Reconfiguration Time Aware Processing on FPGAs

Florian Dittmann and Marcelo Götz
Heinz Nixdorf Institute, University Paderborn, Germany
{roichen,mgoetz}@upb.de

Considering nowadays FPGAs, the reconfiguration time is a non-negligible element of reconfigurable computation. Moreover, run-time environments that ignore the reconfiguration time can quickly lack applicability. Thus, methods to respect this additional time are required. Looking for suitable analogies in already evaluated fields seems to be reasonable and shall be investigated in this work.

In the single machine environment, several scheduling algorithms exist that allow to quantify schedules with respect to feasibility, optimality, etc. In contrast, reconfigurable devices execute tasks in parallel, which intentionally collides with the single machine principle and seems to require new methods and evaluation strategies for scheduling. However, the reconfiguration phases of adaptable architectures usually take place sequentially. Run-time adaptation is realized using an exclusive port, which again is occupied for some reasonable time during reconfiguration. We have to handle the duration and the sequential exclusiveness of reconfiguration phases. Here, we can find an analogy to the single machine environment, as both scenarios must derive a sequential schedule for an exclusive resource. Thus, we investigate the appliance of single processor scheduling algorithms to task reconfiguration on reconfigurable systems in this paper. We determine necessary adaptations and propose methods to evaluate the scheduling algorithms.

1 Introduction

Recently, several authors have proposed similar architectural concepts for fine-grained run-time reconfigurable systems in the reconfigurable computing field. The architectures usually based on Xilinx Virtex FPGAs comprise a specific number of slots, in which tasks are dynamically allocated and executed. The technological capabilities of the Xilinx FPGAs allow a deterministic (e. g., glitch-free) partial reconfiguration of these slots during run-time, without affecting other regions. In addition, the inherent parallelism

Dagstuhl Seminar Proceedings 06141
Dynamically Reconfigurable Architectures
http://drops.dagstuhl.de/opus/volltexte/2006/735

of fine-grained devices enables the implementation of tasks executing in space, i. e., mostly faster than in software. All together, flexibility and performance are merged in a sophisticated run-time environment. This environment can be implemented as a Reconfigurable System-on-a-Chip (RSoC).

In detail, such systems comprise a bus infrastructure for inter-task and external communication. Often, a CPU is included to the system as hard- or soft-core. The CPU or a run-time reconfiguration manager handles the reconfiguration itself. Tasks are executed in the slots of the reconfigurable fabric. Some architectural concepts allow dynamic width assignments of the tasks, while others hold the size fixed. Due to hardware limitations, the slots always span the whole height of the FPGA so far. Future environments based on e. g. the Xilinx Virtex 4 device will be able to define reconfigurable regions that must not span the whole height of the FPGA. Such environments will ease routing, etc. However, their fundamental concept of having regions for the execution of reconfigurable tasks will most likely stay the same.

Efficient executing of tasks on such devices is proved to be not a trivial problem. Apart from area assignment, de-fragmentation and communication problems, which are extensively studied on the above mentioned platforms, the reconfiguration itself demands further investigation. We itemize the main two problems of reconfiguration in the following:

- The system or at least the reconfigurable hardware part is usually implemented on one single chip. Despite the possibility to execute several tasks on this chip in parallel, the reconfiguration of the slots is sequential. There exists one reconfiguration port only, which is used exclusively for the reconfiguration.
- The reconfiguration time itself cannot be neglected. Fine-grained devices, like FPGAs, are primarily designed for fast (i. e., parallel) execution of algorithms. They are not primarily designed for fast run-time reconfiguration, a fact which will barely change in the near future. Thus, the infrastructure provided for the reconfiguration is limited and can be seen as the bottleneck of fine-grained reconfigurable hardware. Note that there have been several proposals to overcome this limitation [9], however, none of them is available commercially yet. Consequently, the practical usefulness of such fast reconfiguring devices is difficult to prove.

Those two characteristics (exclusiveness and reconfiguration time), although fundamentally being a drawback, enable the appliance of methods of the single processor domain to the reconfigurable run-time environments. Scheduling algorithms of the single machine domain sequentially assign a set of tasks to one processing device. The device is used mutually exclusive. Similar, reconfiguration phases must be assigned sequentially to the exclusive reconfiguration port.

