Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices

Christoph Steiger, Herbert Walder, Marco Platzner, Lothar Thiele
Computer Engineering and Networks Lab
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
thiele@tik.ee.ethz.ch

Abstract

This paper deals with online scheduling of tasks to partially reconfigurable devices. Such devices are able to execute several tasks in parallel. All tasks share the reconfigurable surface as a single resource which leads to highly dynamic allocation situations. To manage such devices at runtime, we propose a reconfigurable operating system that splits into three main modules: scheduler, placer, and loader. The main characteristics of the resulting online scheduling problem is the strong nexus between scheduling and placement. We discuss a fast online placement technique and then focus on scheduling real-time tasks. We devise guarantee-based schedulers for two scenarios, namely tasks with arbitrary and synchronous arrival times. The schedulers exploit the knowledge about task properties to improve the system's performance. The experiments show that the developed schedulers lead to substantial performance gains at an acceptable runtime overhead.

1. Introduction

Reconfigurable computing systems usually consist of a host processor and a reconfigurable device, e.g., an SRAM-based Field-Programmable Gate Array (FPGA). Such systems map algorithms or parts thereof to circuits which are configured and executed on the reconfigurable device.

While early reconfigurable devices were rather limited in their densities and reconfiguration capabilities, todays devices provide millions of gates and enable partial reconfiguration and readback. This allows to configure and execute a circuit onto the device without affecting other, currently running circuits. We denote such circuits as hardware *tasks* to express their dynamic nature. An application is basically a collection of tasks which are subject to timing, precedence, and resource constraints.

While the technique of partial reconfiguration can increase device utilization, it also leads to complex allocation situations for dynamic task sets. This clearly asks for well-defined system services that help to efficiently design applications. Such a set of system services forms a *reconfigurable operating system* [1] [2] [3]. From the designer's point of view, an operating system is an additional layer of abstraction in the design process that hides details of the underlying hardware. The benefits of using the additional layer are increased design productivity, portability and resource utilization. On the other hand, these benefits are paid for by overheads in the required area and computation time. Reconfigurable operating systems are a rather new line of research. Many problems have not been addressed yet. This work deals with the resource management part of a reconfigurable operating system. In particular, we focus on the problem of scheduling real-time tasks to a partially reconfigurable device.

Section 2 discusses the modeling of the reconfigurable device, the tasks, and the operating system. Section 3 is dedicated to the placement problem and explains the implementation of our placer module. We then focus on task scheduling in Section 4. An experimental evaluation of the proposed scheduling techniques is presented in Section 5. Section 6 summarizes our findings and points to further work.

The main contributions of this paper are:

- the development of fast online placement techniques
- the development of guarantee-based scheduling algorithms for real-time tasks
- a comprehensive set of simulation experiments showing the performance and runtime efficiency of the scheduling algorithms

2. Problem Modeling and Related Work

This section first presents the modeling of the reconfigurable resource and the tasks. Then, we introduce our operating system model and define the problems of task placement and scheduling. Finally, we discuss practical limitations of these models under current technology and related work in 2D mesh-connected multiprocessors.

2.1. Resource and Task Models

A reconfigurable resource basically consists of a number of reconfigurable units (RCUs) arranged in a rectangular grid of area $W \times H$ and an interconnect between the units. Both the RCUs and the interconnect are programmable. For fine-grained reconfigurable technology, e.g., FPGAs, an RCU typically consists of combinatorial and sequential logic elements such as lookup tables, multiplexors, and flip-flops. The RCUs form the elementary resource unit that can be allocated by, at most, one task at a given time.

A task is a function synthesized to a digital circuit that can be programmed onto the reconfigurable device. A task has a *size* and a *shape*. The size gives the area requirement of the task in RCUs. We assume all task shapes to be rectangles, i.e., a task T_i is modeled as rectangular area of RCUs given by its width and height, $w_i \times h_i$. Depending on the application scenario, further task properties can be modeled. Assuming we know the number of clock cycles a task will execute in advance, we can determine the task execution time e_i by multiplying this number with the period of the system clock. Tasks can have precedence constraints or be independent with synchronous or arbitrary arrival times a_i. Real-time tasks carry deadlines d_i, $d_i \geq a_i + e_i$. Finally, the task execution can be preemptive or non-preemptive.

The mapping of tasks to a reconfigurable device includes *scheduling* and *placement* as related subproblems. The difficulty of the scheduling and placement problems strongly depends on the *area model* of the reconfigurable resource. In the following, we briefly survey area models that have been proposed (Fig. 1) with their advantages and disadvantages.

The *flexible 2D* area model (Fig. 1a)) allows to allocate rectangular tasks anywhere on the 2D reconfigurable surface. This model has been used by many authors [1] [4] [5]. The advantage is a high device utilization as tasks can be packed tightly. On the other hand, the high flexibility of this model makes scheduling and placement rather difficult and practical implementation on current technology is challenging. The development of online placement algorithms that find a good – or even feasible – allocation site for a task is not trivial. Bazargan et al. [4] address this problem and devise efficient data structures and heuristics. Placing tasks on arbitrary positions results in an *external fragmentation* of the remaining free area which may prevent the placement of subsequent tasks although sufficient free resources are available. To combat this external fragmentation, Diessel et al. [5] investigate defragmentation techniques such as local

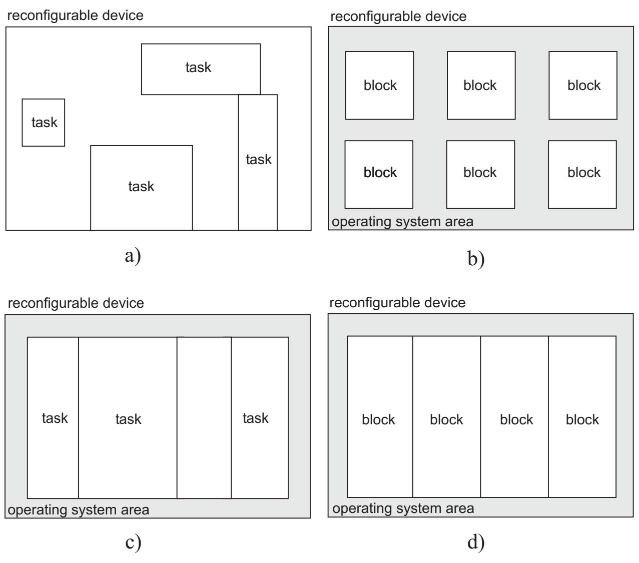

Figure 1. Reconfigurable area models: a) flexible 2D, b) partitioned 2D, c) flexible 1D, d) partitioned 1D

repacking and ordered compactation. These techniques require a preemptive task model.

