Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks

Christoph Steiger, Herbert Walder, *Member*, *IEEE*, and Marco Platzner, *Member*, *IEEE*

Abstract—Today's reconfigurable hardware devices have huge densities and are partially reconfigurable, allowing for the configuration and execution of hardware tasks in a true multitasking manner. This makes reconfigurable platforms an ideal target for many modern embedded systems that combine high computation demands with dynamic task sets. A rather new line of research is engaged in the construction of operating systems for reconfigurable embedded platforms. Such an operating system provides a minimal programming model and a runtime system. The runtime system performs online task and resource management. In this paper, we first discuss design issues for reconfigurable hardware operating systems. Then, we focus on a runtime system for guarantee-based scheduling of hard real-time tasks. We formulate the scheduling problem for the 1D and 2D resource models and present two heuristics, the *horizon* and the *stuffing* technique, to tackle it. Simulation experiments conducted with synthetic workloads evaluate the performance and the runtime efficiency of the proposed schedulers. The scheduling performance for the 1D resource model is strongly dependent on the aspect ratios of the tasks. Compared to the 1D model, the 2D resource model is clearly superior. Finally, the runtime overhead of the scheduling algorithms is shown to be acceptably low.

Index Terms—FPGA, partial reconfiguration, operating system, online scheduling, real-time.

1 Introduction

Embedded computing platforms are composed of a variety of different processing elements, memories, I/O devices, sensors, and actors. The choice of processing elements includes instruction-set processors, application-specific fixed-function hardware (ASICs), and (re)configurable hardware devices. SRAM-based reconfigurable devices, typically field-programmable gate arrays (FPGAs), store the hardware configuration in static memory cells. These devices can be reconfigured arbitrarily often with reconfiguration times on the order of milliseconds.

SRAM-based FPGAs were introduced as high-end devices for implementing random logic in the mid-1980s. In the following years, FPGAs have found a number of additional and novel uses in the design of embedded systems. One example is rapid prototyping and emulation. More recently, reconfigurable hardware is being used as an ASIC replacement with much shorter time-to-market and the novel ability to update hardware after product deployment. Today, the increasing densities of reconfigurable devices and the trend to integrate them with processors, memories, and special function blocks on configurable systems on a chip (CSoC) [1], [2] advocate more *dynamic uses* of reconfigurable hardware in embedded systems. A higher degree of dynamics is further facilitated by partial reconfiguration, a technique that allows us to reconfigure only a fraction of the reconfigurable hardware resources while the other part continues to execute [3].

Many promising application domains for *reconfigurable embedded systems* combine high performance demands with frequent changes of their workloads. Examples for such application domains are found in wearable computing [4], mobile systems [5], and network processors [6]. The dynamics in these systems are caused by user requests and packet flows in the communication networks. Mobile and wearable systems additionally operate in changing physical environments and contexts, which reflects in different types of workloads. Consequently, neither the set of functions nor the time at which these functions will be executed is exactly known in advance. A classic system design process with complete design-time synthesis and optimization is no longer possible. The required degree of flexibility paired with high computation demands asks for partially reconfigurable hardware that is operated in a true multitasking manner.

Multitasking reconfigurable hardware raises a number of novel issues, ranging from programming models to runtime systems. A programming model defines the executable objects and their interaction and provides the developer with a set of well-defined system services. A runtime system efficiently operates the system and resolves conflicts between executable objects. A programming model, together with a runtime system, forms a *reconfigurable hardware operating system* [7], [8]. A reconfigurable hardware operating system can be compared to embedded real-time operating system (RTOS) kernels for microprocessors. Such kernels have been industry standard for years and offer a minimal programming model by specifying a set of objects, e.g., tasks, buffers, semaphores, timers, and their possible

- *C. Steiger is with ESA/ESOC, Robert-Bosch-Strasse 5, 64293 Darmstadt, Germany. E-mail: Christoph.Steiger@esa.int.*
- *H. Walder and M. Platzner are with the Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, 8092 Zurich, Switzerland. E-mail: walder@tik.ee.ethz.ch, marco.platzner@computer.org.*

Manuscript received 1 Dec. 2003; revised 11 Apr. 2004; accepted 8 June 2004. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0251-1203.

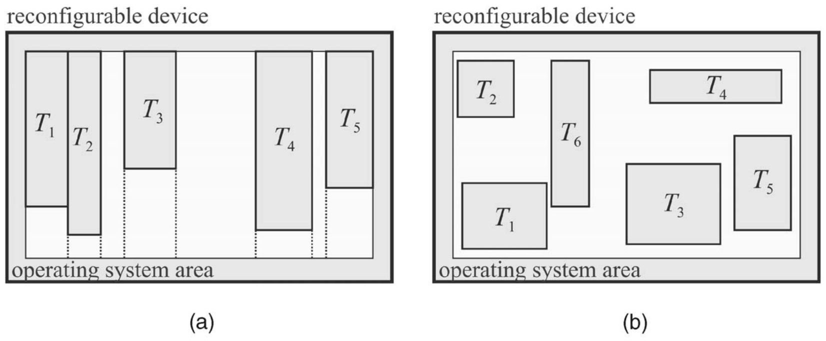


Fig. 1. Reconfigurable resource models: (a) 1D area model; (b) 2D area model.

interactions. The main abstraction is the introduction of a smallest unit of execution, which is mostly denoted as task (alternatively, the smallest unit of execution is called process or thread).

Our long-term goal is to develop an operating system for reconfigurable embedded platforms by building the same abstractions for partially reconfigurable hardware as an RTOS builds for microprocessors. Besides enabling *hardware multitasking*, the benefits of such an operating system are increased productivity and portability. The abstractions of system objects and tasks enable the reuse of tested and reliable code and circuitry, which can considerably speed up development cycles and shorten time-to-market. Application ports are greatly simplified, provided the operating system is available on different target platforms. Eventually, all processing elements of an embedded reconfigurable platform should be managed by one operating system that deals with software tasks running on a microprocessor and hardware tasks running on reconfigurable hardware. Tasks that are available for both software and hardware facilitate dynamic system repartitioning [9].

Reconfigurable hardware operating systems are a rather new line of research, where only a few issues have been addressed yet. In this paper, we first discuss conceptual and practical aspects of reconfigurable hardware operating system design. Then, we concentrate on the runtime functions of such an operating system. The main contribution of this paper is the development and evaluation of heuristics for online scheduling of hard real-time tasks to partially reconfigurable devices. The proposed scheduling heuristics are applicable to both commonly used reconfigurable resource models, the 1D and 2D area model.

The remainder of this paper is structured as follows: In Section 2, the basic operating system abstractions—task and resource models—are presented, followed by a survey of related work and a discussion of the limitations of current FPGA technology. The design of our reconfigurable hardware operating system is outlined in Section 3. Section 4 defines the online scheduling problem and presents two heuristics to solve it. An experimental evaluation of the proposed heuristics based on synthetic workloads is done in Section 5. Finally, Section 6 concludes the paper.

2 Models and Limitations

This section first presents the basic models for hardware tasks and reconfigurable devices that are being used for conceptualizing and constructing operating systems for reconfigurable platforms. Then, we review related work in the area. Finally, we discuss the limitations of task and resource models when it comes to practical implementation in currently available technology.

2.1 Basic Task and Device Models

A hardware task is a synthesized digital circuit that has been preplaced and prerouted. The task is stored in a position-independent way and can be relocated to different locations on the reconfigurable device by the operating system. Hardware tasks have several characteristics. The functional characteristic captures the task behavior and is not visible to the operating system. The structural and timing characteristics, however, are exposed to the runtime system for scheduling and placement.

The main structural characteristics are size (area) and shape. Hardware tasks have a certain area requirement, given in numbers of reconfigurable units (RCUs). The shape of a hardware task is mostly modeled by a rectangle including all RCUs as well as the routing resources used by the task. Compared to more complex shapes, such as polyominoes [10], [11], rectangular shapes simplify task placement. However, rectangular shapes also lead to *internal fragmentation*, i.e., the unused area inside the rectangle.

The main timing characteristic of a hardware task is the clock range at which the task can run. Design tools usually report an upper bound for the clock rate. A task may, however, require a specific clock rate, for example, to derive a timer object that relates events to physical time. A task might further require a clock rate in a certain interval to preserve timing requirements of I/O devices or memory. Further timing characteristics, that might or might not be known in advance, are the number of clock cycles to execute and the deadline of a real-time task.

The complexity of mapping tasks to devices depends heavily on the *area model* used. The two main area models are the 1D and the 2D model shown in Fig. 1. In both models, the reconfigurable device is represented by a rectangular area of RCUs. A part of the reconfigurable resource is reserved for operating system functions (denoted as operating system area in Fig. 1). The remaining part is the hardware task area. In the simpler 1D area model, tasks can be allocated anywhere along the horizontal device dimension; the vertical dimension is fixed and spans the total height of the hardware task area. The 1D area model leads to simplified scheduling and placement problems. However, the model suffers from two types of *external fragmentation*. The first type of external fragmentation is the area wasted when a task does not utilize the full height of the task area. The second type of fragmentation arises when the remaining free area is split into several small but unconnected vertical stripes. External fragmentation can prevent the placement of further tasks although sufficient free area exists.

The more complex 2D area model allows us to allocate tasks anywhere on the hardware task area and suffers less from external fragmentation. Consequently, a higher device utilization can be expected. On the other hand, the high flexibility of this model makes scheduling and placement rather involved.

Formally, a task T_i is modeled as a rectangular area of reconfigurable units given by its width and height, $w_i \times h_i$. Tasks arrive at arbitrary times a_i. Real-time tasks require execution times e_i and carry deadlines d_i, $d_i \geq a_i + e_i$. The hardware task area of the reconfigurable device is modeled as a rectangular area $W \times H$ of reconfigurable units.

2.2 Related Work

A substantial body of work has been done in offline optimization of reconfigurable embedded systems. Examples are temporal partitioning for nonpartially reconfigurable systems, e.g., by Purna and Bhatia [12], or 3D placement of tasks in time and space dimensions for partially reconfigurable devices, e.g., by Fekete et al. [13]. In an offline scenario, one can afford to spend the time to derive optimal or near-optimal solutions. In contrast, operating systems work on online scenarios and require efficient algorithms.