The proposed run-time environments, as well as RSoCs in general are intended to be used in the area of embedded systems. The idea of having many components on the same die is basically motivated by resource efficiency, i. e., limitations in various dimensions (power, memory, size, weight, etc.) are significant. Additionally, most such systems are real-time systems, whose correctness not only depends on the correct result, but also on

the time when the result is made available. Thus, real-time scheduling strategies become a vital fact in these systems.

In this paper, we focus on such a real-time execution of tasks in slots of reconfigurable systems respecting execution and reconfiguration time constraints. We investigate several scheduling strategies known from single processor real-time systems, where tasks can arrive at the same or arbitrary times to the system. We investigate independent task sets and propose a novel approach where a task may be preempted in its reconfiguration phase, in order to achieve a feasible schedule. We base our research on already known schedule algorithms and show how to adapt them for our scenario. Additionally, we explain how guarantee tests can be realized.

The rest of the paper is organized as follows. After summarizing related work, we will abstract the scenario and introduce helpful parameters to solve the problem. In Section 4, we first investigate a set of independent tasks having synchronous arrival time. We show the analogy and limitation to the single environment schedule, which would be used in such a scenario. Then, we enhance the scenario to tasks having arbitrary arrival times. Again, the comparison to the single machine environment is drawn. Here, we propose to introduce preemption to the reconfiguration phases. Finally, we conclude and give an outlook.

2 Related Work

Voluminous amount of work has already been done in online scheduling of real-time tasks on reconfigurable architectures. Most of them divides the problem into two main problems: task scheduling and task placement.

In the work presented in [2], the area occupied is optimized, respecting the task time constraints, where tasks are not allowed to be preempted. In the same scenario, the authors of [12] and [10] analyze the effect of overall response time and guarantee-base scheduling when tasks comprise different shapes. When task preemption is allowed (e. g., [1] and [13]), the task acceptance rate is improved. However, hardware task preemption represents additional costs due to still non-efficient techniques and methods available to do it. All those concepts are based on the assumption that the reconfigurable devices may be partial configured, which is true for some available FPGAs (e. g., Xilinx). 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 (e. g., [13]). Nevertheless, for comparable reconfiguration and execution times, the behavior of the single reconfiguration port may decrease the system performance and needs to be taken into consideration. Therefore, in our paper we propose the inclusion of the reconfiguration phase into the scheduling of real-time tasks on reconfigurable devices. The placement problem is not considered since we assume that every task comprises the same size.

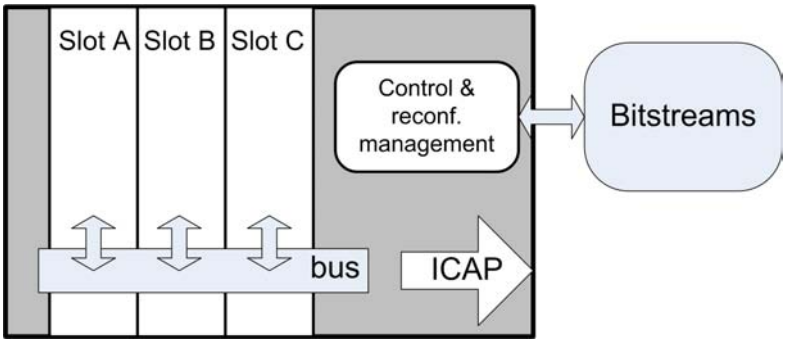

Figure 1: Exemplary architectural concept comprising three slots, a control mechanism, an external storage place for bitstreams and the (ICAP) port for the (internal) reconfiguration.

3 Problem Abstraction

We rely on the above mentioned RSoCs and abstract them first. If we have n tasks to be executed, each in one of the m slots, and $m < n$, i. e., the number of slots is smaller than the number of tasks to be executed, we have to reuse the same slot for multiple tasks. Moreover, all tasks are loaded (by means of slot reconfiguration) through one single port. In order to handle this limitation of resources, we need a suitable mechanism to schedule the tasks. Such a schedule may satisfy an optimization criteria like the minimization of the overall response time or the maximum lateness, etc., under the limited number of resources and the mutual exclusiveness of the configuration port.