Figure 1b) shows a *partitioned 2D* model where the reconfigurable surface is split into a statically-fixed number of allocation sites, so-called blocks. Each block can accommodate at most one task at a time. Such a partitioning has been proposed by Merino et al. [6] and Marescaux et al. [7]. Pre-partitioning the area greatly simplifies allocation. There is no placement issue at all, and the resulting scheduling problem is identical to multiprocessor scheduling. Further, the model facilitates practical implementation as the residual area can be reserved for operating system runtime functions. The disadvantage of the partitioned area model is the *internal fragmentation*, i.e., the area wasted when tasks are smaller than blocks.

In the *flexible 1D* area model shown in Fig. 1c) tasks can be allocated anywhere along the horizontal device dimension; the vertical dimension is fixed. This approach matches well current FPGA technology that is partially reconfigurable in vertical, chip-spanning columns, e.g., the Xilinx Virtex family [8]. The flexible 1D model leads to a placement problem of reduced complexity which is similar to problems of file sector allocation in storage systems. On the downside, the model suffers from both internal and external fragmentation which asks for defragmentation techniques [9]. Figure 1d) finally shows a *partitioned 1D* area model which combines the characteristics of a partitioned model, i.e., no placement problem, with the implementation advantages of the model in Figure 1c). Again, the disadvantage lies in the potentially high internal fragmentation.

In this paper, we consider the problem of *online scheduling and placement* of real-time tasks to the flexible 2D area model. The tasks have known execution times and deadlines, and there are no dependencies. We investigate tasks arriving at arbitrary times as well as synchronously arriving task sets.

2.2. Operating System Model

Fig. 2 shows the proposed model of a reconfigurable operating system. The target architecture comprises a host processor and the reconfigurable device. Arrived tasks are stored in a queue where they await further processing. The resource management is implemented by three modules running on the host, the scheduler, the placer, and the loader:

- *Scheduler*: The scheduler receives incoming tasks and uses a scheduling algorithm to assign starting times to the tasks. The scheduler relies on the placer as a subfunction.
- *Placer*: The placer manages the free space on the device and tries to find feasible placements for the tasks. A poor placer leads to a badly fragmented device with many small areas scattered over the surface. A good placer achieves a low fragmentation of the device surface which allows for the execution of more tasks in parallel.
- *Loader*: Whenever a suitable location for a task has been found, the loader is called to conduct all steps necessary for configuring this task onto the reconfigurable device.

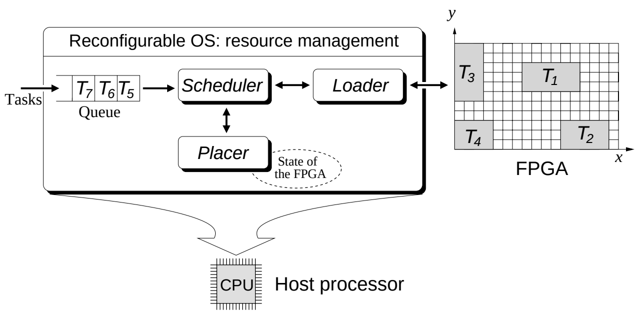

Figure 2. System model

The nexus of scheduling and placement is the main characteristics for the problem of mapping tasks to partially reconfigurable devices. There is no guarantee that the placer finds a feasible placement for every task that has been selected for execution. As a consequence, the scheduling algorithm must incorporate a placement algorithm.

In this work we focus on a non-preemptive system model – once a task is loaded onto the device it runs to completion. Although preemption is a very useful technique that yields efficient and fast online scheduling algorithms, we consider it too expensive for current reconfigurable technology. Preempting and resuming tasks requires context switches. Contrary to processor-based systems where context switches involve saving rather small task control blocks, including the program counter and other registers, the context of a reconfigurable task comprises the states of all state-holding logic elements of the task's circuit. A reconfigurable task's context can easily amount to several kBytes of data. Preempting and restoring such a task would not only introduce a considerable delay but also require additional external memory for storing contexts.

2.3. Scheduling, Placement, and Allocation

The following paragraphs define the problem of scheduling tasks with deadlines to a device using the flexible 2D area model. The device's surface is indexed with a coordinate system such that each reconfigurable unit (RCU) receives the coordinates $\langle x,y \rangle$, with x being the column and y being the row index. Column and row indices increase from left to right and from bottom to top, respectively. Each task T_i executing on the device is identified by its *placement*:

Definition 1 (Placement) *A placement $p(T_i)$ for a task T_i is given by the bottom-left RCU $\langle x_i, y_i \rangle$ of the rectangular area the tasks occupies on the reconfigurable device.*

Tasks must not overlap with the device boundaries and other currently placed tasks. Hence we define a *feasible placement* as:

Definition 2 (Feasible Placement) *A feasible placement $p^f(T_i, \mathcal{T})$ for a task T_i and a set of currently placed tasks $\mathcal{T}$ is a placement $\langle x_i, y_i \rangle$ that satisfies following constraints:*

- $x_i + w_i \leq W; \quad y_i + h_i \leq H$
- $\forall T_j \in \mathcal{T}$:
  $(x_i + w_i \leq x_j) \vee (x_j + w_j \leq x_i) \vee$
  $(y_i + h_i \leq y_j) \vee (y_j + h_j \leq y_i)$

In the following we will only be interested in feasible placements and, hence, use the term placement to denote a feasible placement for the sake of brevity. At any given time the placements of the currently placed tasks $\mathcal{T}$ form the *allocation*.