Brebner [7], [14] was among the first to propose an operating system approach for partially reconfigurable hardware. He defines swappable logic units (SLUs), which are position-independent tasks that are swapped in and out by the operating system. Jean et al. [15] discuss an online scenario where a resource manager schedules arriving tasks to a farm of FPGAs. Since each task occupies exactly one FPGA, there is no partial reconfiguration and placement is not an issue. Merino et al. [16], [17] split the reconfigurable surface into an array of predefined subareas, so-called slots. The operating system schedules tasks to these slots based on a task allocation table that keeps track of currently loaded tasks. As each task fits into one slot, there is again no placement problem involved. Simmler et al. [18] discuss task switching in a preemptive environment. The authors further propose critical sections, i.e., periods of time during which a task must not be preempted. However, their device is not partially reconfigured. Burns et al. [19] describe several operating system functions, including a 2D transform manager that performs translation and rotation operations on tasks to better fit them to the device. Shirazi et al. [20] propose three runtime modules, a monitor, a loader, and a configuration store, to manage reconfigurable designs. The authors discuss trade offs between reconfiguration time and circuit quality depending on the reconfiguration method used and information about the configuration sequence that is available at compile time.

Several authors have addressed the problem of external fragmentation and tackled it by compaction, i.e., rearranging executing tasks in order to maximize the contiguous free area. While these techniques increase the chance of successful future placements, compaction requires preemptive systems and can delay task execution for a considerable period of time. Diessel et al. [21], [22] investigate compaction in the 2D area model. They perform task rearrangement by techniques denoted as local repacking and ordered compaction. Compton et al. [10], [23] discuss task relocations and, additionally, task transforms to reduce fragmentation. Task transforms consist of a series of rotation and flip operations. The authors also propose a novel FPGA architecture that supports efficient row-wise relocation. A different compaction approach for the 1D area model is presented by Brebner and Diessel in [24], where an FPGA itself contains circuitry that determines the relocation positions.

The problem of placement in the 2D area model has been addressed by Barzagan et al. [25]. The authors investigate efficient data structures and algorithms for fast online placement. Simulation experiments for variants of first-fit, best-fit, and bottom-left bin-packing algorithms are conducted. The 2D placer we use in this paper relies on our previous improvements [26] of the work of Bazargan et al. Further, we have also experimented with the placement of tasks that consist of a number of rectangular subtasks [11]. Such shapes result naturally from a core-oriented design style and can be modified by footprint transforms, i.e., by arranging the subtasks differently.

Recent work in reconfigurable hardware operating systems has covered a broader range of runtime functionalities as well as prototype construction. Wigley and Kearney [27] identify a set of services that should be offered by reconfigurable operating systems, including partitioning, allocation, placement, and routing. Mignolet et. al [9] present a networked reconfigurable platform for multimedia appliances that enables multitasking in hardware and software. In [28], the authors discuss interconnection networks on reconfigurable devices. Design issues for a reconfigurable hardware operating system have been presented in [29] and a prototype implementation in [5].

Like some of the related work, the main part of this paper addresses online task and resource management for partially reconfigurable devices. This work is an extension of a previously published conference contribution [30]. The difference from related work is that we target a real-time scenario where each incoming task is either accepted with a guarantee to meet the deadline or rejected. For such a scenario, we compare different scheduling heuristics under both the 1D and 2D area models. The feasibility of the 1D area model is backed by our prototyping work on Xilinx Virtex technology. The 2D model is not enabled by currently available FPGA technology. Despite this fact, the 2D area model is very popular and has often been used in related work.

2.3 Limitations of Current Technology

The main abstraction is that tasks are modeled as relocatable rectangles that can be placed anywhere on the device in the 2D area model and anywhere along the horizontal device dimension in the 1D area model. While the latest FPGA design tools [31] allow us to constrain tasks to rectangular areas at the price of some internal fragmentation, the relocatability raises a number of questions. These questions concern the device homogeneity, task communication and timing, and the partial reconfigurability.

The placement and scheduling algorithms of the runtime system assume homogeneous reconfigurable devices, which is in contrast to modern FPGAs that contain special resources such as dedicated memories and embedded multipliers [1]. Although these special resources are

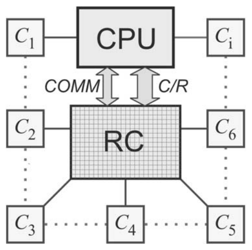


Fig. 2. Target architecture comprising a host CPU, a reconfigurable device (RC), and a number of external components C_i.

distributed over the array of logic blocks, they are not available at every location on the device. This limits the relocatability of hardware tasks. However, an operating system takes many of these resources (e.g., dedicated memories) away from the user tasks and puts them under operating system control. The operating system uses these memories to implement memory objects such as FIFO queues, mailboxes, and memory maps. Tasks must use predefined interfaces to access these objects [9], [8]. Further, the placement algorithms used in this paper can be easily extended to handle placement constraints, e.g., to relocate tasks at different levels of granularity or even to place some tasks at fixed positions. The basic problems and approaches will not change, but the resulting performance will.

Arbitrarily relocated tasks that communicate with each other and with I/O devices require online routing and online delay estimation of their external signals, neither of which is sufficiently supported by current tools. Some prototypes [9], [8] overcome this problem by resorting to a slightly different 1D area model that partitions the reconfigurable surface into a number of fixed-size slots [16]. The operating system provides predefined and static communication interfaces to these slots. In our recent prototype [32], we rely on a communication infrastructure that enables variable-sized slots. The problems of online delay estimation are avoided by using asynchronous communication protocols for task communication and I/O.

For the 2D area model, communication is an unresolved issue. Related work mostly assumes that sufficient resources for communication are available [22] or that tasks communicate via configuration and readback (which is feasible but presumably inefficient) or propose to leave some space between tasks and perform online routing of communication channels.

The partial reconfiguration capabilities of the Xilinx Virtex FPGA family, which reconfigures a device in vertical chip-spanning columns, perfectly fits the 1D area model. While the implementation of a somewhat limited 2D area model on the same technology seems to be within reach, ensuring the integrity of nonreconfigured device areas during task reconfiguration is cumbersome.

In summary, given current technology, the 1D area model is realistic, whereas the 2D model faces unresolved issues. Most of the related work on 2D area models targets the (meanwhile withdrawn) FPGA series Xilinx XC6200 that is reconfigurable on the level of single logic blocks and has a publicly available bitstream architecture. Requirements for future devices supporting the 2D area model include block-based reconfiguration and a built-in communication network that is not affected by user logic reconfigurations. As we will show in this paper, the 2D area model has great

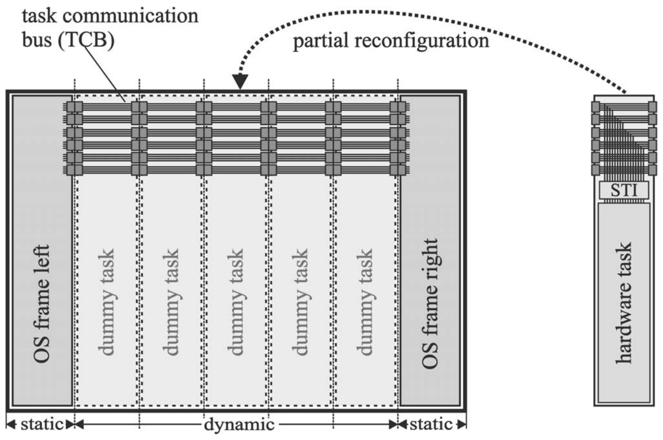


Fig. 3. Partitioning of the reconfigurable device into two static operating system frames and a dynamic hardware task area.

advantages over the 1D area model in terms of scheduling performance. For these reasons, we believe that it is worthwhile to investigate and develop algorithms for both the 1D and 2D area models.

3 Practical Operating System Design

This section briefly overviews our work toward a practical realization of a reconfigurable hardware operating system. We discuss the considered target architecture, the partitioning of the reconfigurable resource, and the main operating system modules. More detailed descriptions of our operating system prototypes can be found in [8], [32]. The primary purpose of this section is to show that the 1D area model is realistic in current FPGA technology. This validates the main part of this paper on scheduling real-time tasks.

3.1 Target Architecture

We consider the system architecture sketched in Fig. 2. A partially reconfigurable device is coupled to a host CPU by two bidirectional channels: The configuration and readback port (C/R) gives the CPU complete control over the configuration of the reconfigurable device, i.e., full and partial configurations can be downloaded to the device and full and partial device states can be read back. The communication port (COMM) provides a number of general-purpose signals between the host CPU and the reconfigurable device. This port is used for connecting the operating system functions mapped to the CPU and to the reconfigurable device. Both the CPU and the reconfigurable device further connect to a number of external components C_i, such as memories and I/O devices. The architectural concept of Fig. 2 can be implemented by distinct CPU and RC devices or as an integrated configurable system on a chip (CSoC).

3.2 Reconfigurable Resource Partitioning

We partition the reconfigurable area into two regions, the *operating system (OS) frames* and the *hardware task area*. The partitioning is shown in Fig. 3. The OS frames accommodate functions that constitute the runtime part of the operating system. These functions are required for task communication and I/O. The OS frames are static during the system's operation. As we target Xilinx Virtex technology, which is partially reconfigurable in vertical chip-spanning columns, the OS frames are positioned at the left and right device edges. The hardware task area accommodates the dynamically allocated and loaded hardware tasks.

Task communication and I/O is established via the task communication bus (TCB) that runs horizontally through all hardware tasks and connects to both OS frames. As the TCB must remain static during the system's operation, we divide the hardware task area into a number of dummy tasks, as indicated in Fig. 3. Dummy tasks are placeholders for hardware tasks. The width of a hardware task is an integer multiple of the width of a dummy task. Each hardware and dummy task has to implement a part of the TCB. Our design tool flow automatically generates the dummy tasks and provides the hardware task designer with a VHDL wrapper that contains the TCB structure. During the configuration of a hardware task, the other loaded hardware tasks remain executing. Technically, we implement the TCB by means of so-called bus macros [32], which constrains task relocatability to a 4-RCU grid. Except for this small modification, this area model matches the 1D area model discussed in Section 2.1.

