Heuristics for Online Scheduling Real-time Tasks to Partially Reconfigurable Devices*

Christoph Steiger, Herbert Walder, and Marco Platzner

Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
platzner@tik.ee.ethz.ch

Abstract. Partially reconfigurable devices allow to configure and execute tasks in a true multitasking manner. The main characteristics of mapping tasks to such devices is the strong nexus between scheduling and placement. In this paper, we formulate a new online real-time scheduling problem and present two heuristics, the *horizon* and the *stuffing* technique, to tackle it. Simulation experiments evaluate the performance and the runtime efficiency of the schedulers. Finally, we discuss our prototyping work toward an integration of scheduling and placement into an operating system for reconfigurable devices.

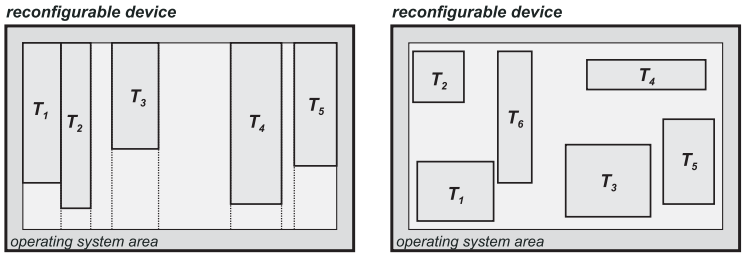
1 Introduction

Todays reconfigurable devices provide millions of gates capacity and partial reconfiguration. This allows for true multitasking, i.e., configuring and executing tasks without affecting other, currently running tasks. Multitasking of dynamic task sets can lead to complex allocation situations which clearly asks for well-defined system services that help to efficiently operate the system. Such a set of system services forms a *reconfigurable operating system* [1] [2]. This paper deals with one aspect of such an operating system (OS), the problem of online scheduling hard real-time tasks.

Formally, a task T_i is modeled as rectangular area of reconfigurable logic blocks given by its width and height, $w_i \times h_i$. Tasks arrive at arbitrary times a_i, require execution times e_i, and carry deadlines d_i, $d_i \geq a_i + e_i$. The reconfigurable device is also modeled as rectangular area $W \times H$ of logic blocks. We focus on non-preemptive systems – once a task is loaded onto the device it runs to completion.

The complexity for mapping tasks to such devices depends heavily on the used *area model*. We use two different area models, a 1D and a 2D model as shown in Figure 1. In the simpler 1D area model, tasks can be allocated along the horizontal device dimension; the vertical dimension is fixed. The 1D area model suffers badly from two types of fragmentation. The first type is the area wasted when a task does not utilize the full device height. The second type occurs when the remaining free area is split into several unconnected vertical

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



(a) 1D area model (b) 2D area model

Fig. 1. Reconfigurable resource models

stripes. Fragmentation can prevent the placement of a further task although a sufficient amount of free area exists. The more complex 2D area model allows to allocate tasks anywhere on the 2D reconfigurable surface and suffers less from fragmentation.

The main contribution of this paper is the development of online hard real-time scheduling heuristics that work for both the 1D and the 2D area model. The limitations of the models and related work are discussed in Section 2. Section 3 states the online scheduling problem and presents the two heuristics. An experimental evaluation is done in Section 4. Section 5 shows our work toward a prototype implementation and, finally, Section 6 summarizes the paper.

2 Limitations and Related Work

Our task and device models are consistent with related work in the field [1] [3] [4] [5] [6]. 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 surface uniformity which is in contrast with modern FPGAs that contain special resources such as block memories and embedded multipliers. However, a reconfigurable 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][2]. Further, the algorithms in this paper can easily be extended to handle additional placement constraints for tasks, e.g., to relocate tasks at different levels of granularity or even to place tasks at fixed positions. The basic problems and approaches will not change, but the resulting performance.

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] [2] overcomes this problem by using a

slightly different area model that partitions the reconfigurable surface into fixed-size blocks. These OSs provide predefined communication interfaces to tasks and asynchronous intertask communication. The same technique can be applied to our 1D area model. For the 2D model, communication is an unresolved issue. Related work mostly assumes that sufficient resources for communication are available [4].

