Hard Real-Time Reconfiguration Port Scheduling

Florian Dittmann and Stefan Frank
Heinz Nixdorf Institute, University of Paderborn
Fuerstenallee 11, 33102 Paderborn, Germany
{roichen, sfrank}@upb.de

Abstract

When modern partially and dynamically reconfigurable FPGAs are to be used as resources in hard real-time systems, the two dimensions area and time have to be considered in the focus of availability and deadlines. In particular, area requirements must be guaranteed for the tasks' duration. While execution environments that abstract the space demand of tasks exist and methods for occupancy of resources over time are discussed in the literature, few works focus on another fundamental bottleneck, the reconfiguration port*. As all resource requests are served by this mutually exclusive device, profound concepts for scheduling the port access are vital requirements for FPGA real-time scheduling. Nevertheless, as the port must be accessed sequentially, we can inherit and apply monoprocessor scheduling concepts that are well researched. In this paper, we introduce monoprocessor scheduling algorithms for the reconfiguration port of FPGAs.*

1 Introduction

Requirements of embedded systems are manifold. Among them we find dependability, efficiency, flexibility, and real-time constraints. They are reactive systems and thus continuously interact with the environment and execute at a pace determined by that environment. In order to guarantee real-time constraints thereby, system response has to be explained without statistical arguments.

Nowadays, embedded systems comprise increasingly programmable logic, especially FPGAs. FPGAs combine performance and flexibility and offer the freedom to compute in space and time. Despite resource re-use, or in field upgrades that come with reconfiguration, area and time both together pose additional problems such as fragmentation or communication requirements. Furthermore, the reconfiguration process itself demands resources and consumes time. In order to use partially reconfigurable FPGAs within real-time systems, all these aspects must be considered.

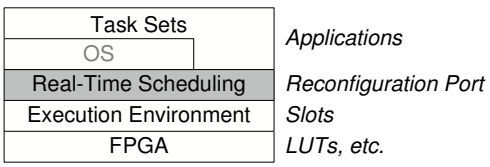

Figure 1. Layers for FPGA real-time systems.

We base our approach on execution environments that abstract from resource requests, handle fragmentation, and thus enable reconfiguration managers or operating systems to efficiently use FPGA resources. Within such execution environments, tasks run in parallel and most often occupy fixed sized slots, which can be reconfigured. The results of [5]—unrealizable schedules if linear task placement is ignored—supports the usage of such execution environments. As displayed in Figure 1, the real-time scheduling proposed in this work is built upon an abstracting execution environment. We process task sets (optionally dispatched by an operating system) and schedule them according to our scheduling algorithm. Based on the workload, task sets are accepted or rejected.

In contrast to others, we propose to schedule the reconfiguration port access. The motivation for this unusual approach is based on several aspects. First, the reconfiguration of even partial areas consumes a significant amount of time that must be respected. Being in the range of milliseconds, the reconfiguration can be even longer than the computation time. Furthermore, the single reconfiguration port of FPGAs demands exclusive sequential activation. Both, the reasonable time demand and the sequentiality motivated us not only to see the reconfiguration as integral part, but also to investigate the reconfiguration in the focus of mono*device* scheduling. Thus, in this work we discuss the assets and drawbacks of applying scheduling algorithms of the monoprocessor domain to the reconfiguration port access, including concepts for real-time guarantee.

The rest of the paper is organized as follows. First, we summarize related work, before we discuss fundamental considerations. Then, we explain the reconfiguration port

978-3-9810801-2-4/DATE07 © 2007 EDAA

scheduling in more detail relying on aperiodic task sets. In the main Section 5, we discuss periodic task scheduling and how to guarantee feasibility for this most common scenario of real-time scheduling. Before a final conclusion, we report on experiments conducted in the context of this work.

2 Related Work

Scheduling tasks on partially reconfigurable FPGAs is targeted differently in the literature. Some works schedule area and time together [12], while others focus on the area only [21], or schedule tasks by a specific manager [15].