Similar to a software task that calls kernel functions to make use of operating system services, a hardware task accesses a *standard task interface (STI)* to connect to the reconfigurable hardware operating system. The STI provides control and data ports. All hardware tasks that are managed by the operating system must implement the same STI.

For prototyping reconfigurable hardware operating systems, we have developed the XF-Board [33], a platform that employs a Xilinx Virtex-II XC2V3000 and a multitude of memory and I/O connected to the left and right device edges. The resource partitioning shown in Fig. 3 has been tested on this prototype. The complete FPGA can be reconfigured in $19.88\ ms$; the smallest possible hardware task is configured in $1.42\ ms$. The CPU is implemented as a MicroBlaze soft CPU core on a second FPGA, allowing for fast partial reconfiguration. Xilinx's *ISE Foundation* in combination with the *Modular Design* package [31] serve as development environment for the OS frames and the hardware tasks. Our prototype runs an embedded networking application that has been described in greater detail in [8]. The application is packet-based audio streaming of encoded audio data (12kHz, 16bit, mono) with an optional AES decoding. A receiver task checks incoming Ethernet packets and extracts the payload to FIFOs. Then, AES decryption and audio decoding tasks are started to decrypt, decode, and stream the audio samples. The task deadlines depend on the minimal packet interarrival time and the FIFO lengths.

3.3 Operating System Modules

Fig. 4 presents the set of modules that provide the operating system services. The modules are partitioned between the CPU and the reconfigurable device. The CPU modules can be categorized into three levels.

The highest level of modules is responsible for task and resource management. The *task scheduler* decides on a starting time for each task, based on some scheduling policy. Scheduling policies can be offline or online. An offline schedule is suitable for statically defined applications and is reflected by a rather simple task sequence table. Online schedulers are priority-driven and split into nonpreemptive and preemptive schedulers. The *resource manager* keeps track of all dynamically assigned resources, such as the hardware task area and memory. The main

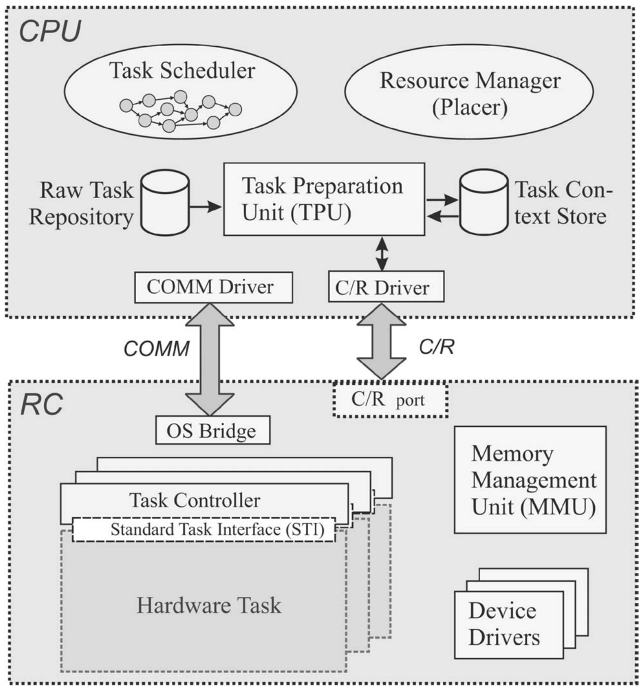


Fig. 4. Operating system modules mapped to the CPU and the reconfigurable device.

subfunction of the resource manager is the placer. Once the task scheduler decides to execute a task, the placer is responsible for providing a suitable free space in the hardware task area.

The intermediate level of operating system modules performs the handling of task bitstreams and tasks states. The *raw task repository* stores task circuits in their raw form, i.e., in a position-independent form which is generated by the task design flow. Before a raw task can be downloaded to the reconfigurable device, it must be relocated to a specific location in the hardware task area. The *context store* holds the task contexts that have previously been extracted from preempted tasks. The *task preparation unit (TPU)* generates and analyzes partial bitstreams that represent the tasks. The lowest level of operating system modules on the CPU deals with communication and configuration. Since both the CPU and the reconfigurable device accommodate parts of the operating system, a communication channel between the two devices is required. The *COMM driver* establishes this channel and allows the operating system modules to exchange commands and data. The *C/R driver* provides device-independent configuration and readback services to the TPU. The services comprise full and partial configuration as well as full and partial readback.

The operating system modules mapped to the reconfigurable device are comprised of the following units: The *task controller* is the operating system's counterpart of the standard task interface and forms the connection between operating system frames and hardware task area. There is one task controller assigned to each task. The *memory management unit (MMU)* offers memory services to the tasks, such as FIFO queues with specific access modes (blocking/nonblocking), private memory blocks, or shared memory blocks. The memory structures are implemented with internal block memories and externally connected memory devices. *Device drivers* implement circuitry to control I/O

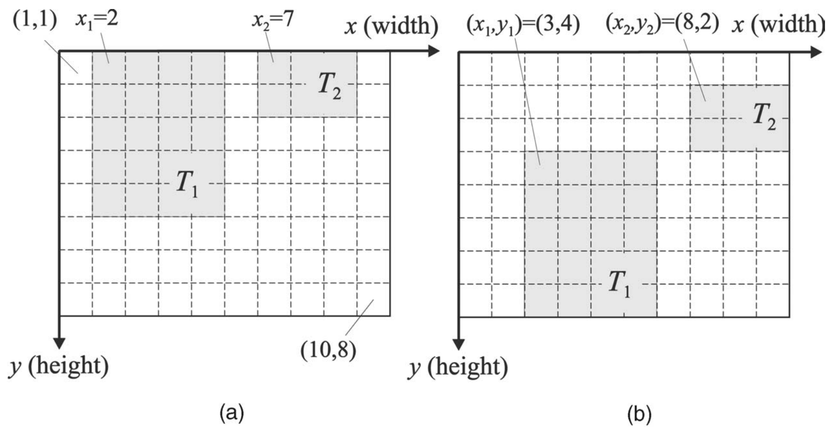


Fig. 5. Placement of rectangular tasks: (a) 1D area model; (b) 2D area model.

devices. Encapsulating I/O access in device drivers offers similar advantages as in software operating systems: The access functions are independent of the actual I/O device and mutual exclusion issues can be resolved. All the operating system modules on the reconfigurable device connect to the modules mapped to the CPU via the OS bridge.

4 Scheduling Real-Time Tasks

Task and resource management play a major role in any operating system. In partially reconfigurable hardware platforms, the problems of task and resource management are even more strongly connected than in processor-based platforms, which makes the design of runtime systems more challenging.

The task, device, and system models adopted in this paper are motivated by our prototyping work on reconfigurable hardware operating systems. Hardware tasks are described as rectangular areas of reconfigurable units (RCUs) with a given bound on the clock frequency. We assume that the tasks have known execution times and deadlines. The reconfigurable device is modeled as a rectangular area of RCUs. We consider both the 1D and 2D area models. While the feasibility of the 1D area model is validated by our prototyping work, the 2D area model is not supported by current technology. The proposed scheduling heuristics are applicable to both area models. Hence, we use the same performance metric—the number of accepted real-time tasks—for both models. The runtime system does not preempt running tasks. Although preemption of hardware tasks has been practically demonstrated, we omit such techniques here because configuration and readback times can easily become dominating when a task has either a short runtime or is preempted and resumed a number of times. In summary, the problem we address is the guarantee-based online scheduling of real-time tasks to partially reconfigurable, nonpreemptive platforms.

In the remainder of this section, we first define the 1D variant of the considered online real-time scheduling problem and present two fast heuristics to solve it. Then, we show the extension of these techniques to the 2D area model. After a discussion of the underlying placement techniques, we finally address the relationship of our scheduling problem to other online placement and scheduling problems.

4.1 The Online Scheduling Problem

A task is characterized by its arrival time a_i, execution time e_i, deadline d_i, width w_i, and height h_i. The online scheduler tries to find a placement and a starting time for a newly arrived task such that its deadline is met. In the 1D area model, a *placement* for a task T_i is given by the x coordinate of the leftmost task cell, x_i, with $x_i + w_i \leq W$. The *starting time* for T_i is denoted by s_i. Fig. 5a shows an example of a device $W \times H = 10 \times 8$ cells with two placed tasks. Task T_1 has the placement $x_1 = 2$; task T_2 is placed at $x_2 = 7$. We assume that the task's height does not exceed the device's height, $h_i \leq H$; otherwise, the task cannot not be executed. The main characteristic of scheduling to dynamically reconfigurable devices is that a scheduled task has to satisfy intertwined timing and placement constraints. We denote the set of scheduled tasks at any point in time with $\mathcal{T}$. Initially, there are no scheduled tasks, $\mathcal{T} = \emptyset$. A scheduled task can be defined as:

Definition 1 (Scheduled Task ID). *A* scheduled task T_i *is a task with a placement* x_i *and a starting time* s_i *such that:*

1. $\forall T_j \in \mathcal{T}, T_j \neq T_i$:

$$[(x_i + w_i) \leq x_j] \vee [x_i \geq (x_j + w_j)] \vee$$
$$[(s_i + e_i) \leq s_j] \vee [s_i \geq (s_j + e_j)]$$

(scheduled tasks must not overlap in space and time).

2. $s_i + e_i \leq d_i$ *(deadline must be met).*

We consider an online *hard real-time* system that runs an acceptance test for each arriving task. A task passing this test is scheduled and thus guaranteed to meet its deadline. A task failing the test is rejected by the scheduler in order to preserve the schedulability of the currently guaranteed task set. Accept/reject mechanisms are typically adopted in dynamic real-time systems [34] and assume that the scheduler's environment can react properly on a task rejection. A proper reaction is to migrate the task to a different computing resource that executes the task with an additional cost. The added cost can be seen as a penalty for the scheduler; the corresponding scheduling goal is to minimize the overall penalty. Such an approach has been used by [25]. We consider a different model where a rejected task leads to a denial of service. Applied to mobile and wearable computing scenarios, this model means that the requested functionality is not available at this time.

