of all task areas.*

*Proof Sketch.* The lemma states that in an overload situation, at most $A_{max}$ area of the RHD stays idle. Assume, in contradiction, that *EDF-FkF* selects the first $k$ jobs out of the list of active jobs $Q = J_1, \ldots, J_m$ for execution, such that the amount of free area $A(H) - \sum_{i=1}^{k} A_i$ would be greater than $A_{max}$. In this case, the next ready job $J_{k+1}$ could also be executed in addition to the first $k$ tasks as its area is bound by $A_{max}$. Then, the assumption must be wrong. □

*Proof of Theorem 3.* We have made the following observations:

1. For the special case, that $\Gamma$ contains a task $T_k$ with $U^T(T_k) = 1$, the condition of Theorem 3 simplifies to $U^S(\Gamma) \leq U^S(T_k)$ and can only be fulfilled for task sets with only one task. Obviously, this task will always meet its deadlines.

2. For the general case we know from Definition 6, that for any task set $\Gamma$ with $U^T_{max} < 1$ there exists a multi-RHD $\pi$ with $Cap(H_k) = U^S(T_k)$ on which $\Gamma$ can be feasibly scheduled by $OPT$.

3. We know from Lemma 2 that the work done until some time $t \geq 0$ by any $\alpha$-work-conserving algorithm on $H$ is at least as much as the work done by $OPT$ on $\pi$. The area of the single RHD $H$ is determined according to Equation 4.4.

4. We know from Lemma 3 that *EDF-FkF* is $\alpha = (1 - \frac{A_{max}}{A(H)})$ -work-conserving. Hence Equation 4.4 holds for *EDF-FkF*.

5. We know by definition that the work done by *EDF-FkF* is always on the most urgent jobs, i.e., the ones with the closest deadlines. It remains to show, that as the work done by *EDF-FkF* is i) never less than the work done by $OPT$ on $\pi$ and ii) always done on the most urgent jobs, all jobs must meet their deadlines under *EDF-FkF* on $H$. Let $I$ be the set of jobs indexed by decreasing *EDF-FkF* priority (non-decreasing deadlines). Considering only the scheduling of $\{J_1\}$, *EDF-FkF*

clearly meets the deadline since it does at least the same work until $d_1$ than the algorithm $OPT$. Assume the *EDF-FkF* schedule of the set $\{J_1, \ldots, J_k\} \subset I$ is feasible. The *EDF-FkF* schedule of the increased set $\{J_1, \ldots, J_{k+1}\}$ will be equivalent to the former one, except for job $J_{k+1}$, thus $J_1$ to $J_k$ will still meet their deadlines. Since by time $d_{k+1}$, *EDF-FkF* has done at least the same work than OPT, also $J_{k+1}$ meets its deadline. It follows by induction, that all jobs of $I$ meet their deadlines under *EDF-FkF* on $H$.

The condition of Theorem 3 is derived from Equation 4.4 by replacing $Cap(\cdot)$ by $U^S(\cdot)$, $S(\cdot)$ by $U^T(\cdot)$, $H_k$ by $T_k$ and, finally, substituting $\alpha$ according to Equation 4.12. $\square$

#### 4.1.3.4 Example Illustrating Tightness of the EDF-FkF Scheduling Condition

This subsection gives an example that illustrates an *EDF-FkF* schedule and shows that the condition of Theorem 3 is tight.

| $T_i$ | $P_i$ | $C_i$ | $A_i$ | $U^T(T_i)$ | $U^S(T_i)$ |
|---|---|---|---|---|---|
| $T_1$ | 20 | $11 + 20\varepsilon$ | 3 | $0.55 + \varepsilon$ | $1.65 + 3\varepsilon$ |
| $T_2$ | 20 | $11 + 20\varepsilon$ | $2 + \varepsilon$ | $0.55 + \varepsilon$ | $1.1 + 2.55\varepsilon + \varepsilon^2$ |
| $T_3$ | 20 | $20\varepsilon$ | 3 | $\varepsilon$ | $3\varepsilon$ |
| $T_4$ | 20 | 9 | 1 | 0.45 | 0.45 |
| | | | | | $3.2 + 8.55\varepsilon + \varepsilon^2$ |

Table 4.1: Example task set $\hat{\Gamma}$

Table 4.1 shows a periodic task set $\hat{\Gamma}$ to be scheduled by *EDF-FkF* onto an RHD $H$ with $A(H) = 8$. Let $\varepsilon << 1$ be a small, but positive number. An example schedule is shown in Figure 4.3. Note that job $T_{4,1}$ does not start execution before $T_{3,1}$ has started and,

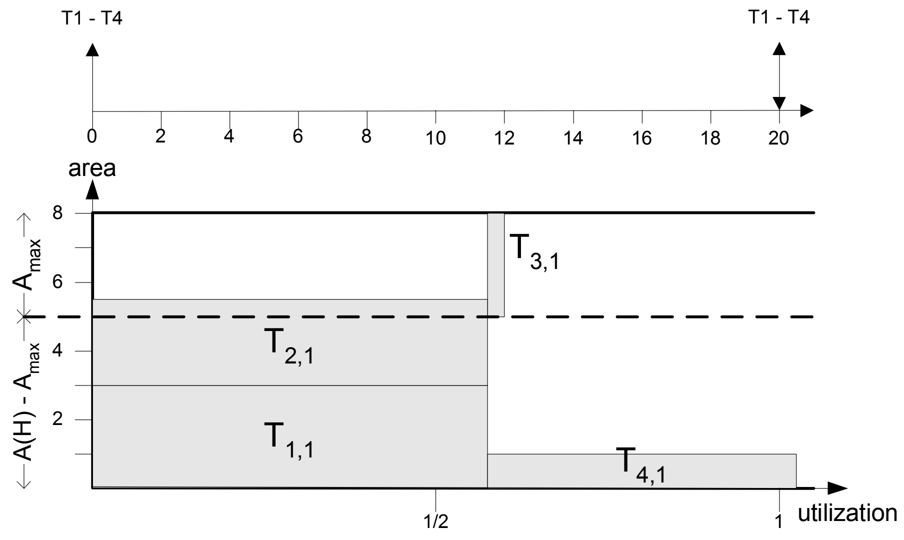


Figure 4.3: *EDF-FkF* schedule of $\hat{\Gamma}$

therefore, will miss its deadline by $\varepsilon$ time units. Evaluating the condition of Theorem 3 for all $k$ shows that task $T_4$ is critical, i.e., for $k = 4$ any infinitesimal positive $\varepsilon$ breaks the scheduling condition.

$$\begin{aligned}
&\forall k : U^S(\Gamma) \leq (A(H) - A_{max}) \cdot (1 - U^T(T_k)) + U^S(T_k)\\
&\quad k = 1 : \quad 3.2 + 8.55\varepsilon + \varepsilon^2 \leq 3.85 - 2\varepsilon\\
&\quad k = 2 : \quad 3.2 + 8.55\varepsilon + \varepsilon^2 \leq 3.35 - 2.45\varepsilon + \varepsilon^2\\
&\quad k = 3 : \quad 3.2 + 8.55\varepsilon + \varepsilon^2 \leq 5 - 2\varepsilon\\
&\quad k = 4 : \quad 3.2 + 8.55\varepsilon + \varepsilon^2 > 3.2
\end{aligned}$$

### 4.1.4 Conclusion on global EDF

We have proposed two adaptations of the global EDF scheduling rule for our application model, namely *EDF-FkF* and *EDF-NF*. The schedulers are called *global*, since a certain job is allowed to execute at any location on the RHD and moreover, after preemption it may resume its execution at a different location. We have been able to develop a scheduling condition, which guaranties that all tasks of a periodic task set will hold their deadlines. The condition can be evaluated in linear time and is valid for both algorithms. How well the algorithm performs will be analyzed in Chapter 5.

The next section will introduce our second scheduling method, which is an adaptation of *partitioned* multiprocessor scheduling approaches to our considered RHD execution model.

## 4.2 Partitioned Scheduling

In the context of multiprocessors, a *non-partitioned* or global schedule is one, where different instances of a periodic task can be executed on different processors and even preempted task instances may migrate to another processor before resuming execution. In contrast, in a *partitioned* schedule, all instances of a task are executed on the same processor. In this section we apply the concept of partitioned scheduling to our RHD execution model. We define a partitioned schedule and the according problem of finding the least resource consuming partitioning, which still guarantees a feasible task schedule. To solve this problem, we apply integer linear programming as well as a heuristic approach.

### 4.2.1 Partitioned Scheduling on RHDs

The concept of partitioned scheduling can be applied to our RHD execution model. We present the following definition:

**Definition 9** (partitioned schedule)**.** *A schedule $R$ of a set of jobs $I$ is said to be* partitioned by $\chi$, *if the following statements hold:*

1. $\chi = \{G_1, G_2, \dots, G_m\}$ *is a* partitioning *of $I$. That is, $\chi$ is a set of disjoint subsets of $I$, called* partition-blocks*, such that the union of all partition-blocks in $\chi$ results in $I$.*
2. *At any point in time, at most one job of each partition-block is in execution on the reconfigurable device. This can be formally expressed by:*

$$\forall t \geq 0 : R(t) \in G_1 \times G_2 \times \cdots \times G_m \tag{4.13}$$

Each partition block is exclusively assigned a certain area of the reconfigurable device. The area requirement of a partition block $A(G_j)$ is determined by its *largest* task; the required overall device area $A(\chi)$ is given by the accumulated area over all partitions:

$$A(G_i) = \max_{J_k \in G_i}(A_k), \quad A(\chi) = \sum_{G_i \in \chi} A(G_i) \tag{4.14}$$

Most of the time we are interested in pure periodic task sets $\Gamma$. In order to apply results from uniprocessor scheduling of periodic task sets to our model, it is most useful to consider the case that all instances of a periodic task belong to the same partition-block. Hence we use the following definition, when considering periodic task sets:

**Definition 10** (periodic partitioned schedule)**.** *If the jobs of $I$ are instances of periodic tasks, a schedule $R$* partitioned by $\chi$ *is said to be* periodic partitioned, *if for each periodic task $T_i$ all of its instances $T_{i,j}$ belong to the same partition-block.*

We use a simplified notation for the partitioning $\chi$ of a period partitioned schedule $R$: If for instance all task instances of $T_1$ and $T_2$ belong to the partition-block $G_i$, we simply write $G_i = \{T_1, T_2\}$ rather than writing $G_i = \{T_{1,1}, T_{1,2}, \ldots, T_{2,1}, T_{2,2}, \ldots\}$. Also we use the notation $T_1 \in G_i$ rather than $T_{1,j} \in G_i$. Hence, $G_i$ denotes a set of periodic tasks, rather than a set of instances of periodic tasks.

Since each partition-block presents an independent task system of its own, we can easily formulate a test whether a given partitioned schedule is feasible by applying the basic results from single-processor EDF scheduling theory:

**Lemma 4** (EDF feasible periodic partitioned schedule)**.** *Let $R$ be a schedule of a periodic task set $\Gamma$ with periodic partitioning $\chi$, where the tasks of each partition $G_i \in \chi$ are scheduled separately by EDF. The schedule is feasible on $H$ if:*

- $A(\chi) \leq A(H)$, *i.e., all partitions fit onto the device and,*
- $\forall G_i \in \chi : U^T(G_i) \leq 1$, *i.e., all partitions have a time utilization of less then 100%.*

*Proof.* The tasks of each partition-block can be separately scheduled by EDF, same as in a uniprocessor system. Since the EDF uniprocessor utilization bound holds for all partitions, all deadlines are met. □

| $T_i$ | $P_i$ | $C_i$ | $A_i$ | $U^T(T_i)$ | $U^S(T_i)$ |
|---|---|---|---|---|---|
| $T_1$ | 4 | 2 | 1/2 | 1/2 | 1/4 |
| $T_2$ | 6 | 5 | 1/4 | 5/6 | 5/24 |
| $T_3$ | 12 | 3 | 3/4 | 1/4 | 3/16 |
| $T_4$ | 12 | 2 | 1/4 | 1/6 | 1/24 |
| | | | | 1.75 | 0.69 |

Table 4.2: Example task set $\Gamma^*$

To illustrate an example of an EDF periodic partitioned schedule, consider the example task set $\Gamma^*$ of Table 4.2. Let some algorithm partition the task set to two partition-blocks, e.g. $\chi = \{G_1 = \{T_1, T_3, T_4\}, G_2 = \{T_2\}\}$. The left-hand side of Figure 4.4 shows the partitioning $\chi$ and illustrates the division of the device area. The according partitioned EDF schedule $R$ is shown on the right-hand side of Figure 4.4. Both partition-blocks are scheduled using a separate EDF scheduler. Since both partition-blocks have a time utilization of less than 100% the schedule is feasibly:

$$(U^T(G_1) = 1/2 + 1/4 + 1/6 \leq 1, \quad U^T(G_2) = 5/6 \leq 1) \tag{4.15}$$

Note, that in the schedule, the task $T_4$ cannot execute in parallel to $T_1$ even if there would be enough free area, since by Definition 9 no more than one task of each partition can execute simultaneously.

Based on Lemma 4 we are able to define an optimization problem in order to find optimal partitioned schedules. To this end, we formulate the problem of finding the least resource-consuming *EDF Feasibly Partitioned Schedule (EFPS)*:

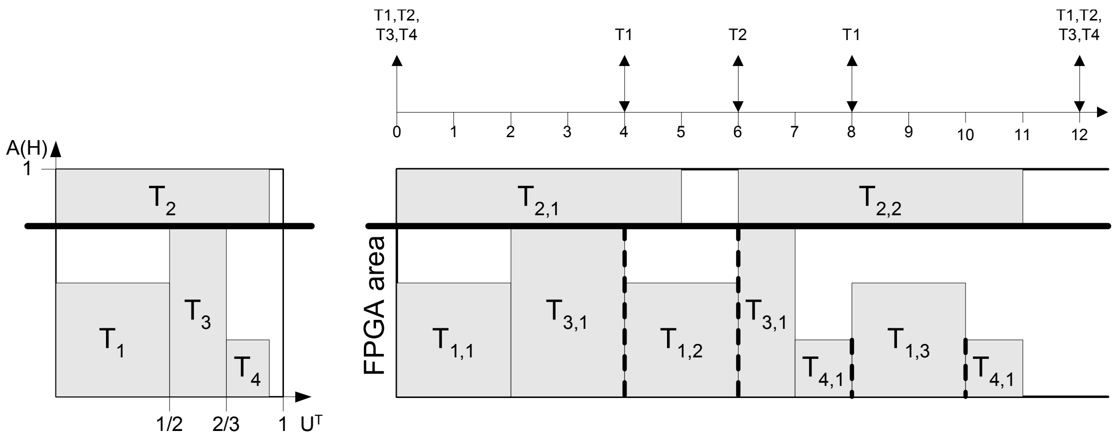


Figure 4.4: Example partitioning of $\Gamma^*$ and the according schedule

**Definition 11** (EFPS problem)**.** *Given a periodic task set* $\Gamma$*, find:*

- *a partitioning* $\chi$ *of* $\Gamma$*,*
- *subject to* $\forall G_i \in \chi : U^T(G_i) \leq 1$*,*
- *minimize* $A(\chi)$

Solving the EFPS problem answers directly the question of the smallest device on which $\Gamma$ can be executed. $\Gamma$ can be feasibly scheduled on any device where $A(H) \geq A(\chi)$ holds. From the resulting partitioning $\chi$, we can easily derive the scheduling function $R$ by applying a separate EDF scheduler to the tasks of each partition-block.

### 4.2.2 Relation to 2-Dimensional Packing Problems

The *EFPS* problem (Definition 11) is equivalent to the *Two-Dimensional Level Strip-Packing Problem (2LSP)*[73], which is a variant of the *Two-Dimensional Strip Packing (2SP)* problem. In 2SP, a set of rectangular items is packed into a strip of given width but infinite height such that the height of the required strip is minimized. The packing must be non-overlapping, orthogonal, and without rotation of the rectangles. In the 2LSP variation of 2SP, the rectangular items have to be packed in rows forming levels. The bottom of the first level is the bottom of the empty strip. The bottom of each other level is determined by a horizontal cut line on top of the highest item in the previous level. Within a level, items are not allowed to be packed on top of each other.

In Definition 11, the tasks $T_i$ can be modeled as rectangular items where the time utilization factor $U^T(T_i)$ corresponds to the rectangle width and the area $A(T_i)$ corresponds to the rectangle height. The width of the strip is set to one, according to the maximal allowed time utilization of an EDF schedule. Each slot of the reconfigurable device – and the set of tasks $G_i$ assigned to it – corresponds to one level of the packing. Finally, the required device area corresponds to the height of the strip. See an example of tasks drawn as rectangles and packed in levels in the utilization diagram on the left side of Figure 4.4.

In the following we will study two approaches to solve the EFPS problem: The first one is to solve EFPS optimally by *Integer Linear Programming (ILP)*, which seems to be reasonable since efficient ILP models which have been successfully applied to 2LSP problems of interesting size have been proposed recently [73]. Since solving an ILP cannot be done in polynomial time, our second approach is a simple heuristic which has been proposed to solve two-dimensional packing problems. We analyze the algorithm in the context of our general scheduling problem described in Chapter 3, and derive a utilization-based scheduling condition which allows a qualitative comparison to our other approaches.

### 4.2.3 Optimal Partitioning by ILP

We can solve our *EFPS* problem optimally by coding the problem as an *integer linear program* and apply a standard solver tool. In an ILP model, we formulate the problem by defining a set of integer decision variables, a set of linear constraint equations and a linear cost function. Formally, an ILP can be formulated in the following form:

Given a rational matrix $A$, a rational constraint vector $b$, and a rational cost vector $c$, determine

$$\{\max c \cdot x | A \cdot x \leq b, x \quad \text{is integer}\}. \tag{4.16}$$

Within this work, ILP-models are never written in the form of Equation 4.16, but use a more readable form were the cost function and the multiple constraint equations are separated according to their meaning.

On modeling the *EFPS* problem, a straight forward way would be to introduce binary decision variables $x_{l,i}$ that indicate, weather task $T_i$ is packed into partition-block $G_l$. The constraint that time-utilization of a partition-block is bounded by 1 (see Definition 11) could be expressed by $\sum_1^n U^T(T_i) \cdot x_{l,i} \leq 1$ for all $l \in \{1, \ldots, n\}$. However, using such simple coding of the decision variables leads to two problems:

- There is no straight forward way to model the cost function.
- The model contains much redundancy. E.g. it is irrelevant for the *EFPS* solution, if some set of tasks is packed into partition-block 1 or into partition-block 2, but this results in different configurations of the ILP decision variables. Such redundancy in coding can cause a large increase in time required by the ILP solver tool.

Therefore, we adopt the slightly more complicated but more efficient ILP model for 2LSP proposed by Lodi at el. [73]. According to the model of Lodi, we make the following basic assumptions without loss of generality:

**Order of tasks:** The tasks $T_i \in \Gamma, i \in \{1, \dots, n\}$ are sorted and numbered by non-increasing area, such that:

$$A(T_i) \leq A(T_{i+1}), i = 1, \dots, n-1. \tag{4.17}$$

**Order of partition blocks:** The task $T_i$ having the largest area within a partition-block is said to *initialize* the partition-block. The index $i$ is also used to index this partition-block. I.e., if the largest task in a partition-block is $T_l$, the partition-block is denoted by $G_l$.

It follows, that a partitioning $\chi$ has $n = |\Gamma|$ potential partition-blocks $G_1 \dots G_n$ (some maybe empty), and that the partition-blocks are sorted by non-increasing areas, i.e. $A(G_l) \geq A(G_{l+1})$. Further, the area of a partition-block is equal to the area of the task that initializes the partition-block, i.e., $A(G_i) = A(T_i)$.

For any task $T_i \in \Gamma$ we can therefore differentiate the following cases:

- $T_i$ initializes a partition-block, i.e. $T_i \in G_i$, or
- $T_i$ is packed within a partition-block with greater area, forcing the partition-block index to be smaller than $i$. I.e. $T_i \in G_1 \cup \dots \cup G_{i-1}$.

The assumptions made are without loss of generality but reduce the search space considerably, since a task $T_i$ cannot be assigned to an arbitrary partition block $G_l$.

When modeling the ILP, the binary decision variables $x_{l,i}$ are used to indicate whether task $T_i$ is packed into partition block $G_l$ or not:

$$x_{l,i} = \begin{cases} 1 & \text{if } T_i \in G_l \\ 0 & \text{if } T_i \notin G_l \end{cases} \qquad l = \{1, \dots, n\}, i = \{1, \dots, n : i \geq l\} \tag{4.18}$$

Index $l$ ranges from 1 to $n$, since there can be at most $n$ partition-blocks. Index $i$ ranges from 1 to $n$, but due to the observations made above, $i$ is always greater or equal to $l$. Therefore, the ILP model uses $n \cdot (n+1)/2$ binary variables.

Note, that the variables $x_{l,i}$ with $l = i$ indicate, whether $G_l$ is initialized or not. If $x_{l,l} = 1$ the partition-block $G_l$ is said to be *initialized* and its largest task is $T_l$. If $x_{l,l} = 0$, $G_l$ is uninitialized and empty:

$$A(G_l) = \begin{cases} A(T_l) & \text{if } x_{l,l} = 1 \\ 0 & \text{if } x_{l,l} = 0 \end{cases} \qquad l = \{1, \dots, n\} \tag{4.19}$$

The final binary ILP model is given in Equation 4.20 and has three main components:

1. The cost function accumulates the area required by all non-empty partition-blocks, hence minimizing the overall area $A(\chi)$.

2. The second equation ensures that each task is packed into one and only one partition-block.

3. The third equation enforces EDF schedulability. Each non-empty partition-block must have a time-utilization less than or equal to 1, i.e.,$U^T(G_l) \leq 1$. Note, that this equation also ensures, that no task can be packed into an uninitialized partition-block, i.e., into a partition-block with $x_{l,l} = 0$.

The final ILP can be written as:

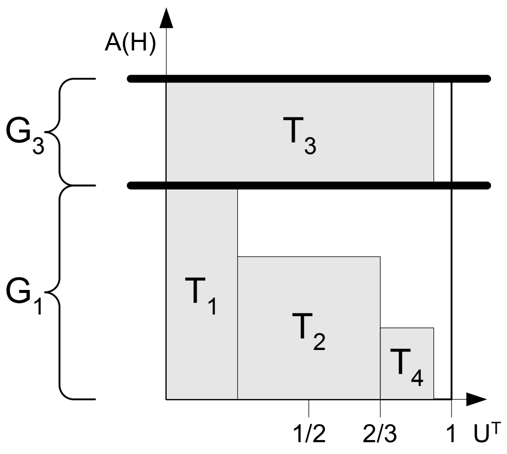
$$\boxed{\begin{aligned} &\min \sum_{l=1}^{n} A(T_l) \cdot x_{l,l} \\ &\quad \sum_{l=1}^{i} x_{l,i} = 1, \qquad i \in \{1, \dots, n\} \\ &\quad \sum_{i=l}^{m} U^T(T_i) \cdot x_{l,i} \leq 1 \cdot x_{l,l}, \qquad l \in \{1, \dots, n\} \end{aligned}} \tag{4.20}$$

Let $n$ be the number of tasks in $\Gamma$, then the complexity of the corresponding ILP model accounts to $n \cdot (n+1)/2$ binary variables and $2 \cdot n$ constraints. Such model complexity is practical. We were able to optimally solve most problem instances with a size of $n$ up to 30 in less than 15 seconds, to provide a rough appraisal. To solve the ILP model, we employed the *lp_solve* library [15] version 4.0 on a 2.8 GHz Pentium 4 machine.

**Example 2.** *To illustrate the ILP model, Figure 4.5 visualizes an example partitioning of a task set $\Gamma = \{T_1, T_2, T_3, T4\}$, that has been partitioned into two partition-blocks, $G_1 = \{T_1, T_2, T_4\}$ and $G_2 = \{T_3\}$. The according variable configuration is shown in the table. $T_1$ initializes partition-block $G_1$ and therefore $x_{1,1} = 1$. The other tasks $T_2, T_4$ in $G_1$ imply, that $x_{1,2} = 1$ and $x_{1,4} = 1$. $T_3$ initializes partition-block $G_3$, thus $x_{3,3} = 1$. No tasks have been assigned to partition-blocks $G_2$ and $G_4$, therefore they do not exist in $\chi$. Note, that the variables marked with "-" are not needed, since by convention each partition-block $G_l$ can only contain tasks $T_i$ with index $i \geq l$.*

The presented ILP (Equation 4.20), which has been adopted from [73], allows us to solve the *EFPS* problem and hence find the optimal partitioned schedule for a periodic task set. We will analyze and compare the performance of this approach to other schedulers in Chapter 5. Moreover, this ILP model will be the basis for an extended ILP, which allows us to solve the enhanced scheduling problem, where numerous implementation variants for each task are considered (see Section 7.1.2).

However, since the ILP model is not practicable for task sets with $n >> 30$, we analyze a fast polynomial time heuristic in the next section.



| $l$ | $x_{l,1}$ | $x_{l,2}$ | $x_{l,3}$ | $x_{l,4}$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 |
| 2 | - | 0 | 0 | 0 |
| 3 | - | - | 1 | 0 |
| 4 | - | - | - | 0 |

Figure 4.5: Example of partitioning $\chi$ and the according decision variable configuration

### 4.2.4 Next-Fit-Decreasing-Area Partitioning

This section presents and analyzes a simple heuristic in the context of our *EFPS* problem. The goal is twofold: On one hand, we want to be able to apply the partitioned scheduling approach to task sets of large size, which is impractical using the ILP model. On the other hand, as we will see, the analysis of the heuristic will help us in the quantitative comparison of the proposed scheduling methods.

To solve the two-dimensional packing problems *2SP* and *2LSP*, Coffman et al. [30] studied the *Next-Fit-Deceasing-Height* algorithm. It first sorts all rectangles according to non-increasing height. The first rectangle is packed left-justified on the bottom of the container *(first level)*. The next rectangle is packed on the right of the former one, if the width of the container is not exceeded. Otherwise, the *level* is closed by a horizontal line on top of the tallest rectangle. A new level is created on top of the last one, and the rest of the items are packed in the same manner.

In [30] was shown, that for any instance of a 2SP problem, i.e. any set of rectangles $X$, the packing height obtained by the *Next-Fit-Deceasing-Height* algorithm (denoted by $NFDH(X)$) is bounded. The bound is given by

$$NFDH(X) \leq 2 \cdot OPT(X) + 1, \tag{4.21}$$

where the height of the rectangles is normalized to 1 and $OPT(X)$ denotes the height of an optimal packing of $X$.

Since a packing computed by the *Next-Fit-Deceasing-Height* algorithm is always done in levels, we can use it to compute a *periodic partitioned schedule* that is feasible by EDF. Since the height of the rectangles corresponds to the area of the periodic tasks, the algorithm is called *Next-Fit-Deceasing-Area (NFDA)* and shown in Algorithm 3.

The *NFDA* algorithm has a time complexity of $O(n \log n)$, since sorting the tasks is the most time consuming part. Hence it can be used to compute the partitioning of very

**Algorithm 3** Next-Fit-Decreasing-Area

```
procedure NFDA(Γ)
    sort_and_index_after_non-increasing_area(Γ)
    $k \leftarrow 1$
    $G_k \leftarrow \emptyset$
    for $i \leftarrow 1, |\Gamma|$ do
        if $U^T(G_k) + U^T(T_i) \leq 1$ then          ▷ EDF utilization condition
            $G_k \leftarrow G_k \cup T_i$             ▷ add to current level
        else
            $k \leftarrow k + 1$                      ▷ open new level
            $G_k \leftarrow \emptyset$
        end if
    end for
    return $\chi \leftarrow \{G_1, \ldots, G_k\}$
end procedure
```

large task sets. In Section 5.2 we will analyze the scheduling performance of *NFDA* on a set of simulation benchmarks.

Since *NFDA* has such low time complexity, a utilization-based scheduling condition is not really necessary for use in praxis. If we want to test whether *NFDA* can feasibly schedule a certain task set $\Gamma$ onto an device $H$, we simply compute the partitioning and check whether it exceeds the device area or not.

However, a utilization based test condition is useful for the further analysis. It gives us a more general comparison to other algorithms. In particular, our concern is the comparison of the performance guaranteed by *NFDA-partitioned-EDF* to the performance guaranteed for the global EDF schedulers by Theorem 3. It turned out in our simulation experiments, that all task sets accepted by the global EDF scheduling test of Theorem 3 where also feasibly scheduled by *NFDA-partitioned-EDF*. Proving this dominance, is the main motivation for the development of a utilization based test condition for *NFDA*. The previously reported analysis and the achieved bound of Equation 4.21 is not useful to provide a scheduling test, since the packing achieved by $OPT(X)$ is an arbitrary packing and not a packing in levels. Hence, $OPT(X)$ does not provide a valid solution for our partitioning problem. Our utilization based scheduling condition for *NFDA* is presented in the next theorem:

**Theorem 4** (NFDA-partitioned-EDF scheduling condition)**.** *For a periodic task set* $\Gamma$ *and RHD* $H$*, an* EDF feasible periodic partitioning *is found by the* NFDA *heuristic if:*

$$U^S(\Gamma) \leq (A(H) - A_{max}) \cdot (1 - U^T_{max}) + U^S_{max} \tag{4.22}$$

*where* $A_{max}$*,* $U^T_{max}$*,* $U^S_{max}$ *are the largest area, time-utilization and system-utilization of any tasks in* $\Gamma$*.*

Based on Theorem 4 we will be able to to prove the following theorem, which says that the NFDA-partitioned-EDF scheduler always outperforms the global EDF scheduling test developed in Section 4.1.

**Theorem 5** (NFDA-partitioned-EDF vs. EDF-FkF). *For every periodic task set* $\Gamma$ *accepted by the EDF-FkF scheduling test of Theorem 3 an* EDF feasibly partitioned schedule *is found by the* NFDA *heuristic.*

*Proof of Theorem 4.* Consider an arbitrary partitioning $\chi$ of a task set $\Gamma$, which has been computed by *NFDA*. We derive a lower bound on the total system utilization $U^S(\chi)$ of the tasks within partitioning $\chi$, as a function of the required device area $A(\chi)$ and some other task set parameters:

$$U^S(\chi) \geq f(A(\chi), A_{max}, U^T_{max}, U^S_{max}) \tag{4.23}$$

Replacing $A(\chi)$ by an actual device area $A(H)$, the function $f(\cdots)$ presents an upper bound on the task set's system utilization that is guaranteed to be feasibly packed:

$$U^S(\Gamma) \leq f(A(H), A_{max}, U^T_{max}, U^S_{max}) \tag{4.24}$$

To illustrate the system utilization packed in the various partition-blocks (also called levels), Figure 4.6 shows an example partitioning by *NFDA*. Since tasks are sorted with respect to non-increasing areas before being partitioned, it follows that $A(G_{l+1}) \leq A(G_l)$ and that the smallest task in a level $l$ has at least area $A(G_{l+1})$. Moreover, let $x_l$ denote

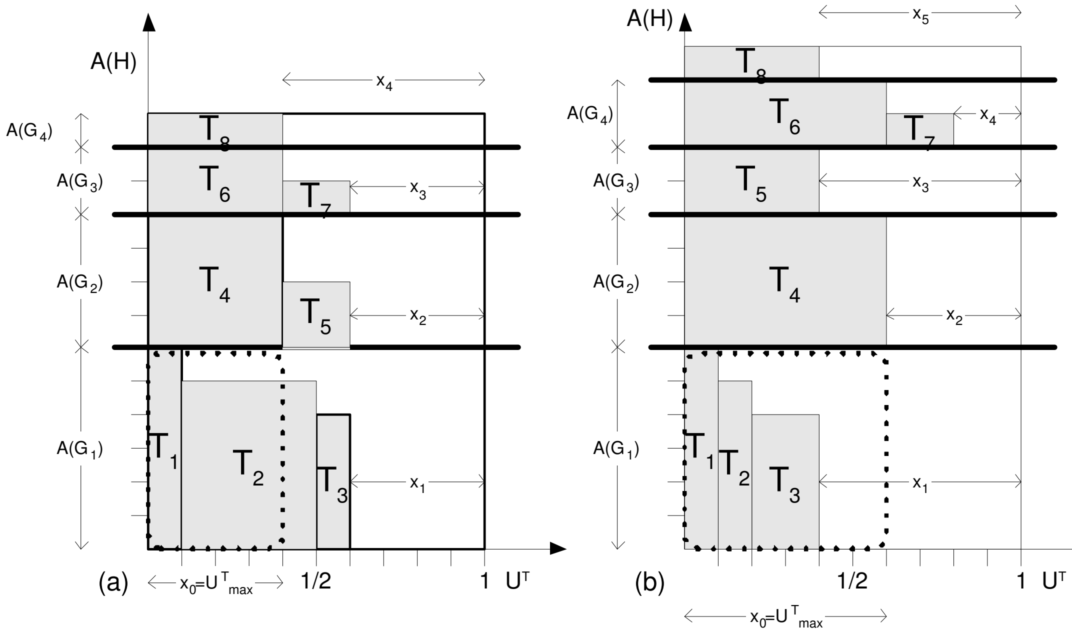


Figure 4.6: Example partitioning of NFDA for (a) a task set with $U^T_{max} \leq 0.5$ and (b) a task set with $U^T_{max} > 0.5$

the fraction of unused time-utilization in level a $l$. From all tasks in a level $l$, the first one has area $A(G_l)$ and the other tasks, if any, have at least an area of $A(G_{l+1})$. The first task in a level $l$ has at least a time utilization equal or greater then $x_{l-1}$, since otherwise the task would have been packed into level $l-1$. The total time utilization in level $l$ is $1-x_l$. Thus, the amount of system utilization packed into level $G_l$ given by ($area \times time\text{-}util.$) of the first task + ($area \times time\text{-}util.$) of other tasks accounts to: [1]

$$\begin{aligned} \forall l \geq 2: \quad U^S(G_l) &\geq A(G_l) \cdot x_{l-1} + A(G_{l+1}) \cdot \max\{(1-x_l-x_{l-1}), 0\} \\ &\geq A(G_l) \cdot x_{l-1} + A(G_{l+1}) \cdot (1-x_l-x_{l-1}) \end{aligned} \tag{4.25}$$

The system utilization for level $l=1$ is slightly different, since we cannot guaranty any time-utilization for the first task, i.e. $x_0$ is undefined since there is no level below the first level. Let $x_0$ be a value in $]0,1]$ and $U^S(0,x_0)$ be the cumulative system-utilization of tasks packed in level 1 from the left boarder up to $x_0$, illustrated by the dotted box in Figure 4.6. The cumulative system-utilization of level 1 is given by the $U^S(0,x_0)$ + ($area \times time\text{-}util.$) of other tasks, if any. It accounts to:

$$\begin{aligned} U^S(G_1) &\geq U^S(0,x_0) + A(G_2) \cdot \max\{(1-x_1-x_0), 0\} \\ &= U^S(0,x_0) + A(G_2) \cdot (1-x_1-x_0) + \underbrace{A(G_2) \cdot \max\{(x_1+x_0-1), 0\}}_{\varrho} \end{aligned} \tag{4.26}$$

Note, that $\varrho$ is a correction term and is equal to zero, as long as $x_0 + x_1 \leq 1$.