Considering the set of tasks in more detail, we deal with tasks arriving at the same or arbitrary time. In the scope of this paper, all tasks are aperiodic and have no precedence constraints. Additionally, the tasks are not preemptive in their execution phase. Concerning the geometrical properties of the tasks, they all occupy a whole slot and comprise the same size. There may be internal fragmentation, which is out of scope of this paper.

An abstract view of the execution platform mentioned in the introduction can be found in Figure 1. Partial reconfiguration capabilities enable a single slot to be reconfigured keeping remaining ones in execution. The concept is similar to several different proposals that we can find in the literature (e. g., [14, 11]). The systems' most important fact in the context of this work is the single reconfiguration port, which enforces us to reconfigure the tasks sequentially.

We model every task of our system with two different phases. The reconfiguration phase (RT) represents the configuration of the hardware itself. The RT phase needs to occur before the second phase, which is the execution phase (EX). Figure 2 shows these two phases. Horizontally, we display the available slots and their occupation over time. RT means that this slot is in reconfiguration, while EX denotes the execution of the task. As all tasks have the same size, the RT phases are of the same duration. Technically speaking, the bitstreams comprise the same number of bits.

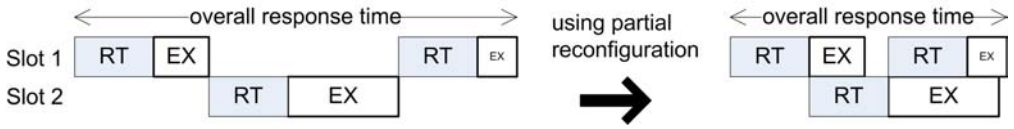

Figure 2: Reconfiguration (RT) and execution phase (EX).

Table 1: Definitions

d_i	execution deadline
$t_{EX,i}$	computation or execution time
t_{RT}	reconfiguration time
L_{max}	maximum lateness
d_i^*	reconfiguration deadline

In Figure 2, we can also find the motivation for partial reconfiguration capabilities of such a system. Partial reconfiguration results in an improved overall response time of the task set, as reconfiguration of new tasks can take place during the execution of current tasks. Thus, we pipeline EX and RT phases of different tasks, hiding the reconfiguration time.

Due to the exclusive usage of the reconfiguration port, no two reconfiguration phases can be scheduled at the same time. However, multiple tasks can execute at the same time. Resource conflicts during task execution (e. g., sharing of the same bus) are out of the scope of this paper.

In order to derive a schedule for task reconfiguration, we specify the parameters of the task t_i in Table 1. The definitions are close to the ones in [3]. We want to emphasize the maximum lateness, which is a known metric for performance evaluation $L_{max} = \max_i(f_i - d_i)$. Further, for our special case, we define a deadline d_i^* that is the deadline for the reconfiguration phases. It is calculated using $d_i^* = d_i - t_{EX,i}$.

4 Scheduling

We investigate the two known aperiodic task scheduling strategies from single processor design: EDD and EDF. Motivated by the similarity of the single processor scheduling and the behavior of the reconfiguration port of the introduced execution platforms, we show in what way we can use these algorithms from the single machine environment in the domain of reconfigurable task scheduling. We do not address the problem of multi processor scheduling, but propose methods for the optimization of the reconfiguration behavior of such systems by focusing on the sequential processing reconfiguration port.

Further, we structure our procedure similar to [3], which is a comprehensive reference for scheduling in the real-time domain. Similar, we make several assumptions on the task set to be able to categorize the problems.

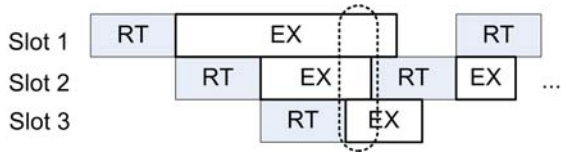

Figure 3: Three slots and n tasks having different execution times.