An approach in the realm of our work [2] optimizes the area occupied, respecting the task time constraints. Tasks are not allowed to be preempted. Similarly, the authors of [19] and [9] analyze the effect of overall response time and guarantee-base scheduling when tasks comprise different shapes. When task preemption is allowed [1, 20], the task acceptance rate is improved. However, hardware task preemption results in additional costs due to still non-efficient techniques and methods available. All concepts are based on partially reconfigurable devices such as Xilinx FPGAs. However, we seldom find concepts that respect the reconfiguration time or even the sequential reconfiguration. Usually, both are neglected due to the assumption that the execution time is much higher than the reconfiguration time [20].

Real-time scheduling on partially reconfigurable FPGAs is considered in [10]. Different to our approach the scheduling is solved by referring to algorithms of the parallel scheduling domain. Furthermore, we focus on the reconfiguration phase as main scheduling problem. First fundamental consideration have been published by us in [11].

Finally, the theory behind our approach can be compared to the *parallel machine problems with a single server* [7], having the slots as parallel machines and the reconfiguration port as server. However, we use preemption, which is not allowed in these works.

3 Fundamental Considerations

Our real-time scheduling layer accepts task sets and dispatches them on an execution environment. Such an execution environment must comprise a number (m) of equally-sized slots, and accept tasks that fit into one of these (homogeneous) slots. Thus, we avoid external fragmentation by accepting internal fragmentation. Furthermore, the environments should abstract communication by offering a suitable communication structure, e. g. such as in the Erlanger Slot Machine approach [6]. In general, the communication with peripherals is either realized via a shared bus large enough to not impose the bottleneck of the system, or distinct communication channels exist. In addition, there is

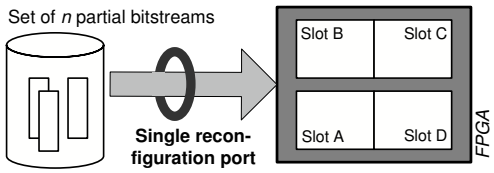

Figure 2. Execution Environment.

a reconfiguration controller for the dispatching. This controller, which can also be located externally, provides the access of our real-time scheduling requests to the execution slots. To summarize, the environment allows us the efficient use of reconfigurable area (ref. to [18, 21]).

We are currently developing a prototyping environment on a Xilinx Virtex 4 FPGA as displayed in Figure 2. We apply the Xilinx EarlyAccess design flow [14]. Concerning the communication, we implement one bus that runs on the highest possible frequency and allows each slot to demand the bus only a fraction of the highest speed (time sharing). As tasks on FPGAs are generally clocked relatively low, this bus is meant to not represent the bottleneck of the system. Further details are out of the scope of this work.

The task sets accepted by our real-time scheduling layer must follow standard requirements of real-time scheduling theory [8]. We schedule a set Γ of n independent tasks τ_i on m slots, with $m < n$. The tasks τ_i have different execution times $t_{EX,i}$ and can have periods T_i. All tasks have relative deadlines D_i. Furthermore, every task has a reconfiguration time $t_{RT,i}$, which is in our case constant for all tasks due to the execution environment that harmonizes area requirements. The run-time reconfiguration results in execution (*EX*) and reconfiguration (*RT*) phases, with an *RT* before each *EX* phase.

As we schedule the reconfiguration port and therefore are interested in when the *RT* phase must have finished, we introduce an additional deadline D^*. This deadline is computed by $D^* = D_i - t_{EX,i}$. In order to schedule the reconfiguration phases, we rely on this deadline D_i^* (relative), or d_i^* (absolute) resp. If we can guarantee the finishing of RT_i before its absolute deadline d_i^*, we also guarantee the completion of EX_i before d_i.

Furthermore, we do not allow the preemption of *EX* phases. Depending on the implementation, a preemption of *EX* results in the need to reconfigure the whole area twice. Even worse, if we have to save states, the preemption of *EX* could result in a need to perform a readback of the slot, thus doubling the additional time in the worst case. However, we allow a preemption of the *RT* phase. Also technically complex to realize, the preemption of *RT* does not substantially increase the whole reconfiguration time. In Figure 3, we display an example where such a preemption is of benefit.