Summing up the system utilization of the first level (Equation 4.26) and all other levels, Equation 4.25 results in the lower bound on the overall system utilization of the entire packing:

$$\sum_{G_l \in \chi} U^S(G_l) \geq U^S(0,x_0) + \varrho + \sum_{l=2}^{m} (1-x_{l-2}) \cdot A(G_l) \tag{4.27}$$

The maximal time utilization of a task in $\Gamma$, denoted by $U^T_{max}$, presents an upper bound to the $x_l$ variables in Equation 4.27. Therefore we can replace them in Equation 4.27 and get a new lower bound on the system utilization in Equation 4.28, which directly leads to Equation 4.29.[2] Note, that $\varrho = 0$ for $U^T_{max} \leq 0.5$, but becomes positive for $U^T_{max} > 0.5$.

$$U^S(\chi) \geq U^S(0, U^T_{max}) + \varrho + \sum_{l=2}^{m} (1-U^T_{max}) \cdot A(G_l) \tag{4.28}$$

$\Leftrightarrow$

$$U^S(\chi) \geq U^S(0, U^T_{max}) + \varrho + (1-U^T_{max})[\sum_{l=1}^{m} A(G_l) - A(G_1)] \tag{4.29}$$

[1]We define $A(G_l) = 0 : l > m$, i.e. the area of non existing levels above the top level is zero. Hence, the system utilization of the top level is according to Equation 4.25 given by $U^S(G_m) \geq A(G_m) \cdot x_{m-1}$.

[2]In our assumptions $x_0$ can take any value from $]0,1]$. Hence we are allowed to choose $x_0 = U^T_{max}$.

Substituting $\sum_{l=1}^{m} A(G_l) \rightarrow A(\chi)$ and $A(G_1) \rightarrow A_{max}$ leads to:

$$U^S(\chi) \geq U^S(0, U^T_{max}) + \varrho + (1 - U^T_{max})(A(\chi) - A_{max}) \tag{4.30}$$

Now let $T_k$ be the task with greatest system utilization, i.e. $U^S(T_k) = U^S_{max}$. We distinguish between two cases, where $T_k$ can be packed by NFDA:

1. $T_k$ is packed in the first level within the interval $[0, U^T_{max}]$. Obviously, it follows $U^S(0, U^T_{max}) \geq U^S(T_k)$.
2. $T_k$ is placed only partially within the interval $[0, U^T_{max}]$ of the first level, or placed in a different level. Then, all tasks packed within the interval $[0, U^T_{max}]$ have an area greater than or equal to $A_k$, since tasks are packed by decreasing areas.
   a) Level 1 is filled from left to right at least up to $U^T_{max}$. Hence, the system utilization is at least as much as that of $T_k$, i.e. $U^S(0, U^T_{max}) \geq A_k \cdot U^T_{max} \geq U^S(T_k)$. This case is illustrated in Figure 4.6-(a).
   b) Level 1 is not filled up to $U^T_{max}$, which can only be the case, if $U^T_{max} > 0.5$. In this case, level 1 is at least filled up to $(1 - U^T_{max})$ and hence $U^S(0, U^T_{max}) \geq A_k \cdot (1 - U^T_{max})$. But still $\varrho = A(G_2) \cdot \max\{(2U^T_{max} - 1), 0\}$ has to be added, and is greater than zero in this case. Since $T_k$ is placed in level 2 or above, $A(G_2) \geq A_k$. Adding it up, results in:

   $$U^S(0, U^T_{max}) + \varrho \geq A_k \cdot (1 - U^T_{max}) + A_k \cdot (2U^T_{max} - 1) = A_k \cdot U^T_{max} \geq U^S(T_k) \tag{4.31}$$

   This case is illustrated in Figure 4.6-(b).

Hence, in all cases we can substitute $U^S(0, U^T_{max}) + \varrho$ by $U^S_{max}$ in Equation 4.30, which leads to the lower bound on $U^S(\chi)$ and presents the upper bound on $U^S(\Gamma)$ of Theorem 4.

$$U^S(\chi) \geq U^S_{max} + (1 - U^T_{max})(A(\chi) - A_{max}) \tag{4.32}$$

$$\Leftrightarrow U^S(\Gamma) \leq U^S_{max} + (1 - U^T_{max})(A(H) - A_{max}) \tag{4.33}$$

□

Now that we have proven the scheduling condition for *NFDA-partitioned-EDF* we are ready to prove that this condition dominates our scheduling condition for the global EDF schedulers (Theorem 5).

*Proof of Theorem 5.* The upper bound on the system utilization of the *EDF-FkF* scheduling condition in Theorem 3 is given by:

$$\forall T_k \in \Gamma : U^S(\Gamma) \leq (A(H) - A_{max}) \cdot \big(1 - U^T(T_k)\big) + U^S(T_k) \tag{4.34}$$

On the other hand, the bound given for the *NFDA-partitioned-EDF* algorithm in Theorem 4 is given by:

$$U^S(\Gamma) \leq (A(H) - A_{max}) \cdot (1 - U^T_{max}) + U^S_{max} \tag{4.35}$$

Let w.l.g. $T_1$ be the task with the maximal time utilization, i.e. $U(T_1) = U^T_{max}$. Then the *EDF-FkF* scheduling condition is bounded from above by:

$$U^S(\Gamma) \leq (A(H) - A_{max}) \cdot \left(1 - U^T_{max}\right) + U^S(T_1) \tag{4.36}$$

Since $U^S(T_1) \leq U^S_{max}$, Equation 4.36 is bounded from above by the *NFDA-partitioned-EDF scheduling condition* of Equation 4.35. Therefore, the bound given by the *NFDA-partitioned-EDF scheduling condition* is always greater than that given by the *EDF-FkF scheduling condition.* □

### 4.2.5 Conclusion on Partitioned EDF

In this section we adapted the partitioned approach for multi-processor systems to our RHD execution model. A task set is partitioned off-line. At runtime, the tasks of each partition-block are executed by a separate EDF scheduler on a separated part of the device area. Hence, the challenge is to find a feasible partitioning of the task set. We proposed two methods:

1. The optimal partitioning, i.e. the partitioning that requires the least amount of resources but still guarantees all deadlines, is computed via integer linear programming. This method is feasible for task sets containing up to about 30 tasks.

2. In order to partition sets of $>>$ 30 tasks, we studied the simple *Next-Fit-Decreasing-Area* heuristic. We derived a utilization based scheduling test, which is important for the analytic comparison of the global EDF vs. the partitioned scheduling approaches. In particular we showed that all task sets accepted by our global EDF scheduling condition will also be scheduled by *NFDA-partitioned-EDF*.

It would be interesting, to apply further heuristics, such as the *First-Fit-Decreasing-Height (FFDH)* which has been shown to perform always better than *NFDH* [30]. However, we omit a deeper analysis of *FFDH*, since for our analytic comparison to the global EDF schedulers our results from the *NFDA* analysis have been sufficient. In particular we showed with Theorem 5 that *NFDA-partitioned-EDF* schedules all the task sets accepted by our global EDF scheduling test (and even more). A detailed analysis of the performance achieved by the partitioned schedulers follows in Chapter 5. The next section will introduce our third and last scheduling method.

## 4.3 Server-Based Scheduling

Periodic servers are a well-known concept in real-time scheduling [25](p. 111 ff.). They are usually used to serve aperiodic requests of application tasks in a system mixed of periodic and aperiodic tasks. Here we only keep the basic idea, that a periodic server executes application tasks, and hence use the same term *server*. However, we do not challenge the execution of mixed periodic / aperiodic task sets, but use periodic servers to create a feasible schedule for the *periodic* hardware tasks of our execution model. Within a server, several application tasks execute in parallel. The aim is to find servers that can be sequentially executed on an RHD and hence no partial reconfiguration is required. This makes the method applicable for a wide set of reconfigurable devices.

We present a scheduling technique called *Merge Server Distribute Load (MSDL)*. To construct a schedule, the *MSDL* algorithm uses the concept of *server tasks*, or briefly servers. A server is a periodic task that reserves execution time and RHD area for other tasks. We define a server as $S_i = (V_i, P_i, C_i, A_i)$, where $V_i = \{T_a, T_b, \dots\} \subseteq \Gamma$ is a set of tasks for which execution time and area is reserved. $P_i$, $C_i$, $A_i$ denote the period, the computation time and the area of the server, respectively. The area of a server is set equal to the sum of the areas of tasks represented by the server:

$$A_i = \sum_{T_k \in V_i} A_k \tag{4.37}$$

Consequently, whenever the server $S_i$ is running, all tasks it represents are running.

The rationale of the MSDL algorithm is to construct a set of servers $\Omega$ from the original task set $\Gamma$, such that any feasible schedule for $\Omega$ implies a feasible schedule for $\Gamma$. More specifically, MSDL constructs a set of servers $\Omega$ by properly *merging* tasks together for parallel execution. The resulting servers are then scheduled for *sequential* execution on the RHD with the single processor EDF algorithm. Feasibility of the resulting set of servers is thus efficiently checked by the utilization test:

$$U^T(\Omega) \leq 1 \tag{4.38}$$

### 4.3.1 The Merge-Server Distribute Load (MSDL) Algorithm

Algorithm 4 shows the pseudo code for the MSDL technique. First, each of the initial tasks is turned into a server (line 3). Then the main loop is entered in which, iteratively, a server pair is identified (line 8) and merged (line 12) if possible. If no valid server pair can be found anymore, the algorithm exits and returns $\Omega$ as the final set of servers (line 10). A feasible schedule is found by *MSDL*, if the final set of servers has a time utilization less or equal to one, i.e. $U^T(\Omega) \leq 1$.

The selection of the two servers $S_x$ and $S_y$ that should be merged is done by calling the function *SelectBestPairToMerge()* (line 8).

A pair of servers $S_x$ and $S_y$ is valid for a merge, if the following two predicates hold:

- The set of represented tasks is disjoint:

$$V_x \cap V_y = \emptyset \tag{4.39}$$

- The tasks of both servers fit together onto the RHD:

$$A_x + A_y \leq A(H) \tag{4.40}$$

If a valid pair of servers $S_x$ and $S_y$ has been selected, they are merged. This is done by the *MergeServers()* function. It takes the current server set $\Omega_{old}$ and the servers $S_x, S_y$ to be merged as inputs and returns the new server set after the merge. Without loss of generality, we assume that server $S_y$ has a shorter or equal period than $S_x$. A new server $S_z$ is created representing all tasks of the two original servers (line 16). The period and the computation time for $S_z$ are set equal to those of $S_y$. Therefore, $S_z$ is a full replacement of $S_y$, and $S_y$ can be removed from $\Omega$ (line 19). The computation time of $S_x$ is reduced, since the new server $S_z$ reserves area and computation time for the tasks of $S_x$ as well. The actual reduction of the computation time $C_x$ depends on how often the new server $S_z$ executes within the period of $S_x$. A pessimistic approximation for the reduction of $C_x$ is given by:

$$takeOverTime(S_x, S_z) = C_z(\lfloor P_x/P_z \rfloor - 1) \tag{4.41}$$

For the implementation of the *SelectBestPairToMerge()* function, several heuristics are conceivable. In our current version we employ a greedy strategy that selects the pair of servers giving the greatest reduction in time utilization $U^T(\Omega_{old}) - U^T(\Omega_{new})$ per increase of system utilization $U^S(\Omega_{new}) - U^S(\Omega_{old})$.[3] That is, *SelectBestPairToMerge()* scans each pair of servers $S_i, S_j$ that is valid for a merge (line 28). The temporary new server set $\Omega_{new}$ is computed, by merging $S_i, S_j$. Afterwards, the profit of this merge is computed (line 30). Finally, the *SelectBestPairToMerge()* function returns the pair $S_i, S_j$, that gains the highest profit.

For example, we apply the MSDL algorithm to an example task set of three tasks. Then, in Section 4.3.2, we provide a more involved analysis to compute the exact computation time reduction.

**Example 3.** *Table 4.3 shows the set of servers $\Omega_k^*$ generated in each iteration $k$ of the MSDL algorithm. Initially, the servers $\Omega_0^* = \{S_1, S_2, S_3\}$ are created. In the first iteration, $S_1$ and $S_2$ are selected and merged into $S_4$. $S_2$ receives the new computation time $C_2 \leftarrow C_2 - 2 = 3$. The server with the shorter period, $S_1$, is removed. In the second iteration, the residual $S_2$ and $S_3$ are merged into $S_5$. Not only the server with the shorter period is removed, but also $S_3$ since its computation time is reduced to zero. $\Omega_2^*$ is the final server set, since neither $V_4, V_5$ are disjunct nor $A_4 + A_5 \leq 1$. As shown in Table 4.3, the time utilization factor $U^T(\Omega_2^*) = 1$. Consequently, $\Omega_2^*$ can be feasibly scheduled by EDF. The resulting schedule is shown in Figure 4.7. The figure also indicates the original tasks of* $\Gamma^*$ *executed* inside *the servers. Compared to a global* EDF-NF *schedule*

[3] $\Omega_{old}$ denotes the set of servers before, whereas $\Omega_{new}$ denotes the server set after merging the selected pair of servers.

**Algorithm 4** Merge Server - Distribute Load

```
1: procedure MSDL($\Gamma, H$)
2:     $\Omega \leftarrow \emptyset$
3:     for all $T_i \in \Gamma$ do                                          ▷ init
4:         $S_i \leftarrow (\{T_i\}, P_i, C_i, A_i)$
5:         $\Omega \leftarrow \Omega \cup S_i$
6:     end for
7:     loop
8:         $S_x, S_y \leftarrow$ SelectBestPairToMerge($\Omega, H$)
9:         if no pair found then
10:            return $\Omega$                                                ▷ exit
11:        end if
12:        $\Omega \leftarrow$ MergeServers($\Omega, S_x, S_y$)
13:    end loop
14: end procedure

15: function MergeServers($\Omega_{old}, S_x, S_y$)
16:    $S_z \leftarrow (V_x \cup V_y, P_y, C_y, A_x + A_y)$                  ▷ $P_y \leq P_x$
17:    $C_x \leftarrow C_x - takeOverTime(S_x, S_z)$
18:    $\Omega_{new} \leftarrow \Omega_{old} \cup S_z$                          ▷ add server
19:    $\Omega_{new} \leftarrow \Omega_{old} \setminus S_y$
20:    if $C_x \leq 0$ then
21:        $\Omega_{new} \leftarrow \Omega_{old} \setminus S_x$
22:    end if
23:    return $\Omega_{new}$
24: end function

25: function SeclectBestPairToMerge($\Omega_{old}, H$)
26:    $bestProfit \leftarrow 0$
27:    for all $S_i \neq S_j \in \Omega_{old}$ do
28:        if $P_i < P_j$ and $V_i \cap V_j = \emptyset$ and $A_i + A_j \leq A(H)$ then   ▷ for all valid pairs
29:            $\Omega_{new} \leftarrow$ MergeServers($\Omega_{old}, S_j, S_i$)          ▷ do temporary merge
30:            $profit \leftarrow \frac{U^T(\Omega_{old}) - U^T(\Omega_{new})}{U^S(\Omega_{new}) + U^S(\Omega_{old})}$   ▷ compute profit
31:            if $profit > bestProfit$ then
32:                $bestProfit \leftarrow profit$
33:                $S_x \leftarrow S_j$
34:                $S_y \leftarrow S_i$
35:            end if
36:        end if
37:    end for
38:    return $S_x, Sy$
39: end function
```

*of the task set, MSDL requires only two RHD programming files. Table 4.3 also lists the system utilization factor $U_i^S$ which increases over the iterations, since larger servers will reveal more idle areas and times inside their reservations. In essence, MSDL trades system utilization for time utilization to allow for an efficient schedulability test and to reduce the number of RHD configurations.*

| $S_i$ | $V_i$ | $P_i$ | $C_i$ | $A_i$ | $U_i^T$ | $U_i^S$ |
|---|---|---|---|---|---|---|
| $S_1$ | $T_1$ | 4 | 2 | 1/2 | 1/2 | 1/4 |
| $S_2$ | $T_2$ | 6 | 5 | 1/4 | 5/6 | 5/24 |
| $S_3$ | $T_3$ | 12 | 3 | 3/4 | 1/4 | 3/16 |
| $\sum$ | | | | | 1.58 | 0.65 |
| ~~$S_1$~~ | $T_1$ | 4 | 0 | 1/2 | 0 | 0 |
| $S_2$ | $T_2$ | 6 | 3 | 1/4 | 1/2 | 1/8 |
| $S_3$ | $T_3$ | 12 | 3 | 3/4 | 1/4 | 3/16 |
| $S_4$ | $T_1, T_2$ | 4 | 2 | 3/4 | 1/2 | 3/8 |
| $\sum$ | | | | | 1.25 | 0.69 |
| ~~$S_2$~~ | $T_2$ | 6 | 0 | 1/4 | 0 | 0 |
| ~~$S_3$~~ | $T_3$ | 12 | 0 | 3/4 | 0 | 0 |
| $S_4$ | $T_1, T_2$ | 4 | 2 | 3/4 | 1/2 | 3/8 |
| $S_5$ | $T_2, T_3$ | 6 | 3 | 1 | 1/2 | 1/2 |
| $\sum$ | | | | | 1 | 0.88 |

Table 4.3: Servers generated for the example task set $\Gamma^*$ by the *MSDL* algorithm

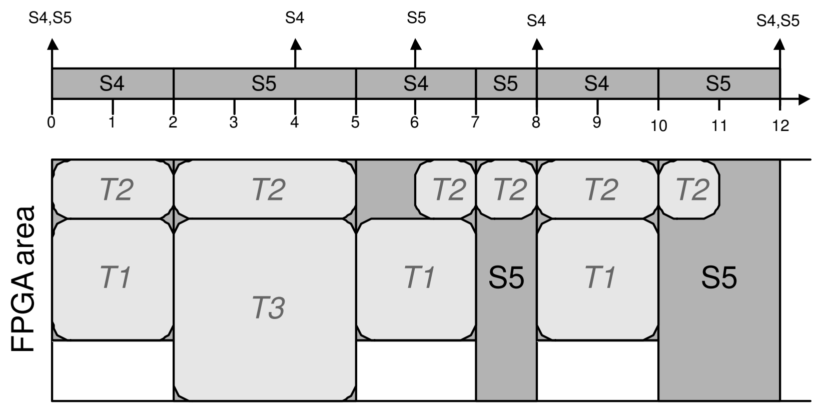


Figure 4.7: Schedule of server task set generated by MSDL

### 4.3.2 Exact Evaluation of Computation Time Reduction

In Equation 4.41, we made the pessimistic assumption that a server $S_z$ with $P_z \leq P_x$ executes only for $m = \lfloor P_x/P_z \rfloor - 1$ times between the release time and deadline of server $S_x$. Therefore, the computation time of $S_x$ was reduced by $mC_z$. A further reduction of $C_x$ is possible, if we take into account that the server instances of $S_z$ which are not fully contained between the release time and deadline of server $S_x$ can still be useful.

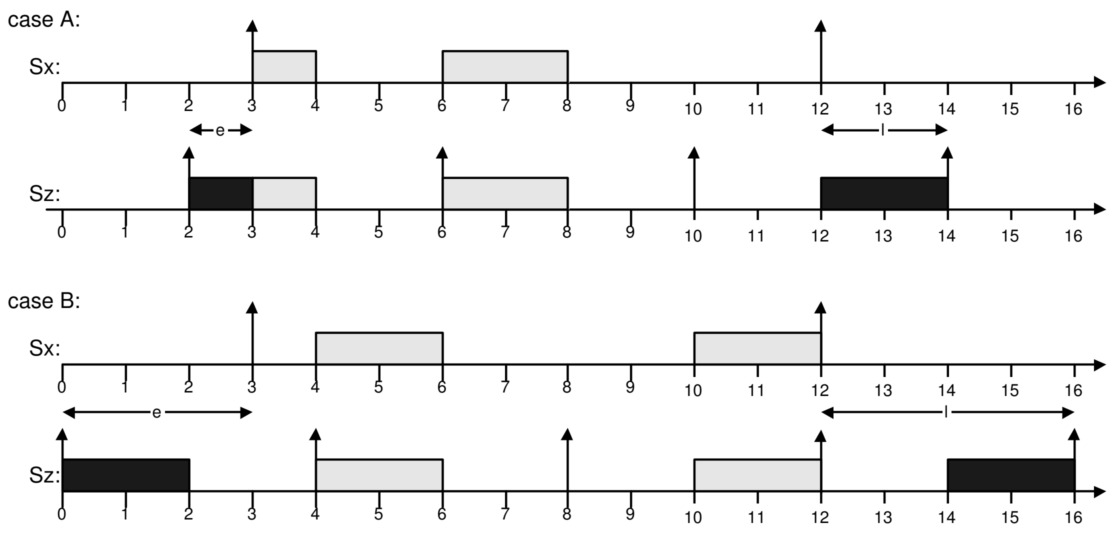


Figure 4.8: Case analysis of computation time reduction

The precise amount of this reduction depends on the actual phase between $S_x$ and $S_z$. Figure 4.8 illustrates the two cases that have to be distinguished:

**Case A** shows an example with $P_x = 9$ and $P_z = 4$. Pessimistically, the server $S_z$ is guaranteed to execute $m = \lfloor \frac{9}{4} \rfloor - 1 = 1$ times between the release time and deadline of $S_x$. As Figure 4.8-A illustrates, there is one instance of server $S_z$ being $e$ time units too *early* to be included in the considered period of $S_x$, and one instance of server $S_z$ being $l$ time units too *late*. In a worst-case schedule, the server $S_z$ executes at the beginning of its *early* instance and at the end of its *late* instance, resulting in some wasted computation time (denoted by black boxes) for $S_x$. However, some amount of the computation time of $S_z$ may still be useful to execute $S_x$, as denoted by the gray box of the first instance of $S_z$. Let $\delta = e + l$ denote the amount of time of the early and late server instances which are *outside* the considered period of $S_x$. The time $\delta$ can be computed by $\delta = (m + 2) * P_z - P_x$. Let $C^{el}$ denote the computation time of the *early* and *late* server instances, which is guaranteed to be within the considered period of $S_x$. Then, $C^{el}$ can be computed by $C^{el} = \max(2 \times C_z - \delta, 0)$. Therefore, in *case A* the time $C^{el}$ is exactly the amount by which $C_x$ can be reduced in addition to Equation 4.41.

**Case B** of Figure 4.8 illustrates the second case, where the server $S_z$ is executed $m+1$ times within the considered period of $S_x$. In this case, $\delta$ changes to $\tilde{\delta} = (m + 3) * P_z - P_x$ and $C^{el}$ changes to $\tilde{C}^{el} = \max(2 \times C_z - \tilde{\delta}, 0)$, respectively. It follows that in *case B* the time by which $C_x$ can be reduced in addition to Equation 4.41 is given by $C_z + \tilde{C}^{el}$.

Since we have to consider the worst case (out of *case A* and *case B*), the precise reduction of the computation time is determined by:

$$\begin{aligned} takeOverTime(S_x, S_z) = \min\{ & \\ & C_z(\lfloor P_x/P_z \rfloor - 1) + \max\big[2C_z - \underbrace{((\lfloor P_x/P_z \rfloor + 1)P_z - P_x)}_{\delta}, 0\big], \\ & C_z \lfloor P_x/P_z \rfloor + \max\big[2C_z - \underbrace{((\lfloor P_x/P_z \rfloor + 2)P_z - P_x)}_{\tilde{\delta}}, 0\big]\} \end{aligned} \tag{4.42}$$

### 4.3.3 Properties of MSDL

We designed *MSDL* especially for the full reconfiguration approach. Hence the number of the required bit-streams for a schedule should be small. Here we show, that it is bounded by the number of tasks.

**Lemma 5** (Number of generated Servers by MSDL)**.** *Let* $\Gamma$ *be a periodic task set and* $\Omega$ *the set of servers generated by the* MSDL *algorithm. Then, the number of servers is bounded by the number of tasks, i.e.*

$$|\Omega| \leq |\Gamma| \tag{4.43}$$

*Proof.* Initially, *MSDL* starts with as many servers as tasks in the task set. In every loop iteration, one server is added to $\Omega$ but one or two servers are removed from $\Omega$ (line 19). Hence, $|\Omega| \leq |\Gamma|$ will hold in each iteration. □

The bounded number of generated servers is one major advantage of *MSDL* when it comes to system realization. Each server can be implemented as one RHD configuration file (bit-stream), and the configurations are scheduled and executed sequentially onto the RHD employing a full reconfiguration model. We will discuss these realization issues in detail in Chapter 6.

**Time complexity of MSDL:** The time complexity of the MSDL algorithm is polynomial. Without giving a formal proof we describe how MSDL can be computed in $O(n^4)$ time, where $n$ is the number of tasks in $\Gamma$.

The *MergeServers()* function can be evaluated in $O(1)$ time, if we assume that the union of the task sets $V_x$ and $V_y$ is done by copying references.

The loop of the *SelectBestPair()* function iterates through all server pairs. Since the number of servers are bounded by $n$, the loop has at most $n^2$ iterations. Within the loop body, comparing the periods, checking the area constraint, calling *MergeServers()* and computing the profit would take $O(1)$ time. Checking, whether the represented task sets $V_x, V_y$ are disjoint, would take $O(1)$ as well, if we store for each server pair a disjoint-flag. These flags can be updated, after each *MergeServers()* call in the *MSDL* procedure. Hence, the runtime of *SelectBestPair()* has the order $O(n^2)$.

The *MSDL* procedure itself first initializes one server for each task, which takes time of the order $O(n)$. After initialization, there are $(\frac{n^2-n}{2})$ server pairs with disjunct task sets (as many as edges in a complete graph). The main loop iterates, until no server pair can be merged anymore. Each merge reduces the number of disjunct server pairs by at least one. Hence the main loop runs at most $(\frac{n^2-n}{2})$ times. The time for the loop body is dominated by the time for the *SelectBestPair()* function. Hence, the total runtime of *MSDL* has the order $O(n^4)$.

The *MSDL* worst case runtime complexity of order $O(n^4)$ is acceptable, since the servers for a task set are constructed at design time. In practice, we have been able to evaluate *MSDL* on task sets of 200 tasks and more within some minutes. If needed, the worst case complexity can be reduced, if less time consuming heuristics for the *SelectBestPair()* function are used.

### 4.3.4 Conclusion on Server-Based Scheduling

The main idea of server based scheduling is, to execute the original tasks within server tasks. With the *MSDL* algorithm, we have presented a heuristic that computes at design time a set of servers for a given task set. At runtime, the servers will be scheduled by sequential *EDF*. Hence, the simple scheduling condition of *EDF* can be applied to the set of servers. The main advantage of *MSDL* is, that it creates at most as many servers as tasks existing in the original task set. Hence, we can use this approach with a simple full reconfiguration model, for which we compile one RHD configuration for each server.

For selecting the servers to be merged, as well as for the assignment of computation time and period to the server created in a merge, several heuristics can be developed. However, we omit studying further heuristics for the *MergeServers()* function, since our current implementation guaranties that only $n$ servers are generated. Furthermore, for such a greedy server generation method, our *SelectBestPair()* function promises the highest scheduling performance. It may be possible to develop scheduling heuristics that achieve a higher scheduling performance and/or create fewer servers than *MSDL*, by creating the server set using meta-heuristics such as genetic algorithms or simulating annealing. We leave this option for future work.

## 4.4 Chapter Conclusion

Within this chapter we have presented our three main approaches to schedule periodic real-time tasks to RHDs. We successfully adopted the idea of global scheduling and presented the two global EDF scheduling algorithms *EDF-NF* and *EDF-FkF*. Based on a resource augmentation approach, we developed a linear-time scheduling condition that holds for both algorithms.

Also the second popular multiprocessor scheduling approach, partitioned scheduling, has been successfully adapted for our RHD execution model. We formulated the partitioning problem and presented the ILP based *optimal-partitioned-EDF* and the heuristic *NFDA-*

*partitioned-EDF* to solve it.

As third method, we developed the server based approach for the scheduling of periodic tasks and presented the *MSDL* heuristic. This novel approach is specially suited for systems which do not support partial reconfiguration of the RHD.

While within this chapter we introduced, defined and provided basic analysis for each of our three main scheduling methods, we said few about how good the methods perform. Hence the next chapter will analyze and compare the performance of all approaches. This is done in two ways, analytically as well as by simulation studies.

CHAPTER 5

# Comparison of Algorithm Scheduling Performance

Within this chapter we will compare the scheduling performance of the proposed algorithms using two methods. First, we will present an analytic comparison in Section 5.1 and derive qualitative statements that tell us whether one algorithm dominates another or not. Together with scheduling conditions achieved in earlier analysis within this work we will be able to present a VENN-diagram which shows the relations among the sets of task sets, scheduled by the various algorithms.

Second, we will present the quantitative results from numerous simulation experiments in Section 5.2. They show which algorithm performs better in average, dependent on various parameters of the task sets. These results support a designer in choosing the appropriate algorithm for a particular system.

## 5.1 Analytical Comparison and Scheduling Anomalies

In an analytical comparison of the algorithms we are interested in whether one dominates the other or if the algorithms have unequal scheduling power in the sense, that there are task sets that can be feasibly scheduled by the first approach but not by the second and vice versa.

In our further analysis, we use $\mathit{\Gamma}^{\text{algo}}$ to denote the set of all task sets $\Gamma$ feasibly scheduled by some algorithm *algo*.

### Partitioned EDF vs. Global EDF

**Theorem 6.** *Neither the* optimal-partitioned-EDF *algorithm is dominant to* EDF-NF *nor vice versa.*

*Proof.* We first show, that we can construct example task sets $\Gamma$, which cannot be scheduled by any partitioned EDF algorithm but are feasibly scheduled by global *EDF-NF* and even by *EDF-FkF*. Hence, we prove with our example that $\digamma^{\text{EDF-FkF}} \not\subseteq \digamma^{\text{opt.-part.-EDF}}$.