The partial reconfiguration capabilities of the Xilinx Virtex family, which reconfigures a device in vertical chip-spanning columns, fits perfectly 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 is realistic whereas the 2D model faces unresolved 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. 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. As we will show in this paper, the 2D model has great advantages over the 1D 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 Scheduling Real-time Tasks

3.1 The Online Scheduling Problem

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 left-most task cell, x_i, with $x_i + w_i \leq W$. The *starting time* for T_i is denoted by s_i. The main characteristics of scheduling to dynamically reconfigurable devices is that a scheduled task has to satisfy intertwined time and placement constraints:

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

i) $[(x_i + w_i) \leq x_j] \vee [(s_i + e_i) \leq s_j] \vee [x_i \geq (x_j + w_j)] \vee [s_i \geq (s_j + e_j)]$
$\forall T_j \in \mathcal{T}, T_j \neq T_i$ *(scheduled tasks must not overlap in space and time,* $\mathcal{T}$ *denotes the set of scheduled tasks)*

ii) $s_i + e_i \leq d_i$ *(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 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 [8] 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.

Our scheduling problem shares some similarity with orthogonal placement problems, so-called strip packing 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. Translated to our scheduling problem, the width of the strip corresponds to the device width and the vertical dimension corresponds to time. The offline strip packing problem is NP-hard [9] and many approximation algorithms have been developed for it. There are also some online algorithms with known competitive ratios. However, to the best of our knowledge, there is no published online algorithm with a proven competitive ratio for the problem described in this paper which differs in following characteristics: First, our optimization goal is to minimize the number of rejected tasks, based on an acceptance test that is run at task arrival. Second, time proceeds as tasks are arriving. We cannot schedule tasks beyond the current timeline, i.e., into the past. Finally, tasks must not be rotated or otherwise modified.

The simplest online method is to check whether a newly arrived task finds an immediate placement. If there is none, the task is rejected. This crude technique needs to know only about the current allocation situation but will show a low performance. We include this method as a reference point in our experimentation. Sophisticated online 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. In the following sections we discuss two online methods, the *horizon* technique and *stuffing* technique. These planning methods are runtime efficient as they do not reschedule previously guaranteed tasks.

3.2 The Horizon Technique

The horizon technique implements scheduling and placement by maintaining two lists, the *scheduling horizon* and the *reservation list*. The scheduling horizon is a set of intervals that fully partition the spatial resource dimension. Each horizon interval is written as $[x_1, x_2]@t_r$, where $[x_1, x_2]$ denotes the interval in x-dimension and t_r gives the last release time for the corresponding reconfigurable resources. The set of intervals is sorted according to increasing release times.

Figure 2 shows an example for a device of width $W = 10$. At time $t = 2$, two tasks (T_1, T_2) are running on the device and further four tasks (T_3, T_4, T_5, T_6) are scheduled. The resulting scheduling horizon is given by the four intervals shown in Figure 3a) and indicated as dotted lines in Figure 2.

When a new task arrives, the scheduler walks through the list of intervals and checks whether the task can be appended to the horizon. When a horizon interval $[x_1, x_2]$ @ t_r is hit that is large enough to accommodate the task, the task is scheduled to placement x_1 and starting time t_r and the planning process stops. Otherwise, the scheduler tries to merge the interval with already released and adjacent horizon intervals to form a larger interval. If this larger interval

does not fit either, the next interval in the horizon is considered. Should several intervals fit at some point, the scheduler applies the best-fit rule to select the interval with the smallest width.

In the example of Figure 2, a new task T_7 arrives with $(a_7, w_7, e_7, d_7) = (2, 3, 2, 20)$ which gives a planning period of $[a_7, (d_7 - e_7)] = [2, 18]$. The first horizon interval to be checked is $[7, 7]$ @ 3, which is too small to fit T_7. The scheduler proceeds to $[6, 6]$@8, which is too small again. Then, these two intervals are temporarily merged to $[6, 7]$ @ 8 which is still insufficient. The next interval is $[1, 5]$ @ 18 which allows to schedule T_7 to $(x_7, s_7) = (1, 18)$.