Furthermore, as the reconfiguration of current FPGAs does not offer to interrupt the reconfiguration phase at any

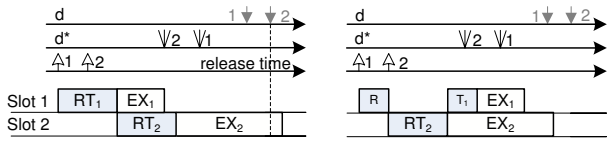

Figure 3. Scheduling according to d (a) and d^* (b) using preemption.

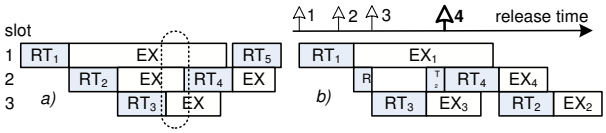

Figure 4. a) Delay due to *full load of slots*, and b) killing due to *full reconfiguration capacity*.

instance of time, but only after frames, we respect this minimum reconfiguration unit $RT_{min} = \Delta$ within our calculations below. Δ also is the smallest time-step for arriving tasks, etc., of our system.

By applying d^* on a set of aperiodic tasks in the next section, we detail the reconfiguration port scheduling.

4 Aperiodic Task Scheduling

In the case of aperiodic tasks that have no dependencies, we distinguish the two cases of synchronous and asynchronous task arrival. The former allows us to perform a schedulability analysis before executing the tasks, while the latter requires dynamic adaptation of the schedule.

4.1 Synchronous Arrival

Considering synchronous arrival, we schedule the tasks according to JACKSON's rule (earliest due date EDD) referring to d_i^* as deadlines. The algorithm executes the tasks in order of non-decreasing deadlines and is optimal w. r. t. minimizing the max. lateness. However, we suffer an abnormality. Due to the avoidance of *EX* phase preemption, all slots can be occupied (from here on called *full load of slots: fls*). Figure 4 a depicts an example, where the start or RT_4 must be delayed.

This problem can be improved by noting that the potential *fls* can be reduced in one single case. If for two subsequent tasks τ_i and τ_{i+1}: $t_{EX,i} > t_{EX,i+1}$, $t_{EX,i} \geq t_{RT}$, and $t_{EX,i} < t_{RT} + t_{EX,i+1}$, we can swap τ_i and τ_{i+1}. Thus, the starting times of the next two *RT* phases will be improved. This holds for $m \geq 2$.

For the schedulability analysis already presented in [11], we construct the schedule referring to a vector that displays the current slot occupancy. Despite the a priori knowledge of the tasks, we have to dynamically react on the *fls* phases. These phases can be compared to dynamic arriving jobs, which however are ordered according to JACKSON's rule and do not cause preemption, only additional *delay*.

4.2 Asynchronous Arrival

In order to schedule asynchronously arriving jobs, preemption is required if NP-completness shall be avoided. The monoprocessor domain proposes to use EDF (earliest deadline first), which dispatches at any instance the task with the earliest absolute deadline.

When we apply EDF to our scenario relying on d^*, in addition to the *fls* abnormality, which occurs here as well, we experience the *full reconfiguration capacity* (*frc*) abnormality (ref. to Figure 4 b). Here, at least one slot is in *RT* phase, while all slots are occupied (by either *RT* or *EX*). We have two possibilities when a new high priority job arrives. We either can delay the job similar to the *fls* abnormality, or we can force a preemption of one of the slots currently in *RT* mode. Such a preemption will result in the killing of the preempted job, as all already reconfigured parts of this task are abandoned in favor of the high priority task. The monoprocessor EDF does not include such an abnormality.