Instead of assigning penalties, we set the scheduling goal to minimize the number of rejected tasks.

Formally, the acceptance test is encapsulated into a scheduling function $\sigma(T_i)$:

$$\sigma(T_i) = \begin{cases} \texttt{accept} \; with \; (x_i, s_i) \\ \texttt{reject}. \end{cases} \qquad (1)$$

The simplest online scheduling method is to check whether a newly arrived task T_i finds an immediate placement, i.e., $s_i = a_i$. If there is none, the task is rejected. This crude technique will show low performance, but greatly reduces the complexity of the scheduler-placer interaction as it needs to know only about the current device allocation. Despite its poor performance, we include this method as a *reference method* in our experimentation.

Sophisticated online scheduling methods increase the acceptance ratio by *planning*, i.e., looking into the future. We may delay starting a task for its laxity (until $s_{i-latest} = d_i - e_i$) and still meet its deadline. The time interval $[a_i, s_{i-latest}]$ is the planning period for a task T_i. At each point in time, a planning scheduler manages two sets of tasks: the set of tasks currently executing on the device and the set of reserved tasks, i.e., tasks that have been previously guaranteed but not yet started. Planning techniques split into two groups. The first group are schedulers that may rearrange already guaranteed tasks to find guarantees for all these tasks plus the newly arrived one. Such scheduling problems are known to be NP-hard. Additionally, our scheduling policy poses the constraint that previously guaranteed tasks must not be rejected, i.e., if a task is to be rejected, it may only be the newly arrived one. The second group are schedulers that try to *fit* the newly arrived task into the set of already guaranteed tasks rather than attempting to rearrange them. Schedulers of the second group are potentially weaker in performance but can be expected to lead to more efficient algorithms. As we target online scenarios where the scheduling runtime is of importance, we focus on planning techniques of the second group. In the following sections, we discuss two such online techniques, the *horizon* technique and the *stuffing* technique.

4.2 The 1D Horizon Technique

As any planning scheduler, the horizon scheduler maintains an *execution* list and a *reservation* list. The execution list E contains all currently executing tasks T_i with their finishing times, $f_i = s_i + e_i$, and placements, x_i. The list entries, (T_i, x_i, f_i), are sorted in order of increasing finishing times. The reservation list R stores all scheduled but not yet executing tasks. The list entries (T_i, x_i, s_i) hold the placements and starting times for the scheduled tasks. The reservation list is sorted in order of increasing starting times.

In addition to the execution and reservation lists, the horizon technique employs a third list, the *scheduling horizon* H. The scheduling horizon consists of elements $h_i = ([x_1, x_2] \,@\, t_r)$, where $[x_1, x_2]$ denotes an interval in the device's *x*-dimension and t_r gives the maximum of the last release time for this interval and the current time. The intervals in the horizon fully partition the spatial resource dimension. The horizon is sorted according to increasing release times.

The pseudocode for the horizon scheduler is shown in Algorithm 1. When a new task T_i arrives, the horizon scheduler walks through the list of horizon intervals and checks whether the task can be appended to the horizon. At any point during this walk, the list L contains those intervals of the horizon that are under consideration for finding a placement. L is sorted according to increasing *x*-coordinates and initialized with all horizon intervals at the current time (line 2 of Algorithm 1). In line 4, the function $BestFit()$ selects the interval from L with the smallest width, among all intervals that are large enough to accommodate T_i. The function $BestFit()$ returns either an empty set when no such interval is found or the placement x for T_i. When a suitable interval is found, the task is scheduled to placement x and starting time t and the planning process stops. Otherwise, the scheduler proceeds to the next horizon interval. In line 10, the function $MergeIntervals()$ tries to merge adjacent horizon intervals to form larger intervals.

TABLE 1
Example Task Set

parameter	T_1	T_2	T_3	T_4	T_5	T_6	T_7
a_i	0	0	1	1	2	2	3
e_i	20	3	12	3	2	3	2
d_i	30	10	15	10	10	20	20
w_i	3	7	3	2	3	5	3
h_i	3	5	5	2	4	1	2
f_i (1D horizon)	20	3	15	6	8	18	20
f_i (1D stuffing)	20	3	15	6	8	18	10
f_i (2D horizon)	20	3	15	4	5	5	5

The parameters a_i, e_i, d_i, and f_i are given in time units; the widths w_i and heights h_i are measured in reconfigurable units (RCUs).

Algorithm 1: 1D Horizon Scheduler $\sigma_{1D-horizon}(T_i, H)$

```
1.  $t \leftarrow a_i$
2.  $L \leftarrow$ horizon intervals with $t_r = t$
3.  while ($t \leq s_{i-latest}$) do
4.      $x = BestFit(L, w_i)$
5.      if $x \neq \emptyset$ then
6.          add reservation $(T_i, x, t)$
7.          $return$(ACCEPT)
8.      end if
9.      $t \leftarrow t_r$ of the next horizon interval
10.     $L \leftarrow MergeIntervals(L, H, t)$
11. end while
12. $return$(REJECT)
```

At each time t_x, the online method first checks for terminating tasks, i.e., tasks with finishing times $f_i = t_x$, and removes them from the execution list. Then, waiting tasks with $s_i = t_x$ are started and moved from the reservation list to the execution list. Finally, for each newly arrived task T_i, the scheduling function $\sigma_{1D-horizon}(T_i, H)$ is called which either rejects the task or accepts it.

Table 1 shows an example for a task set consisting of seven tasks. The resulting schedule under the 1D horizon method on a device of dimensions $W \times H = 10 \times 6$ RCUs are displayed in Fig. 6. The scheduling horizon and the

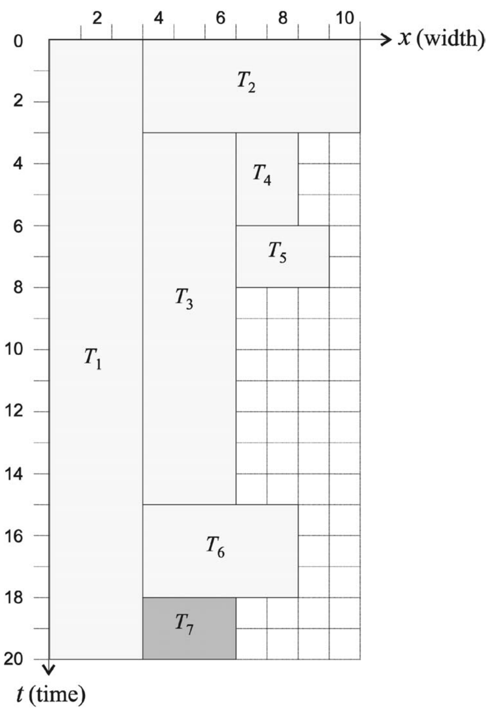
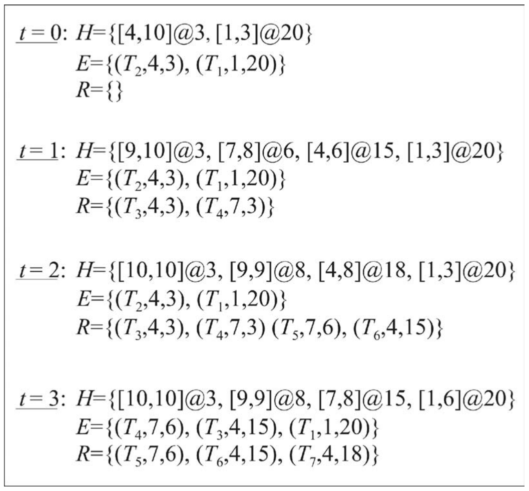


Fig. 6. Schedule for the task set of Table 1 under the 1D horizon technique and a device of dimensions 10×6 RCUs.


$t = 0$: H={[4,10]@3, [1,3]@20}
E={(T_2,4,3), (T_1,1,20)}
R={}

$t = 1$: H={[9,10]@3, [7,8]@6, [4,6]@15, [1,3]@20}
E={(T_2,4,3), (T_1,1,20)}
R={(T_3,4,3), (T_4,7,3)}

$t = 2$: H={[10,10]@3, [9,9]@8, [4,8]@18, [1,3]@20}
E={(T_2,4,3), (T_1,1,20)}
R={(T_3,4,3), (T_4,7,3) (T_5,7,6), (T_6,4,15)}

$t = 3$: H={[10,10]@3, [9,9]@8, [7,8]@15, [1,6]@20}
E={(T_4,7,6), (T_3,4,15), (T_1,1,20)}
R={(T_5,7,6), (T_6,4,15), (T_7,4,18)}

Fig. 7. Scheduling horizon H, execution list E, and reservation list R for the task set of Table 1 under the 1D horizon technique and a device of dimensions 10×6 RCUs.

execution and reservation lists for $t = 0, 1, 2, 3$ are depicted in Fig. 7. At time $t = 2$, two tasks (T_1, T_2) are executing on the device and four tasks (T_3, T_4, T_5, T_6) hold reservations. At time $t = 3$, task T_2 terminates, tasks T_3 and T_4 start, and task T_7 arrives with $(a_7, e_7, d_7, w_7, h_7) = (3, 2, 20, 3, 2)$. The planning period for T_7 is $[a_7, (d_7 - e_7)] = [3, 18]$. The first horizon interval to be checked is $L = \{[10, 10]\}$ at $t = 3$, which is too small to fit T_7. The scheduler proceeds to $t = 8$, where two horizon intervals are merged to $L = \{[9, 10]\}$, which is too small again. The next check takes place at $t = 18$ with $L = \{[4, 10]\}$, which allows us to accept T_7 with the reservation $(T_7, 4, 18)$. The finishing times for all tasks are given in Table 1.

The horizon technique ensures that new tasks are only inserted into the reservation list when they do not overlap in time or space with other tasks in the list. The key to this is that tasks can only be *appended* to the horizon. It is not possible to schedule tasks before the horizon as the scheduling procedure maintains no knowledge about the time-varying allocation between the current time and the horizon intervals. The advantage of this technique is that maintaining the scheduling horizon is simple compared to maintaining all future allocations of the device.