We construct the example task set $\Gamma^A$ of three tasks, each with the same period, e.g. $P_i = 5$. The first task has an arbitrary small computation time, e.g. $C_1 = \epsilon$ but requires 100 % of the device area, i.e. $A_1 = A(H) = 5$. The two other tasks both require an arbitrary small amount of RHD area, e.g. $A_i = \epsilon$ but have a computation time of half their period, i.e. $C_i = 2.5$. The resulting task set including utilization values is shown in Table 5.1.

| $T_i$ | $P_i$ | $C_i$ | $A_i$ | $U^T(T_i)$ | $U^S(T_i)$ |
|---|---|---|---|---|---|
| $T_1$ | 5 | $\epsilon$ | $A(H) = 5$ | $\epsilon/5$ | $\epsilon$ |
| $T_2$ | 5 | 2.5 | $\epsilon$ | $1/2$ | $0.5\epsilon$ |
| $T_3$ | 5 | 2.5 | $\epsilon$ | $1/2$ | $0.5\epsilon$ |
| | | | | $1 + \frac{\epsilon}{5}$ | $2\epsilon$ |

Table 5.1: Example of a task set $\Gamma^A$ without feasible partitioning

Clearly, no feasible partitioning of $\Gamma^A$ exists. We cannot execute all tasks within one partition block, e.g. $G_1 = \{T_1, T_2, T_3\}$, since the time utilization $U^T(G_1)$ exceeds 1 and one task would miss its deadline as shown in Figure 5.1-(a) by the speckled task. Also a partitioning with more partition-blocks such as $\chi = \{G_1 = \{T_1, T_2\}, G_2 = \{T_3\}\}$ is not feasible, since the partition-blocks do not fit together onto the device, i.e. $A(\chi) > A(H)$, as shown in Figure 5.1-(a) by the hashed task.

On the other hand, $\Gamma^A$ will be feasibly scheduled by the global *EDF-NF* scheduling algorithm as shown in Figure 5.1-(b). The reason is that once all tasks are active, *EDF-*

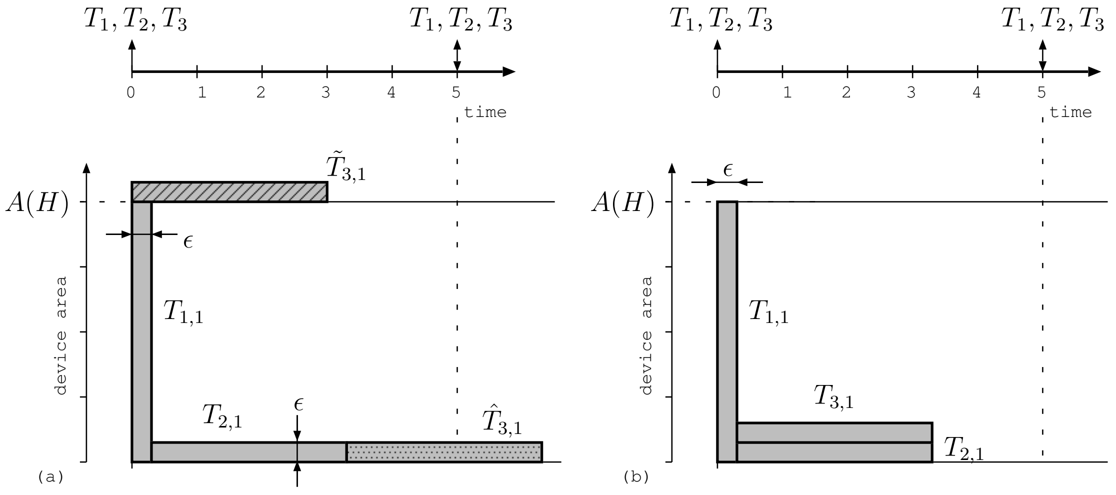


Figure 5.1: Scheduling anomaly: A task set infeasible by *any* partitioned schedule, but feasibly by global *EDF-NF*.

*NF* will select either $T_1$ or $T_2$ and $T_3$ for execution. If we assume that *EDF-FkF* selects tasks instances with equal deadlines in order of their indices, $\Gamma^A$ will also be feasibly scheduled by *EDF-FkF*.

Our observations hold for any $\epsilon$ between $0 < \epsilon \leq 2.5$. Hence, we can create task sets of any relative system utilization $\frac{U^S(\Gamma)}{A(H)} \in (0, 1]$ which are feasibly scheduled by *EDF-FkF* but infeasible by *opt.-part.-EDF*.

Now we prove that $F^{\text{opt.-part.-EDF}} \nsubseteq F^{\text{EDF-NF}}$ by constructing task sets $\Gamma^B$ which are feasible by the partitioning algorithm *opt.-part.-EDF* but infeasible by the global *EDF-NF* scheduler. $\Gamma^B$ consists of three tasks, where $T_1$ and $T_2$ have an arbitrary small computation time of e.g. $C_1 = C_2 = \epsilon$ but each requires 50% of the device area, e.g. $A_1 = A_2 = A(H)/2$. The third task requires an arbitrary small area of $A_3 = \epsilon$ but has a computation time equal to its period, e.g. $C_3 = P_3 = 5$. Let the period of the tasks $T_1, T_2$ be smaller then that of $T_3$, e.g. $P_1 = P_2 = 4$. The resulting task set considering a device of area $A(H) = 4$ is shown in Table 5.2.

| $T_i$ | $P_i$ | $C_i$ | $A_i$ | $U^T(T_i)$ | $U^S(T_i)$ |
|---|---|---|---|---|---|
| $T_1$ | 4 | $\epsilon$ | $0.5A(H) = 2$ | $0.25\epsilon$ | $0.5\epsilon$ |
| $T_2$ | 4 | $\epsilon$ | $0.5A(H) = 22$ | $0.25\epsilon$ | $0.5\epsilon$ |
| $T_3$ | 5 | 5 | $\epsilon$ | 1 | $\epsilon$ |
| | | | | $1 + 0.5\epsilon$ | $2\epsilon$ |

Table 5.2: Example task set $\Gamma^B$

The *NFDA* heuristic produces a partitioning of $G_1 = \{T_1, T_2\}$ and $G_2 = \{T_3\}$ which obviously can be feasibly scheduled as shown in Figure 5.2-(a). On the other hand, the schedule created by global *EDF-NF* is infeasible. As illustrated in Figure 5.2-(b),

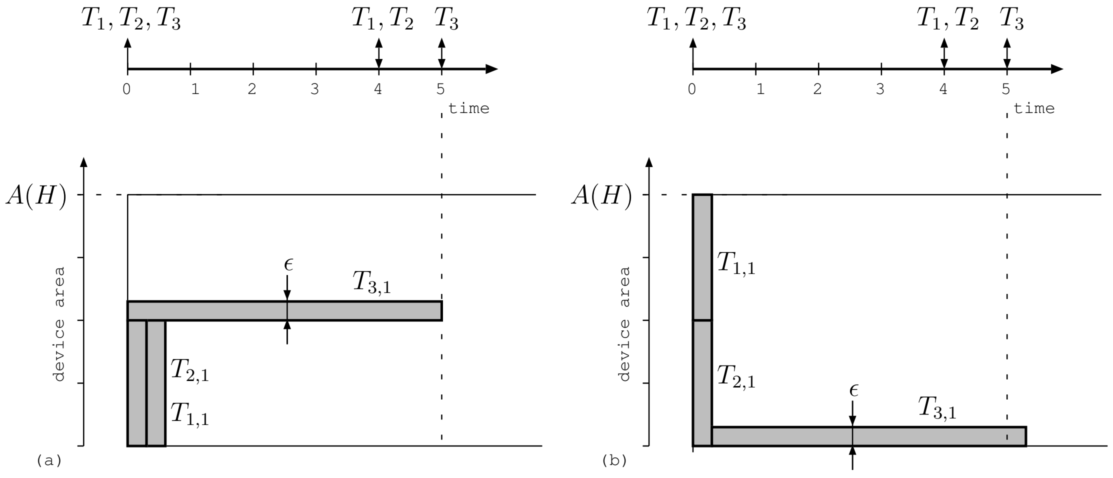


Figure 5.2: Scheduling anomaly: A task set infeasible by global *EDF-NF* but feasible by partitioned EDF *NFDH*.

*EDF-NF* will execute the tasks $T_1$ and $T_2$ first, which leads to the deadline miss of $T_3$. Our observation holds for all $\epsilon$ within $0 < \epsilon \leq 2$. Hence, we can create task sets of any relative system utilization $\frac{U^S(\Gamma)}{A(H)} \in (0, 1]$ which can be scheduled by *NFDA-part.-EDF* but not by *EDF-NF*.

□

### Partitioned EDF and Global EDF vs. MSDL

**Theorem 7.** *Neither the* optimal-partitioned-EDF *algorithm nor* EDF-NF *algorithm dominate the* MSDL *algorithm nor vice versa.*

*Proof.* We first show that there exists very simple task sets, that are feasibly scheduled by *optimal-partitioned-EDF* and *EDF-NF*, but our proposed *MSDL* algorithm fails. Afterwards we provide two example task sets. Both are feasibly scheduled by our *MSDL* heuristic, but the first one is infeasible by *opt.-part.-EDF* and the second by *EDF-NF*.

Consider the task set $\Gamma^C = \{T_1, T_2, T_3\}$, where all three task have the same period, e.g. $P_i = 10$ and same computation time e.g. $C_i = 4$ and an arbitrary small area $A_i = \epsilon$. Clearly the tasks cannot be scheduled entirely sequential, since the total time utilization is given by $U^T(\Gamma^C) = 1.2$. However, merging tasks into servers as used by the *MSDL* approach is not feasible. Let w.l.g. $T_1$ and $T_2$ be the tasks selected for merging into a server $S_3$ with period $P_S = 10$ and computation time $C_S = 4$. $S_3$ fully replaces $T_1$, but the $takeovertime(T_1, S_3)$ of Equation 4.42 evaluates to zero, i.e. the server $S_3$ cannot take over any amount of the computation time of task $T_2$. Hence, the overall time utilization $U^T(\Gamma^C)$ cannot be reduced to $\leq 1$ by server merging. On the other hand it is obvious, that the global *EDF-NF* and *EDF-FkF* methods as well as the *optimal-partitioned-EDF* and also the *NFDA-partitioned-EDF* method feasibly schedule $\Gamma^C$, since all three tasks can be executed in parallel. Hence, we showed that $\mathcal{F}^{\text{EDF-FkF}} \not\subseteq \mathcal{F}^{\text{MSDL}}$ and that $\mathcal{F}^{\text{opt.-part.-EDF}} \not\subseteq \mathcal{F}^{\text{MSDL}}$.

Next we consider the task set $\Gamma^D$ of Table 5.3. which is infeasible by any partitioned

| $T_i$ | $P_i$ | $C_i$ | $A_i$ | $U^T(T_i)$ | $U^S(T_i)$ |
|---|---|---|---|---|---|
| $T_1$ | 5 | 1 | $A(H)$ | 0.2 | $0.2A(H)$ |
| $T_2$ | 5 | 3 | $\epsilon$ | 0.6 | $0.6\epsilon$ |
| $T_3$ | 10 | 3 | $\epsilon$ | 0.3 | $0.3\epsilon$ |
| | | | | 1.1 | |

| $S_i$ | $P_i$ | $C_i$ | $A_i$ | $U^T$ | $U^S$ |
|---|---|---|---|---|---|
| $S_1 = \{T_1\}$ | 5 | 1 | $A(H)$ | 0.2 | $0.2A(H)$ |
| $S_4 = \{T_2, T_3\}$ | 5 | 3 | $2\epsilon$ | 0.6 | $1.2\epsilon$ |
| | | | | 0.8 | |

Table 5.3: Example task set $\Gamma^D$ without feasible partitioning and feasible set of servers $\Omega^D$

scheduler for the same reason[1] as it is for task set $\Gamma^A$. The *MSDL* scheduler will merge task $T_2$ and $T_3$ into a server $S_4$ with period $P_4 = 5$ and computation time $C_4 = 3$, hence $T_2$ is fully replaced by the server. The reduced computation time of $T_3$ can be computed according to Equation 4.41 and evaluates to zero.

$$C_3 \leftarrow C_3 - C_4 \cdot (\lfloor \frac{P_4}{P_3} \rfloor - 1) = 0 \tag{5.1}$$

Since the final set of servers $\Omega^D = \{S_1, S_4\}$ has a time utilization below one, it can be feasibly scheduled. Hence we showed that $\digamma^{\text{MSDL}} \not\subseteq \digamma^{\text{opt.-part.-EDF}}$.

Next we consider the example task set $\Gamma^E$ given in Table 5.4. As shown in the *EDF-NF* schedule in Figure 5.3-a, the first instance of task $T_3$ will miss its deadline. The reason is, that $T_3$ has a large period and hence a far deadline for the first instance $T_{3,1}$. Therefore, it gets preempted 10 times by the tasks $T_1$ and $T_2$ which have shorter periods and hence closer deadlines.

| $T_i$ | $P_i$ | $C_i$ | $A_i$ | $U^T(T_i)$ | $U^S(T_i)$ |
|---|---|---|---|---|---|
| $T_1$ | 100 | 12 | $0.5A(H)$ | 0.12 | 0.12 |
| $T_2$ | 100 | 12 | $0.5A(H)$ | 0.12 | 0.12 |
| $T_3$ | 1000 | 900 | $\epsilon$ | 0.90 | $0.9\epsilon$ |

Table 5.4: Example task set $\Gamma^E$

When applying the *MSDL* algorithm on $\Gamma^E$, it merges task $T_1$ and $T_3$ into a server $S_4$ with $P_4 = 100, C_4 = 12$. $T_1$ gets fully replaced and the computation time of $T_3$ is reduced by 9 times the computation time of the server.

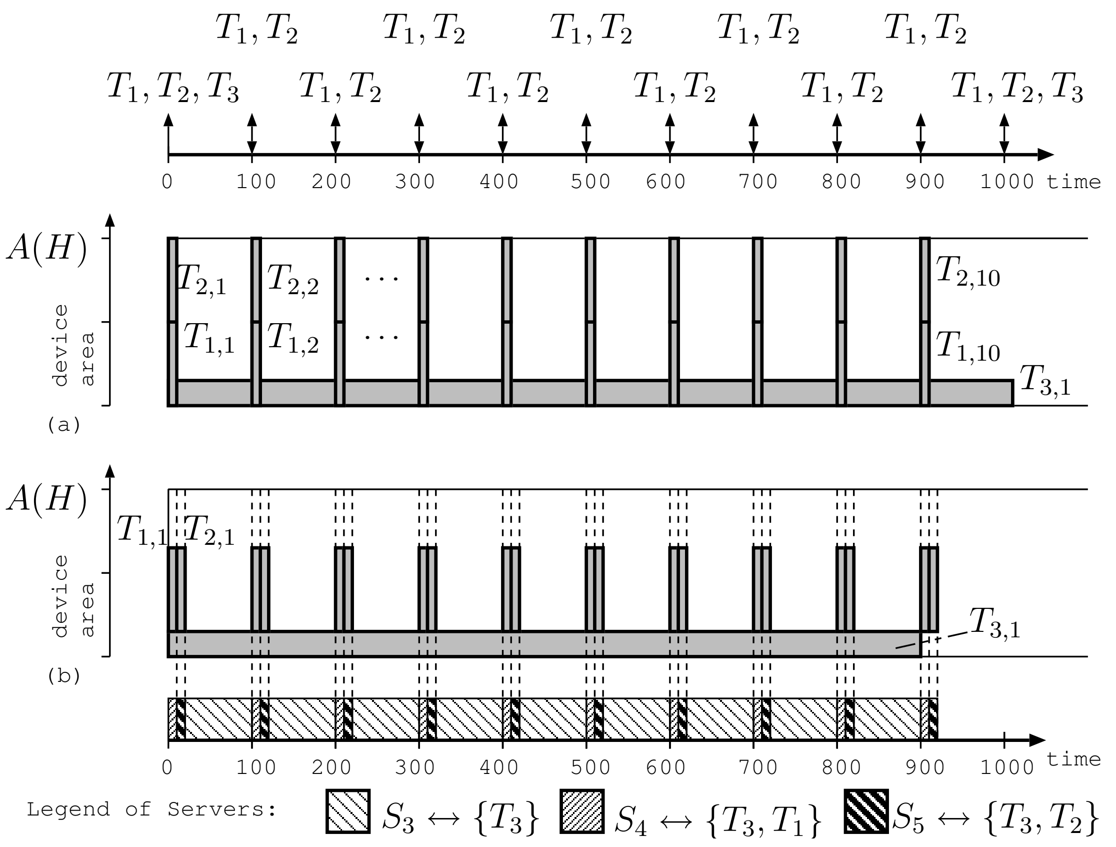
$$C_3 \leftarrow C_3 - C_4 \cdot (\lfloor \frac{P_4}{P_3} \rfloor - 1) = 900 - 12 \cdot 9 = 792 \tag{5.2}$$

In the next merge, $T_2$ and the remaining part of $T_3$ are merged into $S_5$, and $C_3$ is reduced further by 108 units to 684. The final set of servers shown in Table 5.5 has a time utilization of 0.924. The schedule of the servers and the tasks *inside* the servers is shown in Figure 5.3-b. Hence we showed by the example, that $\digamma^{\text{MSDL}} \not\subseteq \digamma^{\text{EDF-NF}}$.

| $S_i$ | $P_i$ | $C_i$ | $A_i$ | $U^T$ |
|---|---|---|---|---|
| $S_3 = \{T_3\}$ | 1000 | 684 | $\epsilon$ | 0.684 |
| $S_4 = \{T_1, T_3\}$ | 100 | 12 | $0.5A(H)$ | 0.120 |
| $S_5 = \{T_2, T_3\}$ | 100 | 12 | $0.5A(H)$ | 0.120 |
| $\sum$ | | | | 0.924 |

Table 5.5: Feasible set of servers created by *MSDL* for task set $\Gamma^E$

$\square$



$T_1, T_2$
$T_1, T_2, T_3$
0 100 200 300 400 500 600 700 800 900 1000 time
$A(H)$
device area
$T_{2,1}$ $T_{2,2}$ ...
$T_{1,1}$ $T_{1,2}$ ...
$T_{2,10}$
$T_{1,10}$
$T_{3,1}$
(a)
(b)
Legend of Servers:
$S_3 \leftrightarrow \{T_3\}$
$S_4 \leftrightarrow \{T_3, T_1\}$
$S_5 \leftrightarrow \{T_3, T_2\}$

Figure 5.3: Scheduling anomaly: A task set infeasible by global *EDF-NF* but feasible by *MSDL*.

### Relation Between all Algorithms

At the end of our analysis we are able to present the relations between all proposed scheduling methods as a *big picture*. Figure 5.4 shows the sets of feasibly scheduled task sets by the different algorithms as VENN-diagram. The rectangle presents the set of all task sets $\Gamma$, for which a feasible schedule exists. Of course none of our proposed methods schedules all these task sets, since they all are suboptimal algorithms. The region in the center represents all task sets accepted by the *EDF-FkF* scheduling condition of Theorem 3. It is a true subset of the task sets de facto scheduled by *EDF-FkF* which again is a true subset of the task sets de facto scheduled by *EDF-NF*:

$$\mathcal{F}^{\text{EDF-FkF-condition}} \subset \mathcal{F}^{\text{EDF-Fkf}} \subset \mathcal{F}^{\text{EDF-NF}} \tag{5.3}$$

On the other hand we showed by Theorem 5 that all task sets accepted by the *EDF-*

[1] One partition-block is infeasible, since the time utilization is greater than one. Two or three partition-blocks are infeasible, since the device area is exceeded.

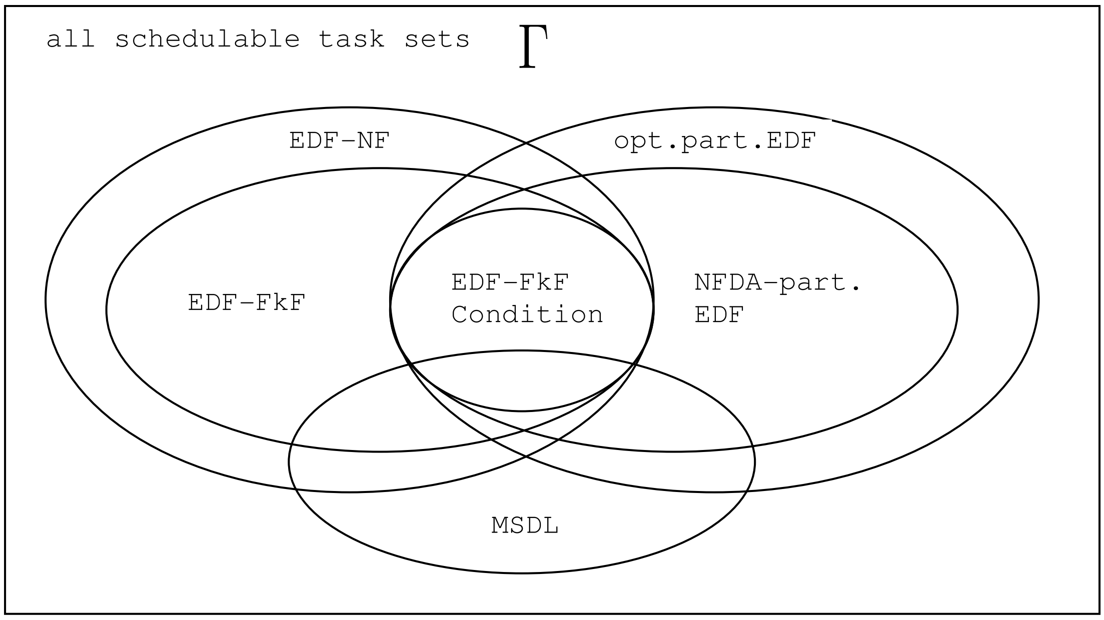


Figure 5.4: The relation between feasibly scheduled task sets by the proposed algorithms as VENN-diagram.

*FkF* scheduling condition are also feasibly scheduled by the partitioned EDF scheduler using the *NFDA*-partitioning heuristic. Obviously, the *optimal-partitioned-EDF* scheduler dominates the *NFDA-partitioned-EDF* algorithm. These relations are shown in Equation 5.4 and correspond to the right side of the VENN-diagram:

$$\mathcal{F}^{\text{EDF-FkF-condition}} \subset \mathcal{F}^{\text{NFDA-partitioned-EDF}} \subset \mathcal{F}^{\text{optimal-partitioned-EDF}} \tag{5.4}$$

Finally we showed by constructing counter examples, that the task sets scheduled by three basic approaches *EDF-NF*, *optimal-partitioned-EDF* and *MSDL* are not subsets of each other. In particular we showed, that *EDF-FkF* is no subset of *optimal-partitioned-EDF* or *MSDL*, that *NFDA-partitioned-EDF* is no subset of *EDF-NF* or *MSDL* and that *MSDL* is no subset of *EDF-NF* or *optimal-partitioned-EDF*. [2]

[2]An complete proof of the set-relations shown in the VENN-diagram would require the construction of a task set for each of the 19 intersections. Beside the presented task sets for the major relations we abstain from presenting further task sets at this place. However, we found task sets for each intersection during our intensive simulations on random generated benchmarks.

## 5.2 Simulation Results

Due to our analytic comparison and analysis we are able to tell whether one algorithm dominates another one or not, and furthermore know scheduling conditions for some of the proposed algorithms. However, it tells us little about how good the algorithms perform on task sets appearing in practice, nor does it tells us anything about which method performs better for task sets of certain characteristics.

In order to answer those questions, an experimental evaluation on application benchmarks becomes necessary. The best one could do is using benchmarks which are widely accepted in the community and consist of applications appearing in practice. Not surprisingly, those benchmarks do not exist, since we consider novel methods for a novel and rarely studied problem. Hence, we have to rely on synthetically created benchmarks. In order to construct meaningful task sets one idea is to look at the parameters of typical FPGA tasks, such as the area of *Discrete Wavelet Transform* design or an *MPEG 2 Video Decoder*. On the other hand we can consider the amount of resources provided by today's modern FPGAs as proper values for the device area. We concluded that this strategy is of little use, since today's device and design sizes are very variable. For example, the *Discrete Wavelet Transform* design requires only 20% of an XILINX Virtex II XC2V3000 FPGA, but will exceed the size of the smaller but still modern XILINX Spartan III FPGAs. Also we know little on typical task computation times, since they depend considerably on the amount of data to be processed. As conclusion we decided to create benchmarks of pure synthetic task sets, but tried to cover wide ranges of the task set parameters.

The next paragraph precisely describes our benchmark generation methods and discusses some characteristics of the created task sets. This should enable the reader to reproduce the benchmarks, validate results and compare our algorithms to other methods.

Afterwards, we present results from simulations that show the performance of the algorithms and the impact of the tasks' area and time-utilization values, the impact of the number of tasks in a task set and how the performance increases when various algorithms are combined.

### 5.2.1 Creation of Benchmarks

Recall that a task set $\Gamma$ consist of periodic tasks $T_i$ each specified by a triple parameter $T_i(P_i, C_i, A_i)$. The main characteristics of the tasks are $U^T(T_i) = C_i/P_i$ as well as $U^S(T_i) = C_i/P_i \cdot A_i$ whereas our main interest in the task set's characteristics are the number of task $n = |\Gamma|$ and the overall system utilization $U^S(\Gamma) = \sum_{i=1}^{n} U^S(T_i)$.

A benchmark creation method creates a given number of task sets $\{\Gamma_1, \Gamma_2, \ldots\}$ by choosing the task parameters and task set sizes according to some rules. To define meaningful rules is rather difficult. For example we cannot choose all task parameters from intervals with a uniform (or normal) distribution, since the parameters depend on each other. In our first task set generation method we decided to keep the $A_i$ and $U_i^T$ values of the tasks uniformly distributed.

Our first task set generation method works as follows: Each task set is composed of randomly generated tasks $T_i$. For the creation of tasks a benchmark $BM$ specifies an interval $[c_{min}, c_{max}]$ of natural numbers for the task's computation times, an interval $[a_{min}, a_{max}]$ for the task's areas, and an interval $[u^T_{min}, u^T_{max}]$ for the task's time-utilization factor. A task is created by randomly choosing its parameters from the intervals with a uniform distribution. Its period is computed according to its computation time and time-utilization. In order to create the task sets $\Gamma \in BM$ with system-utilization distributed from 0% to 100%, randomly generated tasks are added to a task set until a specified task set system-utilization $U^S(\Gamma)$ is reached. Furthermore, a task set is dismissed when its hyper period exceeds a given limit. This is necessary to keep the simulation time of the global EDF algorithms within reasonable bounds. The pseudo code of the algorithm creating one task set is shown in Algorithm 5.

**Algorithm 5** Benchmark Creation Method 1

```
1: procedure CREATETASKSET($U^S_{bound}$, $HP_{bound}$)
2:     repeat
3:         $\Gamma \leftarrow \emptyset$
4:         repeat
5:             $T \leftarrow$ CREATETASK
6:             $\Gamma \leftarrow \Gamma \cup T$
7:         until $U^S(\Gamma) > U^S_{bound}$
8:         $\Gamma \leftarrow \Gamma \setminus T$
9:     until hyperperiod $(\Gamma) \leq HP_{bound}$
10: end procedure
11: procedure CREATETASK
12:     $C \leftarrow$ rand($[c_{min}, c_{max}]$ )
13:     $A \leftarrow$ rand($[a_{min}, a_{max}]$ )
14:     $tempU^T \leftarrow$ rand($[u^T_{min}, u^T_{max}]$ )
15:     $P \leftarrow$ round($\frac{C}{tempU^T}$)
16:     return $T(P, C, A)$
17: end procedure
```

While the values for the tasks' area, computation time and time-utilization are uniformly distributed within some interval, the periods, the system-utilization and the number of tasks is a result of the creation method. To get a deeper idea of the characteristics of the benchmarks created by the described method we study the distribution of all parameters of interest. We create our standard benchmark $BM_{std}$ using the following parameters:

$$param(BM_{std}) = \{c_{min} = 1, c_{max} = 30, a_{min} = 0.1, a_{max} = 0.5, u^T_{min} = 0.1, u^T_{max} = 0.5\} \tag{5.5}$$

The characteristics of the created tasks are illustrated in Figure 5.5. The histograms show the various task parameters, their values and their relative frequency. As specified in the benchmark parameters, the task computation times $C_i$ and area values $A_i$ are uniformly distributed in $[1, 30]$ and $[0.1, 0.5]$ respectively. Since we required a uniform distribution of the time-utilization $U^T(T_i) \in [0.1, 0.5]$, the periods $P_i$ will result with values between 2 and 300 with a distribution as shown in the second histogram of Figure 5.5. Moreover, since the $A_i$ and $U^T(T_i)$ are uniformly distributed in $[0.1, 0.5]$, the system utilization of the tasks $U^S(T_i)$ is between 0.1 and 0.25 and distributed as shown in the fifth histogram. The last plot in the figure shows the average number of tasks $n$ in a task set $\Gamma$, when creating task sets of various system-utilization $U^S(\Gamma)$.

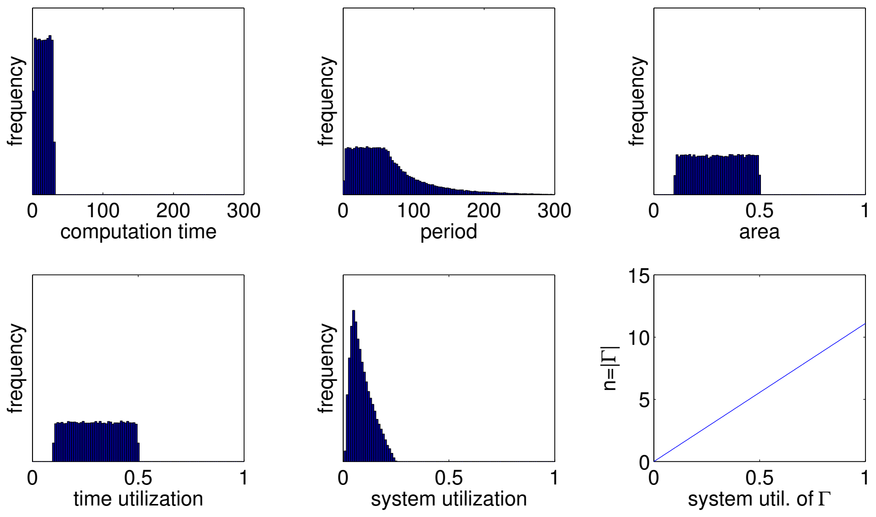


Figure 5.5: Characteristics of $BM_{sdt}$ tasks and task sets

## 5.2.2 Performance on Standard Benchmark $BM_{std}$

In our first simulation experiment, we created our standard benchmark $BM_{std}$ with ten thousand task sets with various system-utilization values. The hyper-period was bounded by 100000, to keep the simulation time reasonably small.

Figure 5.6 compares the scheduling performance of all proposed algorithms on the benchmark $BM_{std}$. The graphs where created by partitioning the 10000 task sets into 20 classes according to their $U^S(\Gamma)$ value. For each class and algorithm one data point was determined. The y-values of the data points present the *success-rate*, i.e. the fraction of feasibly scheduled task sets of a class by the particular algorithm. The x-values present the mean system-utilization of the task sets of a class.

**Global EDF Scheduling:** The global *EDF-NF* approach is the algorithm with the best performance on our standard benchmark. It is able to schedule about 50% of the task sets of the class with a mean system utilization factor around 81% and accepts almost all task sets with $U^S$ less than 70%. The performance drop of the global *EDF-FkF*

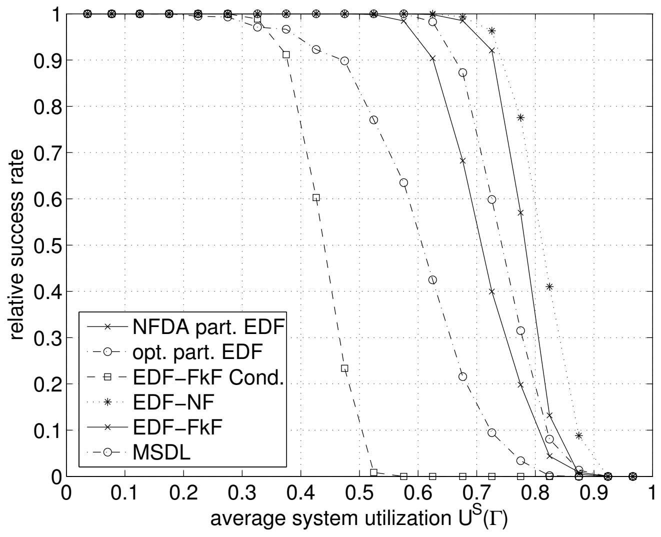


Figure 5.6: Performance of algorithms on $BM_{sdt}$

scheduler compared to *EDF-NF* seems to be rather small. However, the difference is significant for task sets with a system utilization within $[0.75, 0.9]$, where *EDF-NF* generates feasibly schedules for four times as many task sets than *EDF-FkF*. In contrast to the relative good performance of the global schedulers *EDF-FkF* and *EDF-NF* it is shown that our scheduling test *(EDF-FkF-condition)* developed in Section 4.1.3 is rather pessimistic. It accepts almost all task sets with $U^S(\Gamma) < 0.5$ but rejects almost all task sets with $U^S(\Gamma) > 0.5$. However, this large discrepancy between the amount of task sets actually feasibly scheduled and the amount accepted by the scheduling test is in conformance to the results reported for the related multiprocessor scheduling tests [16].