4.3 Experimental Results

We have conducted several test sets in order to rate the performance of EDD and EDF in our scenario. We have set up a reconfiguration port simulator that allows us to randomly generate task sets, schedule them according to various algorithms, respecting different priorities, and displays the schedules for visual control. Our test rows have been on each 1000 randomly generated task sets, with $\frac{t_{EX}}{t_{RT}} \approx l$, and $l = .25, .5, 1, 2, 3, 4$. The number of slots has been in the range of 3 to 5, while the number of tasks is significantly higher, so that tasks have to share slots. We have scheduled both EDD and EDF with d^* and d as deadlines.

Concerning the schedulability of EDD, we find a schedulability of approx. 90 % for d^* and 70 % for d among the schedulable task sets. However, the schedulability concerning d^* significantly increases with $l \ll 1$, and decreases with $l \gg 1$, while d behaves oppositional. Obviously, this is due to a dominance of the *EX* phase. The schedulable task sets not found are due to the *fls* abnormality. When EDF is applied, both abnormalities can occur. Thus, the performance is slightly worse, again with d^* outperforming d. Finally, for both EDD and EDF the approach performs best if $t_{EX} \leq t_{RT} \cdot (m-1)$.

5 Fixed Priority Periodic Task Scheduling

Periodic activities (sensory data acquisition, control loops, etc.) often represent the major computational de-

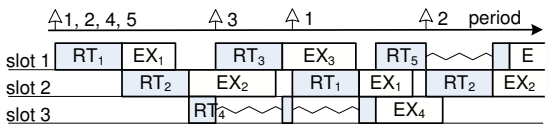

Figure 5. Fixed priority example.

mand of embedded systems. In real-time scheduling theory priorities are principally applied to such jobs. Contention for resources is resolved in favor of the job with the higher priority that is ready to run.

Our tasks have the relative deadlines D_i equal to their periods T_i. However, as we schedule the *RT* phases, we derive the priorities of our jobs by referring to the relative deadline D_i^*. The task with the shortest D_i^* gets the highest priority. We thus have to schedule a set of periodic tasks with deadlines less than periods. A similar monoprocessor scheduling algorithm is denoted deadline monotonic (DM) [13]. It is an extension of the more common rate monotonic scheduling scheme. According to the DM algorithm, each task is assigned a priority inversely proportional to its relative deadline. Thus, at any instant, the task with the shortest relative deadline is executed. Figure 5 shows an example.

5.1 Schedulability Analysis

The sufficient and necessary schedulability test of a DM algorithm can be done by the response time analysis [3, 4], with the longest response time R_i computed at the critical instance as the sum of its computation time and the interference due to preemption by higher-priority tasks: $R_i = t_{RT,i} + I_i$, where $I_i = \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil t_{RT,j}$. If $R_i < D_i^*$ for all tasks, the set is schedulable. We can derive step-wise solutions for this problem. The critical instance occurs when all tasks are released simultaneously.

When scheduling the reconfiguration port, however, we face several circumstances where the response time analysis would not produce the correct result. Both abnormalities presented above, the *fls* and the *frc* scenarios demand special care, as they both can impose additional delays to the scheduling of the jobs, i. e., the increase of I_i. The abnormalities can hardly be calculated in advance. Thus, we propose alternative strategies to guarantee the schedulability.

One possibility is to schedule the hyper-period of a task set. The hyper-period equals the least common multiplier of all periods. Schedulability of one hyper-period guarantees that subsequent hyper-periods can also be scheduled, i. e., the whole task set is schedulable. Thereby, we can solve the question of schedulability, as the *fls* and *frc* abnormalities will occur similarly in all hyper-periods. However, there might exist cases, were no hyper period exists, or it is too large to be constructed in advance.

As second solution, we handle the abnormalities by

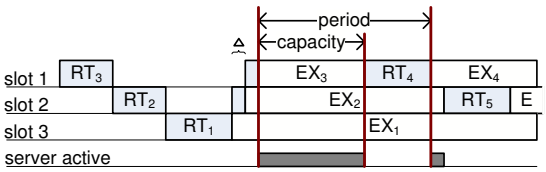

Figure 6. Server for *fls*: Worst Case.

known and appropriate techniques of real-time scheduling that harmonize with the standard DM response time analysis. We derive the two suitable techniques in the following.