4.3 The 1D Stuffing Technique

The stuffing technique schedules tasks to arbitrary free areas that will exist in the future, including areas that will be used later by tasks currently in the reservation list. Compared to the horizon technique, the stuffing technique is more complex as it has to identify such potential placements and to check these potential placements for conflicts with existing reservations.

Besides the execution and reservation lists, the stuffing scheduler employs a *free space list* F. The free space list is a set of intervals $[x_1, x_2]$ that identify currently unused resource intervals, sorted according to increasing x-coordinates. The pseudocode for the stuffing scheduler is shown in Algorithm 2. When a new task T_i arrives, the scheduler starts walking through the task's planning period, mimicking all future allocations of the device by *simulating* future task terminations and starts together with the underlying free space management. In line 1 of Algorithm 2, the current free space is copied to a simulated free space list, F_S, which is then modified during the scheduling process. At any given time, the scheduler first checks for terminating tasks (lines 4-6). Then, reserved tasks are started (lines 8-10). In line 12, the function $BestFit()$ reports all intervals in the simulated free space list that can accommodate T_i or the empty set if no such interval exists. The reported intervals are then checked for conflicts with existing reservations in best-fit order. If an interval without conflict is found, T_i is accepted and planning stops. Otherwise, the scheduler proceeds to the next event.

Algorithm 2: 1D Stuffing Scheduler $\sigma_{1D-stuffing}(T_i, F)$

```
1.  $F_S \leftarrow F$; $t \leftarrow a_i$
2.  $check \leftarrow$ TRUE
3.  while $(t \leq s_{i-latest})$ do
4.      for all $T_j \in E$ with $(f_j = t)$ do
5.          $TerminateTask(T_j, F_S)$
6.          $check \leftarrow$ TRUE
7.      end for
8.      for all $T_j \in R$ with $(s_j = t)$ do
9.          $StartTask(T_j, F_S)$
10.     end for
11.     if $check$ then
12.         $\mathcal{X} \leftarrow BestFit(F_S, w_i)$
13.         for all $(x \in \mathcal{X})$ do
14.             if $((x, t, x + w_i, t + e_i)$ is not conflicting any
                    reservation in $R)$ then
15.                 add reservation $(T_i, x, t)$ to $R$
16.                 $return$(ACCEPT)
17.             end if
18.         end for
19.         $check \leftarrow FALSE$
```

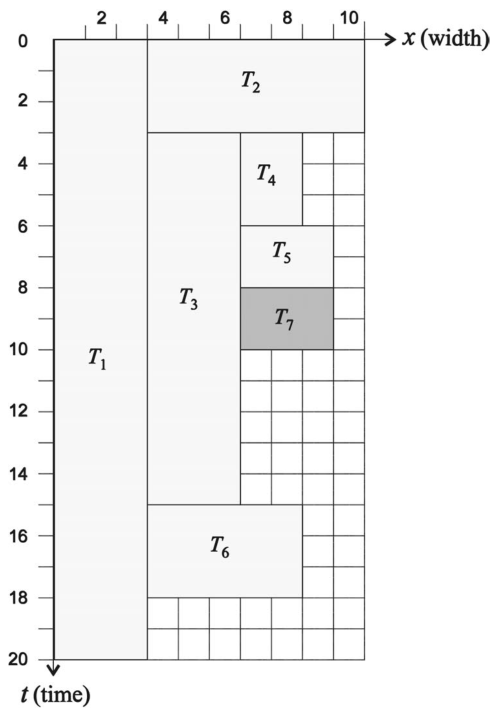


Fig. 8. Schedule for the task set of Table 1 under the 1D stuffing technique and a device of dimensions 10×6 RCUs.

```
20.         end if
21.         $t \leftarrow$ next event from $E \cup R$
22.     end while
23. $return$(REJECT)
```

At each time t_x, the online method first checks for terminating tasks, i.e., tasks with finishing times $f_i = t_x$, and removes them from the execution list. Then, waiting tasks with $s_i = t_x$ are moved from the reservation list to the execution list and are started. Finally, for each newly arrived task T_i, the scheduling function $\sigma_{1D-stuffing}(T_i, F)$ is activated which either rejects the task or moves it to the reservation list. On all three events, the free space list is updated and adjacent free intervals are merged to form larger free intervals.

Fig. 8 shows the resulting schedule for the task set of Table 1 under the 1D stuffing method on a device of dimensions $W \times H = 10 \times 6$ RCUs. When task T_7 arrives at $t = 3$, the free space list consists of one free interval, $F_S = \{([9, 10])\}$, which is too small to fit T_7. In the course of planning, the free space list is modified to $F_S = \{[10, 10]\}$ at $t = 6$ and $F_S = \{[7, 10]\}$ at $t = 8$. At this point, there exists an interval that can accommodate T_7 and does not conflict with any reserved task. Hence, T_7 is scheduled to placement $x_7 = 7$ and starting time $s_7 = 8$.

The stuffing technique leads to improved performance over the horizon method, as shown in Table 1. The drawback of the stuffing method is the increased complexity as we need to simulate future task terminations and planned starts to identify free space.

4.4 Extension to the 2D Area Model

The scheduling problem and the heuristics discussed so far extend naturally to the 2D area model. A 2D placement can be given by the coordinates of the top-left task cell, (x_i, y_i), with $x_i + w_i \leq W$ and $y_i + h_i \leq H$. Fig. 5b shows an example of a device $W \times H = 10 \times 8$ cells with two placed tasks. Task T_1 has the placement $(x_1, y_1) = (3, 4)$; task T_2 is placed at $(x_2, y_2) = (8, 2)$. Accordingly, the definition of a scheduled task in the 2D area model extends to:

Definition 2 (Scheduled Task 2D). *A* scheduled task T_i *is a task with a placement* (x_i, y_i) *and a starting time* s_i *such that:*

1. $\forall T_j \in \mathcal{T}, T_j \neq T_i$:

$$[(x_i + w_i) \leq x_j] \vee [(x_i \geq (x_j + w_j)] \vee \\ [(y_i + h_i) \leq y_j] \vee [(y_i \geq (y_j + h_j)] \vee \\ [(s_i + e_i) \leq s_j] \vee [(s_i \geq (s_j + e_j)]$$

(scheduled tasks must not overlap in space and time).

2. $s_i + e_i \leq d_i$ *(deadline must be met).*

Formally, the acceptance test is encapsulated into a 2D scheduling function $\sigma(T_i)$:

$$\sigma(T_i) = \begin{cases} \texttt{accept}\ with\ (x_i, y_i, s_i) \\ \texttt{reject}. \end{cases} \tag{2}$$

Switching from the 1D to the 2D area model mainly affects the management of free space which is done by the placer. The scheduling techniques remain basically unchanged. In the following, we detail the extension of the horizon technique to the 2D area model. The stuffing technique is extended accordingly.

Like the 1D variant, the 2D horizon technique uses three lists. The execution list E contains all currently executing tasks T_i with their finishing times $f_i = s_i + e_i$ and placements (x_i, y_i), sorted in order of increasing finishing times. The reservation list R stores all scheduled but not yet executing tasks with their placements (x_i, y_i) and starting times s_i, sorted in order of increasing starting times. The scheduling horizon H in the 2D area model is a list of elements $h_i = ([(x_1, y_1), (x_2, y_2)] \ @ \ t_r)$, where $[(x_1, y_1), (x_2, y_2)]$ denotes a rectangle on the device's surface and t_r gives the maximum of the last release time for this rectangle and the current time. The rectangles in the horizon fully partition the spatial resource. The horizon is sorted according to increasing release times.

Compared to the 1D version, the 2D horizon scheduler works with rectangles instead of intervals and with 2D placements (x, y) instead of 1D placements (x). The function $BestFit()$ in Algorithm 1 selects the rectangle from L with the smallest area among all rectangles that are large enough to accommodate T_i. Instead of the function $MergeIntervals()$, the function $MergeRectangles()$ tries to merge adjacent horizon rectangles to form larger rectangles. There is a subtle difference between the 1D and 2D area models with regard to this merging step. While merging two adjacent intervals is a unique process, the placer has several ways to merge adjacent rectangles. In fact, the 2D placers that have been proposed in related work differ strongly in the way they manage the free space and combine adjacent free rectangles (see Section 4.5).

Fig. 9 continues the example from Table 1 and presents the execution list, the reservation list, and the 2D scheduling horizon for $t = 0, 1, 2, 3$. Each diagram of Fig. 9 displays some of the lists graphically in a width-height diagram. Fig. 9a shows the execution list at time $t = 0$. Two tasks, T_1

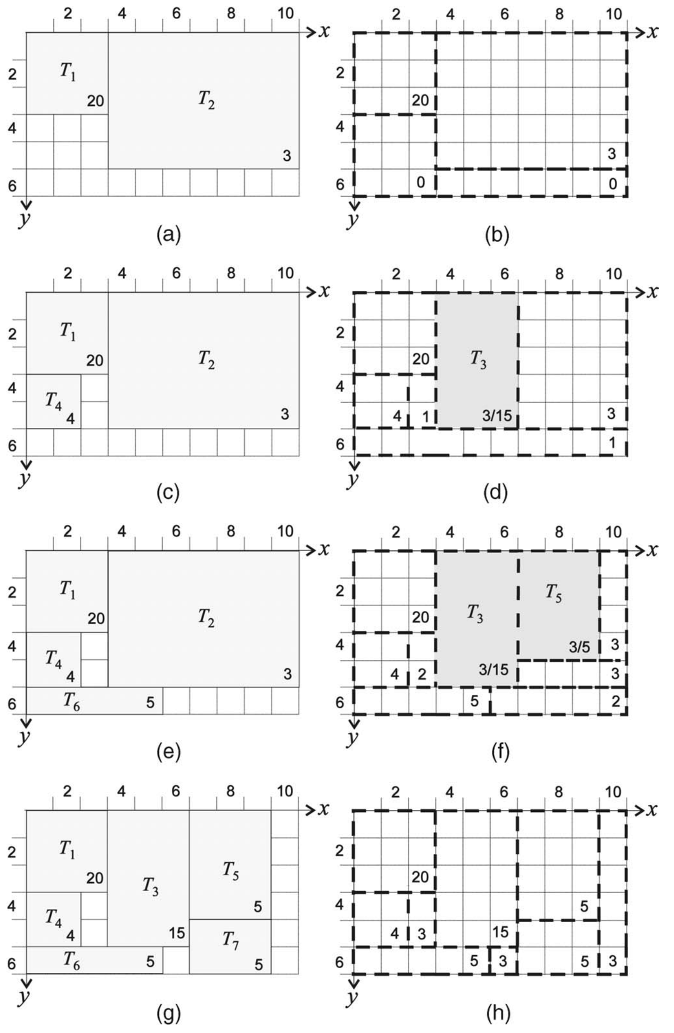


Fig. 9. Execution lists E (a), (c), (e), (g) as well as scheduling horizons H and reservation lists R (b), (d), (f), (h) under the 2D horizon technique at time $t = 0, 1, 2, 3$ for the task set of Table 1 and a device of dimensions 10×6 RCUs.