Definition 3 (Allocation) *An allocation A is the set of placements of all currently placed tasks:*

$$A = \{p(T_i) \,|\, \forall\, T_i \in \mathcal{T}\} \tag{1}$$

A *schedule* for a task set $\mathcal{T}$ is defined as follows:

Definition 4 (Schedule) *A schedule assigns a starting time s_i to each task $T_i \in \mathcal{T}$.*

Finally, a *feasible schedule* for a task set $\mathcal{T}$ is a schedule that meets all deadlines and includes placements for all tasks during their complete execution periods. In other words, for every two tasks there must be at least one dimension (out of the spatial dimensions x and y and the time) in which the tasks do not overlap:

Definition 5 (Feasible Schedule) *A feasible schedule assigns a starting time s_i to each task $T_i \in \mathcal{T}$ such that*

- $s_i + e_i \leq d_i$
- $\forall T_j \in \mathcal{T}, T_j \neq T_i$:
  $(x_i + w_i \leq x_j) \vee (x_j + w_j \leq x_i) \vee$
  $(y_i + h_i \leq y_j) \vee (y_j + h_j \leq y_i) \vee$
  $(s_i + e_i \leq s_j) \vee (s_j + e_j \leq e_i)$

In an online scenario new tasks arrive during the system's runtime. For each arriving task T_i the scheduler has to find a feasible schedule for the task set $\mathcal{T} \cup T_i$, given $\mathcal{T}$ is the currently scheduled task set. In this paper, we consider an online *hard real-time* system that runs an acceptance test for each arriving task. A task passing this test is 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. The scheduling goal is to minimize the number of rejected tasks. Accept/reject mechanisms are typically adopted in dynamic real-time systems [10] and assume that the scheduler's environment can react properly on a task rejection, e.g., by migrating the task to a different computing resource.

2.4. Limitations

The discussed task and device models are consistent with related work in the field [1] [9] [2] [5] [4] [11]. However, we also have to discuss the limitations when it comes to practical realization on currently available technology. The main abstraction is that tasks are modeled as relocatable rectangles. While the latest design tools allow to constrain tasks to rectangular areas, the relocatability rises questions concerning the i) device homogeneity, ii) task communication and timing, and iii) partial reconfigurability.

We assume device homogeneity (surface uniformity) which is in contrast with modern FPGAs that contain special resources such as block memories and embedded multipliers. However, a reconfigurable operating system (OS) takes many of these resources (e.g., block RAM) away from the user task area. Tasks must use predefined communication channels to access such special resources [7] [12].

Arbitrarily relocated tasks that communicate with each other and with I/O devices would require online routing and delay estimation of their external signals, neither of which is sufficiently supported by current tools. The state-of-the-art in reconfigurable OS prototypes [7] [12] overcomes this problem by using area models that partition the reconfigurable surface into fixed-size blocks. These OSs provide predefined communication interfaces to tasks and asynchronous intertask communication. For the flexible 2D model, communication remains to be a complex issue. Related work mostly assumes that sufficient resources for communication are available [5]. Practical implementations either communicate via the task areas using configuration and readback mechanisms, which is feasible but presumably inefficient, or propose to leave some space between tasks for communication channels and perform online routing.

The partial reconfiguration capabilities of the Xilinx Virtex family, which reconfigures a device in vertical chip-spanning columns, 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 non-reconfigured device areas during task reconfiguration is tricky.

In summary, given current technology the 1D area model has already been demonstrated whereas the 2D model faces some challenging issues. Most of the related work on 2D models targets the (meanwhile withdrawn) FPGA series Xilinx XC6200 that is reconfigurable on the logic block level and has a publicly available bitstream architecture. As the flexible 2D model has great advantages over the other models in terms of scheduling performance, we believe that it is worthwhile to investigate and develop algorithms for it. Requirements for future devices supporting the 2D model include block-based reconfiguration and a built-in communication network that is not affected by user logic reconfigurations.

2.5. Related Work in (Multi-)Processors

The scheduling problem we consider in this paper differs from most classical single and multiprocessor scheduling problems due to the nexus between scheduling and placement. A single CPU executes only one task at any given time and there is no placement issue concerning the CPU resource. 1D placement problems arise in this context when it comes to memory allocation.

There is a class of multiprocessor problems which shows similarity to our scenario: allocating and scheduling parallel tasks to 2D mesh-connected multiprocessors. The processors of the 2D mesh correspond to our reconfigurable units (RCUs). A parallel task requires a rectangular grid of processors (submesh), like a reconfigurable task requires a rectangular area of RCUs. Although there exists a significant body of work in multiprocessor allocation and scheduling, see e.g., [13], there are some important differences to our work:

- reconfigurable tasks must not be rotated
- reconfigurable tasks are not malleable
- reconfigurable tasks cannot be preempted or restarted
- related work concentrates on placement rather than scheduling (which is mostly FIFO)
- related work focuses on general-purpose work loads with the goal to minimize the makespan or average response time

The placement algorithms used in the related work 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 the scanning placers is that they are *recognition-complete*, see e.g. [13]. This means they are optimal in the sense that they find a feasible placement for a new task if sufficient resources (idle processor submeshes) are available. The drawback of the scanning placers is their high runtime complexity. In the worst case, they have to consider $W \times H$ potential placements. In the case of partially reconfigurable devices and realistic scenarios (see Section 5) two orders of magnitude higher runtimes are observed in comparison to the placement strategies proposed in this paper.

In a recent paper [14], real-time tasks and their mapping onto a 2D parallel computer are considered. In comparison to the approach presented here, task preemption is partly assumed and the overall run-time complexity of the scheduling and placement algorithm is too high for the application scenarios of this paper.

3. Task Placement

The main function of the placer module is the management of the free space on the reconfigurable device. There exist two different approaches, *scanning* and *maintaining free rectangles*.

Scanning techniques have been used in the context of allocating parallel tasks to 2D mesh-connected multiprocessors (see Section 2.5). Managing the free space by a set of free rectangles is possible because the device and the tasks are modeled as rectangles and the placements have to be orthogonal and oriented. Bazargan et al. [4] use such a technique and present two placement methods. The first method keeps all maximal free rectangles, i.e., rectangles that are not contained in any other rectangles. This has been shown to be recognition-complete. However, the method has to manage $O(n^2)$ rectangles for n placed tasks. Therefore a second method was presented that sacrifices optimality for shorter runtime and keeps $O(n)$ non-overlapping rectangles.

The work presented in this paper uses an extension of Bazargan's $O(n)$ placement method which is appropriate for an online scheduler. We have done various runtime comparisons with scanning placers. The results show that the runtimes of scanning placers are two orders of magnitude higher that the runtimes for the fast placers based on free rectangle lists.

Generally, a placer contains two subfunctions. The *partitioner* maintains the free rectangle list and performs insert and delete operations. Whenever a task is to be placed and there is more than one free rectangle that leads to a feasible placement, the *fitter* selects one. Several fitting strategies can be applied. *Best-fit*, for example, chooses the smallest free rectangle which is big enough to accommodate the task.