5.2 A Server for *Full Load of Slots* Sections

During the *fls* abnormality, all slots are in *EX* phase while a new instance of a task τ_i should be scheduled. τ_i, independent of its priority, demands for the reconfiguration port. Despite the availability of the port, but due to the occupancy of the slots, we cannot schedule this task. It is delayed.

In order to respect such a delay, we rely on the notions of aperiodic job scheduling. We assume an aperiodic job τ_a arriving at the same time as the new instance of job τ_i arrives. τ_a stands for the delay of τ_i and has the corresponding computation demand. For scheduling τ_a, we use a server, which must have both enough capacity to serve τ_a completely and a higher priority than τ_i to be active immediately. Such a server can handle aperiodic tasks with hard deadlines. Here, the aperiodic tasks are executed virtually only.

As the server must always execute in favor of any other job (forced by *fls*), we assign the highest priority to the server, i. e., the server will have the maximum relative deadline $D_{S,max} = \min(D_i^*) - \Delta$. As the server does not have an *EX* phase, its deadline is equal to its period and $D_S^* = D_S$. In order to calculate the exact deadline of the server, we have to know the capacity.

In Fig. 6, we display the worst case *fls* scenario. It occurs when all slots have started with the *EX* phases as close as possible. This nearly simultaneous activation occurs when the *RT* phases of m tasks end nearly synchronously. Then, the server must have enough capacity C_S to schedule an aperiodic job that has the computation requirement of the m-th longest t_{EX}, thus C_S =m-th $\max(t_{EX})$. We can improve the server capacity, if we have a large Δ compared to the *EX* phases. If the largest t_{EX} is smaller than $(m-1)\Delta$, the server capacity becomes 0. This holds for the k-th largest t_{EX} and $t_{EX,k} < (m-k)\Delta$, while $k < m$.

As the capacity must always be available, we require a server that preserves its capacity. Furthermore, as two *fls* abnormalities could occur successively only separated by one *RT* phase, the capacity of the server must always be replenished as soon as possible. The *Sporadic Server* (SS) [17] solves our requirements. The SS algorithm creates a high-priority task for servicing aperiodic requests and pre-

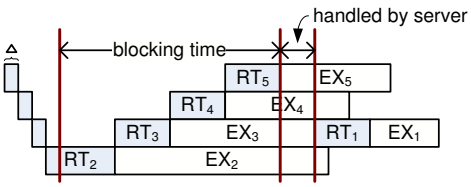

Figure 7. Blocking Time for *frc*: Worst Case.

serves the server capacity at its high-priority level until an aperiodic request occurs. SS replenishes its capacity only after it has been consumed by aperiodic task execution.

Finally, the period (= relative deadline) of the server is the minimum of the above introduced $D_{S,max}$, and the capacity + *RT* phase, i. e., $C_S + t_{RT}$. Thus, the server will have the highest priority among all other tasks. Furthermore, the feature of the SS allows us to have always enough capacity available, as even partially consumed capacity is replenished after the server's period. This is always early enough, as between two consecutive occurrences of *fls* abnormalities always a complete *RT* phase will be scheduled.

5.3 Resource Access Protocol for *Full Reconfiguration Capacity* Sections

When the *frc* abnormality occurs, all slots are occupied and at least one slot is in *RT* phase. If the newly arriving instance of a task τ_i has lower priority than the just reconfiguring task τ_j, nothing will happen and τ_i will be sorted into the list of ready tasks. However, if the priority is higher, τ_i could either be scheduled and τ_j would be killed, or delayed until the reconfiguration port is free again.

For our schedulability analysis, we disallow killing as it harms the assumptions of DM and would complicate the computation of the interference time I_j. Thus, we delay τ_i, i. e., higher priority tasks will suffer a blocking due to lower priority tasks, as the *reconfiguration port* is occupied. This is similar to a critical section of resource sharing. In order to avoid (unbounded) priority inversion, a resource access protocol is necessary. As we only face direct blocking and will not suffer chained blocking or deadlocks [8], we can apply the *Priority Inheritance Protocol* PIP [16]. The protocol modifies the priorities of those tasks that cause blocking. In our case, τ_j would temporarily inherit the priority of τ_i.