and T_2, are executing. T_1 has a placement $(1,1)$ and a finishing time $f_1 = 20$ and T_2 has a placement $(4,1)$ and finishing time $f_2 = 3$, respectively. The finishing times are denoted by the numbers in the task rectangles. Fig. 9b presents the scheduling horizon, which consists of four list entries. The numbers inside the rectangles denote the last release times. The horizon rectangles $[(1,4),(3,6)]$ @ $0)$ and $[(4,6),(10,6)]$ @ $0)$ are currently free rectangles. The horizon rectangles $[(4,1),(10,5)]$ @ $3)$ and $[(1,1),(3,3)]$ @ $20)$ represent the areas currently occupied by executing tasks.

Apparently, there are several ways to partition the current free area (e.g., given by horizon rectangles with release time of 0 in Fig. 9b) into rectangles. This partitioning (as well as the merging of free rectangles) is hidden in the placer module and will be discussed in Section 4.5. The chosen placement technique will strongly influence the resulting scheduling performance and runtime. The scheduling algorithm, however, only requires a list of suitable rectangles for a new task and does not have to know in which way the placer manages the free area. In this sense, the proposed scheduling heuristics are independent of the actual placement techniques. At time $t = 1$, the two tasks T_3 and T_4 arrive. Task T_3 receives a reservation which is shown as a shaded rectangle in Fig. 9d. The numbers in the shaded rectangle denote the starting time of the corresponding task and the last release time for the occupied rectangle, respectively. Task T_4 can be immediately placed and executed. Fig. 9c displays the resulting execution list at time $t = 1$. Overall, the scheduling horizon now consists of six rectangles. Fig. 9e and Fig. 9f show the lists at time $t = 2$. Tasks T_5 and T_6 have arrived. In the course of planning T_5, the scheduling algorithm found a rectangle of sufficient size at $t = 3$. Thus, T_5 received the reservation $(T_5, 7, 1, 3)$. Task T_6 can be started immediately. The horizon at $t = 2$ consists of nine rectangles overall. Finally, Fig. 9g and Fig. 9h display the lists at time $t = 3$. Task T_2 has terminated and all reserved tasks have been started. Additionally, the newly arrived task T_7 could be immediately placed and started. Six tasks are executing in parallel at $t = 3$.

The resulting task finishing times are given in Table 1. As could be expected, the 2D area model leads to an improved resource utilization. For the task set of Table 1, the finishing times for the 2D stuffing technique and 2D horizon technique are identical.

4.5 Placement Algorithms

The placer performs several functions. The basic function is free space management, i.e., representing the unused area of the reconfigurable device with a proper data structure. When tasks start or terminate execution, the placer updates this data structure to reflect the new allocation. When a task arrives, the placer has to search for suitable free areas on the device. The horizon technique further asks the placer to merge adjacent free areas to larger ones to increase the chances of finding suitable free areas. The stuffing technique requires the placer to mimic future task starts and terminations, which is done by regular task insert and delete operations on a copy of the original data structure (see Algorithm 2).

In the 1D area model, the placer manages free space by lists of *maximal free intervals*, i.e., adjacent free intervals are immediately merged. All placer operations described above are therefore bound by $\mathcal{O}(n)$, where n is the number of currently placed tasks.

In the 2D area model, the placer works with lists of *free rectangles*. Partitioning the free 2D space into free rectangles can be done in several ways, as was discussed by Bazargan et al. [25]. An approach that corresponds to the 1D placement is to manage the free space by a list of *maximal free rectangles*. A maximal free rectangle is a rectangle which is not contained in any other free rectangle. In contrast to maximal free intervals, maximal free rectangles are overlapping. For example, the free area in Fig. 9b is defined by the two overlapping rectangles $[(1,4),(3,6)]$ and $[(1,6),(10,6)]$. For any given allocation, the set of maximal free rectangles is unique and optimal in the sense that the placer will find a suitable rectangle for an arriving task if there is sufficient free space. Such a placer is also called *recognition-optimal* [35]. However, the runtime complexities of the placer operations have been shown to be quadratic in the number of currently placed tasks [25].

A different approach to 2D placement is to partition the free space into a set of *nonoverlapping free rectangles*. Such a set is neither unique nor optimal in the previous sense. The main advantage is that the number of free rectangles is bound by $\mathcal{O}(n)$, which can be expected to lead to efficient online implementations. In this work, we employ placers that work with nonoverlapping free rectangles, as shown in Fig. 9. The used placement techniques originate from [25] and have previously been improved by us [26].

All online schedulers discussed in this paper avoid rescheduling of previously guaranteed tasks. Although we sacrifice some performance by this restriction, we gain an important benefit. As soon as a task is scheduled, we know its definite placement. This allows the operating system to perform the bitstream manipulations necessary to prepare (relocate) the task for execution *directly after* its acceptance. The relocation step can be done when the system is idle, which helps reduce the operating system overhead for multitasking.

4.6 Relation to Other Placement and Scheduling Problems

The scheduling heuristics described in this paper are *online algorithms*. Among the vast literature on online algorithms, e.g., see [36] or [37], there are many online placement and scheduling problems with which our work shares similarities. However, our specific scheduling problem seems to differ due to the specific task and system models and the strong nexus between scheduling and placement. In the following, we discuss the most important related problems.

Strip packing tries to place a set of two-dimensional boxes into a vertical strip of width W by minimizing the total height of the strip. Strip packing problems come in many variations, including rotations of boxes, boxes with modifiable aspect ratios, tasks with gravity [37], and extensions to higher dimensions. The offline strip packing problem is NP-hard [38] and many approximation algorithms have been developed for it. There are also some online algorithms with known competitive ratios. Compared to our scheduling problem, the width of the strip corresponds to the width of the reconfigurable device and the vertical dimension corresponds to the time. For the 2D area model, the resulting scheduling problem relates to 3D strip packing [13]. The crucial difference between our scheduling problem and strip packing is that, in our problem, one of the dimensions is the time which proceeds as tasks are arriving. We cannot schedule tasks beyond the current timeline, i.e., into the past. In strip packing, the bottom of the container is fixed and rectangles can be packed to any position of the container anytime, provided there is sufficient free space.

In classical *online processor scheduling*, a processor executes only one task at any given time and usually there is no placement issue. In [39], competitive ratios for online scheduling of nonpreemptive jobs with deadlines are derived. Such a system and task model have applications in admission control for high-speed networks. The only resource is the processor and the objective is the maximization of the total processor utilization rather than the maximization of the number of tasks that meet their deadlines. In [40], the online scheduling of tasks to a system with a number of resources is considered. Resources are grouped into active resources (e.g., processors) and passive resources. Passive resources can be used in either exclusive or shared mode. Similar to our approach, the tasks have deadlines, the algorithms proposed are heuristics, and the evaluation is done experimentally. In contrast to our work, tasks can be preempted in case they are being blocked on accesses to passive resources or tasks with higher priority have to be executed. In the terminology of [40], each of our hardware tasks requires a *contiguous area* of reconfigurable units as an *active resource*.

Allocating and scheduling *parallel tasks* to 2D mesh-connected multiprocessors is presumably the closest related problem. The processors of a 2D mesh correspond to the reconfigurable units (RCUs). A parallel task requires a rectangular grid of processors (submesh), whereas a hardware task requires a rectangular area of RCUs. Although there exists a significant body of work in multiprocessor allocation and scheduling, see, e.g., [35], there are the following important differences from our work: First, our hardware tasks must not be rotated and are not malleable. Second, in the system described in this paper, hardware tasks cannot be preempted or restarted. Third, most work in mesh-connected multiprocessors concentrates on placement rather than scheduling (which is mostly FIFO) and targets general-purpose workloads with the goal of minimizing the makespan or average response time. One recent publication [41] considers mapping of real-time tasks onto a 2D parallel computer. Task preemption is partly assumed and the overall runtime complexity of the scheduling and placement algorithm is too high for the application scenarios considered in this work. The placement algorithms used in mesh-connected multiprocessors are usually *scanning* approaches. The array of processors is scanned from bottom to top and left to right and each element is considered a potential placement. The main advantage of scanning placers is that they are *recognition-complete* [35]. The drawback of the scanning placers is their high runtime complexity. In the worst case, they have to consider $W \times H$ potential placements, which makes them less attractive for online scheduling. For partially reconfigurable devices and the scenarios considered in this paper, two orders of magnitude higher runtimes are observed in comparison with the placement techniques used in this work.

5 Evaluation

An efficient and accurate method to evaluate the performance of the proposed scheduling techniques would be a trace-based simulation. Unfortunately, such traces are not available as reconfigurable hardware operating systems are a new area where only first prototypes are being built. As a consequence, we have to resort to the simulation of randomly generated, i.e., synthetic, workloads. Anyhow, we leverage on our experience with hardware tasks, the operating system prototype, and case studies (see Section 3) to derive realistic value ranges for the different parameters. We consider coarse-grained hardware tasks in our work. Examples for such tasks span from a simple UART with 50 RCUs over a DCT core with 600 RCUs to a 3-DES core with 800 RCUs. The task execution times in our prototype range from a few tens of ms to some hundred ms.