**Partitioned EDF Scheduling:** On our standard benchmark the partitioning schedulers show a significant worse performance than the global schedulers. From task sets with $U^S(\Gamma)$ around 0.78, only 20% could be scheduled by the *optimal-partitioned-EDF* algorithm, while *EDF-NF* schedules up to 80%. However, *opt.-partitioned-EDF* still schedules 75% of task sets with $U^S(\Gamma)$ around 0.7 and almost every task set with $U^S(\Gamma) < 0.6$. When using the simple *Next-Fit-Decreasing-Area* heuristic to partition the task sets, the performance drops only slightly compared to the optimal partitioning achieved by solving an ILP model. It has to be mentioned, that the graphs for the partitioning approaches represent the task sets which could be feasibly partitioned and hence can be guaranteed to be feasibly scheduled, while the global EDF graphs represent the results achieved by simulating the schedule over the entire hyper-period. While the partitioning can be done in reasonable time, the simulation over the hyper-period may often be too time consuming.

**Server Based Scheduling:** The approach based on executing tasks within periodic servers presented in Section 4.3 shows the worst performance on our standard benchmark.[3] MSDL is able to schedule only few task sets with a $U^S$ exceeding 0.7, and achieves an acceptance rate of 50% for task sets with a $U^S$ around 0.6. However, with this relatively poor performance MSDL may still be reasonable and the method of choice, since it has various advantages when it comes to system realization (See Chapter 6).

### 5.2.3 Impact of Area and Time-utilization

In the next experiment we will study the impact of the tasks area and time-utilization values on the performance of the various scheduling algorithms. Hence we created two more benchmarks. In the $BM_{smallAbigU^T}$ benchmark, the average task area has been halved while time-utilization has been doubled compared to the $BM_{std}$ benchmark. For the $BM_{bigAsmallU^T}$ benchmark we did the same vice versa. Table 5.6 shows the parameters used for the benchmark generation in detail.

| benchmark | $c_{min}$ | $c_{max}$ | $a_{min}$ | $a_{max}$ | $u^T_{min}$ | $u^T_{max}$ |
|---|---|---|---|---|---|---|
| $BM_{std}$ | 1 | 30 | 0.10 | 0.50 | 0.10 | 0.50 |
| $BM_{smallAbigU^T}$ | 1 | 30 | 0.05 | 0.25 | 0.20 | 1.00 |
| $BM_{bigAsmallU^T}$ | 1 | 30 | 0.20 | 1.00 | 0.05 | 0.25 |

Table 5.6: Benchmark parameters

Using this parameters we achieved benchmarks comparable to $BM_{std}$ in terms of tasks' system-utilization and task sets size, but with different distribution of the area and time-utilization values. Especially the ratio of $A_i/U^T(T_i)$ is different. In the $BM_{std}$ some few tasks have an extreme $A_i/U^T(T_i)$ ratio of 5 to 1 and 1 to 5 respectively, most tasks have a 1 to 1 ratio as shown in the first histogram of Figure 5.7. In contrast, the tasks

[3]except the EDF scheduling condition which we do not treat as scheduling algorithm

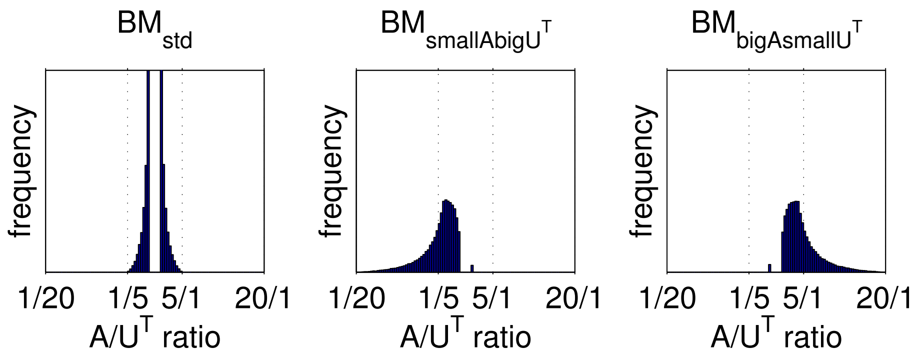


Figure 5.7: Distribution of the tasks' $A_i$ to $U^T(T_i)$ ratio of the $BM_{sdt}$,$BM_{smallAbigU^T}$ and $BM_{bigAsmallU^T}$ benchmarks.

of $BM_{smallAbigU^T}$ can have a $A_i/U^T(T_i)$ ratio as small as 1 to 20. The second histogram of Figure 5.7 shows, that for this benchmark the most frequent ratio is about 1 to 5. The tasks of the $BM_{bigAsmallU^T}$ benchmark have exactly the converse distribution of the $A_i/U^T(T_i)$ ratio, which is shown in the last plot of Figure 5.7.

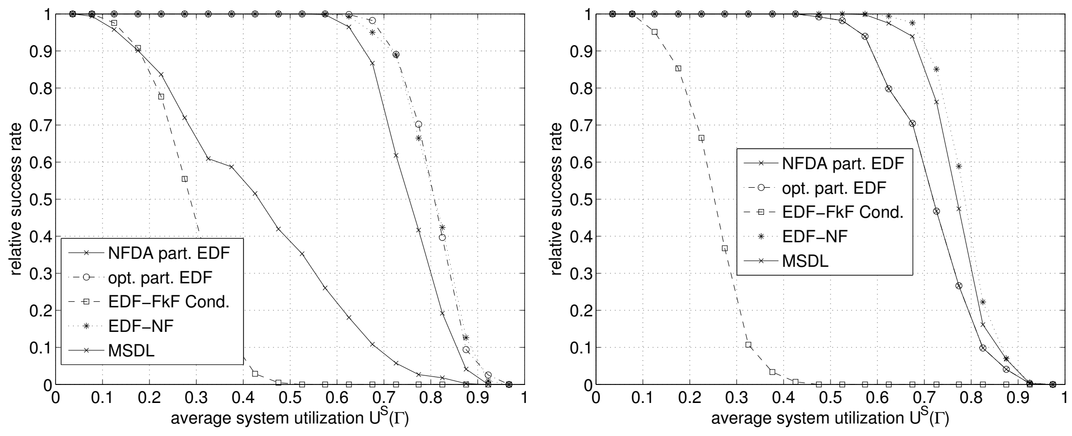


Figure 5.8: Performance of algorithms on $BM_{smallAbigU^T}$ and $BM_{bigAsmallU^T}$ benchmarks.

The performance evaluation of the scheduling algorithms on the $BM_{smallAbigU^T}$ and $BM_{bigAsmallU^T}$ is shown in Figure 5.8. [4]

The *opt.-partitioned-EDF* and the global *EDF-NF* show almost identical performance on the $BM_{smallAbigU^T}$ benchmark. Compared to the results of the $BM_{std}$ benchmark, the performance of *opt.-partitioned-EDF* has considerably improved while the performance of global *EDF-NF* stays almost constant. One possible reason for the improvement of the *opt.-partitioned-EDF* is that the benchmark contains no tasks with large area and hence situations as shown by the anomaly in Figure 5.1 cannot occur. The opposite effect can be seen in the performance evaluation of the $BM_{bigAsmallU^T}$ benchmark: Here, the performance gap between the *opt.-partitioned-EDF* and the *EDF-NF* has slightly grown compared to the $BM_{std}$ benchmark.

Figure 5.9 compares the performance on the three benchmarks for the algorithms *EDF-NF*, *opt.-part.-EDF* and *MSDL* separately. It can be seen that while the performance of *EDF-NF* is almost constant on the three benchmarks, the performance of *opt.-part.-EDF* is better on tasks with small area and gets worse on tasks with large area. However, the greatest sensitivity on the ratio of task area to time-utilization is shown by *MSDL*. The performance on the $BM_{smallAbigU^T}$ benchmark drops dramatically compared to the standard benchmark. The reason is, that in order to find a feasible set of servers, *MSDL* has to fulfill a high number of server merges since the tasks have rather small area values. Each merge adds some overhead into the set of servers. The opposite effect takes place

[4] From the global schedulers we present only the better *EDF-NF* and leave out the evaluation of the *EDF-FkF*, since our simulations did not show interesting differences among the two global schedulers in the sensitivity on the $A_i/U^T(\Gamma_i)$ ratio.

on task sets with large area values. On the $BM_{bigAsmallU^T}$ benchmark *MSDL* performs as well as *EDF-NF* and significantly better than *opt.-part.-EDF* (see Figure 5.8 right side). Here, only few server merges are required to find the final set of servers and thus little overhead is introduced. However, the experiment also indicates that *MSDL* may perform poor on task sets with a large number of tasks, since these may require a large number of server merges as well. This assumption is confirmed by the next experiment.

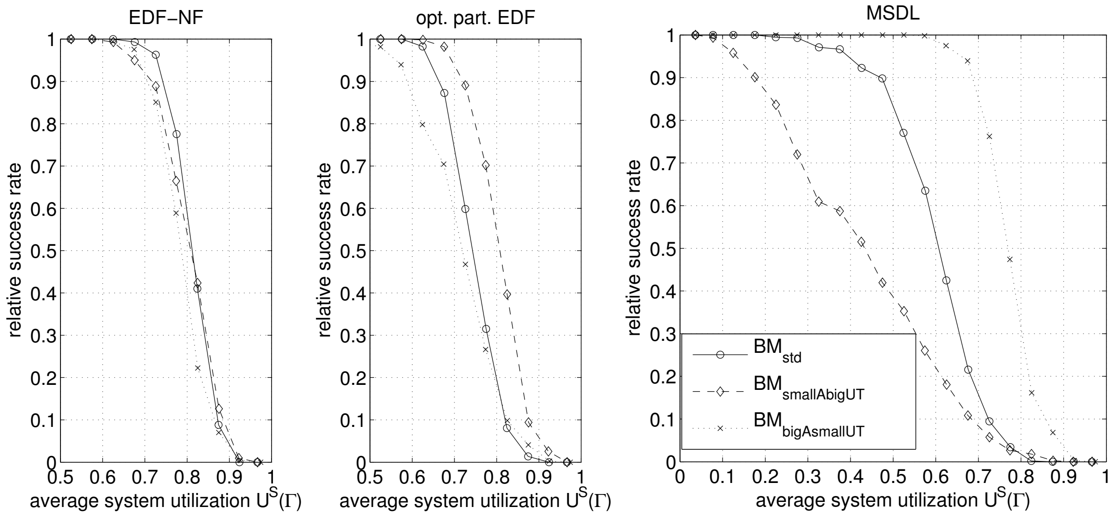


Figure 5.9: Performance of algorithms *EDF-NF*, *opt.-part.-EDF* and *MSDL* on the $BM_{std}$,$BM_{smallAbigU^T}$ and $BM_{bigAsmallU^T}$ benchmarks.

### 5.2.4 Performance of Combined Algorithms

In Section 5.1 we showed that the major scheduling approaches do not dominate each other. Hence, combining two or more of the scheduling algorithms might increase the scheduling performance compared to the single algorithms. We analyzed the performance of all promising algorithm combinations as they are: *EDF-NF + opt.-part.-EDF*, *EDF-NF + MSDL*, *opt.-part.-EDF + MSDL* and *EDF-NF + opt.-part.-EDF + MSDL*. Combinations including the algorithms *EDF-FkF* and *NFDH-part.-EDF* have a lower performance and thus have been omitted in our analysis.

The performance evaluation has been done on all three benchmarks $BM_{std}$,$BM_{smallAbigU^T}$ and $BM_{bigAsmallU^T}$. However, only in two cases the combination of algorithms yielded a non-negligible increase in performance:

When applying *EDF-NF + opt.-part.-EDF* to the $BM_{smallAbigU^T}$ benchmark, much more task sets could be feasibly scheduled in comparison to the task sets scheduled by only *EDF-NF* or only *opt.-part.-EDF*. Especially when the task set's system utilization is in the range of 0.75 and 0.85 this algorithm combination increases the performance by about 10% as shown in Figure 5.9-left. However on the other two benchmarks, where the performance of *EDF-NF* is considerably higher than that of *opt.-part.-EDF*, the combination of both algorithm did only show negligible performance increase compared to *EDF-NF*.

The combination of *EDF-NF + MSDL* applied to the $BM_{bigAsmallU^T}$ benchmark showed a small increase in performance compared to *EDF-NF*. Figure 5.9-right shows, that this improvement stays always less then about 3%. On the other benchmarks, we got no benefit from combining *EDF-NF + MSDL*.

The combinations *opt.-part.-EDF + MSDL* did not show performance improvement on any of the three benchmarks. On $BM_{std}$ and $BM_{smallAbigU^T}$ the performance was not better than that of *opt.-part.-EDF* and on $BM_{bigAsmallU^T}$ was not better than that of *MSDL*. Also considering all three algorithms did not show an advantage compared to the performance achieved by *EDF-NF + opt.-part.-EDF* and *EDF-NF + MSDL* respectively.

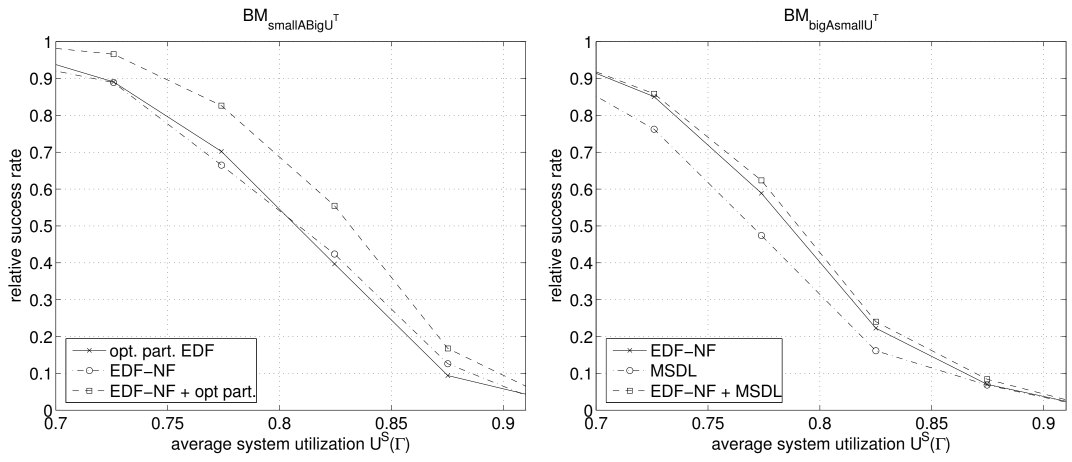


Figure 5.10: Performance of combined algorithms on $BM_{smallAbigU^T}$ and $BM_{bigAsmallU^T}$ benchmarks.

### 5.2.5 Impact of Number of Tasks

In order to analyze how well the proposed algorithms scale we created task sets with various numbers of tasks. Our benchmark creation method of Section 5.2.1 is not able to create task sets with a large number of tasks but a small hyper-period, since the task periods are arbitrary. Moreover, the number of tasks in a task set is not constant within a benchmark. Hence we choose a different method that creates task sets with a given number of tasks and keeps the hyperperiod small by choosing periods from a given set of harmonic related values.

The entire creation method is shown as pseudo code in Algorithm 6. First we choose for all $n$ tasks a period from a given set and a temporary area and time-utilization value. We compute a scaling factor *scale* by dividing the desired task set's system utilization by the actual task set's system utilization. Afterwards, we scale all area values and select the computation time, such that the final task set Γ has the desired system utilization.

**Algorithm 6** Benchmark Creation Method 2

```
 1: procedure CREATETASKSET($n, U^S_{nominal}, P_{eriodSet}$)
 2:     $U^S_{actual} \leftarrow 0$
 3:     for all $i \in \{1, n\}$ do        ▷ Choose period and temp. area and time utiliz.
 4:         $P_i \leftarrow$ chooseFrom $P_{eriodSet}$
 5:         $a_i \leftarrow$ rand([0,1])
 6:         $u_i \leftarrow$ rand([0,1])
 7:         $U^S_{actual} \leftarrow U^S_{actual} + a_i \cdot u_i$
 8:     end for
 9:     $scale \leftarrow U^S_{nominal} / U^S_{actual}$
10:     for all $i \in \{1, n\}$ do        ▷ Scale area values and select comp times
11:         $A_i \leftarrow a_i \cdot \sqrt{scale}$
12:         $C_i \leftarrow \lceil P_i \cdot u_i \cdot \sqrt{scale} \rceil$
13:         $\Gamma \leftarrow \Gamma \cup T_i(P_i, C_i, A_i)$
14:     end for
15:     return $\Gamma$
16: end procedure
```

Using this, three benchmarks of 10000 task sets each have been created, where all task sets of $BM_{10Tasks}$ contain 10 tasks, all task sets of $BM_{20Tasks}$ contain 20 tasks and the task sets of $BM_{50Tasks}$ contain 50 tasks respectively. The periods have been chosen from $P_{eriodSet}$ = {100, 200, 400, 600, 800, 1000, 2000, 4000, 6000, 8000, 10000}, such that the hyper period is bounded by 120000 and the simulation time for each task set is reasonably small.

Figure 5.11 compares the performance of the three benchmarks for the algorithms *EDF-NF*, *NFDA-part.-EDF* and *MSDL* separately. [5] It can be seen that the performance of *EDF-NF* and *NFDA-part.-EDF* improves on the task sets with greater number of tasks. The reason is, that the tasks of $BM_{20Tasks}$ and $BM_{50Tasks}$ have in average smaller area and time-utilization values as the tasks of $BM_{10Tasks}$. When *EDF-NF* selects the tasks for execution or *NFDA-part.-EDF* packs tasks into levels, the amount of wasted device resources by fragmentation is small compared to task sets with *"larger"* tasks.

In contrast, the performance of *MSDL* methods gets worse, as the number of tasks in the task sets increases (Figure 5.11, right side). As mentioned earlier, each *merge* of two tasks or servers introduces an overhead into the task set, since one of the tasks is afterwards executed with a period smaller than its original period. The greater the number of tasks of the task sets, the greater the overall time-utilization $U^T(\Gamma)$ of the task set (in average) is. Hence, it requires more merges to reduce the $U^T(\Gamma)$ to less than or equal to one and it becomes more likely that the method terminates before a feasible set of servers is found.

[5] From the partitioned EDF methods we present only the *NFDA* heuristic since the optimal partitioning for task sets of 50 tasks could not be computed in reasonable time.

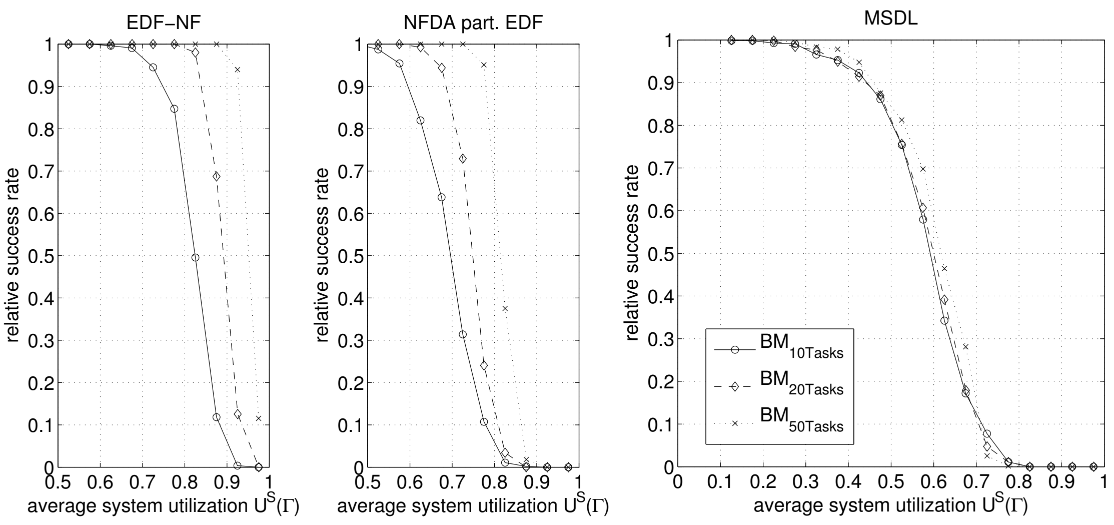


Figure 5.11: Performance of algorithms *EDF-NF*, *NFDA-part.-EDF* and *MSDL* on task sets with 10, 20 and 50 tasks.

## 5.3 Conclusion on Algorithm Performance

We have compared the performance of the proposed algorithm analytically as well as by simulations on synthetic benchmarks. The main conclusions concerning the pure scheduling performance of the algorithms are summarized below:

**dominance:** The three basic approaches, global EDF scheduling, partitioned EDF scheduling and server based EDF scheduling do not dominate each other. However, the simulation study showed that combining the algorithms yields a rather low performance improvement.

**robustness:** For each of the basic scheduling algorithms there exist task sets, that are infeasible by the particular algorithm, have an infinitesimal small system-utilization, but can be feasibly scheduled by some algorithm.

**pessimistic analysis of global EDF:** The scheduling test for the global EDF algorithms is pessimistic in the sense, that each accepted task set will also be scheduled by the *NFDA-partitioned-EDF* method.

**average performance:** In most cases, the global EDF approach shows the best performance followed by the partitioned EDF approaches. The performance of *MSDL* and the global EDF scheduling test is much worse.

**sensitivity:** The performance of global EDF seems to be insensitive on the ratio of the task's area to the task's time-utilization values, while the partitioned EDF improves with tasks with small area and *MSDL* improves with tasks with large area.

**scaling:** The performance of the global and partitioned EDF methods significantly improve with task sets with many but *"small"* tasks, while the performance of *MSDL* does not.

While this performance analysis of this chapter totally neglected any overheads, the next chapter describes the execution models and associated overheads in detail.

CHAPTER 6

# Execution Model and Overhead Analysis

Up to now, we have neglected any system overheads due to reconfiguration, context switching, and running the scheduling routines in our analysis. Neglecting such overheads is a common practice in microprocessor-based real-time systems, but only justified when the time overhead is small compared to the task computation times. For reconfigurable hardware, the time overheads can only be discussed in conjunction with the actually used reconfiguration mode. In this chapter we outline the reconfiguration models, list the requirements on the runtime system and model the resulting reconfiguration overheads for each of our three scheduling methods. Hence we will be able to provide off-line scheduling tests which consider the reconfiguration overhead. Finally we will repeat some of our simulation experiments on the algorithm performance *including reconfiguration overheads.*

We presented the overhead analysis for *global EDF* in [DP06a], for *partitoned EDF* in [DP06b] and for *MSDL* in [DMP06].

## 6.1 Global EDF Overhead Model

### 6.1.1 Reconfiguration Modes and Task Placement

In this section, we outline an execution model for a reconfigurable hardware platform that matches the global EDF scheduling techniques developed in Section 4.1. We further pose some requirements on the runtime system managing the hardware platform. Then we become in a position that enables us to model and analyze the overheads involved.

Recall that generally reconfigurable hardware devices can be reconfigured using either a full or a partial reconfiguration mode. During a full reconfiguration, the complete device undergoes the reprogramming process. We surveyed different full- and partial reconfiguration models in Section 2.3.1 of the background chapter. From the three full reconfiguration models, only model **c** *free space - time sharing* matches our task model requirements. However, using our global EDF methods with the full reconfiguration mode shows some severe drawbacks that will restrict *EDF-NF* and *EDF-FkF* to very small task sets:

- According to the global EDF schedule, we have to compile a separate full configuration bit-stream for each set of running tasks before runtime. This can easily be done using standard device vendor tools, and full reconfiguration is supported by all SRAM-based reconfigurable devices. Compiling several tasks into one configuration bit-stream justifies our scalar area model, i.e., the tasks fit onto the device if $\sum_{J_i \in R} A_i \leq A(H)$. Unfortunately, we have to simulate the schedule for the complete hyper period to determine the sets of running tasks. For larger task sets, this cannot be done in reasonable time. Further, we have to store all the full configuration bit-streams during runtime. This is unacceptable especially for embedded systems with stringent memory constraints.
- In the full reconfiguration model **c**, the preemption of a specific task affects all tasks currently running, independent of their actual priority. This not only adds huge overheads to the actual tasks' execution time, but makes the analysis of the reconfiguration overheads overly pessimistic. The resulting schedulability test would be rendered useless for all except the smallest task sets.

A partial reconfiguration model allows us to reprogram only a fraction of the device during a reconfiguration step, while the other parts of the device remain computing.

From the three resource models of Section 2.3.1 which use partial reconfiguration, only the *1D partitioning model* **(e)** and the *2D partitioning* model **(f)** allow tasks with arbitrary area values.

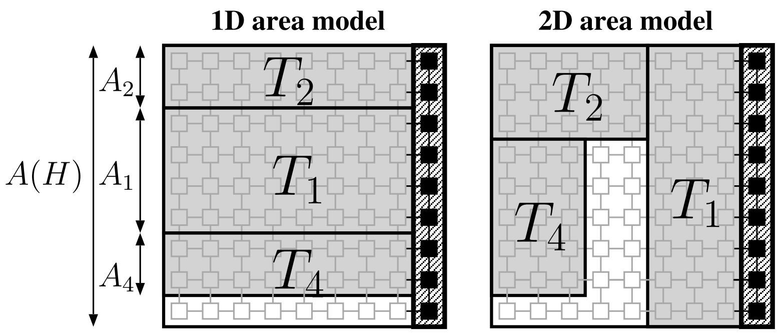


Figure 6.1: Partial reconfiguration models supporting arbitrary task areas

In the 2D area model, tasks have rectangular footprints and can be placed anywhere on the device. An example is illustrated in the right-hand side of Figure 6.1. Placing such rectangular footprints leads to fragmentation problems, i.e., a new task cannot be placed onto the device although there is sufficient free area. While 2D defragmentation techniques have been proposed [37][39], they are hard to implement and cannot completely overcome the fragmentation problem. Hence, our area constraint, $\sum_{J_i \in R} A_i \leq A(H)$, is not sufficient to guarantee feasible task placements.

In contrast, the 1D area model requires tasks to have rectangular footprints which span the entire device width, but may have arbitrary heights. An example is illustrated in the left-hand side of Figure 6.1. The task height is then proportional to the task area $A_i$. In design practice, such footprints can be obtained by applying appropriate constraints during circuit synthesis. To justify our scalar area model, we still have to ensure that the device is unfragmented at any time. For this we propose the following placement and reconfiguration scheme:

- Tasks are stacked on the device from bottom to top. The order of tasks corresponds to their scheduling priority, such that tasks with closer deadlines are placed below tasks with deadlines further away.
- Whenever a task terminates, all tasks above are shifted downwards by a relocation process. Tasks below, i.e., the ones with closer deadlines, remain unaffected.
- Whenever a task starts or resumes from preemption, it is placed according to its deadline. Specifically, all tasks with greater deadlines are shifted upwards. Again, tasks with closer deadlines remain unaffected.

Figure 6.2 shows an example of a schedule using the proposed placement and reconfiguration scheme. Whenever a task is being preempted or resumed, only those tasks with later deadlines are preempted or relocated and receive an overhead. In Figure 6.2, solid lines at the end of task executions denote terminations; dashed lines denote preemptions due to scheduling decisions. Note that task $T_4$ is preempted by the scheduler at time 7, while task $T_3$ is just relocated for two times.

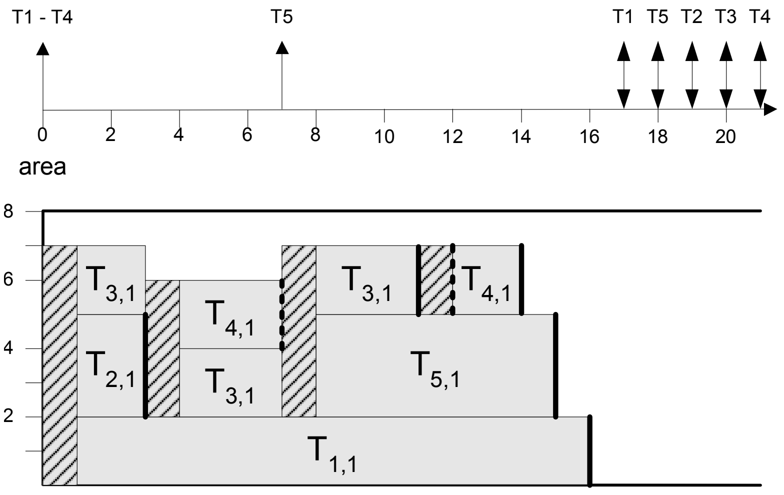


Figure 6.2: A schedule including reconfiguration times

The proposed partial reconfiguration scheme offers the following key advantages: First, we do not have to preempt all running tasks on a task preemption, which results in an improved scheduling performance. Second, the time required to reconfigure a part of the device is substantially smaller than the time required for a full reconfiguration. Thus, the reconfiguration time overhead is lowered. Third, the system has to store only one partial configuration bit-stream for each task, greatly reducing the required memory capacity. The fact that the tasks' bit-streams are independent opens up the possibility of online scheduling scenarios, where new periodic tasks enter the system at runtime and have to pass an acceptance test.

### 6.1.2 Runtime System Requirements

For the global EDF scheduling algorithms, we require a reconfigurable hardware execution model and runtime environment featuring the following characteristics:

- the device is partially reconfigurable using the 1D area model
- the tasks are relocatable
- the runtime system supports task preemption

While some of these features are research issues for themselves, their feasibility has already been demonstrated in a number of projects [102][21][97]. Among these characteristics, hardware task preemption is perhaps the least-investigated one. The context of a hardware task is given by all of its state-holding elements, i.e. registers and memory elements. Depending on the actual task instance, the state-holding elements might be distributed over its complete logic area. Related work has demonstrated the feasibility of preemptive multitasking by two different approaches:

The first approach realizes task preemptions by stopping the clock and reading back the device configuration. Appropriate tools can then be used to extract the task's state from the read-back bit-stream. The state information is used to initialize the storage elements of the task before it is resumed [61].

The second approach shifts the responsibility for saving and restoring the context to the hardware task itself. To that end, the task must be extended with an interface which allows stopping and starting the task and signaling the task to save or restore its state [Dan04b, Dan04c]. This interface together with some kind of logic that is necessary for task communication and I/O is indicated by the right-most logic cell columns in Figure 6.1.

### 6.1.3 Reconfiguration Overheads

For the following analysis, we lump together all overheads due to reconfiguration, context save and restore, and running the scheduler. This is sensible as the reconfiguration overhead will be by far the dominating part. The basic approach to account for the reconfiguration overhead is to increase the tasks' computation times by the amount of time spent in the reconfiguration process. We denote $t_{reconfig}$ as the time required to reconfigure the entire device with a new task set. In an EDF schedule, a task preemption can only occur when other tasks are released. For a periodic task set, we know in advance how many task releases can appear within a certain period of time and thus we can derive the worst-case number of preemptions $N_i$ for a task instance $T_i$. While a task instance of $T_i$ is in execution, a task $T_k$ may be released $\lceil \frac{P_i}{P_k} \rceil$ times at maximum. However, only $\lfloor \frac{P_i}{P_k} \rfloor$ releases of $T_k$ can have a higher priority and hence cause a preemption of $T_i$. Summing up, the maximal number of preemptions $N_i$ a task instance $T_i$ may experience accounts to:

$$N_i = \sum_{T_k \in \Gamma} \left\lfloor \frac{P_i}{P_k} \right\rfloor - 1 \tag{6.1}$$

Analyzing the proposed 1D placement and reconfiguration scheme, we obtain the following observations:

- An instance of a task $T_i$ is shifted upwards or preempted only when higher priority task instances are released. In the worst case, this may happen $N_i$ times.
- An instance $T_{i,j}$ of a task $T_i$ is shifted downwards or resumes from preemption only when higher priority task instances terminate. On one hand, a number of higher priority tasks will already be running when $T_{i,j}$ starts execution. These tasks might terminate and cause $T_{i,j}$ to shift downwards. In the worst-case, there might be as much as $n-1$ such tasks, $n$ being the overall number of tasks. On the other hand, we can have $N_i$ higher-priority task instances released and terminated while $T_{i,j}$ is running. Again, the terminations cause $T_{i,j}$ to shift downwards. The total number of times a task may be shifted downwards or resumes from preemption is then bounded by:

  $$M_i = N_i + (n-1) \tag{6.2}$$

  Generally, the number of tasks already running when $T_{i,j}$ starts will be much smaller than $n-1$. We can derive a better bound employing an area argument: Obviously, the tasks running at the same time as $T_i$ cannot occupy more than $A(H) - A_i$ of the device's area resource.

  $$O_i = \max_{\forall R_l \subseteq \Gamma \setminus T_i} |R_l| \quad : \sum_{T_k \in R_l} A_k \leq A(H) - A_i \tag{6.3}$$

  Thus, we can bound the number of tasks causing a shift downward or a resume by:

  $$M_i = N_i + O_i \tag{6.4}$$

The total number of times a task instance may have to be reconfigured is bounded by its upward shifts plus its downward shifts plus one for the first start. The time overhead can be accounted for by increasing the computation time of each task to:

$$\tilde{C}_i = C_i + (1 + 2 \cdot N_i + O_i) \cdot t_{reconf} \tag{6.5}$$

Note that we are using the full device reconfiguration time to derive this bound although the underlying model is based on partial reconfiguration. This is necessary as one task start or termination event can potentially lead to several task preemptions, resumes or shifts. As practical reconfigurable devices have only one reconfiguration port, the single tasks' reconfigurations have to be serialized. In the worst case, we have to pessimistically assume that the whole device undergoes reconfiguration. A more detailed analysis could try to exploit a more realistic area-proportional partial reconfiguration time.