The schedulability analysis of the PIP is based on the response time analysis. Therefore, the blocking time B_i is added to the recurrent equation: $R_i = t_{RT,i} + B_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil t_{RT,j}$. Note that this test becomes only sufficient, as tasks could actually never experience blocking. In order to calculate the blocking time B_i, we rely on the worst case blocking time as displayed in Figure 7. The worst case occurs when a high priority task suffers a blocking due to m lower priority tasks in their *RT* phase. Thus, the blocking time will not be longer than $B_{max} = (t_{RT} - \Delta)(m - p)$, while $p = m$ for the lowest priority task and decreases by 1 with every priority increase until $p = 0$.

Algorithm 1 Blocking Time

```
1:  L ⇐ Γ, L̄ ⇐ ∅, p ⇐ m
2:  while L ≠ ∅ do
3:      τ_i ⇐remove_lowest_priority_task(L)
4:      B_i ⇐ (t_RT − Δ)(m − p)
5:      if p > 0 then p ⇐ p − 1
6:      for (k ⇐ 1; k < (m − 1) ∧ k < number_of τ_j in L̄;
        k ⇐ k + 1) do
7:          B_tmp ⇐ k-th longest t_EX,j + k(t_RT − Δ)
8:          if B_tmp < B_i then B_i ⇐ B_tmp
9:      end for
10:     L̄ ⇐ L̄ ∩ τ_i
11: end while
```

We notice that the blocking time will be large for a large m. However, B_i will never be longer than the largest $t_{EX,j} + t_{RT} - \Delta$ among all tasks with lower priority than τ_i. In fact, the k-th longest *EX* phase among the tasks τ_j added to a non-avoidable fraction of $k(t_{RT} - \Delta)$ will denote B_i, if it is smaller than B_{max}. We calculate B_i by Algorithm 1.

5.4 DM + SS + PIP Schedulability Test

For the schedulability test, we thus have to combine the response time analysis for DM with the PIP and the server. From a scheduling point of view, SS can be replaced by a periodic task having the same utilization factor. As our server will never come active during a *frc* abnormality, the server does not have a blocking time ($B_S = 0$). Thus, we have to solve the recurrent equations $R_i = t_{RT} + B_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil t_{RT}$ for the task set $\Gamma' := \Gamma \cap \tau_S$.

6 Experiment

We have randomly generated periodic task sets and tested their schedulability using our simulator. Fig. 8 shows an example. Our approach using PIP access protocol and the SS server on one hand correctly guarantees the schedulability of the task sets. On the other hand, however, the approach is pessimistic, as neither the server capacity, nor the blocking time are consumed completely among the majority of our task sets. Thus, we have also scheduled task sets that are not feasible according to our test, but according to the standard DM response time analysis. Such schedules are often still feasible within the reconfiguration port scenario. The same empirical results form the aperiodic task sets hold: our approach performs best if $t_{EX} \leq t_{RT} \cdot (m - 1)$. Furthermore, there exist few cases, where killing of tasks can increase the performance of the schedule (reducing the max. lateness) or even result in feasibility.

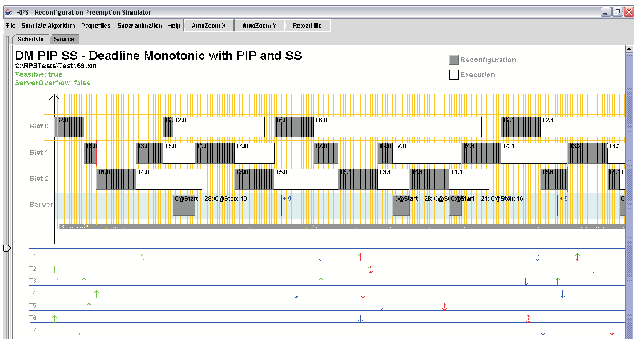

Figure 8. Simulator.