For evaluation, we have devised a discrete-time simulation framework. The framework consists of several modules such as the simulator kernel, the operating system modules with hooks for different scheduling and placement functions, a random task generator, a module for the collection of execution statistics, and a graphical display of the current allocation situation. The simulator can be parameterized to model reconfigurable devices of different size and generate random task sets with defined distributions. Simulation

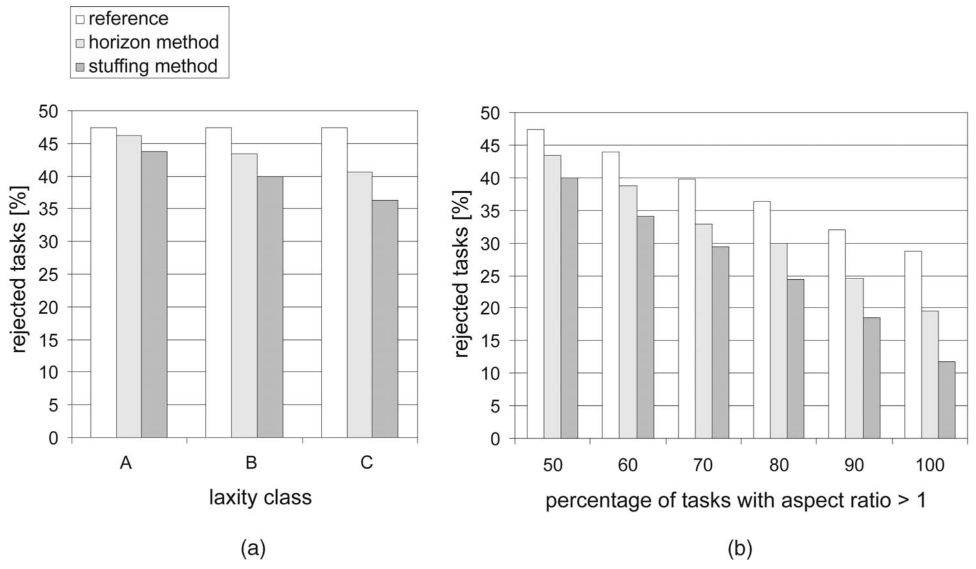


Fig. 10. Scheduling performance of the proposed scheduling heuristics for the 1D area model: (a) scheduling performance as function of the laxity class, 50 percent of the tasks have an aspect ratio larger than 1; (b) scheduling performance as function of the aspect ratio for laxity class B.

time proceeds in discrete time steps. For the simulation experiments described here, we have neglected the runtimes of the online schedulers. In Section 5.2.3, we conduct runtime measurements to justify this simplification.

5.1 Simulation Setup

We have run a number of simulation experiments with different parameter settings. The results presented in this section are typical and are based on following settings: The simulated device consists of 96×64 reconfigurable units (Xilinx XCV1000). The task areas are uniformly distributed in $[50, 500]$ reconfigurable units. The execution times are uniformly distributed in $[5, 100]$ time units. We assume a time unit of the discrete-time simulation to be $10\ ms$, which results in execution times for the hardware tasks between $50\ ms$ and $1\ s$. The aspect ratios are distributed between $[5, 0.2]$. We have defined three task classes, A, B, C, with laxities uniformly distributed in $[1, 50]$, $[50, 100]$, and $[100, 200]$ time units, respectively.

The runtime measurements have been conducted on a Pentium-III 1000MHz, taking advantage of Visual C++'s profiling facilities. All the simulations presented below use a 95 percent confidence level with an error range of ±3 percent.

5.2 Results

5.2.1 Scheduling to the 1D Area Model

Fig. 10a compares the performance of the reference scheduler with the performance of the horizon and the stuffing techniques. The scheduling performance is measured by the percentage of rejected tasks, which is to be minimized. Fig. 10a displays the scheduling performance of the three schedulers for the three laxity classes A, B, and C. For this experiment, the aspect ratios have been distributed such that 50 percent of the tasks are taller than wide (standing tasks) and 50 percent are wider than tall (lying tasks). The reference scheduler does not plan into the future. Hence, its performance is independent of the laxity class. As expected, the stuffing method performs better than the horizon method, which, in turn, is superior to the reference scheduler. The differences between the methods grow with increasing laxity because longer planning periods provide more opportunity for improved scheduling. For laxity class C, the horizon scheduler outperforms the reference scheduler by 14.46 percent; the stuffing scheduler outperforms the reference scheduler by 23.56 percent.

Fig. 10b shows the percentage of rejected tasks as a function of the aspect ratio, using laxity class B. For the 1D area model, standing tasks are clearly preferable. The rightmost data samples in Fig. 10b denote an all standing task set, i.e., the percentage of tasks with aspect ratio larger than 1 is 100 percent. The results demonstrate that all schedulers benefit from standing tasks. The differences again grow with the aspect ratio. For 100 percent standing tasks, the horizon method results in a performance improvement of 32 percent over the reference method, the stuffing method outperforms the reference method even by 59 percent.

It has to be mentioned that a guarantee-based real-time scheduling system with rejection ratios of some 40 percent, as shown in Fig. 10, would be considered useless. We achieve these high rejection ratios because we distribute the arrival times of the tasks such that the schedulers are reasonably loaded and the differences between the scheduling techniques become visible. If the task interarrival times are long compared to the task execution times, the device is mostly unused and any scheduler will be able to accept the tasks.

The main conclusion from this simulation experiment is the great influence of the aspect ratio distribution on the performance of the schedulers. While all schedulers improve with standing tasks, the stuffing schedulers gain most. As a consequence for the 1D area model, the creation of standing tasks must be facilitated during synthesis whenever possible by providing proper placement and routing constraints. If the creation of standing tasks is not facilitated and we have to assume that some 50 percent of the tasks are lying, planning schedulers are in vain and one could resort to the simple reference method without too much loss in performance.

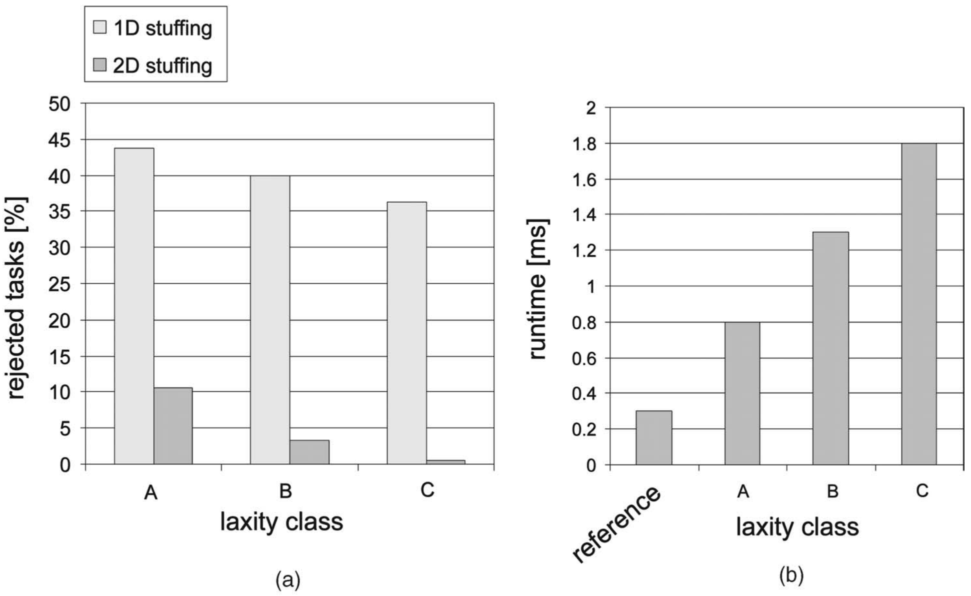


Fig. 11. (a) Scheduling performance of the 1D and 2D stuffing methods for different laxity classes, 50 percent of the tasks have an aspect ratio larger than 1. (b) Runtimes of the reference and stuffing schedulers to schedule one task, depending on the laxity class.

5.2.2 Comparison of 1D and 2D Area Models

Fig. 11a compares the scheduling performance of the 1D and 2D area models for the stuffing technique, depending on the laxity class. The aspect ratios are distributed such that 50 percent of the tasks are standing. The results clearly show the superiority of the 2D area model. For laxity class A, the performance improvement in going from 1D to 2D is 76 percent; for laxity class C, it raises to 98 percent. Even for 100 percent standing tasks, the 1D stuffing method stays above 10 percent rejected tasks (see Fig. 10b), whereas the 2D stuffing method achieves a rejection ratio of less than 5 percent (see Fig. 11a). A further result which is not shown in the figures is that the scheduling performance for the 2D model depends only weakly on the aspect ratio distribution. Due to the 2D resource management, both mixed and standing task sets are handled equally well.

5.2.3 Runtime Efficiency

Fig. 11b presents the average runtime required to schedule one task for both the 2D reference and 2D stuffing methods, depending on the task laxities. The 2D stuffing scheduler is the most complex of all implemented techniques. As expected, the stuffing scheduler's runtime increases with the length of the planning period. However, with $1.8\ ms$, at most the absolute values are small. In our simulation, we assume a time unit of the discrete-time simulation to be $10\ ms$, which gives hardware tasks running from $50\ ms$ to $1\ s$. The runtime overheads for the online scheduler ($1.8\ ms$ at most) and for partial reconfiguration (in the order of a few ms) are negligible and justify our simulation setup that does not model these overheads.

6 Conclusion and Further Work

In this paper, we have discussed design issues for reconfigurable hardware operating systems and the problem of online scheduling hard real-time tasks to partially reconfigurable devices. We have developed two online scheduling heuristics that work for both the 1D and 2D area models. Simulations have shown that the heuristics are effective in reducing the number of rejected tasks. While the 1D schedulers strongly depend on the tasks' aspect ratios, the 2D schedulers do not. In all cases, the 2D area model dramatically outperforms the 1D model. Finally, the scheduler runtimes are so small that the more complex stuffing technique will be the method of choice for most application scenarios.