Based on the modified task computation times of Equation 6.5, we can compute new time- and system-utilization values for the tasks:

$$\tilde{U}_i^T = \frac{\tilde{C}_i}{P_i} \tag{6.6}$$

$$\tilde{U}_i^S = \tilde{U}_i^T \cdot A_i \tag{6.7}$$

$$\tag{6.8}$$

Using these modified utilization values in our *EDF-FkF* scheduling test of Theorem 3 results in the scheduling test which considers the reconfiguration overhead:

$$\begin{aligned} &\forall T_k \in \Gamma : \\ &\tilde{U}^S(\Gamma) \leq (A(H) - A_{max}) \cdot \left(1 - \tilde{U}^T(T_k)\right) + \tilde{U}^S(T_k) \end{aligned} \tag{6.9}$$

We will statistically evaluate the performance of the *global EDF* scheduler considering reconfiguration overheads later within this chapter.

## 6.2 Partitioned EDF Overhead Model

### 6.2.1 Reconfiguration Modes and Task Placement

The partitioned scheduling approach matches perfectly the 1D partial reconfiguration model and even task relocation is not required. Like in the 1D area partial reconfiguration model used for global EDF, the tasks have rectangular footprints which span the entire device width, but may have arbitrary height. In our model, the height is proportional to the tasks area.

The device resources are statically partitioned into slots - one for each partition-block $G_k \in \chi$ determined by the partitioning algorithm. The task with the largest area requirement $A_i$ among all tasks of a partition-block defines the size of its slot. At runtime, for each partition-block a separate EDF scheduler decides which task is executed. Recall, that only one task per partition-block can be in execution at a time. For each partition-block the task selected by EDF is configured to the device area of the according slot. It is executed until it terminates or gets preempted by a higher priority task of *the same* partition-block. Using such an execution model, a task is always executed in the same position on the device area and this position is determined at design time. Hence, task relocation is not an issue.

Tasks are loaded to the slots using partial reconfiguration. However, since it may happen that more then one slot has to be reconfigured at the same time, the reconfigurations have to be serialized which increases the delay.

### 6.2.2 Runtime System Requirements

The *partitioned EDF* scheduling approach requires a runtime environment featuring the following characteristics:

- the device is partially reconfigurable using the 1D model
- the tasks do not have to be relocatable
- the runtime system supports task preemption

Such a reconfiguration model is one of the simplest, among the partial reconfiguration models (see 2.3.1). Numerous prototypes implement the partial 1D reconfiguration based on slots of static size. Since the task position is determined at design time, the approach may also deal with devices, which have a not fully homogeneous resources array.

### 6.2.3 Reconfiguration Overheads

Accounting to the reconfiguration overhead for the partitioned schedule is much easier, than it has been for the global schedule. The reason is that each partition-block represents a task system on its own. Whenever preemption occurs, only the according slot is reconfigured and the tasks executing in other slots do not get interrupted.

Let $G_l$ be the partition block which contains $T_i$. The maximum number of times an instance of task $T_i$ may become preempted is given by:

$$N_i = \sum_{T_j \in G_l} \left\lfloor \frac{P_i}{P_j} \right\rfloor - 1 \tag{6.10}$$

Equation 6.10 has the same form, as Equation 6.1 but considers only the tasks within partition-block $G_l$ rather than the tasks of the entire task set $\Gamma$. Hence, the number of preemptions $N_i$ for each task will generally be smaller than in the global scheduling case.

Now we are ready to account for the reconfiguration overhead by increasing the computation time of all tasks in $\Gamma$. We have to add $(1 + N_i)$ times the required time for the device reconfiguration[1], once for reconfiguring the task before it starts execution and $N_i$ times when it needs to resume its execution after preemption (Equation 6.11). The time utilization of the task increases according to Equation 6.12.

$$\tilde{C}_i^k = C_i^k + (1 + N_i) \cdot t_{reconf} \tag{6.11}$$

$$\tilde{U}^T(T_i^k) = U^T(T_i^k) + \frac{(1 + N_i) \cdot t_{reconf}}{P_i} \tag{6.12}$$

Obviously, a partitioning $\chi$ of a task set $\Gamma$ might be feasible to schedule on neglecting the reconfiguration overhead, but may become infeasible on considering the increased time utilization of Equation 6.12. To decide whether a feasible partitioning $\chi$ including reconfiguration overhead exists, is a difficult problem:

The optimal partitioning $\chi$ of tasks depends on the tasks' time utilization $\tilde{U}^T$, but the actual time utilization of a task depends on the partitioning, i.e., on the tasks in the same partition-block. One might think about packing rectangles that change their widths during the packing process, depending on the other rectangles currently packed to the same level. We cannot reasonably account for this cross dependency in our ILP model.

Making the simplifying assumption that the overheads due to preemption are similar for all partition blocks, the following two steps seem to be a reasonable heuristic:

1. Given the task set $\Gamma$ and device area $A(H)$, compute the partitioning $\chi$ that minimizes the maximal time utilization over all partition blocks:

$$\min\left(\max_{G_l \in \chi}\left(U^T(G_l)\right)\right) \tag{6.13}$$

2. Account for the reconfiguration overhead of each task according to Equation 6.12. If the updated time-utilization factors for each partition block $\tilde{U}^T(G_l)$ are still less or equal to one, the partitioned schedule is still feasible.

In Section 6.4 we will statistically evaluate the performance of the partitioned EDF scheduler considering reconfiguration overheads, where the partitioning is computed according to the described approach.

[1] Note, that we have to account the time $t_{reconf}$ for a full device reconfiguration, even if we assume partial reconfiguration. Although the time to reconfigure a small portion of the device will be shorter than $t_{reconf}$, in the worst case all partition blocks may request a reconfiguration simultaneously.

## 6.3 Server Based MSDL Overhead Model

### 6.3.1 Reconfiguration Modes and Task Placement

The server based *MSDL* scheduler has been developed targeting systems, which support no partial but only full device reconfiguration. Particularly, the resource model **(c)** of Section 2.3.1 which we called *free space partitioning - time sharing* is used. For a set $\Gamma$ of $n$ tasks, the *MSDL* algorithm creates at maximum $n$ servers $S_1, S_2, \ldots$. Each server $S_i$ represents a subset $V_i \subseteq \Gamma$ of the original tasks, and whenever the server executes, all of its represented tasks can execute. Hence, each server becomes one RHD configuration, i.e. one device bit-stream. The configurations are loaded sequentially onto the RHD. Which configuration is loaded is determined by a sequential EDF schedule of the according servers. Whenever a server $S_i$ gets preempted by a higher priority server $S_j$ all tasks of $V_i$ become interrupted. Some tasks of $S_i$ may also be included in server $S_j$, and hence they are only interrupted for the time the device is being reconfigured. The other tasks are preempted and will be resumed later on when $S_i$, or another server including these tasks, is scheduled for execution again. However, even if a task $T_k$ is included in $S_i$ and $S_j$ its state has to be stored before and restored after the reconfiguration process, since the circuit-elements of $T_k$ might be placed and routed at completely different locations on the device within the bit-streams of $S_i$ and $S_j$ (see Figure 2.5).

### 6.3.2 Runtime System Requirements

The *server based MSDL* scheduling approach requires a runtime environment featuring the following characteristics:

- the device supports full reconfiguration during runtime (partial reconfiguration is not required)
- no task relocation during runtime is required
- the runtime system supports task preemption

### 6.3.3 Reconfiguration Overheads

Reconfiguration overhead in an *MSDL* schedule occurs due to the start or resumption of the servers. Since the periodic servers of $\Omega$ are scheduled by the *EDF* exactly like periodic tasks, we know that the maximum number of times a server $S_i$ gets preempted by a higher priority server is given by Equation 6.14, where $P_i, P_j$ denotes the periods of the servers $S_i, S_j$ and not the periods of the original tasks $T_i, T_j$.

$$N_i = \sum_{S_j \in \Omega} \left\lfloor \frac{P_i}{P_j} \right\rfloor - 1 \tag{6.14}$$

Hence, we can account for the time overhead due to device reconfiguration by increasing the computation time of each server. Each server gets reconfigured one time when it starts execution and may be reconfigured $N_i$ times, when it resumes from preemption.

$$\tilde{C}_i = C_i + (1 + N_i) \cdot t_{reconf} \tag{6.15}$$

After increasing the server computation time, we can update the time-utilization of the server set $\Omega$.

$$\tilde{U}^T(\Omega) = \sum_{S_i \in \Omega} \frac{\tilde{C}_i}{P_i} \tag{6.16}$$

The time-utilization $\tilde{U}^T(\Omega)$ includes the worst case time-overhead that can occur due to device reconfiguration. Hence, a scheduling test for an *MSDL* generated server set which accounts for the reconfiguration overhead is given by:

$$\tilde{U}^T(\Omega) \leq 1 \tag{6.17}$$

## 6.4 Performance Evaluation Including Overhead

In this section, we evaluate the influence of the reconfiguration overhead on the scheduling performance of the proposed algorithms. Therefore, we simulated all algorithms on our standard benchmark $BM_{std}$, but this time including the reconfiguration overhead.[2] The reconfiguration time has been varied between $0.1, 1, 10$ time-units, which corresponds to 1%, 10% and 100% of the computation time of the shortest task.

### 6.4.1 Global EDF Overhead Evaluation

First, we have applied our new EDF schedulability test including reconfiguration overheads of Equation 6.9 to our standard benchmark $BM_{std}$. Figure 6.3-left shows the success rates for the original scheduling test without reconfiguration overheads and with device reconfiguration time varied between $0.1, 1, 10$ time-units, which corresponds to 1%, 10% and 100% of the computation time of the shortest task. The results show that the effect of device reconfiguration can almost be neglected, when the reconfiguration time is about 1% of the runtime of the shortest task. Even with a reconfiguration time in the range of 10% of the runtime of the shortest task, a considerable number of task sets could be guaranteed. Only when the reconfiguration time increases further, a stronger performance loss can be observed.

The same evaluation has been done on a second benchmark called $BM_{std2}$. It has been generated using the same method than for $BM_{std}$, but the time-utilization factors

[2]The computation times and periods have been scaled by a factor of 10 to provide a higher resolution for scheduling simulations.

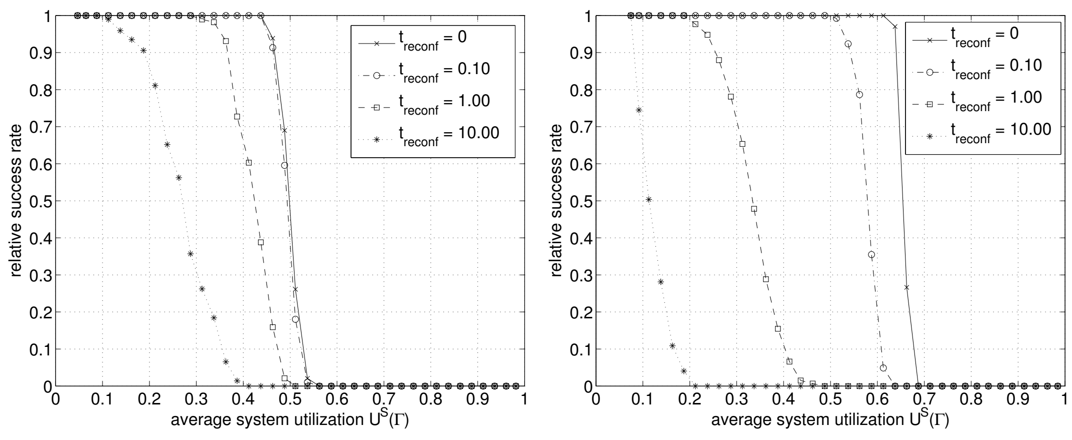


Figure 6.3: Performance of the EDF-FkF scheduling test including reconfiguration overheads on the benchmark $BM_{std}$ (left) and $BM_{std2}$ (right)

$U^T(T_i)$ and areas $A_i$ have been chosen from the interval $[0.01, 0.2]$. Hence, the tasks in $BM_{std2}$ could be denoted *light-weight*. On the other hand, the number of tasks per task set increased to about 100. Figure 6.3-right shows the results. For small tasks with small time-utilization, our scheduling condition is less pessimistic and accepts task sets of 65% system utilization and more. However, the more tasks we have in the task set, the more preemptions can occur and the higher the reconfiguration overheads are. For benchmark $BM_{std2}$, a device reconfiguration time of 1% of the shortest task runtime already reduces the amount of accepted task sets considerably.

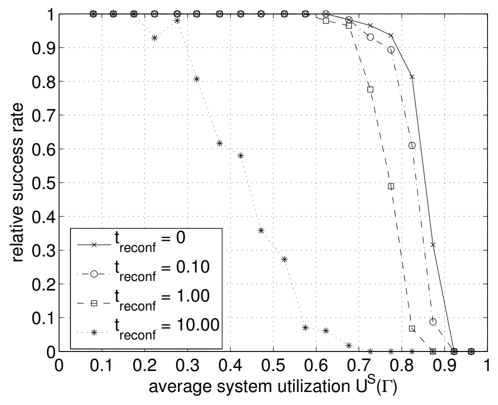


Figure 6.4: Performance of EDF-NF obtained by exact simulation, including reconfiguration overheads ($BM_{std}$)

Figure 6.4 shows the scheduling performance for the *simulated EDF-NF* scheduler on benchmark $BM_{std}$ including reconfiguration overheads. Again, the device reconfiguration time $t_{reconfig}$ has been varied between 1%, 10% and 100% of the computation time of the shortest task. The simulated execution of the *EDF-NF* scheduler does not suffer from the pessimism of our scheduling test. Therefore, we get reasonably small losses in performance for reconfiguration times of values one and more orders of magnitude smaller than the smallest task computation time. For reconfiguration times in the same range as the task computation times, the performance drop is again quite dramatic.

### 6.4.2 Partitioned EDF Overhead Evaluation

To evaluate the overhead in case of partitioned scheduling methods, we applied the *opt.-part.-EDF* algorithm again on our standard benchmark ($BM_{std}$). The feasibility of a task set including reconfiguration overheads was determined according to the heuristic method introduced in 6.2. The results are shown in Figure 6.5 with a device reconfiguration time varied between $0.1, 1, 10$ time-units, which corresponds to 1%, 10% and 100% of the computation time of the shortest task.

The results show that the effect of device reconfiguration can almost be neglected, when the runtime of the shortest task is about one magnitude higher than the device reconfiguration time. Even at reconfiguration times in the range of the runtime of the shortest task, a considerable number of task sets could be feasibly scheduled.

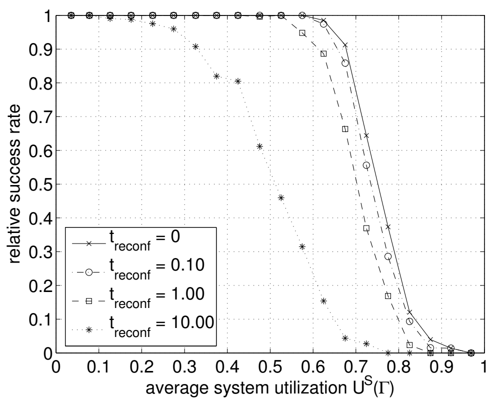


Figure 6.5: Performance of opt.-partitioned-EDF including reconfiguration overheads ($BM_{std}$)

### 6.4.3 Server Based MSDL Overhead Evaluation

Also the impact of reconfiguration overheads on the *MSDL* scheduling method has been evaluated on the $BM_{std}$ benchmark. The results in Figure 6.6 show that a reconfiguration time of 0.1, which corresponds to 1% of the computation time of the shortest task, decreases the performance of *MSDL* almost negligibly. For a reconfiguration time of 1.0, the performance drop becomes considerable, but even for a reconfiguration time of 10, a considerable amount of task sets can be feasibly scheduled.

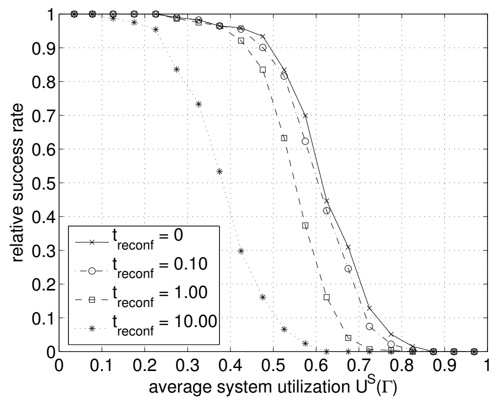


Figure 6.6: Performance of MSDL including reconfiguration overheads ($BM_{std}$)

## 6.5 Conclusion on Execution Models and Overhead

For the three proposed scheduling approaches, this chapter discussed the runtime requirements, modeled the reconfiguration overhead, included it within the schedulability tests and finally evaluated the impact of the overhead in simulation experiments.

**Runtime Requirements:** We discussed the execution models and the according overhead for all of our proposed scheduling algorithms. We showed how the schedule obtained by a certain algorithm can actually be executed on a reconfigurable device and what are the resulting requirements to the runtime-system. While all scheduling approaches create preemptive schedules and hence require a mechanism to store and recover the state of tasks, the requirements concerning the reconfiguration model are different. The *global EDF* schedulers have the hardest requirements - the runtime system has to support 1D partial reconfiguration and task relocation. This makes the implementation rather difficult, and requires target architectures with a homogeneous arrangement of resources. The *partitioned EDF* schedulers require 1D partial reconfiguration as well, but no task relocation, since a task will always have the same place on the device area. A static partitioning of the device area into slots is suitable - which matches perfectly the XILINX supported partial design flow for Virtex FPGAs [71]. Moreover, the simplicity of the model suggests that it can be implemented on most devices that support partial reconfiguration in any form. The *server based MSDL* scheduler has the least requirements to the reconfiguration model. When switching from one server to another, the entire device area is reconfigured. Hence, this scheduler can be used with any device that supports runtime reconfiguration.

**Reconfiguration Overheads:** Concerning the time overheads due to the device reconfigurations at runtime, we have derived extended scheduling conditions for all our scheduling approaches. The basic approach was, to derive an upper bound on the number of preemptions or relocations a task or server may undergo in the worst case. Doing so, we could account for the overhead by increasing the computation time with a time penalty for each preemption or relocation. The scheduling condition for each algorithm could be used with the modified computation time, and hence take the configuration overheads into consideration. Partial device reconfiguration is usually faster then configuring the entire device. Up to now we failed to take advantage from this fact, since in the worst case all tasks have to be configured onto the device at the same time. However, we can expect that the overall reconfiguration time penalty can be reduced by a deeper analysis.

Our performance evaluation shows that the reconfiguration time has considerably more impact on the *global EDF* schedulers, than on the *partitioned EDF* or the *server based MSDL* schedulers. One reason is, that in a *global EDF* schedule each preemption may not only effect the preempted task but also cause a relocation of other running tasks. In a *partitioned EDF* schedule, only tasks within the same partition can preempt each other. Moreover, a preemption within one partition does not disturb the execution of the tasks in other partitions. In a *MSDL* schedule, the maximum number of preemptions depends on the number of servers, which is usually smaller then the number of tasks.

However, for all schedulers the reconfiguration overheads increase more or less with the number of tasks. Partly, a drop in performance can be compensated. Larger task sets are easier to schedule, at least for *global EDF* and *partitioned EDF*, since they contain smaller tasks which leads to less resource fragmentation.

In conclusion, the evaluation on our standard benchmark shows that the performance drop due to the reconfiguration overhead is acceptable, when the task computation time is at least one order of magnitude higher than the device reconfiguration time.

CHAPTER 7

# Model Extensions

Within this chapter we will present two important extensions to our simple task model of Chapter 3. By capturing more of the typical characteristics of hardware-tasks, we aim to improve the resource management of the reconfigurable system.

The first model extension allows specifying not only one but several implementation variants per task – each with different area and computation time. These variants can be considered in a task schedule and our results show a significant improvement in scheduling performance. These results have been also presented in [DP06b].

The second model extension captures the memory requirements of application tasks. When tasks require a considerable amount of data memory, they will typically use dedicated RAM which can be either block RAM inside the reconfigurable device or external RAM banks. We will show that sharing such RAM banks among tasks without jeopardizing timing constraints is feasible, reduces the overall memory requirements and can even be used for inter task communication. The main results of this section have been also presented in [DP05b].

## 7.1 Periodic Tasks with Variants

One, if not the fundamental characteristic of reconfigurable hardware, is that a certain computational job can be implemented in many alternative ways, exploring the tradeoffs between fast but resource-consuming and slower but resource-saving versions. Even third party IP-cores can often be obtained as several variants, differing in area and speed, as Example 1 of the DCT computation in Section 2.2 has shown. Assume a system consisting of several periodic real-time tasks has to be realized, and for all or some of the tasks we have alternative implementations to choose from. This raises a question, which implementation variant to choose for which task. We may gain an advantage by selecting particular variants, since we may be able to greatly reduce the resource fragmentation and hence increase the scheduling performance.

First we extend our task model in such a way that makes us able to deal with implementation variants. Afterwards, we present for the *partitioned EDF* scheduler a detailed method to obtain the optimal task variant selection. It follows a performance evaluation which demonstrates the performance increase compared to task sets scheduled without considering variants (Section 7.1.4). For the *global EDF* and the *server bases* scheduler, we have not yet developed methods to deal with task variants. Hence, we will discuss only rough approaches at the end of this section.

### 7.1.1 Variant-Rich Tasks

We consider the scheduling of a set of $n$ *variant-rich* periodic tasks $\digamma = \{\Gamma_1, \Gamma_2, \ldots \Gamma_n\}$ onto a reconfigurable device $H$. Each *variant-rich* periodic task $\Gamma_i \in \digamma$ refers to some computation which has to be performed periodically. The instances $\Gamma_{i,j}$ of task $\Gamma_i$ are released with period $P_i$. Each task instance is associated with an absolute deadline $d_{i,j}$. We assume that the relative deadline of $\Gamma_i$ equals its period.

Each *variant-rich* task $\Gamma_i$ exists in one or more implementation variants $\Gamma_i = \{T_i^1, T_i^2, \ldots\}$. Each variant $T_i^k \in \Gamma_i$ is specified by a worst-case computation time $C_i^k$ and the amount of required reconfigurable logic resources $A_i^k$. Table 7.1 shows an example task set with four periodic tasks $\digamma^* = \{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4\}$. The first two tasks have two implementation variants.

| $\Gamma_i$ | $T_i^k$ | $P_i$ | $C_i^k$ | $A_i^k$ | $U^T(T_i^k)$ | $U^S(T_i^k)$ |
|---|---|---|---|---|---|---|
| $\Gamma_1$ | $T_1^1$ | 12 | 3 | 6 | 1/4 | 3/2 |
| | $T_1^2$ | | 6 | 3 | 1/2 | 3/2 |
| $\Gamma_2$ | $T_2^1$ | 4 | 2 | 4 | 1/2 | 2 |
| | $T_2^2$ | | 1 | 8 | 1/4 | 2 |
| $\Gamma_3$ | $T_3^1$ | 6 | 5 | 3 | 5/6 | 5/2 |
| $\Gamma_4$ | $T_4^1$ | 12 | 2 | 2 | 1/6 | 1/3 |

Table 7.1: Example of a *variant-rich* task set $\digamma^*$

The notations used for *variant-rich* periodic real-time tasks and their instances are summarized in the following list:

| | |
|---|---|
| $\digamma$ | denotes a set of periodic or sporadic *variant-rich* tasks |
| $\Gamma_i$ | denotes a generic *variant-rich* periodic task |
| $\Gamma_{i,j}$ | denotes the $j$the instance of *variant-rich* task $\Gamma_i$ |
| $P_i$ | denotes the period of $\Gamma_i$ |
| $D_i$ | denotes the relative deadline of $\Gamma_i$ |
| $T_i^k$ | denotes the $k$th implementation variant of $\Gamma_i$ |
| $C_i^k$ | denotes the computation time of variant $T_i^k$ |
| $A_i^k$ | denotes the area requirement of variant $T_i^k$ |

$r_{i,j}$ denotes the absolute release or arrival time of $\Gamma_{i,j}$
$d_{i,j}$ denotes the absolute deadline of $\Gamma_{i,j}$
$s_{i,j}$ denotes the start time, when $\Gamma_{i,j}$ begins its execution
$f_{i,j}$ denotes the finishing time of $\Gamma_{i,j}$

### Feasible Schedule of Variant-Rich Tasks

In order to perform the requested execution of a task instance $\Gamma_{i,j}$, one of the task's implementation variants $T_i^k \in \Gamma_i$ has to be loaded (configured) onto the device and executed for $C_i^k$ time units. During this execution, an amount $A_i^k$ of the device area is occupied.

Generally speaking, we would like to schedule and execute a set of periodic tasks on the reconfigurable device such that each task instance meets its deadline. Let $\Gamma$ denote the unified set of all implementation variants of all *variant-rich* tasks.

$$\Gamma = \{\Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_n\} \tag{7.1}$$

A schedule for a set of periodic variant-rich tasks $\digamma$ is given by a function $R : \mathbb{R}^+ \to \mathbf{P}(\Gamma)$. $R(t)$ denotes the set of task implementations from $\Gamma$ running at time $t$. A schedule is called *feasible*, if for each request $r_{i,j}$ of task $\Gamma_i$ one of its implementations $T_i^k \in \Gamma_i$ is scheduled for execution for at least $C_i^k$ time units within the interval given by the release time and deadline of $\Gamma_i$.

We consider only schedules $R$, where the selection of variants is static. Hence, the same variant $T_i^k$ of a task $\Gamma_i$ is used for the entire schedule. We express a selection by the function $f(i)$, that selects exactly one implementation variant $T_i^{f(i)}$ for each task $\Gamma_i$. The set of all selected implementations is denoted by $\Gamma_{|f} = \{T_1^{f(1)}, T_2^{f(2)}, \ldots\}$.

For example, Figure 7.1 displays a possible schedule for the variant-rich task set shown in Table 7.1 on a device with an area of 10 units. The upper part of Figure 7.1 indicates the release times and deadlines of the tasks. The lower part illustrates which implementation variants of the tasks are executed, and how they share the device area over time. In this example schedule always the first variant of each task is chosen. It occurs one task preemption at time 4, which is denoted by a dotted line. The schedule shown can easily be proven feasible, because every task instance meets its deadline for the entire *hyper-period* of the task set (which amounts to 12 time units). The hyper-period is the least common multiple of all task periods. Recall that a feasible schedule defined over the hyperperiod can be repeated an infinite number of times.

### Utilization Metrics for Variant-Rich Tasks

We have already introduced the time-utilization $U^T$ and the system-utilization $U^S$ for tasks and task sets of our basic model (without variants). In our extended model (with variants), the metrics are defined for implementation variants $T_i^k$ of a variant-rich task $\Gamma_i$ and not for the task itself.

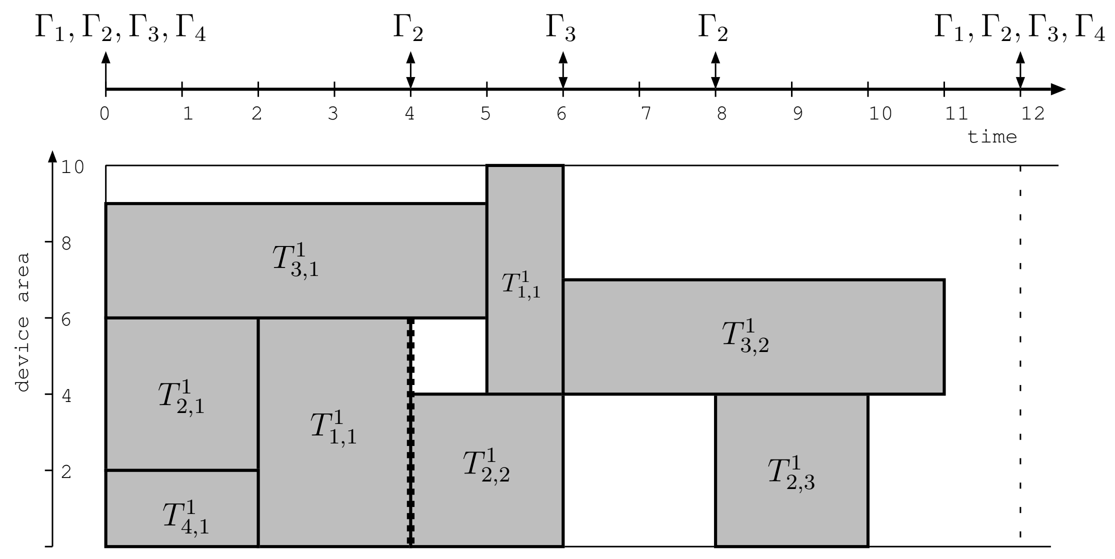


Figure 7.1: Preemptive global schedule of variant-rich tasks

The time-utilization given by

$$U^T(T_i^k) = \frac{C_i^k}{P_i} \tag{7.2}$$

is the fraction of time a certain task implementation $T_i^k$ occupies the device in order to complete its execution. The cumulative time-utilization for a set of task-variants $\tilde{\Gamma} \subseteq \Gamma$ is defined as

$$U^T(\tilde{\Gamma}) = \sum_{T_i^k \in \tilde{\Gamma}} U^T(T_i^k). \tag{7.3}$$

Obviously, a set of task implementations $\tilde{\Gamma}$ cannot be executed sequentially if $U^T(\tilde{\Gamma}) > 1$. In the same way we define the system-utilization metric, which captures the degree by which the device is utilized by $T_i^k$:

$$U^S(T_i^k) = U^T(T_i^k) \cdot A_i^k \tag{7.4}$$

In order to measure the load generated by a variant-rich task set $\mathcal{F}$, we now define the total system-utilization of $\mathcal{F}$ by summing up the system utilization factors of all variant-rich tasks. If a task has several alternative implementations, we account for the one with the minimal system-utilization:

$$U^S(\mathcal{F}) = \sum_{\Gamma_i \in \mathcal{F}} \min_{T_i^k \in \Gamma_i} \left( U^S(T_i^k) \right) \tag{7.5}$$

Thus, $U^S(\mathcal{F})$ is the minimum amount of combined area and time required by the variant-rich task set. If $U^S(\mathcal{F}) > A(H) \cdot 1$, no feasible schedule exists since the system is utilized by more than 100%. We use the $U^S$ metric in our simulation study, in order to rate the performance of the proposed scheduling algorithm.

**Differences between the model with- and without implementation variants**

What has been a task $T_i$ in our old model, is now represented by a variant-rich task $\Gamma_i = \{T_i^1, T_i^2, \ldots, T_i^m\}$. Each implementation variant of the new model $T_i^k$ has the same structure as a task of the old model (period, deadline, computation time, area) - that is why we use the same symbol $T$.

What has been a task set $\Gamma$ in our old model, is now represented by a set of variant-rich tasks $\digamma = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_n\}$. Since a variant-rich task $\Gamma_i$ of the new model has the same structure as a periodic task set in our old model, we use the same symbol $\Gamma$.

### 7.1.2 Partitioned Scheduling of Variant-Rich Tasks

We can apply the concept of partitioned scheduling straight forward to our generalized task model with implementation variants.