Algorithm 1 Earliest Due Date for Reconfigurable Slot Architectures

1: **if** reconfiguration port is inactive (i. e., no RT phase is active) **then**
2: Find slot where no EX is active
3: **if** all slots are in EX phase **then**
4: Wait until at least one slot is available
5: **end if**
6: Reconfigure slot r, $(r = \text{ind}\,(\min\{z_1, z_2, \ldots, z_m\}))$
7: **end if**

4.1 Synchronous arrival of independent tasks

A set of n aperiodic tasks has to be loaded into m slots ($m < n$), using the mutual exclusive reconfiguration port. The tasks have synchronous arrival time, but can have different execution time $t_{EX,i}$ and deadline d_i. Note that schedules for this scenario do not need preemption as no new tasks will enter the system at run-time (synchronous arrival time).

The sequential scheduling problem of the RT phases is solved in the single processor environment with respect to minimizing the maximum lateness using JACKSON's algorithm, also called *earliest due date (EDD)*. The algorithm executes the tasks in order of non-decreasing deadlines. We apply EDD to our scenario, using d_i^* as deadlines.

We have to extend EDD due to the fact that the seamless scheduling of RT phases can be blocked when all slots are in EX phases, as displayed in Figure 3. We may be forced to wait to start the next reconfiguration due to full slot occupancy. A waiting period may be enforced, which we denote as δ_i. In this case, the optimality of EDD cannot be guaranteed. However, every slot is executing, i. e., the FPGA is fully utilized and does not waste free space. The algorithm respecting such waiting phases looks as displayed in Algorithm 1.

If we can guarantee at least one free slot at the beginning of each RT phase, all results of EDD of the single machine environment hold and EDD is optimal in our scenario with respect to minimizing the maximum lateness. A sufficient but not necessary condition to guarantee a free slot is $\forall i : t_{EX,i} < t_{RT} \cdot (m-1)$. Moreover, using this formula, we can estimate the number of slots needed.

4.1.1 Guarantee

When we want to make a guarantee test, i. e., to guarantee that a set of tasks can be feasibly scheduled, we need to show that, in the worst case, all tasks can complete before

their deadlines. The guarantee test for EDD in the single processor case is $\forall i = 1, \ldots, n : \sum_{k=1}^{i} t_{EX,k} \leq d_i$. In our scenario, due to the possible delays when all slots are occupied and the next RT phase is postponed (see Figure 3), we must extend every scheduled task by a possible additional δ_i. Thus, it must hold

$$\forall i = 1, \ldots, n : \quad \sum_{k=1}^{i} (t_{RT,k} + \delta_k) + t_{EX,i} \leq d_i.$$

The δ_k depend on the current occupation of all slots of the system and are difficult to compute. Therefore, our guarantee test avoids the explicit calculation of the δ_k by computing the slot occupancies iteratively. The idea is to use a vector $\mathbf{z}$ that holds the current status of each slot. In the vector fields, we sum up the individual occupancy with respect to the global availability of the reconfiguration port. Therefore, we sequentially run through the schedule produced by EDD of Algorithm 1 filling this vector $\mathbf{z}$, whose entries z_l represent the slots of the reconfigurable fabric. The vector is updated each time a new RT phase starts. After the update, the vector's entries display when (time) their corresponding slots can be reconfigured next, also concerning the global condition, i. e., the occupancy of the reconfiguration port. Thus, by extracting the field with the smallest value, we can determine the next slot r for reconfiguration of the next task t_j. We apply

$$r = \text{ind}\left(\min\left\{z_1, z_2, \ldots, z_m\right\}\right),$$

while *ind* is the index function (see also Line 6 of Algorithm 1). If multiple z_l are minimal, the selection is arbitrarily.

In detail, the entries of the vector are updated ($z_{r,\text{old}} \Rightarrow z_{r,\text{new}}$) as follows: We add t_{RT} and $t_{EX,j}$ to the field of the selected slot (z_r): $z_{r,\text{new}} = z_{r,\text{old}} + t_{RT} + t_{EX,j}$. In order to update all other fields $z_l, l \neq r$, the following equation holds:

$$z_{l,\text{new}} = \max\left\{z_{l,\text{old}}, (z_{r,\text{old}} + t_{RT})\right\}. \tag{1}$$

Thus, if the finishing time of the RT phase of slot r is larger than $z_{l,\text{old}}$, slot l may be reconfigured, when the currently started reconfiguration has finished ($z_{l,\text{new}} = z_{r,\text{old}} + t_{RT}$). Otherwise, if slot l will still be in EX phase when slot r has finished reconfiguration, we must not select slot l for reconfiguration. Therefore, z_l keeps its value ($z_{l,\text{new}} = z_{l,\text{old}}$), which is larger than $z_{r,\text{new}}$ indicating its next availability for reconfiguration.