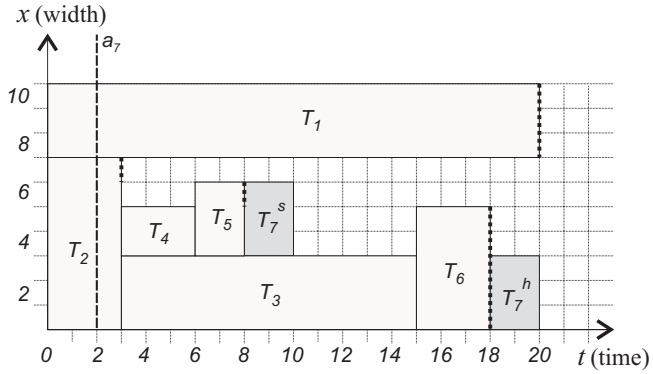
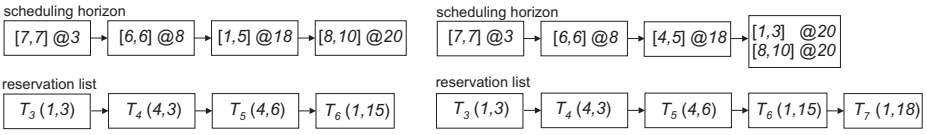


Fig. 2. 1D allocation with scheduling horizon (*dotted lines*) before accepting T_7 and placements for T_7 in the horizon (T_7^h) and stuffing methods (T_7^s)

The *reservation list* stores all scheduled but not yet executing tasks. The list entries are denoted as $T_i(x_i, s_i)$ and hold the placement and starting time. The reservation list is sorted in order of increasing starting times. 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 scheduler is activated whenever a new task arrives, a running task terminates, or a scheduled task is to be started. On each event, the horizon is updated. For the example of Figure 2, the reservation list at time $t = 2$ is displayed in Figure 3a). Figure 3b) shows the updated horizon and reservation lists after scheduling and accepting T_7 at time $t = 2$.

The central point of the horizon scheduler is that tasks can only be *appended* to the horizon. Particularly, it is not possible to schedule tasks before the horizon, as the procedure maintains no knowledge about the time-varying allocation between the current time and the horizon. The advantage of this technique is that maintaining the horizon is simple compared to maintaining the complete future.



(a) scheduling horizon and reservation list at t=2, before scheduling T_7

(b) scheduling horizon and reservation list at t=2, after scheduling T_7

Fig. 3. Horizon method: scheduling horizon and reservation list

3.3 The Stuffing Technique

The stuffing technique schedules tasks into arbitrary free rectangles that will exist in the future. The implementation uses again two lists, the *free space list* and the reservation list. The free space list is a set of intervals $[x_1, x_2]$ that denote currently unused resource rectangles with height H. The free spaces are ordered according to their x-coordinates.

On arrival of T_i, we assume n tasks are executing on the device and m tasks are waiting in the reservation list. Two types of events occur during the planning period of T_i. First, $n', n' \leq n$ tasks terminate which generates new free areas. Second, $m', m' \leq m$ previously guaranteed tasks are started which reduces the free area. When a new task T_i arrives, the scheduler starts walking through the task's planning period, simulating all future allocations of the device by mimicking task terminations and placements together with the underlying free space management. On arrival of T_i and the termination of the n' placed tasks, the placer is called to find a feasible interval. If one is found, the scheduler accepts and adds a new reservation for T_i, and planning stops. If no sufficient interval is found, the scheduler proceeds until $s_{i-latest}$. During the planning process, the scheduler merges adjacent free spaces. Should several intervals fit, the best-fit rule is applied.

For T_7 in the example of Figure 2 planning proceeds until time $t = 8$, where the free space list contains the interval $[4, 7]$ which fits T_7. The stuffing technique leads to improved performance over the horizon method. The drawback is the increased complexity as we need to simulate future task terminations and planned starts to identify free space. Both schedulers use two lists. They differ, however, in the planning process for an arriving task. While the horizon method updates the horizon list only once per task arrival, the stuffing method updates the free space list for $n' + m'$ times.

3.4 Extension to the 2D Area Model

The concepts and methods discussed so far extend naturally to the 2D area model. A 2D placement is given by the coordinates of the task's bottom-left cell, (x_i, y_i), with $x_i + w_i \leq W$ and $y_i + h_i \leq H$, and the resulting scheduling problem relates to 3D strip packing problems [6]. The main difference compared to the 1D model lies in the placer. Instead of keeping lists of intervals, we need

to manage lists of rectangles. The 2D scheduling horizon is a set of rectangles that fully partition the device rectangle, together with the last release time for each rectangle. The 2D stuffing method maintains the free space as a list of free rectangles. The placers we use to implement the 2D horizon and stuffing methods are based on the approach presented by Bazargan et al. [5] and have been described in [10].