**Definition 12** (partitioned schedule of variant-rich task sets)**.** *A schedule $R$ for a periodic variant-rich task set $\digamma$ is said to be* partitioned by $\chi$ *with selection $f$, if the following three statements hold:*

1. *$f(i)$ is a function that selects exactly one implementation variant $T_i^{f(i)}$ for each task $\Gamma_i$. The set of all selected implementations is denoted by $\Gamma_{|f} = \{T_1^{f(1)}, T_2^{f(2)}, \ldots\}$.*
2. *$\chi = \{G_1, G_2, \ldots, G_m\}$ is a* partitioning *of $\Gamma_{|f}$. That is, $\chi$ is a set of disjoint subsets of $\Gamma_{|f}$, called* partition blocks*, such that the union of all partition blocks results in $\Gamma_{|f}$.*
3. *At any point in time, at most one task implementation of each partition block is executed on the reconfigurable device.*

Each partition block is exclusively assigned a certain area of the reconfigurable device. The area requirement of a partition block $A(G_j)$ is determined by its *largest* task; the required overall device area $A(\chi)$ is given by the accumulated area over all partitions:

$$A(G_j) = \max_{T_i^k \in G_j} (A_i^k), \quad A(\chi) = \sum_{G_j \in \chi} A(G_j) \tag{7.6}$$

Since each partition block presents an independent task system of its own, we can easily formulate a test to figure out whether a given partitioned schedule is feasible or not by applying the basic results from single-processor EDF theory:

**Lemma 6** (EDF feasibly partitioned schedule of variant-rich tasks)**.** *Given a periodic variant-rich task set $\digamma$, let $f$ be a selection function and $R$ be a schedule of the selected task variants $\Gamma_{|f}$ with partitioning $\chi$, such that the tasks of each partition $G_i \in \chi$ are scheduled separately by EDF. The schedule is feasible on the reconfigurable device $H$ if:*

- *$A(\chi) \leq A(H)$, i.e., all partitions fit onto the device, and*
- *$\forall G_i \in \chi : U^T(G_i) \leq 1$, i.e., all partitions have a time utilization of less than 100%.*

*Proof.* The tasks implementations within each partition-block can be separately scheduled by EDF, such as in a uniprocessor system. Since the EDF uniprocessor utilization bound holds for all partitions, all deadlines are met. □

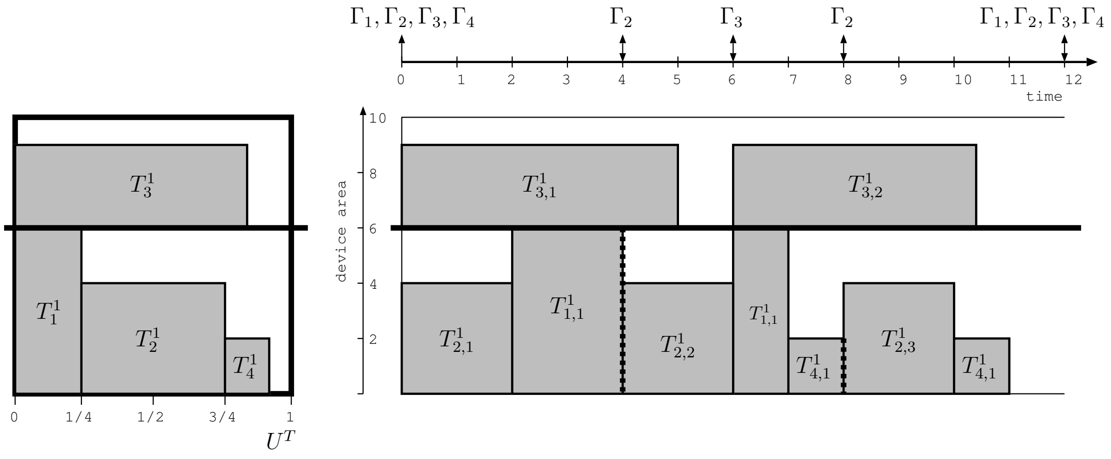


Figure 7.2: Example of a partitioned EDF schedule of a variant-rich task set

Figure 7.2 illustrates an EDF partitioned schedule on the example of the task set $\Gamma^*$ shown in Table 7.1. In this example, the first variant for each task has been selected, i.e., $\Gamma^*_{|f} = \{T_1^1, T_2^1, T_3^1, T_4^1\}$, and two partition-blocks have been generated, e.g., $\chi = \{G_1 = \{T_1^1, T_3^1, T_4^1\}, G_2 = \{T_2^1\}\}$. The left-hand side of Figure 7.2 presents the partitioning $\chi$ and illustrates the division of the device area. The according partitioned EDF schedule $R$ is shown on the right-hand side of Figure 7.2. Since both partition blocks have a time utilization of less than 100%, the schedule is feasible:

$$U^T(G_1) = 1/2 + 1/4 + 1/6 \leq 1, \quad U^T(G_2) = 5/6 \leq 1$$

Based on Lemma 6 we are able to define an optimization problem in order to answer our scheduling questions from Section 3.2. To this end, we formulate the problem of finding the least resource-consuming *EDF Feasibly Partitioned Schedule (EFPS)* for a variant-rich task set:

**Definition 13** (EFPS of variant-rich tasks problem)**.** *Given a periodic variant-rich task set $\digamma$, find:*

- *a function $f$ that selects a variant for each task and*
- *a partitioning $\chi$ of the selected variants $\Gamma_{|f}$,*

- *subject to* $U^T(G_i) \leq 1 : \forall G_i \in \chi$,
- *minimize* $A(\chi)$.

Solving the *EFPS of variant-rich tasks* problem answers directly our formulated scheduling questions: The smallest device that allows to feasibly execute $\mathcal{F}$ is of size $A(H) := A(\chi)$. The according decision problem is also solved. Given $\mathcal{F}$ and a device $H$, $\mathcal{F}$ can be feasibly scheduled if $A(H) \geq A(\chi)$. From the resulting partitioning $\chi$, we can easily derive the scheduling function $R$.

This problem can be optimally solved using an integer linear programming approach, which we will present in the next section. The development and evaluation of further heuristics is omitted at this place and left for future work.

### 7.1.3 Optimal Partitioning by ILP

In this section we develop an extension of the binary ILP model for the 2-dimensional level strip-packing problem [73], which we used to compute partitioning for tasks without variants in Section 4.2. Our extension is able to deal with implementation variants for tasks. Instead of modeling each task by a single rectangle, we apply the ILP model on the *set of rectangles* corresponding to all task variants. The ILP constraints are modified, such that one and only one variant of each task is packed.

According to the model of [73], we make the following basic assumptions:

**Order of task variants:** We sort and renumber the set of all variants $\Gamma$, according to non-increasing area of the task variants. Let $\bar{\Gamma}$ denote the sorted set:

$$\Gamma = \{T_1^1, T_1^2, T_1^3, \ldots, T_2^1, T_2^2, \ldots, T_n^1, \ldots\} \tag{7.7}$$

$$\bar{\Gamma} = \{\bar{T}_1, \bar{T}_2, \bar{T}_3, \ldots\}, \tag{7.8}$$

$$A(\bar{T}_j) \geq A(\bar{T}_{j+1}), j = 1, \ldots, |\bar{\Gamma}| - 1$$

The index function $\delta(i,k)$ maps the index of a variant $T_i^k$ in $\Gamma$ to the according task variant in $\bar{\Gamma}$, i.e., $\bar{T}_{\delta(i,k)} = T_i^k$.

**Order of partition blocks:** The task implementation $\bar{T}_l$ having the largest area within a partition block is said to *initialize* the partition block. The index $l$ is also used to index the partition block, i.e., if the largest variant in a partition block is $\bar{T}_l$ the partition block is denoted as $G_l$.

It follows, that a partitioning $\chi$ has $m = |\Gamma|$ potential partition blocks $G_1 \ldots G_m$ (some may be empty), and that the partition blocks are sorted by non-increasing areas, i.e., $A(G_l) \geq A(G_{l+1})$. Further, the area of a partition block is equal to the area of the task variant that initializes the partition block, i.e., $A(G_l) = A(\bar{T}_l)$.

Considering an arbitrary solution to our formulated *EFPS of variant-rich tasks* problem, we can now, based on the assumptions made above, differentiate three cases for each task implementation variant $\bar{T}_j \in \bar{\Gamma}$:

- $\bar{T}_j$ is not packed in any $G_l$ since another implementation variant of the task is selected and packed.
- $\bar{T}_j$ initializes a partition block, i.e., $T_j \in G_j$.
- $\bar{T}_j$ is packed into a partition block with greater area, forcing the partition block index to be smaller than $j$.

The assumptions made are without loss of generality but reduce the search space considerably, since a task implementation $T_i^k$ cannot be assigned to an arbitrary partition block $G_l$. When modeling the ILP, we use the binary decision variable $x_{l,j}$ to indicate that the task variant $\bar{T}_j$ is packed into partition block $G_l$:

$$x_{l,j} = \begin{cases} 1 & \text{if } \bar{T}_j \in G_l \\ 0 & \text{if } \bar{T}_j \notin G_l \end{cases} \quad \text{with } l = \{1, \dots, |\Gamma|\}, j = \{1, \dots, |\Gamma| : j \geq l\} \tag{7.9}$$

If $x_{l,j} = 1$ and $j = l$, the partition block $G_l$ is said to be *initialized* and its largest task variant is $\bar{T}_l$. If $x_{l,l} = 0$, $G_l$ is uninitialized and empty. The resulting ILP has three main components:

1. The cost function accumulates the area required by all non-empty partition blocks, hence minimizing the overall area $A(\chi)$.
2. A set of constraints ensures that for each variant-rich task $\Gamma_i$ exactly one variant $T_i^k$ is packed into one of the partition blocks. For the variant-rich task $\Gamma_i$ with $m$ variants this can be expressed by the term:

$$\left(\sum_{l=1}^{\delta(i,1)} x_{l,\delta(i,1)}\right) + \left(\sum_{l=1}^{\delta(i,2)} x_{l,\delta(i,2)}\right) + \cdots \\ + \left(\sum_{l=1}^{\delta(i,k)} x_{l,\delta(i,k)}\right) + \cdots + \left(\sum_{l=1}^{\delta(i,m)} x_{l,\delta(i,m)}\right) = 1 \tag{7.10}$$

   The first term of Equation 7.10 equals one, if $T_i^1$ is packed into some partition block, the second term checks whether variant $T_i^2$ is packed into some partition block, and so on. Note, that by convention a variant $T_i^k$ can only be packed into $G_1, \dots, G_{\delta(i,k)}$. Therefore, the upper bounds of the sum indices are given by $\delta(i,k)$.

3. Another set of constraints enforces EDF schedulability. Each non-empty partition block must have a time utilization less or equal than 1, i.e., $U^T(G_l) \leq 1$.

The final binary ILP can be written as:

$$\boxed{\begin{aligned} &\min \sum_{l=1}^{|\Gamma|} A(\bar{T}_l) \cdot x_{l,l} \\ &\sum_{k=1}^{|\Gamma_i|} \left( \sum_{l=1}^{\delta(i,k)} x_{l,\delta(i,k)} \right) = 1, \qquad i \in \{1,\ldots,n\} \\ &\sum_{j=l}^{n} U^T(\bar{T}_j) \cdot x_{l,j} \leq 1 \cdot x_{l,l}, \qquad l \in \{1,\ldots,|\Gamma|\} \end{aligned}} \tag{7.11}$$

Let $n$ be the number of tasks in $\mathcal{F}$ and $|\Gamma|$ be the cumulative number of variants over all tasks. The corresponding ILP contains $|\Gamma| \cdot (|\Gamma|+1)/2$ binary variables and $n + |\Gamma|$ constraints.

## Example illustrating the binary ILP with task variants

To illustrate the ILP model, we consider again the example task set in Table 7.2.

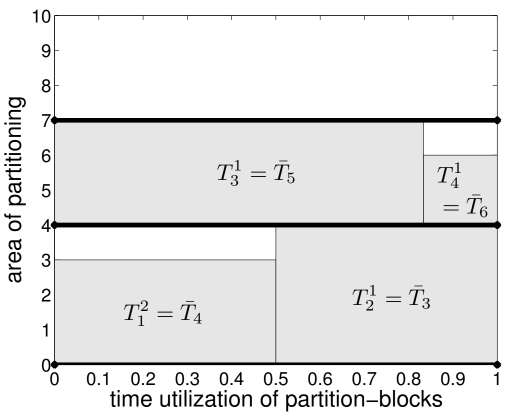
| $\Gamma_i$ | $T_i^k$ | $P_i$ | $C_i^k$ | $A_i^k$ | $U^T(T_i^k)$ | $U^S(T_i^k)$ |
|---|---|---|---|---|---|---|
| $\Gamma_1$ | $T_1^1$ | 12 | 3 | 6 | 1/4 | 3/2 |
| | $T_1^2$ | | 6 | 3 | 1/2 | 3/2 |
| $\Gamma_2$ | $T_2^1$ | 4 | 2 | 4 | 1/2 | 2 |
| | $T_2^2$ | | 1 | 8 | 1/4 | 2 |
| $\Gamma_3$ | $T_3^1$ | 6 | 5 | 3 | 5/6 | 5/2 |
| $\Gamma_4$ | $T_4^1$ | 12 | 2 | 2 | 1/6 | 1/3 |

Table 7.2: Example of a *variant-rich* task set $\mathcal{F}^*$

The four tasks of $\mathcal{F}^*$ have a total of six task implementations (the set $\Gamma$) which get sorted and re-numbered according to non-increasing areas. The resulting set $\bar{\Gamma}$ is given by:

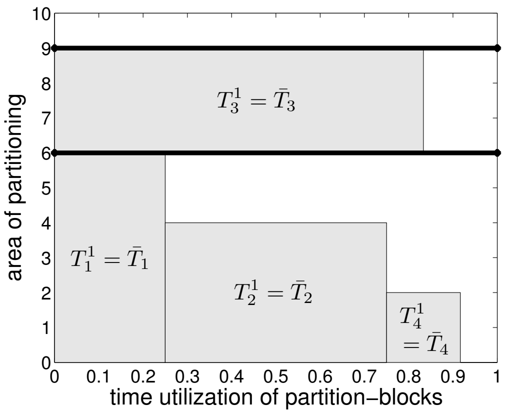
$$\bar{\Gamma} = \{\bar{T}_1 := T_2^2, \bar{T}_2 := T_1^1, \bar{T}_3 := T_2^1, \bar{T}_4 := T_1^2, \bar{T}_5 := T_3^1, \bar{T}_6 := T_4^1\} \tag{7.12}$$

Figure 7.3 shows the optimal partitioning $\chi$, which consists of the two partition blocks $G_3 = \{T_1^2, T_2^1\}$ and $G_5 = \{T_3^1, T_4^1\}$. Since $|\Gamma| = 6$, the ILP model contains $6 \cdot (6+1)/2 = 21$ binary variables. The configuration of the decision variables $x_{l,j}$ is shown in the table beside the figure. The required overall device area is 7 units. For comparison: If each task comes only in one variant (the first one), the optimal partitioning would require 9 units of device area. Figure 7.4 shows the optimal partitioning and the according decision variable configuration, in case only the first variant of each task is considered.



| $l$ | $x_{l,1}$ | $x_{l,2}$ | $x_{l,3}$ | $x_{l,4}$ | $x_{l,5}$ | $x_{l,6}$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | - | 0 | 0 | 0 | 0 | 0 |
| 3 | - | - | 1 | 1 | 0 | 0 |
| 4 | - | - | - | 0 | 0 | 0 |
| 5 | - | - | - | - | 1 | 1 |
| 6 | - | - | - | - | - | 0 |

Figure 7.3: Optimal partitioning of $\Gamma^*$ and the according decision variable configuration



| $l$ | $x_{l,1}$ | $x_{l,2}$ | $x_{l,3}$ | $x_{l,4}$ | $x_{l,5}$ | $x_{l,6}$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 2 | - | 0 | 0 | 0 | 0 | 0 |
| 3 | - | - | 1 | 0 | 0 | 0 |
| 4 | - | - | - | 0 | 0 | 0 |
| 5 | - | - | - | - | 0 | 0 |
| 6 | - | - | - | - | - | 0 |

Figure 7.4: Sub-optimal partitioning of $\Gamma^*$, considering only the first variant of each task.

### 7.1.4 Performance Evaluation for Variant-Rich Tasks

Introducing implementation variants for tasks makes it more likely, that an *EDF feasible partitioned schedule* exists for a reconfigurable device of given size. However, finding such a schedule becomes computationally more complex since the design space grows rapidly with the number of implementation variants. In order to evaluate the benefit we gain by introducing implementation variants we present a simulation study.

To create a benchmark of task sets without and with variants, we used the following procedure: First, we generated periodic task sets $\Gamma$ (with only one variant per task) with various system utilization by composing randomly generated task implementations according to our $BM_{std}$ parameters. For each task $\Gamma_1 = \{T_i^1\}$, the area $A_i^1$ was chosen according to a uniform distribution in $[0.1, 0.5]$ and the computation time and period were chosen[1] such that $U^T(T_i^1)$ is uniformly distributed in $[0.1, 0.5]$. Such a setup creates task sets of up to 15 tasks. Based on these task sets, we created task sets with variants using the following methods:

**3 variants per task:** To the existing variant $T_i^1$ of each task, we added one variant $T_i^2$ with half the computation time but double the area requirement, i.e., $T_i^2 = (P_i^1, \frac{1}{2} \cdot C_i^1, 2 \cdot A_i^1)$, and another variant with doubled computation time but only half the area requirement, i.e., $T_i^3 = (P_i^1, 2 \cdot C_i^1, \frac{1}{2} \cdot A_i^1)$.

**1 to 5 variants per task:** For each task the number of variants was randomly chosen from 1 to 5. Each variant $T_i^k$ was created choosing a form factor $q$ from the continuous interval $[1, 4]$ and a sign $s$ from $\{-1, +1\}$. The computation time and area were set to $C_i^k = C_i^1 \cdot q^s$ and $A_i^k = A_i^1 \cdot q^{-s}$, respectively. Note, that each variant still has the same system utilization as the initial variant since $U^S(T_i^k) = C_i^k/P_i \cdot A_i^k = C_i^1 \cdot q^s/P_i \cdot A_i^1 q^{-s} = U^S(T_i^1)$.

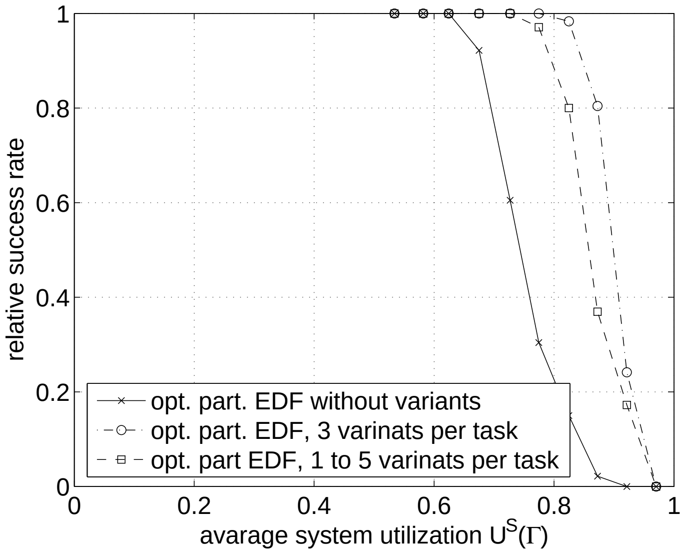


Figure 7.5: Performance of opt.-part.-EDF on task sets with implementation variants

Figure 7.5 shows the relative scheduling success rates, depending on the task sets system utilization, when scheduled onto a device $H$ of size $A(H) = 1$. For an average system utilization of 0.7, the optimally partitioned EDF schedule produced feasible solutions for about 80% of the task sets in the reference case (with only one variant per task). Introducing two additional implementation variants for each task yields an enormous advantage - even task sets with average system utilization of 0.87 could be scheduled with an 80% success rate. If some tasks have more and some tasks have less variants,

[1]First $U^T(T_i^1)$ was chosen from $[0.1, 0.5]$, then $C_i^1$ was chosen from {1,...,30} and the period was set to $P_i^1 = \text{round}\ (C_i^1/U^T(T_i^1))$.

we can also observe a great improvement over the reference case. 80% of the task sets with average $U^S \simeq 0.83$ could be successfully scheduled.

To solve the ILP model, we employed the *lp_solve* library version 4.0 on a 2.8 GHz Pentium 4 machine. We were able to optimally solve most problem instances with a size of $|\Gamma|$ up to 30 in less than 15 seconds. That is, we could compute the schedule for 30 tasks with only one implementation variant, or for 10 tasks with three variants each. In particular, from the 1000 task sets generated for the evaluation in Figure 7.5, only 28 exceeded the 15 second time limit.

### 7.1.5 Approaches for Global- and Server Scheduling of Variant-Rich Tasks

Selecting the right task variants to improve the scheduling performance of our global EDF schedulers or our server based scheduler is rather difficult. Up to now, we have not explored the possibility of choosing variants for these schedulers deeply. We rather present some rough approaches and leave their evaluation for future work.

#### Global EDF for Variant-Rich Tasks

The simulation results of our performance study in Section 5.2 have shown, that the global *EDF-NF* scheduler performance is almost independent of the ratio between area and task computation time. Hence, we cannot expect an improvement of the scheduling performance by following simple rules like *select for each task the variant with the smallest computation time (respectively smallest area).*

A more constructive method can be developed, by considering the *global EDF schedulability test* of Theorem 4 which we have derived in Chapter 4. The schedulability test allows us to rate whether a given selection of implementation variants is guaranteed to be feasible by *EDF-FkF* or not. Hence, we can formulate the following optimization problem:

**Definition 14** (EDF-FkF scheduling condition variant selection problem)**.** *Given a periodic variant-rich task set $\Gamma$, find:*

- *a function $f(i)$ that selects one variant for each task $\Gamma_i$*
- *such that for the set of selected variants $\Gamma_{|f}$ the* EDF-FkF scheduling test *holds, i.e.,*

$$\forall T_i^k \in \Gamma_{|f} : U^S(\Gamma_{|f}) \leq (A(H) - A_{max}) \cdot \left(1 - U^T(T_i^k)\right) + U^S(T_i^k)$$

  *where $A_{max}$ is the largest area of all implementation variants in $\Gamma_{|f}$.*
- *minimize $A(H)$.*

Solving this problem yields the best selection $f$ of implementation variants of our variant-rich tasks in the sense, that it is guaranteed by our scheduling condition that *EDF-FkF* finds a feasible schedule on a device of size $A(H)$. For any other selection $\tilde{f}$ we would require a device of equal or larger size.

One idea to solve such problem is, to apply a suitable meta heuristic such as *simulating annealing*, *tabu search* or a *genetic algorithm*. In this case, the heuristic would propose new selections $f$ and then evaluate the scheduling condition which can be done in linear time. Evaluation of such an approach is subject to future work.

### Server Based MSDL of Variant-Rich Tasks

Similar to the global scheduler, it is rather difficult to find a good variant selection for the *server based MSDL* scheduling method. However, we will formalize the optimization problem and present a simple heuristic as rough draft.

The problem of selecting the right variants for an *MSDL* schedule can be formulated as follows:

**Definition 15** (MSDL scheduling variant selection problem)**.** *Given a variant-rich task set $\digamma$, find a selection function $f$ and a device $H$ such that when* MSDL *is applied to the selected variants it creates a feasible set of servers and the device size $A(H)$ is minimized, i.e.*

- $\Omega = \mathrm{MSDL}(\Gamma_{|f}, H)$ *with* $U^T(\Omega) \leq 1$ *and*
- *minimize* $A(H)$*.*

Also meta-heuristics can be considered for this problem, their runtime may be unacceptably long due to the high time complexity of the *MSDL* procedure which has the order $O(n^4)$. Instead, we find it more reasonable to develop problem specific heuristics. In contrast to the performance of the *global EDF schedulers*, the *server based MSDL* scheduler has appeared to be very sensitive to the ratio of the task area to the task time-utilization (Section 5.2). Figure 5.9 has shown, that *MSDL* performs better on tasks with large area but small time-utilization. Hence a heuristic rule such as *select for all tasks the variant with the smallest computation time* seems to be promising. Once again a detailed evaluation remains subject of future work.

In summary, we have shown that considering task variants considerably improves the scheduling performance. We evaluated this for the partitioned EDF scheduler in detail and also presented rough approaches how the selection of variants could be utilized in the global and server based schedulers.

The next part of this chapter deals with an extension of our basis model that captures the memory access of tasks under real time constraints.

## 7.2 Periodic Tasks with Memory Access

Typical hardware tasks process data in a streaming fashion; hence they demand large memory buffers and access them periodically. Up to now we have neglected the issue of the task's data access under real-time constraints. In this section, we extend our application model by *logical data buffers*, which are used by the tasks. The problem of assigning data buffers to physical memories for a given application is addressed. We show that sharing physical memory banks between tasks is feasible and reduces the required memory resources. Moreover, knowledge of the actual task schedule helps to reduce the memory requirement even further.

Usually, a reconfigurable computing platform provides several relatively large external banks of SRAM to store data processed by hardware tasks. With today's fast-growing FPGA capacities the number of tasks that can run simultaneously will in general be much higher than the number of external memories. Furthermore, hardware tasks optimized for throughput may access several memories at the same time. Hence, a simple one-to-one assignment between tasks and memories is unrealistic. Memories will be shared among tasks, and tasks will access several memories.

The required versatile memory assignment opens up two issues. First, a proper addressing mechanism has to be introduced to ensure that data of one task is stored and addressed separately from the data of other tasks – unless the intention is implementing inter-task communication via shared memory access. We have solved this issue by a page table mechanism that maps logical address to physical addresses [Dan04b].

The second and more challenging issue is resolving conflicts when simultaneously executing tasks attempt to write or read from the same memory bank at the same time. Since the memories are usually single-port memories, only one access can be performed per clock cycle. A simple mutual exclusion mechanism protecting the memory port is insufficient, since this would lead to an unpredictable timing and could enlarge the worst case execution time of a task.

Our approach to solve the problem of *sharing physical RAMs but guarantee timing* is to apply a fix-priority scheduling mechanism to determine the order of memory accesses in case of access conflicts. To this end we assume that the tasks access memory in a deterministic and known pattern. Particularly, we assume periodic RAM accesses such as demonstrated by the DCT-task in Example 1 of Section 2.2.

Before introducing our model, we continue the example to illustrate the problem:

**Example 4.** *Assume two simultaneously executing tasks, each computing an 8-point 1D discrete cosine transform (DCT) on its own data set. Figure 7.6-(a) shows the mapping of the two tasks onto an FPGA. For this example, we use two different variants of the DCT core [107] described in Section 2.2. We repeat the characteristics of the DCT core in Table 7.3.*

Task 1 *has to process a set of 500 words and is implemented using* variant 2 *while* Task 2 *has to process a set of 2000 words and is implemented using* variant 3*. As denoted in the scheduling diagram of Figure 7.6-(b),* Task 1 *reads from and writes to its RAM banks*

| variant | FPGA area (slices) | throughput (sample/clk) | ext. RAM port access period | amount | utiliz. |
|---|---|---|---|---|---|
| 1 | 982 | 1 | 2 clks | 1 ×32 bit word | 50% |
| 2 | 767 | 8 / 9 | 9 clks | 4 ×32 bit words | 45% |
| 3 | 589 | 8 / 17 | 17 clks | 4 ×32 bit words | 24% |

Table 7.3: Size, throughput and memory access pattern of several variants of a XILINX 8 point 1D-DCT IP core [107].

*4 words every 9 clock cycles, while* Task 2 *makes 4 accesses every 17 clock cycles. Due to the FIFO buffers residing within the hardware tasks, the requested RAM accesses do not have to take place exactly in the clock cycles shown in Figure 7.6-(b), but can be scheduled any time within their period of 9 and 17 clock cycles, respectively. The question remains, whether four separate RAM banks are required as shown in Figure 7.6-(a), or if fewer RAM banks can be shared between the tasks. If we assume that all RAMs have a size of 4000 words, the input data of* Task 1 *and* Task 2 *could be stored in SRAM1 and the output data of both tasks in SRAM2. If we assign the accesses of* Task 1 *a higher priority than the accesses of* Task 2, *all memory accesses can be feasibly scheduled, as shown in Figure 7.6-(c).*

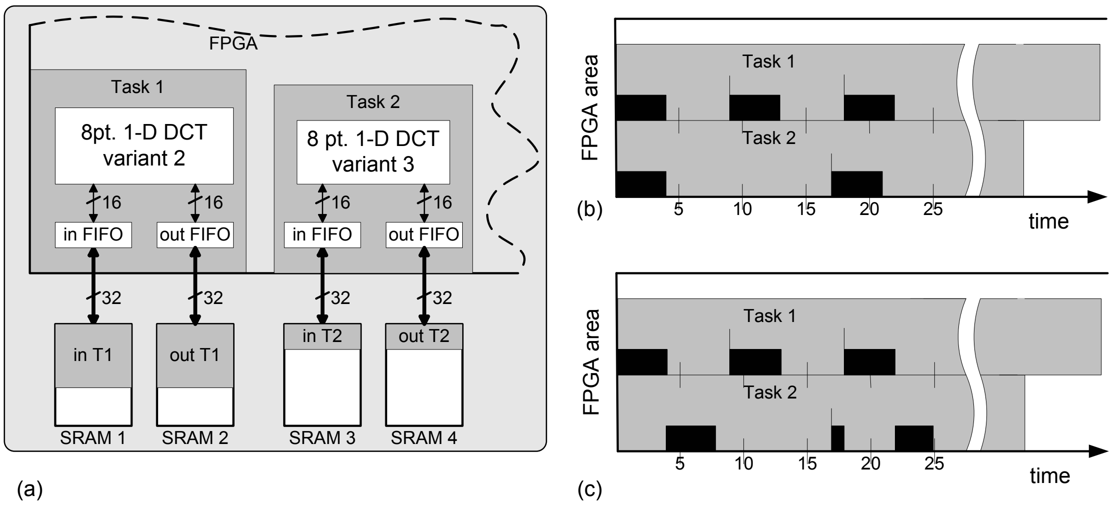


Figure 7.6: (a) Application of Example 4 mapped to an reconfigurable platform; (b) schedule using four memory banks; (c) schedule using two memory banks

### 7.2.1 The Buffer Assignment Problem

The example has shown that physical RAM banks can be shared among simultaneously executing tasks. We now formalize the problem of minimizing the number of required memories. The following assumptions form the basis of our model:

- Tasks use memory in the form of logical buffers, which are accessed periodically. The memory accesses must be completed within the given period, i.e., deadlines equal periods.
- Each task can use many logical buffers.
- A logical buffer can be used by several tasks.
- Each logical buffer is assigned to one and only one physical memory.
- Write and read accesses of tasks to physical memories are scheduled using the fixed-priority *Rate Monotonic (RM)* algorithm.

**Definition 16** (memory-demanding RT application)**.** *A* memory-demanding RT application $\Upsilon$ *is a tuple* $\Upsilon = (\Gamma, B, s, L, \omega, \alpha)$. $\Gamma$ *is a set of periodic tasks.* $B = \{B_1, B_2, \dots\}$ *is a set of logical buffers representing the memory requirements of the application. The size of a buffer is given by* $s : B \to \mathbf{N}$*, where* $s(B_i)$ *denotes the size of* $B_i$ *in terms of words.* $L \subseteq \Gamma \times B$ *is a relation from* $\Gamma$ *to* $B$*, expressing which task is using which buffer, where* $(T_i, B_j) \in L$ *denotes that* $T_i$ *is using buffer* $B_j$*. We assume that a task is using a buffer by a pattern of periodic write or read accesses.* $\omega : L \to \mathbf{N}$ *specifies the access periods in clock cycles, where* $\omega(T_i, B_j)$ *represents the period of accesses of task* $T_i$ *to buffer* $B_j$*.* $\alpha : L \to \mathbf{N}$ *is a function, where* $\alpha(T_i, B_j)$ *denotes the number of requested read or write accesses of task* $T_i$ *to buffer* $B_i$*, which have to be performed within one period* $\omega(T_i, B_j)$*.*

Note, that the relation $L$ between tasks $\Gamma$ and logical buffers $B$ is a *many–to–many* relationship, which allows a task to use several logical buffers, but also one logical buffer to be used by several tasks.

The mapping of the logical buffers to the physical memories of a platform is modeled by a buffer assignment function:

**Definition 17** (buffer assignment:)**.** *A* buffer assignment *of a given application* $\Upsilon$ *is an injective function* $a : B \to M$*, which assigns each logical buffer* $B_i \in B$ *required by the application to one physical memory bank* $M_k \in M$ *of the platform. For simpler notation, we define the binary variables* $\hat{a}_{i,k} \in \{0, 1\}$ *such that* $\hat{a}_{i,k} = 1$ *if* $a(B_i) = M_k$ *and* $0$ *otherwise. I.e.,* $a_{i,k} = 1$*, iff buffer* $B_i$ *is assigned to memory* $M_k$*. A* buffer assignment $a$ *is called* feasible*, if*

1. *the logical buffers assigned to each memory* $M_k$ *do not exceed its size, and*
2. *all resulting access requests to the physical memories are performed within their deadlines, when scheduled by RM.*

Note, that the buffer assignment of Definition 17 is a function, which requires that a logical buffer is mapped to one and only one physical memory. Distributing a logical buffer to many physical memories is not considered. Also the function is injective, requiring that every buffer is assigned to a memory.

As an illustration, an instance of a *memory demanding RT application* $\Upsilon$ and a particular mapping of buffers to physical memories can be drawn as a graph. Figure 7.7 shows the graph according to Example 4. Tasks $\Gamma$, logical buffers $B$ and physical memories $M$ are the graph nodes plotted as circles, squares and rectangles respectively, the latter two labeled with their according size $s$. The edges between tasks and buffers represent the usage relation $L$, labeled with the number of accesses $\alpha$ and periods $\omega$. The *buffer assignment* function $a$, which has to be computed, is represented by edges from $B$ to $M$.