In the following, we first briefly discuss the partitioning technique introduced by Bazargan et al. and then present our enhancements.

3.1. Bazargan's Partitioner

At any point in time, the partitioner keeps a list of non-overlapping free rectangles. Upon arrival of a new task T_i, a rectangle R is selected from the list of rectangles that are big enough to accommodate T_i by some fitting strategy. The placer configures the task in the bottom-left corner of R which splits the residual area of R into at most two smaller rectangles R' and R''. The split can either be done vertically or horizontally as shown in Fig. 3(a) and Fig. 3(b), respectively. Bazargan et al. proposed several heuristics to decide on which of the two splits should be performed. Because a free rectangle can split into two new rectangles at most, a *binary tree* can be used to represent the device state. Each node n_R of the tree represents a rectangle R. n_{R_2} is a child of node n_{R_1} if it was created upon inserting a task into R_1. The leaves of the tree represent the currently free rectangles. If the leaves are additionally linked into a linear list, inserting a task by best-fit – or detecting that it cannot be accommodated – is done in $O(n)$ time, given n is the number of currently placed tasks.

Fig. 4 gives an example, showing the state of the reconfigurable surface as well as the corresponding rectangle tree. Initially, the device is empty and the free space is represented by a single rectangle A (Fig. 4(a)). Then, task T_1 is inserted into A and the remaining free space is split vertically into B and C (Fig. 4(b)). Similarly, Fig. 4(c) shows the result of the subsequent insertion of T_2. Assuming that T_2 finishes execution while T_1 is still executing, the rectan-

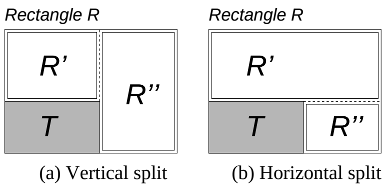

Figure 3. Splitting decisions

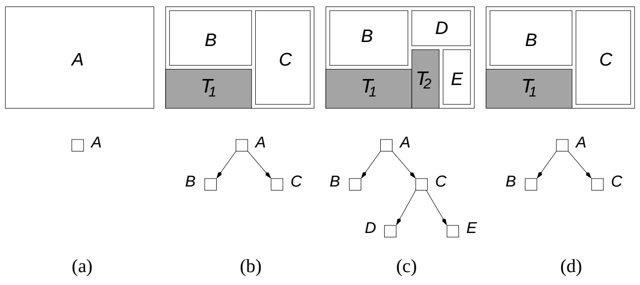

Figure 4. Free space management

gle that held T_2 is merged with D and E to form C again (Fig. 4(d)).

Task deletion is more involved than task insertion. The placer marks a rectangle released by a terminating task as free and then tries to merge it with other free rectangles to form a maximal free rectangle. The merging step is crucial as it avoids managing the free space with smaller and smaller rectangles.

Whenever a task T is deleted from its container rectangle R, a new free rectangle X – covering the area formerly occupied by T – is inserted into the rectangle tree as an additional child of R. Then, the placer tries to merge X with all its *brother rectangles*. The merging step includes deleting the merged rectangles and marking the parent node n_R, which is now a leaf node, as free rectangle. Merging propagates iteratively up the tree as long as all brothers can be merged or the root is reached.

Fig. 5 gives an example. After three tasks T_1, T_2 and T_3 have been placed, the allocation shown in Fig. 5(a) results. Deleting T_1 introduces a new free rectangle X as an additional child of A. At this point, no merge is possible (Fig. 5(b)). Supposing T_2 finishes next, the placer conducts one merge operation: the newly created rectangle in place of T_2 is merged with D and E, i.e., rectangle C becomes free again (Fig. 5(c)). Upon the subsequent termination of task T_3, the free rectangle is merged with F, and B is marked as free again (Fig. 5(d)). This gives room for a next merge: B is merged with its brothers and the initial device state A is reached again (Fig. 5(e)).

The complexity of the delete operation depends on the

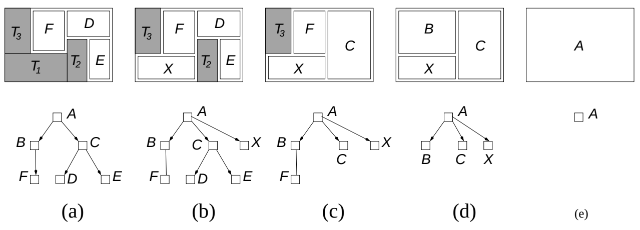

Figure 5. Delete and merge

number of merge steps which is bound by the longest path from a leaf to the root in the rectangle tree. In the worst case, the placer might have to visit all nodes on this path, which results in a complexity of $O(n)$.

3.2. Enhanced Partitioners

We have developed enhanced versions of the partitioner presented by Bazargan et al. [4]. The enhancements increase the average size of the managed free rectangles, raising the chances of successful placements. The performance is greatly improved while the efficiency is completely preserved. An evaluation of the enhancements compared to Bazargan's original partitioner has been published previously [15], but for non real-time scheduling problems. Hence, we concentrate here on presenting the main principle.

Bazargan's placer uses heuristics to decide whether a free rectangle is split vertically or horizontally upon task insertion. No matter how good such a heuristics is, the possibility of conducting the wrong split remains. That is, the next task cannot be placed in one of the resulting rectangles due to a wrong previous split decision. The key to improve the partitioner performance is to delay the split decision. Whenever a task T_1 is inserted into a rectangle, two *overlapping* child rectangles A and B are created as shown in Fig. 6(a). The actual split decision is not made until the next task T_2 for one of the two children, A or B, arrives. If the next task is inserted into A (B), the height (width) of rectangle B (A) is adjusted such that rectangles do not overlap any more (Fig. 6(b) and Fig. 6(c)).

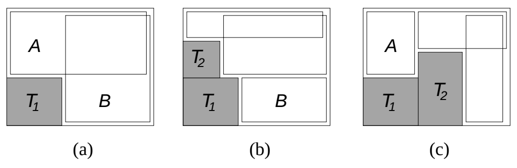

Figure 6. Enhanced partitioner: task insertion

The enhanced partitioner improves quality by taking advantage of keeping non-overlapping rectangles and, thus, avoiding a heuristic split decision at all. On the other hand, the efficiency is preserved because we only overlap free rectangles with the same parent node. The modifications to the previous algorithm are small. When a task is inserted into rectangle R, we check whether R overlaps with its brother. In such a case we resize the width or height of the brother. The task deletion procedure remains unchanged.