Now, we can answer the question of feasibility of a task t_j, i. e., whether the deadline d_j of task t_j can be met. After each update of the vector due to the dispatching of a task t_j, it must hold $z_{r,\text{new}} \leq d_j$. After scheduling all tasks, we can calculate the overall finishing time as the $\max\left\{z_1, z_2, \ldots, z_m\right\}$.

Example

An exemplary sequence of the vector during a guarantee test for the scenario of Figure 3 will look like the following. We start with an empty vector and schedule the first two tasks:

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} : \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} t_{RT,1} + t_{EX,1} \\ t_{RT,1} \end{pmatrix} \rightarrow \begin{pmatrix} t_{RT,1} + t_{EX,1} \\ t_{RT,1} + t_{RT,2} + t_{EX,2} \end{pmatrix} \rightarrow \begin{pmatrix} t_{RT,1} + t_{EX,1} \\ t_{RT,1} + t_{RT,2} \end{pmatrix} \rightarrow$$

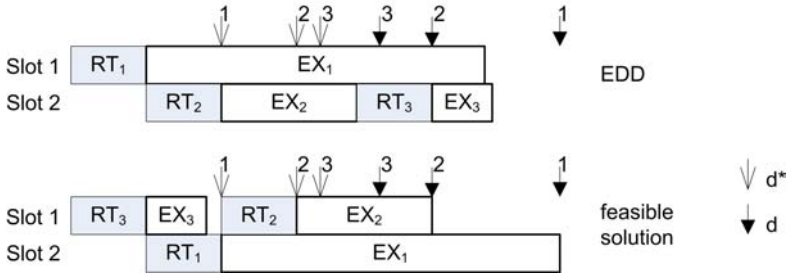

Figure 4: EDD can fail to produce a feasible schedule.

The update of the vector after dispatching the third task will result in keeping of the value of z_2, according to Equation 1.

$$\begin{pmatrix} t_{RT,1} + t_{EX,1} \\ t_{RT,1} + t_{RT,2} + t_{EX,2} \\ t_{RT,1} + t_{RT,2} + t_{RT,3} + t_{EX,3} \end{pmatrix} \rightarrow \begin{pmatrix} t_{RT,1} + t_{RT,2} + t_{EX,2} + t_{RT,4} \\ t_{RT,1} + t_{RT,2} + t_{EX,2} + t_{RT,4} + t_{EX,4} \\ t_{RT,1} + t_{RT,2} + t_{EX,2} + t_{RT,4} \end{pmatrix} \rightarrow \dots$$

After each update of the vector, we can prove the feasibility ($z_{r,\text{new}} \leq d_j$) and determine the next slot ($r = \text{ind}\,(\min\{z_1, z_2, \dots, z_m\})$) and the time instance for reconfiguration ($\min\{z_1, z_2, \dots, z_m\}$).

4.1.2 Limitations

As stated above, when using EDD for the reconfigurable slot scheduling of reconfigurable architectures, we cannot guarantee optimality. In fact, EDD can miss to produce a feasible schedule. Figure 4 shows the problems, which is due to the possible additional δ_i of each task. We can also see that we have to dissociate from the statement that EDD also reduces the maximum lateness in our reconfigurable environment.

To summarize, using EDD, we can guarantee the minimization of the maximum lateness only if no reconfiguration phase is delayed.

4.2 Asynchronous arrival of independent tasks

We now release the restriction of synchronous arrival of all tasks, i. e., tasks can dynamically enter the system. If we have such arbitrary arrival times, preemption becomes an important factor. In the literature, we find that when preemption is not allowed, the problem of minimizing the maximum lateness and the problem of finding a feasible schedule become NP-hard [8, 7, 6]. If preemption is allowed, Horn [5] found an algorithm, called *Earliest Deadline First* (EDF), that minimizes the maximum lateness. The algorithm dispatches at any instance the task with the earliest absolute deadline.