According to Definition 17, a *feasible buffer assignment* is subject to two conditions. Obviously, the memory size condition is fulfilled, if and only if Equation 7.13 holds where $s(M_k)$ denotes the size of the memory bank $M_k$.

$$\sum_{i=1}^{|B|} \hat{a}_{i,k} s(B_i) \leq s(M_k), \qquad \forall M_k \in M \tag{7.13}$$

The second condition concerning the schedulability of access to memory under the RM algorithm is more involved. Since each buffer $B_i$ has been uniquely assigned to a memory $M_k$, and this assignment does not change during runtime, the scheduling condition can be checked for each involved physical memory separately. This problem is equivalent to testing the schedulability of a periodic RT task sets scheduling onto a uniprocessor by RM. A sufficient but not necessary condition can be derived by computing the utilization factor of the processor utilization, which is in our case the utilization factor of the physical memory port. Recall, that any periodic task set can be scheduled by RM, given the processor utilization is less then $ln2$, i.e. less than $\approx 69$ % (see [25]p. 78 ff. and Section 2.4.2).

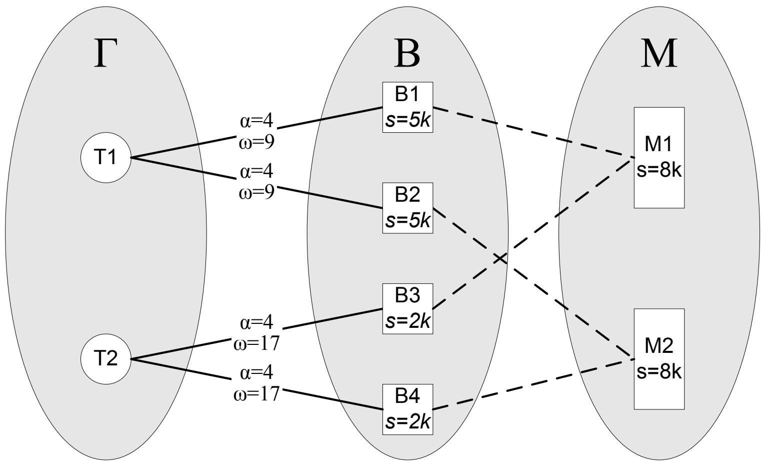


Figure 7.7: Graph representation of the memory demanding RT application $\Upsilon$ of Example 4, with tasks $\Gamma$, logical buffers $B$ and their assignment to physical memories $M$.

Following the same concept, we define two access utilization factors modeling the RAM port utilization: $\mu_B(B_i)$ expressing the utilization of the *virtual* RAM port of a logical buffer $Bi$, and $\mu_M(M_i)$ expressing the utilization of the port of a physical memory $M_i$ as a consequence of the assignment of buffers to that memory.

Let $L(\Gamma, B_i) \subseteq L$ denote all edges from tasks incidence to buffer $B_i$, than the virtual port utilization $\mu_B(B_i)$ is defined as:

$$\mu_B(B_i) = \sum_{l \in L(\Gamma, B_i)} \frac{\alpha(l)}{\omega(l)} \tag{7.14}$$

The utilization factor of a physical memory port $\mu_M(M_i)$ results from summing up all logical utilization factors of buffers assigned to that memory. It is calculated through:

$$\mu_M(M_k) = \sum_{i=1}^{|B|} \hat{a}_{i,k} \mu_B(B_i) \tag{7.15}$$

Applying the test condition from RM scheduling, we can derive a sufficient (but not necessary) condition to guaranty a feasible schedule of all memory accesses:

$$\mu_M(M_k) \leq \ln 2, \qquad \forall M_k \in M \tag{7.16}$$

Based on the two conditions for a *feasible buffer assignment*, we formulate the following optimization problem:

**Definition 18** (2C-BA (two constraint buffer assignment)). *Given a memory demanding application $\Upsilon$ and a set of platform memories $M$ each with a certain cost $c(M_k)$, find a feasible buffer assignment $a : B \to M$ subject to:*

- $\sum_{i=1}^{|B|} \hat{a}_{i,k} s(B_i) \leq s(M_k), \qquad \forall M_k \in M$ *and*
- $\mu_M(M_k) \leq \ln 2, \qquad \forall M_k \in M,$
- $\min \sum c(M_k)$ *of used memories, i.e. minimize the total cost of used memories.*

### 7.2.2 Task Schedule Aware Buffer Assignment

We can weaken the constraints of feasible buffer assignments and therefore extend the set of feasible buffer assignments by taking into account the actual schedule of hardware tasks, i.e. the possible sets of parallel running tasks. Hence, an optimization strategy may find better assignments that require less physical memory banks.

In a given task schedule $R(t)$ not all tasks $T_i \in \Gamma$ are executed in parallel, but only a certain subset $\zeta_j \subseteq \Gamma$ is executed at any given time. The number of different running sets appearing in the schedule is finite, and is denoted by $\zeta = \{\zeta_1, \zeta_2, \dots\}$. Therefore, we can compute the utilization of each virtual buffer port $\mu_B$ and each physical memory port utilization $\mu_M$ separately for each running set $\zeta_j$. Thus, only the usage relations in $L$ from the tasks of a specific running set $\zeta_j$ to a specific buffer $B_i$ – denoted by $L(\zeta_j, B_i) \subseteq L$ – are considered. The virtual memory port utilization and physical port utilization are reduced and change to:

$$\mu_B(B_i, \zeta_j) = \sum_{l \in L(\zeta_j, B_i)} \frac{\alpha(l)}{\omega(l)} \quad \leq \mu_B(B_i) \tag{7.17}$$

$$\mu_M(M_k, \zeta_j) = \sum_{i=1}^{|B|} \hat{a}_{i,k} \mu_B(B_i, \zeta_j) \quad \leq \mu_M(M_k) \tag{7.18}$$

As a result, the access scheduling condition changes from Equation 7.16 to Equation 7.19, which has to be checked for all combinations of memories and running sets.

$$\mu_M(M_k, \zeta_j) \leq \ln 2, \qquad \forall (M_k, \zeta_j) \in M \times \zeta \tag{7.19}$$

We illustrate the advantage of the task schedule aware buffer assignment by extending Example 4.

**Example 5.** *Assume the application $\Upsilon$ of Example 4 extended by two additional tasks $T_3, T_4$ that share another buffer $B_5$ of 1000 words. Both tasks access this buffer once within a period of 5 clock cycles. Figure 7.8 illustrates this application as a graph, where tasks $\Gamma$, logical buffers $B$ and physical memories $M$ are the graph nodes plotted as circles, squares and rectangles, respectively. Assume further that in a given task schedule, either the tasks $\zeta_1 = \{T_1, T_2\}$ or $\zeta_2 = \{T_2, T_3, T_4\}$ are running simultaneously. Buffer $B_5$ is assigned to memory $M_2$, in addition to $B_2$ and $B_4$. Then the utilization of the RAM port of $M_2$ is calculated to be $\mu_M(M_2, \zeta_1) = 4/9 + 4/17 \approx 0.68$ for the first running set and $\mu_M(M_2, \zeta_2) = 4/17 + 2/5 \approx 0.64$ for the second running set. Therefore, the buffer assignment is feasible according to Equation 7.19. In contrast to that, the utilization without considering the running sets amounts to $\mu_M(M_2) \approx 1.08$, making the buffer assignment infeasible according to Equation 7.16.*

The example showed, that the memory banks required by an application can be reduced, when the task schedule is taken into account. This motivates us, to formulate the problem of a schedule aware buffer assignment.

**Definition 19** (schedule aware buffer assignment (SA-BA))**.** *Given a memory demanding application $\Upsilon$ and the finite set $\zeta = \{\zeta_1, \zeta_2, \dots\}$ of the different running sets of its task schedule, find a feasible buffer assignment $a : B \to M$ subject to*

- $\sum_{i=1}^{|B|} \hat{a}_{i,k} s(B_i) \leq s(M_k), \qquad \forall M_k \in M$ *and*
- $\mu_M(M_k, \zeta_j) \leq \ln 2, \qquad \forall M_k \in M, \forall \zeta_j \in \zeta$
- $\min \sum c(M_k)$ *of used memories, i.e. minimize the total cost of used memories.*

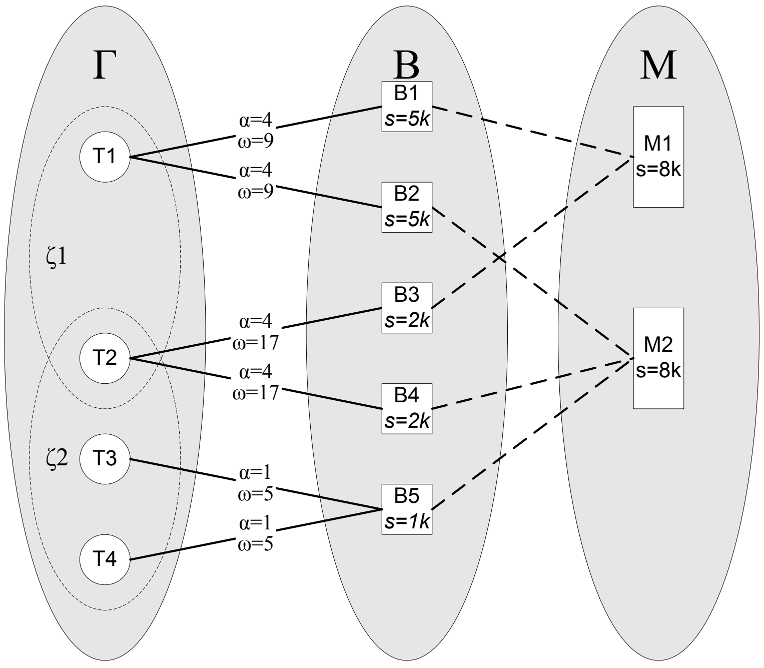


Figure 7.8: Memory demanding RT application from Example 5, illustrating different running sets of a task schedule.

## 7.2.3 Buffer Assignment Based on Integer Programming

Solving the 2C-BA and SA-BA problems to optimality is rather difficult. Both are generalizations of the classical bin-packing problem and hence cannot be solved in polynomial time. For solving instances of large size, suboptimal methods like approximation algorithms and heuristics are needed.

However, in this work we are rather interested in comparing the cost of a *non-schedule aware* buffer assignment to that of a *schedule aware* buffer assignment. Moreover, many applications of interest may be of rather medium complexity, e.g. up to some tens of tasks and buffers.

Hence, we present an integer linear programming approach to solve simplified versions of the 2C-BA and SA-BA problems. The assumptions we make are:

- All platform memories are of equal size and equal cost. Hence we minimize the number of used memories.
- For the RM scheduling condition we use the fix utilization bound of $ln2$, rather than the more precise bound of $n(2^{1/n} - 1)$. This makes our port utilization pessimistic, but enables us to use an ILP formulation.

For this modified versions of 2C-BA and SA-BA problem we present an efficient binary integer program:

Equations 7.20 to 7.23 specify the *non schedule-aware* buffer assignment problem, while replacing Equation 7.23 by Equation 7.24 specifies the *schedule-aware* buffer assignment problem.

The binary variables are $\hat{a}_{i,k} \in \{0,1\}$, where $\hat{a}_{i,k} = 1$ iff buffer $B_i$ is assigned to memory $M_k$. The index range for $i$ and $k$ is $\{1 \dots |B|\}$ with $i \geq k$, which exploits the symmetry of the problem and leads to a complexity of $|B|^2/2$ variables.

$$\min \sum_{i=1}^{|B|} \hat{a}_{i,i} \tag{7.20}$$

$$\sum_{k=1}^{i} \hat{a}_{i,k} = 1, i \in \{1 \dots |B|\} \tag{7.21}$$

$$\sum_{i=k+1}^{|B|} \hat{a}_{k,k}[sM - s(B_k)] - \hat{a}_{i,k} s(B_i) \geq 0, k \in \{1 \dots |B|\} \tag{7.22}$$

$$\sum_{i=k+1}^{|B|} \hat{a}_{k,k}[\ln 2 - \mu_B(B_k)] - \hat{a}_{i,k} \mu_B(B_i) \geq 0, k \in \{1 \dots |B|\} \tag{7.23}$$

$$\sum_{i=k+1}^{|B|} \hat{a}_{k,k}[\ln 2 - \mu_B(B_k, \zeta_j)] - \hat{a}_{i,k} \mu_B(B_i) \geq 0, (k,j) \in \{1 \dots |B|\} \times \{1 \dots |\zeta|\} \tag{7.24}$$

The objective function (Equation 7.20) minimizes the number of memories where buffers have been assigned to, since $a_{i,i}$ is set to 1 iff any buffer is assigned to $M_i$. Equation 7.21 ensures that each buffer is assigned exactly once. Equation 7.22 and Equation 7.23/7.24 consider respectively the maximal size and maximal port utilization of each memory.

### 7.2.4 Simulation Results

In our experiments, we could solve both variants of the ILP for problem sizes of up to $|B| = 15$ in reasonable time. Solving one instance took less than one minute using *lp_solve* version 4.0 on a 2.8 GHz Pentium IV machine. Table 7.4 presents the results for randomly generated problem instances, each with 15 tasks and 15 buffers. Each buffer is used by 1 or 2 tasks. In the various test cases, the virtual buffer port utilization $\mu_B(B_i)$ and the buffer size $s(B_i)$ (in percent of the memory bank size) have been chosen equally distributed from the reported intervals. For the *schedule-aware* buffer assignment problem, a task schedule with 15 running sets, each containing 1 to 3 tasks, has been assumed. For each test case the table reports on the number of required physical memory banks of: the buffer assignment without sharing memory banks; the non schedule aware assignment; and the schedule aware buffer assignment.

| test | parameter | | used memories (in average) | | |
|---|---|---|---|---|---|
| no | equally distributed | | non mem | non sched. | schedule |
| | port util. $\mu_B(B_i)$ | buffer size $s(B_i)$ | sharing | aware | aware |
| 1 | $[0, 0.69]$ | $[0, 100\%]$ | 15 | 10.42 | 8.85 |
| 2 | $[0, 0.69]$ | $[0, 50\%]$ | 15 | 8.90 | 4.49 |
| 3 | $[0, 0.50]$ | $[0, 50\%]$ | 15 | 5.94 | 4.26 |
| 4 | $[0, 0.25]$ | $[0, 50\%]$ | 15 | 4.29 | 4.29 |
| 5 | $[0, 0.69]$ | $[0, 25\%]$ | 15 | 8.90 | 3.47 |
| 6 | $[0, 0.50]$ | $[0, 25\%]$ | 15 | 5.97 | 2.67 |
| 7 | $[0, 0.25]$ | $[0, 25\%]$ | 15 | 3.23 | 2.33 |

Table 7.4: Simulation results of 2C-BA and SA-BA problems.

Without sharing memory banks each buffer is assigned to one physical memory exclusively, thus 15 memories are used in all test cases. In the first test case, the average buffer port utilization and size are chosen relatively large. Hence, only in few cases multiple buffers can be assigned to one memory bank which results in about 10-11 memory banks for the non schedule-aware assignment. A schedule aware assignment results in 8-9 memories which is an further reduction of about 15%. As the port utilization and size of the buffers decreases, the number of used memories is reduced considerably. For all test cases except test no 4, the schedule aware assignment gives a further cost reduction compared to the non schedule aware assignment. In test case no 4, the virtual port utilization is small compared to the relative buffer size. Hence, the RAM size and not the port utilization is the major limitation. Therefore, weaken the RAM port constraints by a schedule aware assignment brings no further improvement. Test case no 5, where the port utilization is choose large and the buffer size is choose small, shows the greatest difference (about 60% further improvement) between the cost of a schedule-aware and a non schedule aware assignment.

### 7.2.5 Concluding Discussion on Memory Demanding RT Applications

In this section we extended our application model by the memory requirements of the tasks. The goal was, to share physical memories among the tasks without jeopardizing timing constraints. We have presented a formal model of a buffer assignment problem and exploited the fact that the conditions for a successful scheduling of memory accesses under RM can be weakened by considering the actual combinations of tasks running simultaneously. This *schedule aware* buffer assignment is particularly interesting in combination with the presented server based task scheduling algorithm MSDL of Section 4.3. As MSDL produces a rather small number of running sets, the resulting ILP is likely to stay within a realistic problem size. The simulation results indicate, that sharing memories between buffers reduces the total memory cost considerably and that further reductions can be achieved by a schedule aware buffer assignment.

For the memory port utilization we have used the simple RM scheduling bound of $\ln 2$. Using better RM scheduling conditions (see Section 2.4.2) would potentially improve our results. However, using the RM hyperbolic bound would bring non-linear conditions into the assignment problem. Hence using an ILP approach would not be straightforward. Using the RM bound of Equation 2.2 for a fix $n$ would be an option for an ILP approach, but even for small $n$ the bound is close to $\ln 2$ and hence we cannot expect considerable improvements. We leave the investigating of non ILP based buffer assignment methods in combination with improved RM scheduling conditions as a subject of future work.

Sharing RAM banks among tasks can further be used to implement inter-task communication. A particular mechanism for the communication among periodic tasks is the use of *Cyclical Asynchronous Buffers (CABs)* (see [25] p.319 ff.). Hard real-time tasks can access CABs without blocking the sender or receiver, for example to exchange messages containing current sensor values. A CAB provides a one-to-many communication channel. Hence CABs can be implemented using our many-to-many logical buffers, extended by some simple handshaking functionality.

## 7.3 Chapter Conclusion

Within this chapter we generalized and extended our task model in two directions in order to respect the special characteristics of hardware-tasks.

When implementing a computational task as a digital circuit on an RHD, the designer can choose several architectures resulting in different tradeoffs between area and computation time. This is captured, by modeling a task as a set of implementation variants. We have shown that considering these variants within the task schedule can considerably increase the performance.

Hardware-tasks typically process high amounts of data which cannot be kept within the task circuit but has to be stored in dedicated RAM resources. In our approach, we extended the model to capture the memory requirements of the tasks. Both, the required RAM size and the utilization of the RAM port have been modeled. Task may share a physical RAM bank, iff its size is not exceeded and the access to the RAM-port is feasible under the RM priority schema. We presented a method to assign task buffers to RAM-banks, which allows the minimization of the memory requirements and makes inter-task communication feasible.

# CHAPTER 8

# Prototype: FPGA Based Real-Time Kernel

This chapter presents our prototype of an FPGA based real-time system. It implements the *server based* scheduler (MSDL) and hence uses the *free space - time sharing* resource model which is based on full FPGA reconfiguration. In contrast to others, the kernel is a hardware-only system with the OS functions entirely implemented in hardware. We first describe the architecture of the system and the kernel and the phases of the scheduler cycle. Afterwards, we present an automated synthesis flow. It takes the task set parameters and the source files of the kernel and the tasks as input and generates the required FPGA bit-streams of the entire system. Finally, results on the system's logic requirements and timings are reported.

Up to now we have build no prototype systems for the *global EDF* and *partitioned EDF* algorithms. These methods are both based on the 1D resource model. Since several prototypes that follow the 1D model and moreover implement task relocation and state saving have been reported previously [60][20][21][61], we found that no additional proof of concept is necessary within this work.

We previously reported on our prototype in [DMP06].

## 8.1 System Architecture

### 8.1.1 An All-Hardware Runtime System

We have implemented an all-hardware runtime system for the execution of MSDL schedules. The decision for implementing the complete runtime system in hardware was driven by the fact that a hardware scheduler can control the user tasks in a very efficient manner and, thus, minimize the runtime overhead and deliver maximum scheduling performance. In a processor-based real-time runtime system, a periodic timer triggers a kernel function that maintains time in terms of system ticks. The smaller the timer period is, the finer the system's time resolution is but the higher the runtime overhead is. In our all-hardware runtime system, the scheduler runs in parallel to the user tasks and, thus, no execution time is wasted for updating the internal time reference.

As the runtime system itself undergoes full device reconfiguration, we have to include it in each single configuration bit-stream. On one hand, this opens up the opportunity for compile-time optimization of parts of the runtime system to specific servers. On the other hand, we also have to save and restore the context of the runtime system.

Our prototype is based on a CELOXICA RC203 board. Figure 8.1 shows the system architecture consisting of an FPGA, a CPLD, a flash memory card and two banks of SRAM. The CPLD is connected to FPGA user I/Os, to the FPGA configuration port, as well as to the flash memory card. This way, the CPLD acts as a reconfiguration controller that reads configuration bit-streams from the flash memory card on request of the logic implemented in the FPGA.

The main runtime system components shown in Figure 8.1 comprise the *timer*, the *scheduler*, the *task management* and an optional *memory management unit (MMU).*

**Timer:** The timer provides the time reference for the system by scaling down the clock to a system tick signal with a user-defined period. All execution times and periods are expressed in terms of system ticks.

**Scheduler:** The scheduler is the central part of the runtime system as it controls which

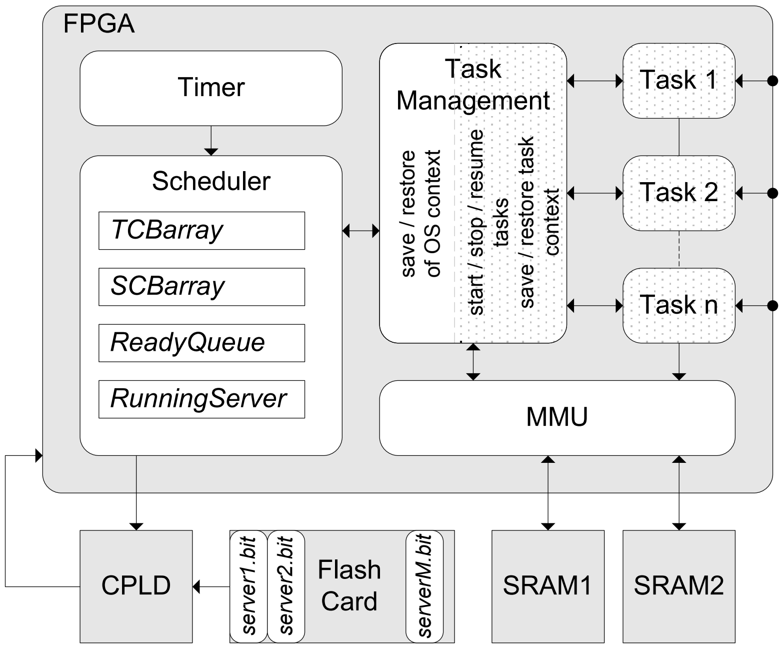


Figure 8.1: System architecture

user tasks are executed at any given time. The scheduler further decides whether the FPGA has to be reconfigured and, if so, what the proper configuration is. To control the execution of servers and tasks, three states are introduced: *READY, RUN,* and *IDLE*. A server (task) is in the *RUN* state, when it is currently being executed on the FPGA. A server (task) in the *READY* state has been released and has not yet terminated, but it is not currently being executed on the FPGA. This implies that another server is currently assigned to the FPGA. Finally, a server (task) enters the *IDLE* state, when its current instance has terminated. In this case, the server (task) has to wait for its next release.

**Task Management:** The task manager controls the task operations and the context switching. Whenever a reconfiguration is required, the scheduler triggers the task manager to save the entire system context. Similarly, the task manager is called to restore the entire system context after the FPGA reconfiguration. The task manager is comprised of a server-independent part and a server-dependent part. The server-independent part is identical in all configuration bit-streams and implements a logic to save and restore the context of the runtime system. The server-dependent part (speckled objects in Figure 8.1) implements a logic to start, stop and resume the user tasks and to save and restore their contexts. Since each server contains a different set of user tasks, and each user task has a different context size, we apply server-specific optimizations in the compilation process.

**MMU:** The memory management unit is an optional component. In a runtime system without MMU, the task manager is connected directly to an SRAM bank. This SRAM is then exclusively used as intermediate context storage. The second SRAM bank is devoted to implement data storage for the user tasks. Using an MMU, such as the one described in [Dan04c, Dan04b, DP05b], the SRAM banks are divided into pages which can be shared among the runtime system and the user tasks.

### 8.1.2 Scheduler Data Structures

Executing an MSDL schedule means applying the sequential EDF scheduling policy to a precomputed set of servers. To this end, the scheduler implements several data structures. The *server control block array (SCBarray)* holds for each server the following constants: ID, period, computation time, the address of the corresponding bit-stream file in the flash memory, and the set of tasks which constitute the server. The variable part of the data structure stores the server's current server state, its absolute deadline, and its remaining computation time. The register *RunningServer* contains the ID of the server in execution.

In general, the task periods may differ from the server periods. Tasks can start and terminate at arbitrary times within their servers. Hence, also the task states can differ from those of the servers, requiring the scheduler to implement another data structure for tasks. The *task control block array (TCBarray)* holds for each task the following constants: ID, period, and the size of the context. The variable fields are the task's current state and its absolute deadline.

The ready queue is used to store a list of servers in the ready state, sorted according to non-decreasing absolute deadlines. We do not implement the ready queue directly, but provide a single register, *ReadyQueueHead*, that points to the entry in *SCBarray* holding the server with the currently earliest deadline. Additionally, each element of *SCBarray* contains a field which points to the next server in the ready queue.

### 8.1.3 The Scheduler Cycle

Time proceeds in system ticks. The period of the system tick corresponds to a user-defined number of clock cycles which is constant during the application's runtime. A scheduling cycle denotes all the operations of the scheduler during a system tick period.

There are basically two different sequences of scheduler operations during a cycle, depending on whether a device reconfiguration is needed or not. These two cases are outlined in Figure 8.2.

In any case, the scheduler cycle starts with the *Run Tasks (RT)* phase. Here, all tasks that are included in the current server and are in the *READY* state are set to the *RUN* state. Consequently, these tasks receive a signal to start or resume their execution. In the example of Figure 8.2, $S_1$ starts execution. Then, the scheduler idles until almost the end of the system tick period. During this time interval, the user tasks are executed and some may terminate.

The first scheduler phase after the delay is called *end cycle*. Here, the scheduler decreases the remaining computation time of the currently running server. If the remaining computation time reaches zero, the server has finished its execution for its current period and its state is set to *IDLE*.

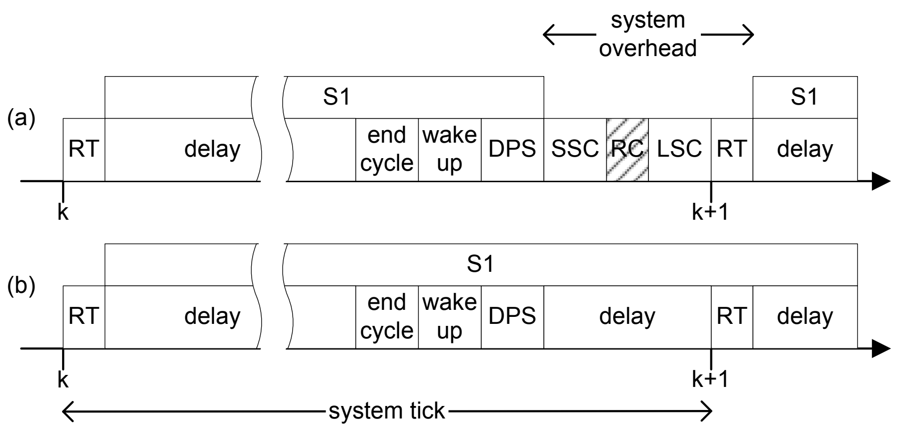


Figure 8.2: Scheduler phases during a system tick period

In the following scheduler phase *wake up*, the scheduler checks if any server or task has reached the end of its period. This is done by comparing the absolute deadlines to the system tick. If a server or task is at the end of its period, it is set to the *READY* state and its deadline is set to $d_i = d_i + P_i$. In case of a server, the remaining computation time is set to the server's initial computation time.

The *Dispatch Server (DPS)* phase increments the system tick and determines the server that should run in the next system tick period following the EDF rule. Since the queue of ready servers is sorted according to non-decreasing deadlines, DPS only compares the deadline of the running server to that of the first server in the ready queue. If the currently running server terminates or has to be preempted, the first server in the ready queue becomes the next running server. In this case, shown in Figure 8.2-(a), the *Save System Context (SSC)* phase is entered next. If the currently running server is determined to continue execution, shown in Figure 8.2-(b), a delay phase is entered next.

If a device reconfiguration is intended, the SSC phase requests all currently running tasks to stop their execution. Then, the contexts of all user tasks are stored in the external SRAM, followed by the context of the runtime system. Upon completion of the context saves, the scheduler triggers the FPGA reconfiguration and the subsequent *Reconfiguration (RC)* phase is entered. After reconfiguration, the scheduler executes the *Load System Context (LSC)* phase and restores first the context of the runtime system and, afterwards, the contexts of all tasks in the new server running. The following RT phase starts the next scheduler cycle (system tick period).

The total time required by the SSC, RC, LCS, and RT phases is denoted as runtime system overhead as it cannot be used for user task execution. This overhead actually combines overheads due to device reconfiguration (RC), task and runtime system context saves and restores (SSC, LCS), and runtime system management overhead (RT).

In case no device reconfiguration is intended, the scheduler enters another delay phase after the DPS phase. The duration of this delay matches exactly the sum of the SSC, RC and LSC phases. By this, we ensure a constant time interval for each system tick. As can be observed from Figure 8.2, the runtime system overhead only occurs if the system switches from one configuration to another. If a server executes for several system ticks, which can be expected to be the regular case, there is no runtime overhead at all.

The presented scheduler cycle assumes that a system tick period is larger than the reconfiguration time. While this constraint on the system tick period simplifies the time keeping and scheduling processes, it also limits the system's time resolution. An alternative implementation could make use of an external, non-reconfigured real-time clock as a time reference and reduce the system tick period considerably.

## 8.2 Synthesis Tool Flow

Figure 8.3 shows the basic steps of the synthesis tool flow that generates the configuration bit-streams for a particular application. We use *CELOXICA's* parallel programming language *Handel-C* to specify the runtime system and the user tasks.

The application, i.e., the task set $\Gamma$, is specified in textual form (*appl.txt*) and contains the period, worst-case computation time, and area requirement for each user task. The

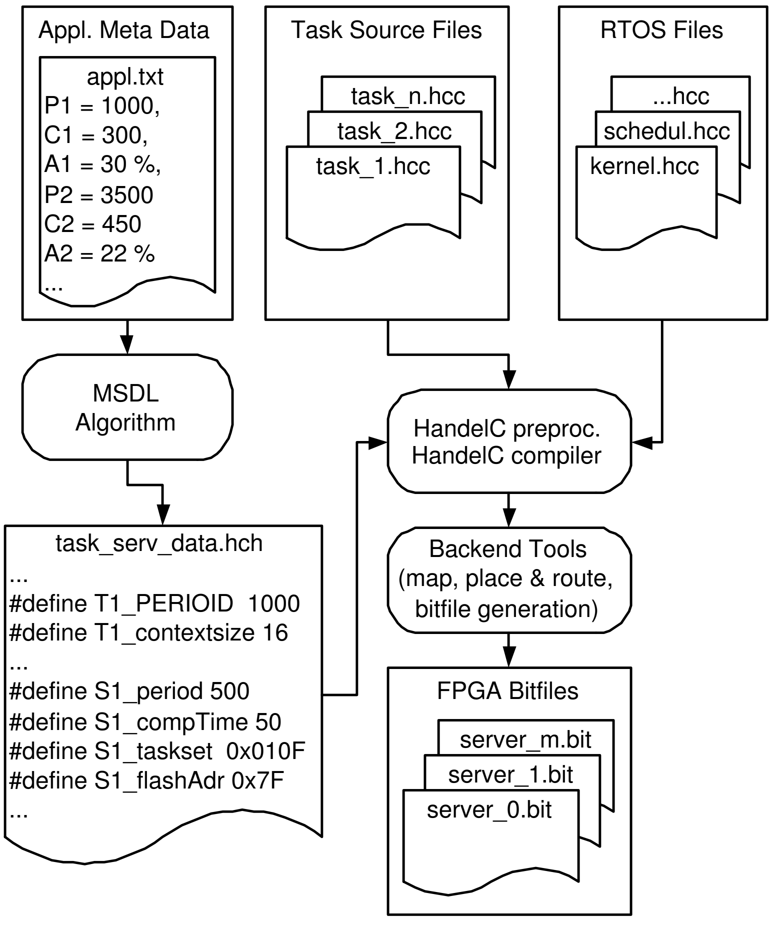


Figure 8.3: Synthesis tool flow

first step of the tool flow calls the MSDL algorithm to compute a feasible set of servers $\Omega$ (see Section 4.3). If such a feasible set of servers is found, the relevant task and server parameters are written into the Handel-C header file *task_serv_data.hch*. This file contains mainly preprocessor macros, defining several properties of the servers and tasks such as the IDs and periods. Furthermore, for each server the computation time, the set of included tasks, and the address of the configuration bit-stream in the flash memory card is defined.

The second step of the tool flow takes this header file together with the Handel-C source files for the user tasks and the runtime system and creates the individual configuration bit-streams for the server tasks. We use the Handel-C compiler (CELOXICA DK4) and standard FPGA vendor tools (XILINX ISE 7.2) for this step.