The split decision can be delayed even further, which leads to our *On-the-fly (OTF)* partitioner. The general idea of this method is shown in Fig. 7. Task T_1 has been inserted and two overlapping child rectangles A and B have been created (Fig. 7(a)). Now task T_2 arrives and is to be placed in rectangle B. The enhanced partitioner resizes rectangle A as shown in Fig. 7(b). However, T_2 does *not* overlap with rectangle A. Therefore, one can leave A at its original size, getting a better partition of the free space (Fig. 7(c)). By this the split decision is deferred to a later point in time.

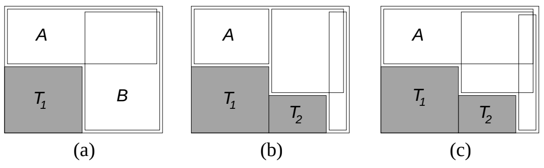

Figure 7. Motivation for OTF partitioning

The price paid is that it might be necessary to resize several rectangles on a task insertion. Whenever a task is inserted into rectangle A (B) which overlaps with brother rectangle B (A), the heights (widths) of rectangle B (A) and all rectangles in the subtree rooted at B (A) are adjusted such that they do not overlap anymore with rectangle A (B). This process might continue with the parent of A and B and can, potentially, propagate up to the root of the rectangle tree. Consequently, the worst-case runtime complexity for inserts increases to $O(n)$.

Fig. 8 gives an illustration by extending the example of Fig. 7(c). Fig. 8(b) shows the result of inserting a task T_3 into rectangle A which overlaps B. The whole subtree of rectangles rooted at B has to be resized, i.e., the height of the rectangles in the subtree is adjusted. Fig. 8(c) shows the insertion of a task into a free rectangle in the subtree rooted at B, triggering the resizing of rectangle A.

The implementation of the OTF partitioner differs from the enhanced Bazargan partitioner in that all rectangles of brother subtrees might be resized at a later point in time. Task deletion remains unchanged.

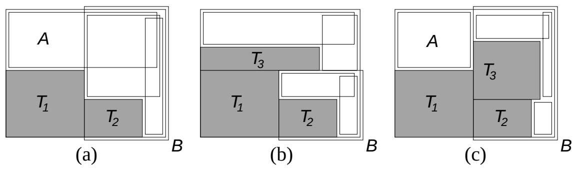

Figure 8. Resizing of brother rectangles

4. Scheduling Real-time Tasks

We consider an online *hard real-time* system that runs an acceptance test for each arriving task. A task passing this test is 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. The scheduling goal is to minimize the number of rejected tasks.

As a reference point, we define a *basic scheduler* that considers arriving tasks one-by-one and accepts a new task T_i with arrival time a_i, execution time e_i, and deadline d_i only if a feasible placement can be found immediately. Otherwise the task is rejected.

Although such a basic scheduler shows limited performance, its implementation is simple and fast as it only calls the placer once for each arriving task. In the following, we discuss ways to improve the scheduling behavior. First, we consider a scenario where tasks arrive at arbitrary times. Then, we look at the case when several tasks arrive synchronously. All presented scheduling algorithms are heuristics and can therefore not guarantee that an optimal schedule, or a schedule with a given competitive ratio compared to an optimal off-line schedule, is established. The principles are general in that they do not dependent on the underlying placer module.

4.1. Arbitrary Arrival Times

The number of accepted tasks can be increased by looking into the future, i.e., by *planning*. Although a task might not be immediately placeable, the scheduler might be able to allocate the task at some future time and still meet the deadline. This situation is depicted in Fig. 9. Neglecting any runtime overheads by the scheduler-placer system and the loader, the last possibility to place a task T_i and meet its deadline is at time $s_{i-latest} = d_i - e_i$. In other words, the planning period for possible task placements is $d_i - a_i - e_i$, the task's *laxity*.

The main difficulty in devising the scheduler lies in the fact that the planning process not only has to find a starting time for the task, but also a free area on the device. For tasks that have such a future placement, the scheduler performs a reservation of the rectangular area on the device during the task execution period. With respect to already

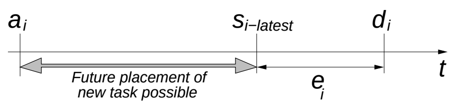

Figure 9. Planning period for real-time tasks

placed and guaranteed tasks, the future of the device is determined. As shown in Fig. 10, there are two types of events during the planning period. First, n' of the n placed tasks terminate which generates new free areas on the device. Second, m' of the m previously guaranteed tasks are placed on the device which reduces the free area.

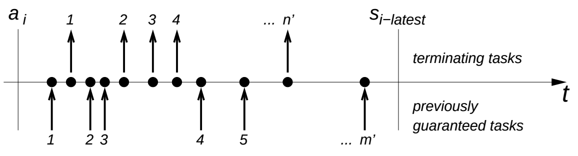

Figure 10. Events during the planning period

The scheduler implementation maintains three data structures:

- a list E of elements $(T_x, f_x, p(T_x))$ that represent currently executing tasks T_x with their finishing times $f_x = s_x + e_x$ and placements $p(T_x)$, sorted according to increasing finishing times
- a list G of elements (T_x, s_x) that represent guaranteed tasks with their starting times s_x, sorted according to increasing starting times
- a list K of elements (R_j, s_j, f_j, T_j) that denote the reserved rectangle R_j (determined by the placement $p(T_j)$) together with the reservation period (s_j to f_j) and a pointer to the corresponding task T_j in G

The two basic operations are the termination of an executing task and the start of a guaranteed task. The function $terminate(T_i, A_C)$ updates the current allocation A_C and removes T_i from E. The function $start(T_i, A_C)$ updates the current allocation, removes T_i from G, adds it to E, and removes the reservation from K.

Algorithm 1 shows the main loop of the scheduler's planning operation. When a new task T_i arrives, the scheduler starts walking through the task's planning period, simulating the future of the device by mimicking the task terminations (line 4) and starts (line 8) together with the underlying free space management (simulated allocation A_S). On arrival of T_i and any task termination, the placer is called to find a free area for T_i (line 11). The placer function $placer(T_i, A_S)$ returns either an empty set when there is no free rectangle that can accommodate T_i in the current simulated allocation A_S, or a list of free rectangles sorted according to the best-fit metrics, i.e., the smallest rectangle fitting T_i is the first element. If a rectangle $R_x \in BF$ exists, the scheduler checks whether R_x is in conflict with an existing reservation – according to Definition 5 of a feasible schedule in Section 2.3. A conflict exists if R_x overlaps with a reserved rectangle *and* the execution period of T_i overlaps with the execution period of the task holding the reservation. If there are no conflicts, a new reservation is added and T_i is accepted. If there is no free rectangle large enough to accommodate T_i or there are conflicts the planning process proceeds. If the planning process exceeds $s_{i-latest}$, T_i is rejected. A task is never accepted in the presence of conflicts. Thus, previously issued guarantees are never changed by any arriving task.