Preemption for tasks executing on hardware is challenging and is not in the scope of this paper. However, we propose to preempt tasks during their RT phase, when the calculation has not started and no context saving, etc. is necessary.

In order to realize such a preemption, we divide the area reconfigured during a RT phase into columns c_j. These columns are of equal size and comprise the equal re-

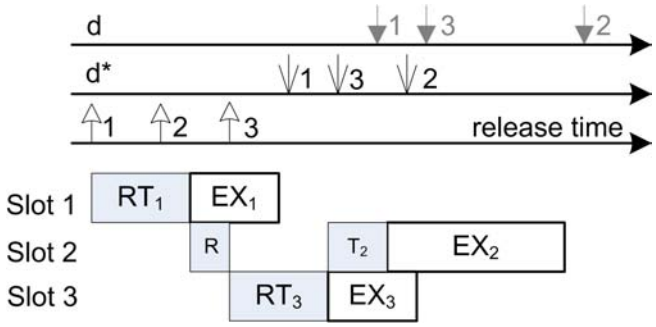

Figure 5: EDF schedule.

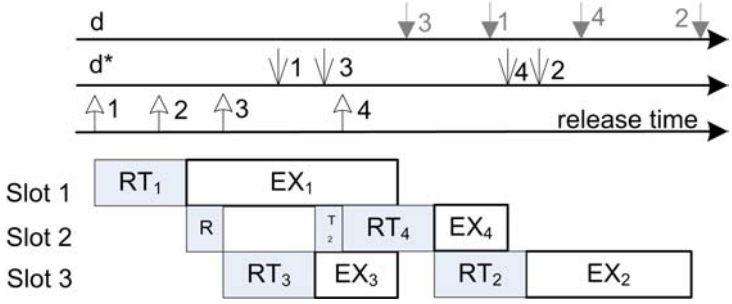

Figure 6: EDF schedule 2.

configuration time $t(c_j)$, which sum up to the reconfiguration time of the whole slot: $\sum t(c_j) = t_{RT}$. The reconfiguration process then looks as follows: gradually all the c_j of task t_v are loaded in slot s_v of the reconfigurable fabric. If a new task t_w enters the system and has an earlier deadline d^*_w, we preempt task t_v, i. e., task t_v frees the reconfiguration port and task t_w starts to reconfigure.

Depending on the current occupation of the fabric, different scenarios for the slot assignment of task t_w are possible. If we have another free slot available ($s_{\text{free}} \neq s_v$), we use this slot. After the RT phase of t_w we can resume the RT phase of t_v at the interrupted point. However, if no free slot is available, we can use slot s_v of the interrupted task. s_v becomes the slot for t_w ($s_w \Leftarrow s_v$) and RT_w overwrites all already configured parts of t_v. Thus, after finishing the reconfiguration of t_w, we cannot resume the reconfiguration phase (RT_v) of the preempted task. Instead, we have to restart RT_v completely, as already loaded parts of the bitstream are lost.

Implementation of EDF

The implementation of EDF (refer to Algorithm 2) for our scenario bases on a queue Q, which orders all tasks according to their deadlines d^*. As mentioned above, if a new task dynamically arrives to the system and its deadline is smaller than the task currently in RT phase ($t_{current}$), we start the preemption process. Note that we put $t_{current}$ at the head of the queue. Depending on the slot we use to reconfigure, we either mark $t_{current}$ as partly loaded and assign the rest of its reconfiguration time to the queue, or, in the

Algorithm 2 Earliest Deadline First for Reconfigurable Slot Architectures

```
1: if d*_new < d*_current then
2:     if all slots are in EX then
3:         wait for next free slot
4:     else if all other slots s_i ≠ slot(t_current) are in EX then
5:         add t_current completely to Q
6:         reconfigure now free slot
7:     else
8:         add rest of t_current to Q
9:         Reconfigure next free slot
10:    end if
11: else
12:    Insert t_new in queue Q
13: end if
```

case of $s_w \Leftarrow s_v$, we put $t_{current}$ and its complete t_{RT} to the head of the queue.

Technical Realization of Bitstream Preemption

Although the concept of stopping reconfiguration processes and starting in another region opens new perspectives to the scheduling in the mentioned execution environments, it is challenging to realize. We have to divide the bitstream for a slot into smaller columns, i. e., bitstreams for fewer LUTs.