Further research activities could extend the scheduling heuristics to include a broader range of task models. For example, non-real-time tasks (i.e., tasks without deadlines) and tasks with unknown execution times can easily be handled by the scheduling algorithms. A task without a deadline is assigned an infinite planning period; a task without a bound on its execution time is assigned a release time of infinity. Periodic tasks could be included by making periodic reservations. This will severely compromise the performance of the horizon technique because a periodic reservation will shift the release time of the corresponding horizon interval (rectangle) to infinity. The stuffing technique, however, should be able to utilize the free areas between subsequent invocations of a periodic task. Other system models, such as preemptive real-time systems, soft real-time systems, or non-real-time systems, will require different scheduling techniques and performance metrics.

Acknowledgments

This work was supported by the Swiss National Science Foundation (SNF) under grant number 2100-59274.99.

References

[1] Xilinx, Inc., "Virtex-II Pro Platform FPGAs: Complete Data Sheet, November 11, 2003,"http://www.xilinx.com.

[2] Altera Corp., "Excalibur Device Overview, May 2002,"http://www.altera.com.

[3] B.L. Hutchings and M.J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications," *Proc. Int'l Workshop Field-Programmable Logic and Applications (FPL)*, pp. 419-428, 1995.

[4] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, L. Thiele, and G. Tröster, "The Case for Reconfigurable Hardware in Wearable Computing," *Personal and Ubiquitous Computing,* pp. 299-308, Oct. 2003.

[5] IMEC Interuniversity Micro Electronic Center, "T-ReCS Gecko," http://www.imec.be, year?

[6] S. Chakraborty, M. Gries, S. Künzli, and L. Thiele, "Design Space Exploration of Network Processor Architectures," *Network Processor Design: Issues and Practices, Volume 1,* pp. 55-89 Morgan Kaufmann, Oct. 2002.

[7] G. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200," *Proc. Int'l Workshop Field-Programmable Logic and Applications (FPL),* pp. 327-336, 1996.

[8] H. Walder and M. Platzner, "Reconfigurable Hardware Operating Systems: From Concepts to Realizations," *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms (ERSA),* pp. 284-287, 2003.

[9] J.-Y. Mignolet, S. Vernalde, D. Verkest, and R. Lauwereins, "Enabling Hardware-Software Multitasking on a Reconfigurable Computing Platform for Networked Portable Multimedia Appliances," *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms (ERSA),* pp. 116-122, 2002.

[10] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck, "Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 10, no. 3, pp. 209-220, June 2002.

[11] H. Walder and M. Platzner, "Non-Preemptive Multitasking on FPGAs: Task Placement and Footprint Transform," *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms (ERSA),* pp. 24-30, 2002.

[12] K. Purna and D. Bhatia, "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers," *IEEE Trans. Computers,* vol. 48, no. 6, pp. 556-564, June 1999.

[13] S. Fekete, E. Köhler, and J. Teich, "Optimal FPGA Module Placement with Temporal Precedence Constraints," *Proc. Design Automation and Test in Europe Conf. (DATE),* pp. 658-665, 2001.

[14] G. Brebner, "The Swappable Logic Unit: A Paradigm for Virtual Hardware," *Proc. IEEE Symp. FPGAs for Custom Computing Machines (FCCM),* pp. 77-86, 1997.

[15] J.S. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook, "Dynamic Reconfiguration to Support Concurrent Applications," *IEEE Trans. Computers,* vol. 48, no. 6, pp. 591-602, June 1999.

[16] P. Merino, J.C. Lopez, and M. Jacome, "A Hardware Operating System for Dynamic Reconfiguration of FPGAs," *Proc. Int'l Workshop Field Programmable Logic and Applications (FPL),* pp. 431-435, 1998.

[17] P. Merino, M. Jacome, and J.C. Lopez, "A Methodology for Task Based Partitioning and Scheduling of Dynamically Reconfigurable Systems," *Proc. IEEE Symp. FPGAs for Custom Computing Machines (FCCM),* pp. 324-325, 1998.

[18] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA Coprocessors," *Proc. Int'l Conf. Field Programmable Logic and Applications (FPL),* pp. 121-130, 2000.

[19] J. Burns, A. Donlin, J. Hogg, S. Singh, M. de Wit, "A Dynamic Reconfiguration Run-Time System," *Proc. IEEE Symp. FPGAs for Custom Computing Machines (FCCM),* pp. 66-75, 1997.

[20] N. Shirazi, W. Luk, and P. Cheung, "Run-Time Management of Dynamically Reconfigurable Designs," *Proc. Int'l Workshop Field-Programmable Logic and Applications (FPL),* pp. 59-68, 1998.

[21] O. Diessel and H. ElGindy, "On Scheduling Dynamic FPGA Reconfigurations," *Proc. Australasian Conf. Parallel and Real-Time Systems (PART),* pp. 191-200, 1998.

[22] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs}," *IEE Proc. Computers and Digital Techniques,* vol. 147, no. 3, pp. 181-188, May 2000.

[23] K. Compton, J. Cooley, S. Knol, and S. Hauck, "Configuration Relocation and Defragmentation for Reconfigurable Computing," *Proc. IEEE Symp. FPGAs for Custom Computing Machines (FCCM),* pp. 279-280, 2001.

[24] G. Brebner and O. Diessel, "Chip-Based Reconfigurable Task Management," *Proc. Int'l Conf. Field Programmable Logic and Applications (FPL),* pp. 182-191, 2001.

[25] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems," *IEEE Design and Test of Computers,* vol. 17, no. 1, pp. 68-83, 2000.

[26] H. Walder, C. Steiger, and M. Platzner, "Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing," *Reconfigurable Architectures Workshop (RAW), Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2003.

[27] G. Wigley and D. Kearney, "The Development of an Operating System for Reconfigurable Computing," *Proc. IEEE Symp. FPGAs for Custom Computing Machines (FCCM),* 2001.

[28] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs," *Proc. Int'l Conf. Field-Programmable Logic and Applications (FPL),* pp. 795-805, 2002.

[29] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an Operating System for a Heterogeneous Reconfigurable SoC," *Reconfigurable Architectures Workshop (RAW), Proc.e Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2003.

[30] C. Steiger, H. Walder, and M. Platzner, "Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices," *Proc. Int'l Conf, Field Programmable Logic and Applications (FPL),* pp. 575-584, 2003.

[31] D. Lim and M. Peattie, "Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations," XAPP 290, Xilinx, 2002.

[32] H. Walder and M. Platzner, "A Runtime Environment for Reconfigurable Hardware Operating Systems," *Proc. Int'l Conf. Field-Programmable Logic and Applications (FPL),* 2004.

[33] H. Walder, S. Nobs, and M. Platzner, "XF-BOARD: A Prototyping Platform for Reconfigurable Hardware Operating Systems," *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms (ERSA),* 2004.

[34] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer, 2000.

[35] D. Feitelsen, "Job Scheduling in Multiprogrammed Parallel Systems," *IBM Research Report,* vol. RC 87657, Aug. 1997.

[36] A. Borodin and R. El-Yaniv, *{Online Computation and Competitive Analysis.* Cambridge Univ. Press, 1998.

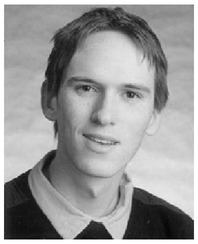
[37] J. Csirik and G. Woeginger, *Online Algorithms: The State of the Art.* Springer-Verlag, 1998.

[38] B. Baker, E. Coffman, and R. Rivest, "Orthogonal Packings in Two Dimensions," *SIAM J. Computing,* no. 9, pp. 846-855, 1980.

[39] S.A. Goldman, J. Parwatika, and S. Suri, "Online Scheduling with Hard Deadlines," *J. Algorithms,* vol. 34, pp. 370-387, 2000.

[40] W. Zhao, K. Ramamritham, and J.A. Stankovic, "Preemptive Scheduling under Time and Resource Constraints," *IEEE Trans. Computers,* vol. 36, no. 8, pp. 949-960, Aug. 1987.

[41] S.-M. Yoo, H. Youn, and H. Choo, "Dynamic Scheduling and Allocation in Two-Dimensional Mesh-Connected Multicomputers for Real-Time Tasks," *IEICE Trans. Information and Systems,* vol. E84-D, no. 5, pp. 613-622, 2001.



Christoph Steiger received the degree in computer science from the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, in 2002. In his final thesis, he worked on task scheduling and placement algorithms for partially reconfigurable devices, a topic which was pursued further in a stay as a research associate at ETH Zurich. In 2003, he joined the European Space Operations Centre (ESOC) of the European Space Agency (ESA), Darmstadt, Germany, where he is working for the ROSETTA mission to comet 67P/Churyumov-Gerasimenko, launched in March 2004. He currently holds responsibility for maintenance and operations of the ROSETTA flight software as a member of the Flight Control Team, as well as contributing to advanced flight software research projects of ESA. His research interests include reconfigurable computing, spacecraft software in general, and flight software operations automation in particular.

Herbert Walder received the Dipl. El.-Ing. degree in electrical engineering from the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, in 1995. From 1995 to 2000, he held several positions in the defense and telecommunication industries, ranging from HW/SW development and operational process optimizations and engineering to security and fraud management. In 2000, he joined the Computer Engineering and Networks Laboratory at ETH, where he started his PhD studies in the field of reconfigurable computing. His research interests are in operating system design for partially reconfigurable devices and cover conceptual, algorithmic as well as practical and experimental levels. He is a member of the IEEE.

Marco Platzner received the Dipl.-Ing. and the PhD degrees in telematics from Graz University of Technology, Graz, Austria, in 1991 and 1996, respectively. From 1991 to 1996, he was with the Institute for Technical Informatics, Graz University of Technology, where he worked on embedded multiprocessors for simulation and digital signal processing applications. From 1996 to 1998, he held a postdoctoral researcher position at GMD—German National Research Center for Information Technology, St. Augustin, Germany, where he developed a robot vision system for autonomously moving vehicles. From 1997 to 1998, he was a visiting scholar at the Computer Systems Laboratory (CSL), Stanford University, Stanford, California, where he started working on reconfigurable computing systems. In 1998, he joined the Computer Engineering and Networks Laboratory at the Swiss Federal Institute of Technology (ETH) Zurich, where he leads several projects in the area of reconfigurable systems. His current research interests include reconfigurable computing, hardware-software codesign, and embedded systems. He is a member of the IEEE and the IEEE Computer Society.