Algorithm 1 : planning (T_i, A_C)

```
1:  $A_S \leftarrow A_C; t \leftarrow a_i$
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:          $terminate(T_j, A_S)$
6:          $check \leftarrow TRUE$
7:      end for
8:      for all $T_j \in G$ with ($s_j = t$) do
9:          $start(T_j, A_S)$
10:     end for
11:     if $check$ then
12:         $BF \leftarrow placer(T_i, A_S)$
13:         for all ($R_x \in BF$) do
14:             if (($R_x, t, t + e_i$) is not conflicting any $K_y \in K$) then
15:                 add ($T_i, t$) to $G$
16:                 add ($R_x, t, t + e_i, T_i$) to $K$
17:                 return(ACCEPT)
18:             end if
19:         end for
20:         $check \leftarrow FALSE$
21:     end if
22:     $t \leftarrow$ next event from $E \cup G$
23: end while
24: return(REJECT)
```

The worst-case complexity for planning one task amounts to $O(n^2 m)$: During the planning period $m' \leq m$ tasks will start and $n' \leq n$ tasks will terminate. Starting a task involves detecting ready tasks which is done in constant time as G is sorted, and inserting into the sorted list E which is done in $O(log(n))$. The insert procedure of the OTF placer runs in $O(n)$ (see Section 3.2). Overall, task starts amount to $O(mn)$. Terminating a task involves detecting finished tasks which is done in constant time as E is sorted, deleting the task placement from the allocation, and calling the placer. The OTF placer's delete operation runs in $O(n)$ time (see Section 3.2). Called to find a feasible placement, the OTF placer will return at most $O(n)$ free rectangles that have to be checked for conflicts. Checking for a conflict requires to compare each rectangle with all reservations from K which takes $O(m)$ time. Additionally, a guaranteed task is inserted into G which takes $O(log(m))$ time, and its reservation is inserted into K which takes constant time. In summary, a task deletion and the subsequent placement effort amounts to $O(nm)$. As the number of terminating tasks is bound by $O(n)$, the overall planning complexity is $O(n^2m)$.

4.2. Synchronous Arrival Times

In this scenario, the scheduler has to deal with several tasks with their corresponding areas, execution time requests, and deadlines. We are interested in fast algorithms that immediately decide on the acceptance or rejection for each task of the arriving task set. Therefore, we cast this scheduling problem as an online problem where we first order the tasks in a list according to some criterion, and then schedule the tasks from the list one-by-one.

The two main issues are to find a criterion for the task order and, characteristic for scheduling to dynamically reconfigurable devices, to handle situations where the placer cannot find a placement for the next task in the list.

We can take the task area as a possible criterion and schedule small tasks first – a heuristic that can be seen as the spatial counterpart of scheduling short tasks first which, on single CPUs, minimizes the average response time. On the other hand, the task deadlines are crucial which points to scheduling tasks with earlier deadlines first – a heuristic which corresponds to Jackson's algorithm (Earliest Deadline Due) for single CPUs. The two concepts of *scheduling small tasks first* and *scheduling tasks with early deadlines first* can clearly be conflicting. We have experimented with two different functions to assign priorities v_i to the tasks T_i of the arriving task set T. The first is a linear weighted combination of the task size $w_i \cdot h_i$, given by number of required reconfigurable units (RCUs), and the deadline d_i:

$$v_i = A_1 \cdot \frac{w_i \cdot h_i}{\max_{T_j \in T} \{w_j \cdot h_j\}} + A_2 \cdot \frac{d_i}{\max_{T_j \in T} \{d_j\}} \tag{2}$$

The function normalizes a task's area to the maximum area for any task in the task set, and the task's deadline to the latest deadline for any task in the task set. The weighting coefficients are constrained by $A_1 \in [0,1], A_2 \in [0,1]$ and $A_1 + A_2 = 1$. In the experiments section, we try to determine settings for these coefficients that yield the best scheduling performance.

The second function is a non-linear combination of the task parameters:

$$v_i = \frac{1}{w_i \cdot h_i \cdot d_i} \tag{3}$$

Whenever a set of tasks T arrives, the scheduler performs the following steps:

1. For every task $T_i \in T$, the priority $v_i = f(w_i, h_i, d_i)$ is calculated.
2. All tasks are sorted in increasing order according to v_i.
3. The tasks are passed on to the placer one-by-one in the order determined in step 2.

Asymptotically, steps 1 and 2 are done in $O(nlog(n))$ time. In step 3, the placer might be unable to find a placement for a task on the reconfigurable device. There are two possibilities to deal with this situation: *strict* order enforcement (SOE) and *relaxed* order enforcement (ROE).

SOE schedules a task onto the reconfigurable device only if all tasks with higher priorities have already been scheduled. If a task cannot be placed in step 3, the scheduler waits until the device allocation changes (due to terminations of already scheduled tasks) and then calls the placer again to find a placement. This is done until the task can be placed or its planning period expires.

ROE puts an unplaceable task on hold and proceeds with the next task in the list. Whenever the device allocation changes, the scheduler processes the waiting tasks in the order established by the list.

Our guarantee-based scheduler makes a decision for each task of the task set immediately after the set arrives. Step 3 of the above procedure includes a planning routine similar to the one in Algorithm 1. The planning routine simulates the future of the device by mimicking task terminations and starts, together with the underlying free space management. After planning, the scheduler knows which tasks will meet their deadlines and which tasks have to be rejected, with respect to the established task order.

5. Simulation Results and Evaluation

5.1. Simulation Setup

We have constructed a discrete time simulation framework in C++ language to experimentally evaluate the performance of the schedulers. Runtime measurements have been conducted on a Pentium-III 1000MHz.