Each partial bitstream is preluded by a small header section which indicates the location of the bitstream on the FPGA. Additionally to this header, each c_j enforces the reconfiguration controller to be active and initiate a new partial reconfiguration, which increases the load of the controller. Further, the overall response time will increase as each RT phase increases due to the additional headers.

The final technical solution of dividing partial bitstreams into small columns is ongoing work and is not presented in this paper. However, the capabilities of the introduced concept of RT phase preemption motivates the effort still needed.

4.2.1 Limitations

EDF in the uniprocessor domain minimizes the maximum lateness. Similar to EDD, applying EDF for the reconfigurable port scheduling of reconfigurable environments, we cannot guarantee this minimization. Again, if all slots are in EX phase, EDF cannot load a dynamically arriving task as executing tasks are assumed to be non-preemptive. As stated above, we deal with an NP-hard scenario in such a case.

Further, Figure 6 displays that a RT phase might have to be restarted completely. This will increase the overall response time and enforces a complex scheduling test to be done online after each new task has entered the system (acceptance test).

5 Conclusion

Referring to the abstract, the main contribution of this work was to investigate the adaption of known and reasonably evaluated concepts to new fields in order to pose new perspective to open problems of current research areas. In particular, in this paper, we have investigated scheduling strategies known from the single machine environment and applied them on reconfigurable devices. We focused on the reconfiguration process, as reconfiguration phases are executed sequentially and thus are applicable for the uniprocessor scheduling algorithms. We have shown that the appliance of the scheduling algorithms is possible and valuable for such scenarios, even though some limitations have to be taken into account.

Moreover, we discuss the possibility to allow task preemption during the reconfiguration phases instead of the execution phases in order to improve the schedule feasibility for tasks with asynchronous arrival time. Also posing a challenge for the implementation, the preemption of the reconfiguration phase opens new perspectives to the appliance of run-time reconfiguration on FPGAs in the real-time domain.

As ongoing work, we are currently exploring the possibilities to divide bitstreams into small columns in an efficient manner. In addition, in a future work, we plan to extend our scenario to aperiodic and periodic real-time task sets also having precedence constraints. Furthermore, we hope that sophisticated caching strategies in combination with the proposed reconfiguration scheduling will improve the results.

Acknowledgements

This work was partially supported by SFB 614 (Self-Optimizing Concepts and Structures in Mechanical Engineering) and SPP 1148 (Reconfigurable Computing) of the DFG (German Research Foundation).

References

[1] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich. Task scheduling for heterogeneous reconfigurable computers. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 22–27, New York, NY, USA, 2004. ACM Press.

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

[3] G. C. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[4] F. Dittmann and M. Götz. Applying Single Processor Algorithms to Schedule Tasks on Reconfigurable Devices Respecting Reconfiguration Times. In *13th Reconfigurable Architectures Workshop (RAW 2006)*, Rhodes Island, Greece, 25 - 26 Apr. 2006.

[5] W. Horn. Some simple scheduling algorithms. *Naval Reasearch Logistics Quarterly*, 21, 1974.

[6] H. Kise, T. Ibaraki, and H. Mine. A solvable case of the one machine scheduling problem with ready and due times. *Operations Research*, 26(1):121–126, 1978.

[7] J. K. Lenstra and A. H. G. Rnnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1977.

[8] J. K. Lenstra, A. H. G. Rnnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[9] R. Maestre, M. Fernandez, F. J. Kurdahi, N. Bagherzadeh, and H. Singh. Configuration management in multi-context reconfigurable systems for simultaneous performance and power optimizations. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 107–113, Washington, DC, USA, 2000. IEEE Computer Society.

[10] C. Steiger. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Comput.*, 53(11):1393–1407, 2004. Member-Herbert Walder and Member-Marco Platzner.

[11] 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 International Conference on Field Programmable Logic and its Applications (FPL2004)*, Antwerp, Belgium, 30 Aug. - 1 Sept. 2004.

[12] 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)*, pages 24–30, June 2002.

[13] 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)*, pages 290–295. IEEE Computer Society, March 2003.

[14] 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)*, pages 831–835. Springer, August 2004.