Besides the $m$ servers generated by MSDL, we create another bit-stream for an idle-server (server_0). The idle server contains only the runtime system but no user tasks. This server is run as initial configuration after system startup and when all other servers are idle. When the synthesis tool flow is completed, all $m+1$ server bit-streams are stored onto the flash memory card at their respective addresses. After system power-on, the CPLD automatically configures the idle server into the FPGA. The scheduler takes over control and decides which server to reconfigure next according to the MSDL schedule.

User tasks are free to use logic, memory and I/O resources of the FPGA, as long as they implement the pre-defined interface to the runtime system. This interface allows the runtime system to start and stop a task and to initiate context save and restore operations. Although our current prototype expects user tasks specified in Handel-C, integrating other HDL components into the synthesis flow is still straight-forward.

## 8.3 Synthesis Results

We have implemented the entire system on CELOXICA's RC203 board, which hosts a XILINX Virtex II 3000 FPGA, two 2-MB SRAM banks and a 64 MB SmartMedia flash memory card. An FPGA configuration bit-stream has a size of approximately 1.25MB, which allows us to store up to 50 server bit-streams on the flash memory card.

To determine the area requirement for the runtime system, we have conducted synthesis experiments with different numbers of tasks and servers. These experiments use a runtime system without MMU and minimal user task implementations in order to reduce the effect of the user task logic on our measurements. The results are presented in Table 8.1. The number of required LUTs grows with the number of servers and ranges from 16% to 45% of the overall LUT resources. The required number of Flip Flops (FFs) grows slower and ranges from 5% to 14% of the overall device FFs. Assigning almost 50% of the device resources to the runtime system might look unrealistic at the first glance. However, this number corresponds to the rather high number of 15 servers and 15 tasks, respectively. Then, these figures have been derived from a first, unoptimized version of the runtime system. For example, all scheduler data structures are currently implemented as registers. Mapping these structures to Block-RAM will reduce the required resources substantially. Finally, in the future we expect that much larger devices will become available, reducing the relative overheads for such a runtime system.

| # tasks | # servers | LUTs (% of device) | FFs (% of device) |
|---|---|---|---|
| 15 | 3 | 7786 (27%) | 2703 (9%) |
| 15 | 6 | 9802 (33%) | 3028 (10%) |
| 15 | 9 | 10297 (35%) | 3390 (11%) |
| 15 | 12 | 11655 (40%) | 3725 (12%) |
| 15 | 15 | 13109 (45%) | 4030 (14%) |
| 3 | 3 | 4470 (16%) | 1336 (5%) |
| 6 | 3 | 5297 (18%) | 1677 (6%) |
| 9 | 3 | 6123 (21%) | 2019 (7%) |
| 12 | 3 | 6948 (24%) | 2361 (8%) |
| 15 | 3 | 7787 (27%) | 2703 (9%) |

Table 8.1: Runtime system size for various numbers of tasks and servers

The time overheads for the runtime system are fairly small. With $s$ as the number of servers and $t$ as the number of overall tasks, we measured a runtime for the SSC phase of $3s + 7t$ clock cycles and for the LSC phase of $3s + 8t$ cycles. The wake up phase takes $3s + t$ cycles and, finally, the end cycle and DPS phases take just one clock cycle each. For a runtime system controlling 3 servers and 9 tasks we achieved a maximal clock frequency of about 50 MHz. Although the Virtex II 3000 could be reconfigured within some 20 ms via the SelectMap interface, the speed of the SmartMedia flash memory card limits the reconfiguration time on the RC203 board to 180 ms.

We further measured the overhead of the logic inside the user tasks needed for storing and loading the tasks' contexts by synthesis experiments. We organized the context registers in form of a shift register chain and measured the number of additional LUTs required to implement the load and store functions. The results reveal an overhead of 2 LUTs per bit for a context register of 32 bit. This overhead decreases to 1.2 LUTs per bit for context register sizes of 256 and more.

## 8.4 Chapter Conclusion

In this chapter, we presented a prototype system that executes periodic real-time tasks on dynamically reconfigurable hardware. We used the *MSDL* scheduling technique which offers an efficient off-line schedulability test and requires a minimal number of device configurations. We showed an implementation of this technique on an FPGA board, including an all-hardware runtime system and the corresponding synthesis tool flow.

For the future, we plan to port the system to an FPGA platform with shorter reconfiguration time. Although we consider it a benefit of our approach that it can work with full device reconfiguration, and thus with all SRAM-based devices on the market, we also intend to investigate partial device reconfiguration. A natural way to utilize partial reconfigurability in our prototype would be to keep the runtime system permanently in the device and only reconfigure between different servers. Alternatively, an embedded CPU could be used to implement the runtime system. Prototypes for the *global EDF* and *partitioned EDF* schedulers may also be subject of future work.

# CHAPTER 9

# Conclusion and Outlook

## 9.1 Summary

In this thesis we presented our investigation on real-time multitasking on reconfigurable architectures. Application tasks were implemented as digital circuits and executed on a runtime reconfigurable hardware device such as an FPGA. While the general concept of executing hardware-tasks with RHDs had been proven feasible in several prototypes [102][57][105], and scheduling and placement of hardware-tasks had been given much attention by the research community (e.g. in [14][94]), only minor results on real-time multitasking on RHDs had been developed so far. We presented several key contributions to this field within our work: We defined the periodic task model for RHDs, where each task requires some amount of logic resources to be executed. An arbitrary set of tasks can run in parallel as long as the cumulative task area does not exceeded the device area. For such a model we presented three major scheduling methods:

1. The *global EDF* algorithms were inspired from global EDF scheduling methods for multiprocessors. The new EDF algorithms were denoted as *EDF-NF* and *EDF-FkF*. They manage resources globally, meaning that any resource can be assigned to any task. Following a resource augmentation approach [81], we presented the first known scheduling test for these algorithms. The test can be computed in $O(n)$ time, $n$ being the number of tasks. Simulations showed that, in conformance to results derived for multiprocessor scheduling, the test is pessimistic for task sets with high system utilization. Further, the simulations showed that the *EDF-NF* algorithm gives a non-negligible improvement over *EDF-FkF*.

2. The *partitioned EDF* algorithms were also inspired by traditional multiprocessor scheduling algorithms. The task set is partitioned into partition-blocks at design time and exclusive device resources are assigned to each partition-block. At

runtime, the tasks of each partition-block are scheduled by a separate EDF scheduler. We showed the relation of this partitioning problem to the two-dimensional level-strip packing problem. We employed a previously reported ILP model to compute the optimal partitioning and a next-fit heuristic to achieve a close to optimal partitioning for large task sets.

3. In the *MSDL* method the computation of the user tasks is distributed into periodic servers tasks. At design time, the tasks are grouped to servers for parallel execution. At runtime, the servers are executed sequentially according to EDF. Since only one server executes at a time, this approach is suitable for systems and devices which do not support partial reconfiguration.

While in our analytic performance comparison we showed that there is no dominance among the three approaches, our simulation studies showed that on average, global EDF outperforms partitioned EDF, which outperforms MSDL. Further it was shown that the global EDF scheduling test is rather pessimistic. Concerning execution model overhead, we analyzed the reconfiguration overhead for all three approaches and included it within the schedulability tests. Simulation experiments showed that global EDF suffers significantly more from overhead when compared to the two other schedulers.

Two extensions of our task model were presented in order to consider the specific characteristics of reconfigurable applications: First we considered tasks for which not only one but several alternative implementations (circuits) exist. We presented an ILP model to select the optimal implementations for the partitioned EDF scheduler, which considerably improved the scheduling performance. Second, we considered tasks memory usage under real-time constraints. We presented a method efficiently sharing RAM-banks without jeopardizing task deadlines, hence reducing the required memory resources.

Finally, we described our prototype FPGA based real-time kernel which, in contrast to others, is a hardware-only system with the OS functions entirely implemented in hardware. The prototype implements the MSDL approach, uses full device reconfiguration, and is hence suitable for most of today's reconfigurable devices.

## 9.2 Drawn Conclusions

From our work and results we draw the following conclusions:

**Practicability of real-time multitasking on RHDs:** Multitasking on RHDs is feasible, even under real-time conditions. However, several technical issues such as reconfiguration overheads, partial runtime reconfiguration support and state saving today still limit the practicability.

The overheads associated with the device reconfiguration are generally not negligible and present the strongest limitation. However, it can be successfully included within the scheduling conditions of all approaches. Simulation results indicate, that the overheads can almost be neglected, if the reconfiguration time is one order of magnitude (two orders in case of *global EDF*) smaller than the computation time of the shortest task. Commercial FPGAs have, depending on their size, reconfiguration times from some hundreds of micro seconds up to some tens of milliseconds. Considering such devices, our results suggest that hardware tasks should have computation times of at least about 5 milliseconds. Moreover, a great deal of research is on its way aiming to a reduction of the reconfiguration time. For example, multi-context architectures [31] store several configurations on-chip and allow rapid switching between them. Yet another approach are coarse-grained reconfigurable devices, based on ALU arrays or ALU/look-up table hybrids and bus-wide interconnects. These devices have reconfiguration times in the order of microseconds. Applying our scheduling techniques to such platforms renders hardware tasks feasibly with computation times as low as some tens of microseconds.

Partial reconfiguration is, if at all, only weakly supported by today's RHD architectures. Often, the tasks footprints have to be rectangularly shaped, spanning the entire device size in one dimension. Inhomogeneous resources make task relocation difficult. However, our scheduling algorithms deal with these limitations: The one-dimensional area model, used by *global EDF* and *partitioned EDF*, can be realized with today's devices. *Partitioned EDF* avoids task relocation and *MSDL* does not rely on partial reconfiguration at all.

Several reported prototypes of runtime environments prove the concept of hardware-multitasking, including task preemption. A working prototype for the *free space partitioned - time sharing* resource model used by *MSDL*, which realizes state saving via dedicated task interfaces, has been presented in this work.

**The algorithms:** All three approaches generate schedules with predictable task timings and achieve an acceptable high device utilization.

*Global EDF* achieves the highest average performance. While this makes it a promising approach, only a fraction of this performance can be guaranteed by our schedulability test. Moreover, the approach requires task relocation and suffers greatly from reconfiguration overheads. Hence, global EDF may only be useful in several special cases where the task sets are small such that the hyper-period can be simulated or cases where online acceptance tests are required. This may change, if further research comes up with less pessimistic schedulability tests.

*Partitioned EDF* combines good performance with low reconfiguration overheads and few requirements to the execution model. The optimal partitioning can be computed in reasonable time for task sets up to 30 tasks and even more. Hence, it should be the chosen method if the system supports partial reconfiguration.

*MSDL* has the lowest performance, but enables multitasking for system that do not support partial reconfiguration.

**Problem relations to multiprocessor systems:** The execution models used for RHDs share many relations to multiprocessor models. In particular, our resource model presents a generalization of a simple multiprocessor model. Our work showed that several scheduling approaches used in multiprocessor environments can be extended for the RHD model. This suggests, that potentially even more ideas from multiprocessor scheduling methods could be used to develop further RHD scheduling algorithms.

**Importance of alternative task implementations:** We can conclude from our results that considering alternative task implementations generally yields a better scheduling performance. We proved this in particular for the partitioned scheduler, where we computed the optimal selection of alternatives via an ILP.

## 9.3 Outlook and Future Work

This work presented key methods for periodic real-time scheduling on RHDs. Since the topic has been only rarely addressed before, we developed only basic methods for simple task models. Hence, the work opens up a great field for future research. At this point we suggest four concrete lines of future work. (More details can be found in the according chapters.)

**Improvement of presented schedulers:** One direction of future work should concentrate on the improvement of the presented methods. For *global EDF*, better schedulability conditions could be developed, based on their multiprocessor counterparts [11]. For the server based scheduler, the presented *MSDL* algorithm involves greedy heuristic choices for the selection of servers to merge. Future work should evaluate other approaches in order to improve the scheduling performance.

For the extended model which considers alternative task implementations, only the *optimal-partitioned-EDF* method has been considered. How task variants can be considered in *global EDF* and *MSDL* has been roughly outlined but needs to be evaluated in future work. Another idea is the development of approximation algorithms that compute partitioned EDF schedules even for large task sets.

**Non-preemptive multitasking:** Scheduling periodic real-time onto RHDs in the non-preemptive case has not been addressed yet. It is useful since the reconfiguration overheads are reduced and all implementation issues of task interruption are avoided. Non-preemptive schedulers generally achieve less device utilization than preemptive schedulers, but the effect of reduced reconfiguration overhead could compensate that drawback.

**Generalizations of the model:** Throughout this work, we assumed that the relative task deadlines $D_i$ are equal to periods $P_i$. However, the basic idea of our approaches should also work with deadlines less than periods. The multiprocessor EDF schedulability test of [48] has been successfully extended to this case in [16], hence we believe we can do the same for our *global EDF* schedulability test. For *partitioned EDF*, we cannot use the simple time utilization EDF bound $U^T \leq 1$ for each partition anymore. Rather, we should replace it by a *processor demand* criteria [25], p.101. This can be easily done for the next fit heuristic, but since this leads to non-linear scheduling conditions the ILP approach will not work anymore.

Moreover, in our model the required task resources are simply modelled by a scalar area value. Future work can extend this to resource vectors, e.g. to cover FPGA resources such as the number of LUTs, FFs, block-RAMs and multiplier-blocks.

**Implementation of prototypes:** This work presented a prototype implementing the *MSDL* scheduler. For *global EDF* and *partitioned EDF*, prototypes could be built on any system supporting the 1D partial reconfiguration model.

In summary, this thesis presented new fundamental work in real-time multitasking on RHDs. While we focused on the development and analysis of scheduling algorithms for periodic tasks, we moreover presented a prototypical system realization. Perhaps just as important this work also showed that much more research is required to utilize the promising runtime reconfiguration capabilities of RHDs in next generation embedded systems. We hope that our work motivates more research is this exciting new area.

# Author's Publications

[BDAT03] Christophe Bobda, Klaus Danne, Ali Ahmadinia, and Jürgen Teich. A new approach for reconfigurable massively parallel computers. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'03)*. IEEE, 15 - 17 December 2003.

[BDAT04] Christophe Bobda, Klaus Danne, Ali Ahmadinia, and Jürgen Teich. Generation of distributed arithmetic designs for reconfigurable applications. In *ARCS 2004 Organic and Pervasive Computing, Workshop Proceedings*, volume P-41, Bonn, 26 March 2004. GI LNI, Köllen Verlag.

[BDD$^+$03] M. Bednara, Klaus Danne, Markus Deppe, Oliver Oberschelp, Frank Slomka, and Jürgen Teich. Design and implementation of digital linear control systems on reconfigurable hardware. *EURASIP Journal on Applied Signal Processing*, 6(6):594–602, May 2003.

[BDL03] Christophe Bobda, Klaus Danne, and André Linarth. Efficient implementation of the singular value decomposition on a reconfigurable system. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL 2003)*, Lisbon, Portugal, September 2003.

[Dan04a] Klaus Danne. Distributed arithmetic FPGA design with online scalable size and performance. In *Proceedings of 17th SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI04)*, pages 135–140. ACM Press, New York, NY, USA, 7 - 11 September 2004.

[Dan04b] Klaus Danne. Memory management to support multitasking on FPGA based systems. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig04) ISBN 970-692-169-9*. Mexican Society of Computer Science, SMCC, 20 - 21 September 2004.

[Dan04c] Klaus Danne. Operating systems for FPGA based computers and their memory management. In *ARCS 2004 Organic and Pervasive Computing, Workshop Proceedings*, volume P-41 of *GI-Edition Lecture Notes in Informatics (LNI)*, Bonn, 26 March 2004. Köllen Verlag.

[DB04] Klaus Danne and Christophe Bobda. Dynamic reconfiguration of distributed arithmetic controllers: Design space exploration and trade-off analysis. In

*Proceedings of the 11th Reconfigurable Architectures Workshop (RAW'04)*. IEEE Computer Society, IEEE Computer Society Press, 26 - 27 April 2004.

[DB06] Klaus Danne and Christophe Bobda. Dynamic reconfiguration of distributed arithmetic designs. *International Journal of Embedded Systems 2006 - Vol. 2, No.1/2 pp. 51 - 61*, 2(1/2):51–61, 2006.

[DBK03a] Klaus Danne, Christophe Bobda, and Heiko Kalte. Increasing efficiency by partial hardware reconfiguration: Case study of a multi-controller system. In Toomas Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, Nevada, USA*, 23 - 26 June 2003.

[DBK03b] Klaus Danne, Christophe Bobda, and Heiko Kalte. Run-time exchange of mechatronic controllers using partial hardware reconfiguration. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL2003), Lisbon, Portugal*, September 2003.

[DMP06] Klaus Danne, Roland Muehlenbernd, and Marco Platzner. Executing hardware tasks on dynamically reconfigurable devices under real-time conditions. In *Proceedings of the FPL06*, 2006.

[DP05a] Klaus Danne and Marco Platzner. A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL05)*, Tampere, Finland, 24 - 26 August 2005. Piscateway, NJ: IEEE.

[DP05b] Klaus Danne and Marco Platzner. Memory-demanding periodic real-time applications on FPGA computers. In *Work-in-Progress proceedings of the 17th EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS (ECRTS 05)*, 6 - 8 July 2005.

[DP05c] Klaus Danne and Marco Platzner. Periodic real-time scheduling for FPGA computers. In Volker Turau and Christophe Weyer, editors, *The Third IEEE International Workshop on Intelligent Solutions in Embedded Systems (WISES'05) at Hamburg University of Technology*. Piscateway, NJ: IEEE, 20 May 2005.

[DP06a] Klaus Danne and Marco Platzner. An edf schedulability test for periodic tasks on reconfigurable hardware devices. In *Proceedings of LCTES 2006, Ottawa*, 2006.

[DP06b] Klaus Danne and Marco Platzner. Partitioned scheduling of periodic real-time tasks onto reconfigurable hardware. In *International Parallel and Distributed Processing Symposium (IPDPS'06), Reconfigurable Architecture Workshop (RAW'06)*, 2006.

[DS05] Klaus Danne and Sven Stühmeier. Off-line placement of tasks onto reconfigurable hardware considering geometrical task variants. In *Proceedings of International Embedded Systems Symposium 2005 (IESS05)*, Manaus, Brazil, 15 - 17 August 2005.

[GDBS04] Jens Gerling, Klaus Danne, Christophe Bobda, and J. Schrage. Distributed arithmetics for recursive convolution of optical intercannects. In *EOS Topical Meeting, Optics in Computing (OIC)*, pages 65–66, Engelberg (Switzerland), April 2004.

# Bibliography

[1] Mobileye vision technologies ltd. http://www.mobileye.com.

[2] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *FPGA*, pages 3–12, 2000.

[3] Altera. Stratix II performance and logic efficency analysis. altera white paper v1.0, February 2004.

[4] Altera. Altera end markets. http://www.altera.com/end-markets/, 2006. This is an electronicdocument. Date retrieved: July 13, 2006.

[5] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, pages 337–346, 2000.

[6] R. Andraka. A survey of cordic algorithms for fpgas. In *FPGA '98. Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200, 1998.

[7] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming models for hybrid fpga-cpu computational components: A missing link. *IEEE Micro*, 24(4):42–53, 2004.

[8] J. Axelsson. A case study in heterogeneous implementation of automotive real-time systems. In *6th International Workshop on Hardware/Software Co-Design (CODES)*. Volvo Technological Development, 1998.

[9] D. Babbar and P. Krueger. On-line hard real-time scheduling of parallel tasks on partitionable multiprocessors. In *ICPP*, pages 29–38, 1994.

[10] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *RTSS*, pages 120–129, 2003.

[11] T. P. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. Technical Report TR-051101, Florida State University Dept. of Computer Science Tallahassee, FL 32306 USA, 2005.

[12] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[13] V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. Pact xpp - a self-reconfigurable data processing architecture. In *ERSA*, Las Vegas, Nevada, June 2001.

[14] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, pages 68–83, Mar. 2000.

[15] M. Berkelaar. *lp_solve - solve (mixed integer) linear programming problems*. Eindhoven University of Technology, 4.0 edition, 2002.

[16] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *proceedings of the 17th ERTCS, Palma de Mallorca, July 5-8 2005*, Palma de Mallorca, Spain, July 2005. IEEE CS.

[17] E. Bini, G. C. Buttazzo, and G. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *IEEE Proceedings of the Euromicro Conference on Real-Time Systems*, pages 59–66, Delft, The Nederlands, June 2001.

[18] C. Bobda. *Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement.* Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, 2003. Euro 35,-, ISBN 3-935433-37-9.

[19] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen. Dynoc: A dynamic infrastructure for communication in dynamically reconfigurable devices. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, pages 153–158, Tampere, Finland, Aug. 2005.

[20] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich. Increasing the flexibility in fpga-based reconfigurable platforms: The erlangen slot machine. In *IEEE 2005 Conference on Field-Programmable Technology (FPT05)*, pages 37–42, December 11-14 2005.

[21] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, J. Teich, S. P. Fekete, and J. van der Veen. The erlangen slot machine: A highly flexible FPGA-based reconfigurable platform. In *Proceeding 2005 IEEE Symposium on Field-Programmable Custom Computing Machines April 17 - April 20, 2005*, 2005.

[22] G. Brebner and O. Diessel. Chip-based Reconfigurable Task Management. In *Field-Programmable Logic and Applications (FPL'01)*, pages 182–191. Springer-Verlag, Berlin, Germany, 2001.

[23] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. Assigning real-time tasks to homogeneous multiprocessor systems. Technical report, University of Virginia, Charlottesville, VA, USA, 1994.

[24] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (second editon)*. Addison-Wesley, second edition edition, 1997.

[25] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications (second edition)*. Kluwer Academic Publishers, 2005.

[26] G. C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.

[27] Celoxica Inc., www.celoxica.com. *Celoxica RC 300 Datasheet*, v 1.2 edition, 2005.

[28] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2:181 – 194, 1990.

[29] E. G. Coffman, M. R. Garey, and D. S. Johnson. *Approximation Algorithms for NP-Hard Problems*, chapter Approximation Algorithms for Bin Packing: a Survey, pages 46–93. PWS Publishing, 1996.

[30] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal of Computing*, 9(4):808–826, November 1980.

[31] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[32] Continental. The new generation of adaptive cruise control acc from wabco and continental temic for trucks and busses. Continental Press Release, September 2004.

[33] A. Cosoroaba and F. Rivoallon. *Achieving Higher System Performance with the Virtex-5 Family of FPGAs: White Paper 245*. XILINX, v1.1.1 edition, July 2006.

[34] CRAY Inc. *Cray XD1 Datasheet*, 1.3 edition, 2005.

[35] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, 1996.

[36] M. L. Dertouzos. Control robotics: the procedural control of physical processes. *Information Processing*, 1974.

[37] O. Diessel and H. ElGindy. Run-time compaction of FPGA designs. In *7th International Conference on Field Programmable Logic and Applications*, pages 131–140, Berlin, Germany, 1997.

[38] O. Diessel and H. ElGindy. On scheduling dynamic FPGA reconfigurations. Technical report, University of South Australia, 1998.

[39] 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.

[40] O. F. Diessel. *On Scheduling Dynamic FPGA Reconfiguration, A Partial Rearrangement Approach.* PhD thesis, The University of Newcastle, Australia, 1998.

[41] H. ElGindy, M. Middendorf, B. Scheuermann, and H. Schmeck. An evolutionary approach to dynamic task scheduling on FPGAs. In Hartenstein and Gr?nbacher, editors, *Field-Programmable Logic and Applications (FPL'00).* Springer-Verlag, Berlin, September 2000.

[42] S. P. Fekete, E. Köhler, and J. Teich. Extending partial suborders. In *Proc. CTW2001, 1st CologneTwente Workshop on Graphs and Combinatorial Optimization.* published as Electronical Notes in Discrete Mathematics (ENDM), June 6-8 2001.

[43] S. P. Fekete, E. Köhler, and J. Teich. Higher-dimensional packing with order constraints. In *Proc. WADS2001, 7th Workshop on Algorithms and Data Structures*, Providence, Rhode Island, U.S.A., August 8-10 2001.

[44] S. P. Fekete, E. Köhler, and J. Teich. Optimale FPGA module placement with temporal precedence constraints. In *Proc. DATE 2001, Design, Automation and Test in Europe*, pages 658–665, Munich, Germany, March 13-16 2001. IEEE Computer Society Press.

[45] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, pages 1–19, 2006.

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

[47] J. Goossens, S. Baruah, and S. Funk. Real-time scheduling on multiprocessor. In *Proceedings of the 10th International Conference on Real-Time System*, 2002.

[48] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, 2003.

[49] S. Govindarajan and R. Vemuri. Tightly integrated design space exploration with spatial and temporal partitioning in sparcs. In *FPL*, pages 7–18, 2000.

[50] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[51] W. A. Horn. Some simple scheduling algorithms. *Naval Res. Logist. Quart.*, 21:177–185, 1974.

[52] E. Horta and J. W. Lockwood. PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs). Technical Report WUCS-01-13, Washington University in Saint Louis, Department of Computer Science, July 6, 2001.

[53] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Design Automation Conference (DAC)*, New Orleans, LA, June 2002.

[54] M. B. I. Gat and A. Shashua. Monocular vision advance warning system for the automotive. In *Intelligent Vehicle Initiative (VI) Technology 2005 - Advanced Controls and Navigation Systems Intelligent Vehicle Initiative (VI) Technology 2005 - Advanced Controls and Navigation Systems*. SAE, 2005.

[55] X. Inc. Xilinx marketing solutions. http://www.xilinx.com/esp/, year unknown. This is an electronic document. Date retrieved: July 13, 2006.

[56] IPFlex. *DAPDNA-2 Dynamically Reconfigurable Processor (0412-DD2E)*. IPFlex, http://www.ipflex.com, 2004.

[57] S. N. Jason Scott and T. Bapty. Runtime environment for dynamically reconfigurable embedded systems. Technical report, Vanderbilt University Nashville, TN 37325, U.S.A., 1999.

[58] J. S. N. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook. Dynamic reconfiguration to support concurrent applications. *IEEE Trans. on Computers*, 48(6):591–602, June 1999.

[59] H. Kalte, M. Köster, B. Kettelhoit, M. Porrmann, and U. Rückert. A comparative study on system approaches for partially reconfigurable architectures. In T. Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, pages 70–76, Las Vegas, Nevada, USA, 21 - 24 June 2004.

[60] H. Kalte, G. Lee, M. Porrmann, and U. Rückert. REPLICA: a bitstream manipulation filter for module relocation in partial reconfigurable systems. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium - Reconfigurable Architectures Workshop*, 2005.

[61] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *15th International Conference on Field Programmable Logic and Applications*, pages 223–228, 24 - 28 Aug. 2005.

[62] H. Kalte, M. Porrmann, and U. Rückert. A prototyping platform for dynamically reconfigurable system on chip designs. In *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*, Hamburg, Germany, 2002.

[63] M. Koester, H. Kalte, and M. Porrmann. Task placement for heterogeneous reconfigurable architectures. In *Proceedings of the IEEE 2005 Conference on Field-Programmable Technology (FPT'05)*, pages 43–50. IEEE Computer Society, December 11-14 2005.

[64] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluewer, 1997.

[65] O.-H. Kwon, J. Kim, S. J. Hong, and S. Lee. Real-time job scheduling in hypercube systems. In *Proceedings of ICPP07*, 1997.

[66] S. Lauzac, R. Melhem, and D. Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proceedings of the 10th Euromicro Workshop on Real Time Systems*, page 188, 1998.

[67] W. Y. Lee, S. J. Hong, and J. Kim. On-line scheduling of scalable real-time tasks on multiprocessor systems. *J. Parallel Distrib. Comput.*, 63(12):1315–1324, 2003.

[68] J. Lenstra, A. R. Kan, and . P. Brucker. Ann. of Discrete Math., 1:343-362. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 7(1):343–362, 1977.

[69] L. Levinson, R. Männer, M. Sessler, and H. Simmler. Preemptive multitasking on FPGAs. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 301. IEEE Computer Society, 2000.

[70] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44(12):1429–1442, 1995.

[71] D. Lim and M. Peattie. *Xilinx Application Note XAPP290: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, May 2002.

[72] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[73] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141:241–252, 202.

[74] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgeford. Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration on xilinx fpgas. In *In proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL)*, pages 12–17, Madrid, Spain, August 2006. IEEE.

[75] G. D. Micheli and M. Sami. Hardware-software codesign. In *Proceedings of the IEEE Volume 85, Issue 3*, pages 349 – 365, 1997.

[76] P. Mohapatra. Dynamic real-time task scheduling on hypercubes. *Journal of Parallel and Distributed Computing*, pages 91–100, 1997.

[77] Y. Oh. *The Design and Analysis of Scheduling Algorithms for Real-Time and Fault-Tolerant Computer Systems.* PhD thesis, University of Verginia, 1994.

[78] Y. Oh and S. H. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical report, University of Virginia, 1995.

[79] R. Pellizzoni and M. Caccamo. Adaptive real-time management of relocatable tasks for fpga-based embedded systems technical report, university of illinois at urbana-champaign, 20. Technical report, University of Illinois at Urbana-Champaign, 2005.

[80] R. Pellizzoni and M. Caccamo. Adaptive allocation of software and hardware real-time tasks for fpga-based embedded systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages pp. 208–220, 2006.

[81] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation in 29 th ann. In *In 29 th Ann. ACM Symp. on Theory of Computing*, pages 140–149", El Paso, Texas, United States, 1997.

[82] L. Sha, T. F. Abdelzaher, K.-E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[83] H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA coprocessors. In *FPL*, pages 121–130, 2000.

[84] H. C. Simmler. *Preemptive Multitasking auf FPGA-Prozessoren.* PhD thesis, Inauguraldissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften der Universität Mannheim, 2001.

[85] H. K.-H. So and R. W. Brodersen. Improving usability of fpga-based reconfigurable computers through operating system support. In *16th International Conference on Field Programmable Logic and Applications (FPL '06)*, 2006.

[86] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[87] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Softw.*, 8(3):62–72, 1991. See Buttazzo Book.

[88] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, 1995.

[89] C. Steiger, H. Walder, and M. Platzner. Heuristics for online scheduling real-time tasks to partially reconfigurable devices. In *In Proceedings of the 13rd International Conference on Field Programmable Logic and Applications (FPL03)*, pages 575–584. September, 2003.

[90] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1392–1407, November 2004.

[91] C. Steiger, H. Walder, M. Platzner, and L. Thiele. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In *Proceedings of 24th IEEE International Real-Time Systems Symposium (RTSS'03)*, 2003.

[92] J. Teich. *Digitale Hardware/ Software-Systeme. Synthese und Optimierung.* Springer, 1997.

[93] J. Teich, S. Fekete, and J. Schepers. Compile-time optimization of dynamic hardware reconfigurations. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1097–1103, Las Vegas, U.S.A., June 1999.

[94] J. Teich, S. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *The J. of Supercomputing*, 19(1):57–75, May 2000.

[95] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *The Journal of VLSI Signal Processing*, 28(1-2):7–27, May 2001.

[96] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *IEEE Symp. on FPGAs and Custom Computing Machines (FCCM)*, pages 22–28, 1997.

[97] M. Ullmann, M. Hubner, B. Grimm, and J. Becker. An fpga run-time system for dynamical on-demand reconfiguration. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Reconfigurable Architectures Workshop*, 2004.

[98] H. Walder, S. Nobs, and M. Platzner. Xf-board: A prototyping platform for reconfigurable hardware operating systems. In *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms (ERSA)*, 2004.

[99] H. Walder and M. Platzner. Non-preemptive multitasking on FPGAs:task placement and footprint transform. Technical report, Swiss Federal Institute of Technology, ETH, 2002.

[100] H. Walder and M. 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)*, pages 284–287. CSREA Press, June 2003.

[101] H. Walder and M. Platzner. Reconfigurable hardware os prototype. Technical Report Technical Report TIK Nr. 168, Swiss Federal Institute of Technology (ETH), April 2003.

[102] H. Walder and M. Platzner. A runtime environment for reconfigurable operating systems. In *In Proceeding 14th International Conference on Field-Programmable Logic and Applications (FPL)*, 2004.

[103] H. H. Walder. *Operating system design for partially reconfigurable logic devices.* (tik-schriftenreihe ; nr. 68/2, diss nr. 15955), ETH Zuerich, 2005.

[104] T. Wiangtong, P. Cheung, and W. Luk. A unified codesign run-time environment for the ultrasonic reconfigurable computer. In *Field-Programmable Logic and Applications (FPL'03)*. p-springer, september 2003.

[105] G. Wigley and D. Kearney. The Development of an Operating System for Reconfigurable Computing. In *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE CS Press, Los Alamitos, CA, USA, 2001.

[106] Xilinx. *Virtex 2.5 V Field Programmable Gate Arrays*, April 2001. http://www.xilinx.com.

[107] Xilinx. *XILINX LogicCore 1-D Discrete Cosine Transformation V2.1*, 2002.

[108] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGA: Functional Description*, 2005. http://www.xilinx.com.