The simulated device consists of $96 \times 64 = 6144$ reconfigurable units (RCU), which corresponds to Xilinx's XCV1000 FPGA [8]. We use task sets with randomly generated areas, execution times, and deadlines. The distributions

reconfigurable core	area [RCUs]
UART [16]	50
100-tap FIR filter [17]	250
ADPCM [18]	250
DCT [19]	600
Triple-DES processor [17]	800
256 point complex FFT [17]	850
Protocol stack: Ethernet-MAC, IP, UDP [16]	1050
Discrete Wavelet Transform [19]	1800
LEON Sparc-V8 core [20]	2000

Table 1. Area requirements for typical cores

are based on assumptions about real-world scenarios. We have simulated four classes of tasks differing in their area requirements. The classes are denoted by C_i and represent tasks of equally distributed areas in $[50,i]$ reconfigurable units (RCUs). Taking the sizes of typical cores mapped to reconfigurable devices into account (see Table 1), we have simulated $C_i : i = \{100,500,1000,2000\}$.

For every C_i, 50 task sets have been generated and the simulation results have been averaged. The number of tasks has been set to achieve a confidence level of 95% with an error range of at most $\pm 4\%$. The scheduling heuristics have been evaluated with all task classes. Since the results show similar trends we concentrate on task class C_{500} in the following. For all task classes, task computation times are equally distributed in $[5,100]$ time units. Under the assumption that one time unit corresponds to $50\,ms$, we consider tasks with computation time of $250\,ms - 5000\,ms$. We consider three different laxity classes A, B, and C, with laxities uniformly distributed in $[1,50]$, $[50,100]$ and $[100,200]$ time units, respectively.

For tasks arriving at arbitrary times, we have additionally generated random arrival times. In the following figures, we give a load parameter instead of the arrival time distribution. The load which the scheduler-placer system experiences is a decisive factor for the performance as it heavily influences the number of missed deadlines. Contrary to CPU scheduling, we demand for a special definition of task load that incorporates task size. We denote the set of tasks arriving in the time interval Δt with $\mathcal{T}$. Every task $T_i \in \mathcal{T}$ occupies $w_i \cdot h_i$ reconfigurable units on the device for e_i time units. The reconfigurable device has a size of $W \times H$ reconfigurable units. We define the load $L_{\mathcal{T},\Delta t}$ as follows:

$$L_{\mathcal{T},\Delta t} := \frac{\sum_{T_i \in \mathcal{T}} w_i \cdot h_i \cdot e_i}{W \cdot H \cdot \Delta t} \qquad (4)$$

This definition takes into account the *capacity* offered by the reconfigurable device as well as the capacity requested by the task set. Δt is set to span from the arrival of the first task to the arrival of the last task plus its execution time. Theoretically, a task set with load $L = 1$ might be schedulable so as to reduce all task waiting times to zero. However, in practice a load of 1 is likely to cause significant waiting times due to (i) internal device fragmentation and (ii) suboptimal placement decisions.

5.2. Arbitrary Arrival Times

Fig. 11 shows the percentage of rejected tasks for the planning scheduler (Algorithm 1) based on the OTF placer (see Section 3.2) and area class C_{500} in dependency of the load and the laxity classes A, B, and C. Further, as a reference we include the basic scheduler (see Section 4) that performs no planning at all. The figure shows that planning is beneficial as the basic scheduler is outperformed by the planning schedulers for all loads. The later the deadlines are, the more planning pays off. For the reasonable load $L = 0.5$ and laxity class C, planning reduces the rejection ratio from 15.5% to 0.6%.

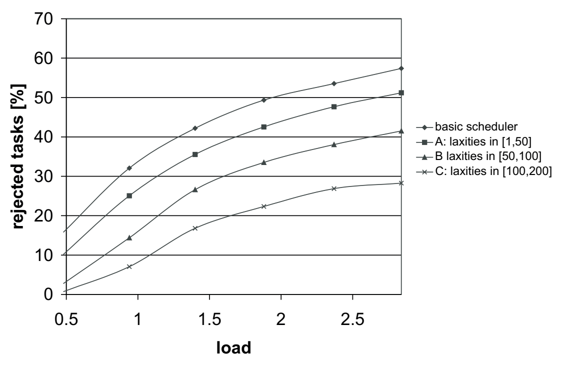

Figure 11. Performance of the schedulers for arbitrarily arriving tasks of class C_{500}

Fig. 12 shows the average runtime spent in the scheduling and placement routines for a single task. The figure pictures the results for task class C_{500} and a load of $L = 0.94$. As expected we observe an increase in the runtime for later deadlines as more planning needs to be done. With at most $1.8\,ms$, the runtime overhead of the scheduler seems to be affordably low compared to task execution times which are in the range $250\,ms - 5000\,ms$ (see Section 5.1).

5.3. Synchronous Arrival Times

Fig. 13 shows the percentage of rejected tasks for the planning scheduler based on the OTF placer (see Section 3.2), task class C_{500}, and laxity class C in dependency of

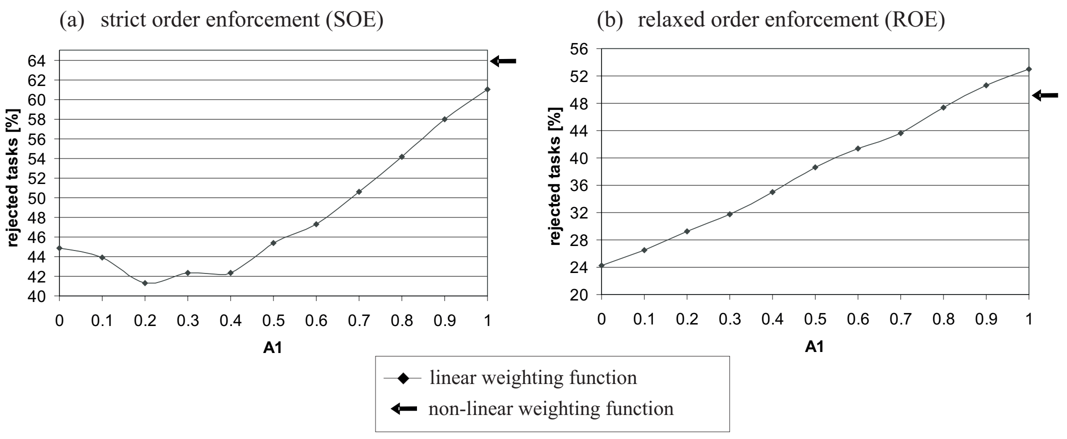

Figure 13. Performance of the scheduler for synchronously arriving tasks; class C_{500}, laxity class C