3.5 Runtime Efficiency

The runtime efficiency of the schedulers depends largely on the underlying placer implementation. Both our 1D free space list and the 2D Bazargan placer [5] keep $\mathcal{O}(n)$ free areas for n currently placed tasks. Thus, the asymptotic worst-case runtime complexity is the same for 1D and 2D area models. The reference scheduler that considers only immediate placements has a complexity of $\mathcal{O}(n)$. It can be shown that the horizon scheduler's complexity is given by $\mathcal{O}(n+m)$, where m denotes the number of guaranteed but not yet scheduled tasks. The complexity of the stuffing method amounts to $\mathcal{O}(n^2m)$.

4 Evaluation

4.1 Simulation Setup

To evaluate the online schedulers, we have devised a discrete-time simulation framework. Tasks are randomly generated. We have simulated a wide range of 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; task execution times are uniformly distributed in $[5, 100]$ time units. 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. Runtime measurements have been conducted on a Pentium-III 1000MHz, taking advantage of Visual C++'s profiling facilities. All the simulations presented below use 95 % confidence level with an error range of ± 3 percent.

4.2 Results

Scheduling to the 1D Area Model: Figure 4a) compares the performance of the reference scheduler with the horizon and stuffing techniques. The aspect ratios are distributed such that 50 % of the tasks are taller than wide (standing tasks) and 50% 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 scheduling. For laxity class C, the horizon scheduler outperforms

the reference by 14.46 %; the stuffing scheduler outperforms the reference by 23.56 %.

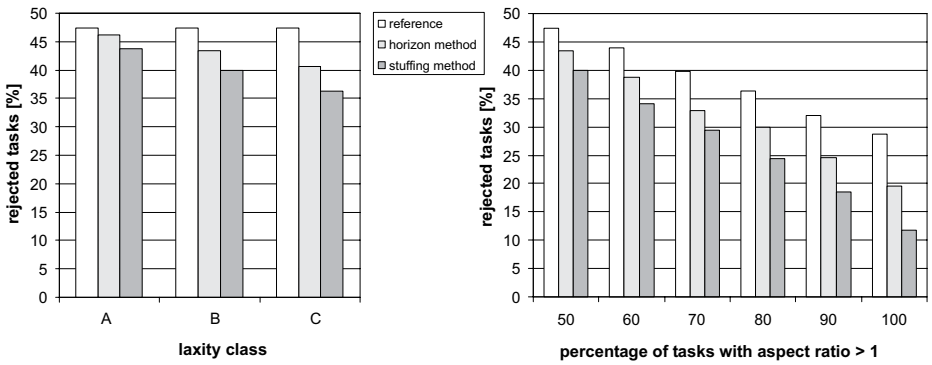


Fig. 4. Performance of the scheduling heuristics for the 1D area model

Figure 4b) shows the number of rejected tasks as function of the aspect ratio, using laxity class B. For the 1D area model standing tasks are clearly preferable. The generation of such tasks can be facilitated by providing placement and routing constraints. In Figure 4b), a percentage of tasks with aspect ratio > 1 of 100 % denotes an all standing task set. The results demonstrate that all schedulers benefit from standing tasks. The differences again grow with the aspect ratio. For 100 % standing tasks, the horizon method results in a performance improvement over the reference of 32%, the stuffing method even in 58.84 %.

Comparison of 1D and 2D Area Models: Figure 5a) compares the performance between the 1D and 2D area models for the stuffing technique. The aspect ratios are distributed such that 50 % 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 75.57 %; for laxity class C it is 98.35 %. An interesting result (not shown in the figures) is that the 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 well.

Runtime Efficiency: Figure 5b) presents the average runtime required to schedule one task for the 2D stuffing method, the most complex of all implemented techniques. The runtime increases with the length of the planning period. However, with 1.8 ms at most the absolute values are small. Assuming a time unit to be 10 ms, which gives us tasks running from 50 ms to 1 s, the runtime overheads for the online scheduler and for reconfiguration (in the order of a few ms) are negligible. This justifies our simulation setup which neglects these overheads.