7 Conclusion

In this paper, we introduced a real-time scheduling layer for partially and dynamically reconfigurable FPGAs. The layer accepts task sets and schedules the reconfiguration port of execution environments. Due to the mutual exclusiveness and the sequentiality of the port, we can apply scheduling algorithms from the monoprocessor domain. We discussed assets and drawbacks of this approach on the basis of aperiodic task sets. With this background, we derived a scheduling algorithm for periodic tasks. The algorithm bases on the deadline monotonic scheduling concept and is extended by a server to handle *full load of slots* abnormalities, and a resource access protocol to handle *full reconfiguration capacity* abnormalities. By theses extensions, we are able to perform a schedulability test on the basis of the tasks' parameters without calculating a whole hyper-period. In our experiments, we could find that our schedulability test helps to derive feasible schedules with a reasonable performance. As the feasibility test still is pessimistic, we are currently looking for an improvement of the test. Furthermore, we plan to investigate dynamic priority task sets.

References

[1] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich. Task scheduling for heterogeneous reconfig. computers. In *SBCCI '04*, pp. 22–27, Pernambuco, Brazil, 2004. ACM.

[2] A. Ahmadinia, C. Bobda, and J. Teich. A Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware. In *ARCS*, pp. 125–139, Augsburg, Germany, 2004.

[3] A. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[4] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, 1991.

[5] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration. In Proc. *DAC '05*, pp. 335–340, San Diego, California, USA, 2005. ACM Press.

[6] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, J. Teich, and J. van der Veen. The Erlangen Slot Machine: A Highly Flexible FPGA-Based Reconfigurable Platform. In *Proc. 13th IEEE FCCM*, pp. 319–320, 2005.

[7] P. Brucker, C. Dhaenens-Flipo, S. Knust, S. A. Kravchenko, and F. Werner. Complexity results for parallel machine problems with a single server. *J. of Scheduling*, 5:429–457, 2002.

[8] G. C. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Appl*. Springer, 2004.

[9] H. Walder, C. Steiger and M. Platzner. Operating systems for reconf. embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Comput.*, 53(11):1393–1407, 2004.

[10] K. Danne and M. Platzner. Executing hardware tasks on dynamically reconfigurable devices under real–time conditions. In *Proceedings of the FPL06*, Madrid, Spain, 2006.

[11] F. Dittmann and M. Götz. Applying Single Processor Algorithms to Schedule Tasks on Reconfigurable Devices Respecting Reconfiguration Times. In *13th RAW*, Rhodes Island, Greece, 2006. IEEE.

[12] S. P. Fekete, E. Köhler, and J. Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Proc. DATE 2001, Design, Automation and Test in Europe*, pages 658–665, Munich, Germany, 2001. IEEE.

[13] J. Leung and J. W. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, pp. 237–250, 2(4), 1982.

[14] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgeford. Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration on XILINX FPGAS. In *Proceedings of the FPL 2006*, Madrid, Spain, 2006.

[15] J. Resano, D. Mozos, D. Verkest, and F. Catthoor. A reconfiguration manager for dynamically reconfig. hardware. *IEEE Design and Test of Computers*, 22(5):452–460, 2005.

[16] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[17] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1(1):27-60, June 1989.

[18] M. Ullmann, M. Hübner, B. Grimm, and J. Becker. On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities. In *Proceedings of the FPL*, pp. 454–463, Antwerp, Belgium, 2004. Springer.

[19] H. Walder and M. Platzner. Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 24–30, June 2002.

[20] H. Walder and M. Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, pp. 290–295. IEEE Computer Society, March 2003.

[21] H. Walder and M. Platzner. A Runtime Environment for Reconfigurable Hardware Operating Systems. In *Proceedings of the 14th International Conference on Field Programmable Logic and Application (FPL'04)*, pp. 831–835. Springer, August 2004.