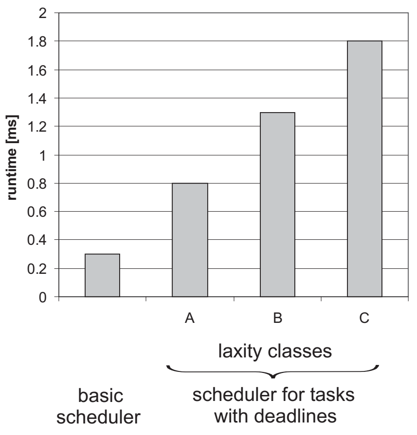

Figure 12. Runtime of the schedulers for arbitrarily arriving tasks; class C_{500}, load $L = 0.94$

the weighting coefficient A_1 (see Section 4.2). Fig. 13(a) presents the data for a scheduler with strict order enforcement (SOE) and Fig. 13(b) for relaxed order enforcement (ROE). As expected, ROE yields much better performance than SOE. Nevertheless, the SOE case is interesting as it most clearly shows the effects of giving the task parameters (size and deadline) different weights in the priority function. All our simulation showed that for ROE the best results are achieved for $A_1 = 0$. As shown in Fig. 13(b) the rejection ratio starts at 24.3% for $A_1 = 0$ and increases almost linearly to 53.1% for $A_1 = 1$, where only task sizes are considered. This means that for relaxed order enforcement one should totally rely on deadlines for prioritizing tasks. Note that ROE implicitly considers the tasks areas as it allows to schedule (smaller) tasks with lower priorities before (larger) tasks with higher priorities. SOE is different in that here both parameters, task size and deadline, should be considered in the priority function. For SOE, the best results were achieved for $A_1 = [0.2, 0.4]$.

The arrows in Fig. 13 indicate the scheduling performance for the non-linear weighting function (see Section 4.2). For neither SOE nor ROE, the non-linear function leads to competitive results.

We further measured the runtime spent in the scheduling and placement routines for the overall task set which was, on the average, about 10 *ms*.

6. Conclusion and Further Work

In this paper, we discussed the problem of online scheduling real-time tasks to partially reconfigurable devices. We presented an operating system model and identified three main modules: the *scheduler*, the *placer* and the *loader*. The main characteristics of the involved scheduling problems is the strong nexus between scheduling and placement. We then focused on online scheduling tasks with deadlines arriving at arbitrary times and synchronously. We presented a fast placement technique and planning schedulers that run an acceptance test for arriving tasks to decide on acceptance and rejection. Finally, we evaluated the schedulers by means of simulation on randomly generates task sets.

Further research activities are planned along the following lines:

- Evaluation of the schedulers with different placement techniques.
- Development of schedulers for tasks with precedence constraints.
- Extensions of the scheduling algorithms to cover more realistic task and device models. For example, issues

of routing, intertask communication and I/O must be incorporated.

- Investigation of online schedulers for the heterogeneous target architecture consisting of the host processor and the reconfigurable device. Besides running the operating system functions such as scheduler, placer, and loader, the host processor can also execute application tasks. This raises questions such as hardware/software partitioning and combining preemptive CPU scheduling techniques with the non-preemptive techniques for reconfigurable devices presented in this paper.

7. Acknowledgments

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

References

[1] Gordon Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proc. 6th Int'l Workshop on Field-Programmable Logic and Applications (FPL)*, pages 327–336, 1996.

[2] Grant Wigley and David Kearney. The Development of an Operating System for Reconfigurable Computing. In *Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM)*, April 2001.

[3] Herbert Walder and Marco Platzner. Non-preemptive Multitasking on FPGA: Task Placement and Footprint Transform. In *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 24–30. CSREA Press, June 2002.

[4] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. In *IEEE Design and Test of Computers*, volume 17, pages 68–83, 2000.

[5] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. In *IEE Proceedings on Computers and Digital Techniques*, volume 147, pages 181–188, May 2000.

[6] Pedro Merino, Margarida Jacome, and Juan Carlos Lopez. A Methodology for Task Based Partitioning and Scheduling of Dynamically Reconfigurable Systems. In *Proc. IEEE Symopsium on FPGAs for Custom Computing Machines (FCCM)*, pages 324–325, 1998.

[7] Théodore Marescaux, Andrei Bartic, Verkest Dideriek, Serge Vernalde, and Rudy Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs. In *Proc. 12th Int'l Conf. on Field-Programmable Logic and Applications (FPL)*, pages 795–805, 2002.

[8] Xilinx, Inc. *Virtex^{RM} 2.5 V Field Programmable Gate Arrays*, December 2002.

[9] Gordon Brebner and Oliver Diessel. Chip-Based Reconfigurable Task Management. In *Proc. 11th Int'l Workshop on Field Programmable Gate Arrays (FPL)*, pages 182–191, 2001.

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

[11] Sandor Fekete, Ekkehard Köhler, and Jürgen Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Proc. Design Automation and Test in Europe (DATE)*, pages 658–665, 2001.

[12] H. Walder and M. Platzner. Reconfigurable Hardware Operating Systems: From Concepts to Realizations. In *Proc. 3rd International Conf. on Engineering of Reconfigurable Systems and Architectures (ERSA)*, 2003.

[13] Dror Feitelsen. Job Scheduling in Multiprogrammed Parallel Systems. In *IBM Research Report*, volume RC 87657, August 1997.

[14] S-M. Yoo, H.Y Youn, and H. Choo. Dynamic Scheduling and Allocation in Two-Dimensional Mesh-Connected Multicomputers for Real-Time Tasks. In *IEICE Trans. Inf. and Syst.*, volume E84-D, pages 613–622, 2001.

[15] Herbert Walder, Christoph Steiger, and Marco Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In *Proceedings of the Reconfigurable Architectures Workshop (RAW'03)*, April 2003.

[16] M. Lerjen and C. Zbinden. Reconfigurable Bluetooth–Ethernet bridge. Master's thesis, Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH) Zurich, 2002.

[17] Xilinx Inc., Virtex Core Generator.

[18] M. Dyer and M. Wirz. Reconfigurable system on FPGA. Master's thesis, Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH) Zurich, March 2002.

[19] Amphion Semiconductor Ltd., `www.amphion.com`.

[20] J. Gaisler. *The LEON Processor User's Manual, Version 2.3.7.* Gaisler Research, August 2001.