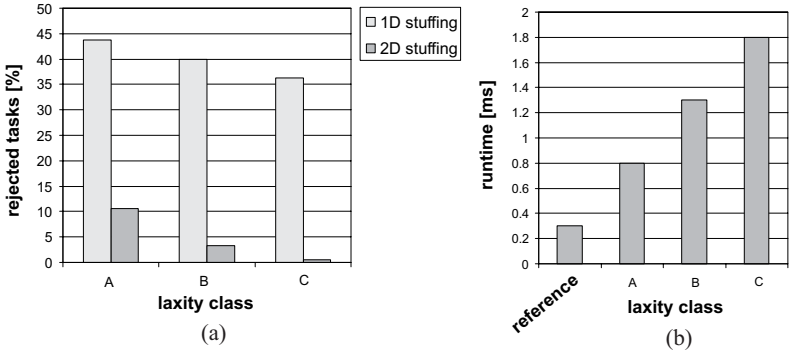
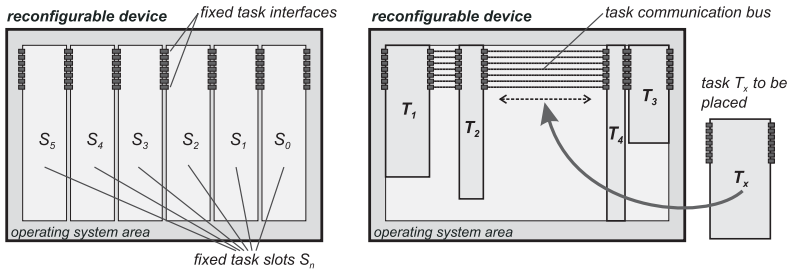


Fig. 5. (a) performance of the 1D and 2D stuffing methods; (b) runtimes for scheduling one task for the reference scheduler and the 2D stuffing method

5 Toward a Reconfigurable OS Prototype

Figure 6(a) shows the 1D area model we use in our current reconfigurable OS prototype. The reconfigurable surface splits into an OS area and a user area which is partitioned into a number of fixed-size task slots. Tasks connect to predefined interfaces and communicate via FIFO buffers in the OS area. The hardware implementation is done using Xilinx Modular Design [11]. The prototype runs an embedded networking application and has been described elsewhere in more detail [2]. 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 inter-arrival time and the FIFO lengths. Our prototype implements rather small FIFOs and guarantees to handle packets with a minimal inter-arrival time of 20 ms.



(a) 1D fixed-size blocks (b) 1D variable placement

Fig. 6. Prototype OS structure

Our prototype proves the feasibility of multitasking on partially reconfigurable devices and its applicability to real-time embedded systems. However,

the 1D block-partitioned area model differs from the model used in this paper which requires task placements to arbitrary horizontal positions as shown in Figure 6(b). While task relocation is solved, prototyping a communication infrastructure for variably-positioned tasks, as proposed by [3], remains to be done.

6 Conclusion and Further Work

We discussed the problem of online scheduling hard real-time tasks to partially reconfigurable devices and developed two online scheduling heuristics for 1D and 2D area models. Simulations show that the heuristics are effective in reducing the number of rejected tasks. While the 1D schedulers depend on the tasks' aspect ratios, the 2D schedulers do not. In all cases, the 2D 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.

References

1. G. Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Int'l Workshop on Field-Programmable Logic and Applications (FPL)*, pages 327–336, 1996.
2. H. Walder and M. Platzner. Reconfigurable Hardware Operating Systems: From Concepts to Realizations. In *Int'l Conf. on Engineering of Reconfigurable Systems and Architectures (ERSA)*, 2003.
3. G. Brebner and O. Diessel. Chip-Based Reconfigurable Task Management. In *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, pages 182–191, 2001.
4. O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. 147(3):181–188, 2000.
5. K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. 17(1):68–83, 2000.
6. S. Fekete, E. Köhler, and J. Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Design Automation and Test in Europe (DATE)*, pages 658–665, 2001.
7. T. Marescaux, A. Bartic, Verkest D., S. Vernalde, and R. Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs. In *Int'l Conf. on Field-Programmable Logic and Applications (FPL)*, pages 795–805, 2002.
8. G.C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 2000.
9. B.S. Baker, E.G. Coffman, and R.L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, (9):846–855, 1980.
10. H. Walder, C. Steiger, and M. Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In *Reconfigurable Architectures Workshop (RAW)*, 2003.
11. D. Lim and M. Peattie. Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations. XAPP 290, Xilinx, 2002.