Applying Single Processor Algorithms to Schedule Tasks on Reconfigurable Devices Respecting Reconfiguration Times*

Florian Dittmann and Marcelo Götz
Heinz Nixdorf Institute, University Paderborn, Germany
Fuerstenallee 11, 33102 Paderborn, Germany, {roichen, mgoetz}@upb.de

Abstract

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 is occupied for some reasonable time during reconfiguration. Thus, we can find an analogy to the single machine environment. In this paper, we investigate the appliance of single processor scheduling algorithms to task reconfiguration on reconfigurable systems. We determine necessary adaptations and propose methods to evaluate the scheduling algorithms.

1 Introduction

Recently in the reconfigurable computing field, several authors have proposed similar architectural concepts for fine-grained run-time reconfigurable systems. The architectures comprise a specific number of slots, in which tasks are dynamically allocated and executed. In addition, the inherent parallelism 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 implemented as a Reconfigurable System-on-a-Chip (RSoC).

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. 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 must be used exclusively. Furthermore, the reconfiguration time cannot be neglected.

Those two characteristics (exclusiveness and reconfiguration time) 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 the processor. Similar, reconfiguration phases must be assigned sequentially to the exclusive reconfiguration port. We investigate several scheduling strategies known from single processor real-time systems and adapt them for our scenario. We rely on independent tasks and propose a novel approach where a task may be preempted in its reconfiguration phase. Additionally, we explain how guarantee tests can be realized.

The rest of the paper is organized as follows. After summarizing related work, we abstract the scenario. Then, we investigate a set of independent tasks having synchronous arrival time and subsequently enhance the scenario to tasks having arbitrary arrival times. We show the analogy and limitation to the single environment schedule. We conclude and give an outlook.

2 Related Work

Significant amount of work has already been done in online scheduling of real-time tasks on reconfigurable architectures. Most works distinguish 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. Similarly, the authors of [8] and [6] analyze the effect of overall response time and guarantee-base scheduling when tasks comprise different shapes. When task preemption is al-
allowed [1, 9], the task acceptance rate is improved. However, hardware task preemption represents additional costs due to still non-efficient techniques and methods available. All those 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 [9]. In our paper, we propose the inclusion of the reconfiguration phase into the scheduling of real-time tasks on reconfigurable devices.

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. The tasks arrive 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. All tasks 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 can be found in Figure 1 (see also [7, 10]). Partial reconfiguration capabilities enable a single slot to be reconfigured keeping remaining ones in execution.

We model every task of our system with two different phases. The reconfiguration phase ($RT$) represents the configuration of the hardware itself. $RT$ must precede the second phase, 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. We also display the motivation for partial reconfiguration.

Partial reconfiguration results in an improved overall response time of the task set, as $RT$ can take place during $EX$. Thus, we pipeline $EX$ and $RT$ of different tasks, hiding the reconfiguration time.

Due to the exclusive usage of the reconfiguration port, only one $RT$ phases can be scheduled at the same time. However, multiple tasks can execute at the same time. Further resource conflicts (e.g., sharing of the same bus) are out of the scope of this paper.

A Task $t_i$ has the computation or execution time $t_{EX,i}$, the reconfiguration time $t_{RT}$ and the deadline $d_i$. We want to emphasize the maximum lateness, which is a known metric for performance evaluation $L_{max} = \max_i \left( f_i - d_i \right)$ (ref. to [3]). Furthermore, 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 Synchr. arrival of independent tasks

A set of $n$ aperiodic tasks is executed in $m$ slots ($m < n$). The tasks have synchronous arrival time, but different execution time $t_{EX,i}$ and deadline $d_i$. We do not need preemption as no new tasks will enter the system during run-time. The problem is solved in the single processor environment w.r.t. minimizing the max. lateness using 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 $\delta_i$. 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. See Algorithm 1.

If we can guarantee at least one free slot at the beginning of each $RT$, all results of EDD of the single machine environment hold and EDD is optimal with
Algorithm 1 EDD for Reconf. Slot Architectures

1: if reconf. port is inactive (i.e., no RT 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:   Reconf. slot $r$, ($r = \text{ind} (\min \{z_1, z_2, \ldots, z_m\})$
7: end if

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 $z_l, l \neq r$, we apply $z_{l, \text{new}} = \max \{z_{l, \text{old}}, (z_{r, \text{old}} + t_{RT})\}$. 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 ($z_{l, \text{new}} = z_{l, \text{old}} + t_{RT}$). 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 $\{z_1, z_2, \ldots, z_m\}$.

As stated above, we cannot guarantee optimality. In fact, EDD can fail to produce a feasible schedule due to the possible additional $\delta_i$ of each task (see Figure 4). Furthermore, 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 only to minimize the maximum latency if no reconfiguration phase is delayed.

5 Asynchr. arrival of independent tasks

We now release the restriction of synchronous arrival of all tasks, i.e., tasks can dynamically enter the system. Now, preemption becomes an important factor. We find that when preemption is not allowed, the problem of minimizing the max. lateness and the problem of finding a feasible schedule become NP-hard [5]. If preemption is allowed, Horn [4] 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 reconfiguration time $t(c_j)$. 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 $d_t^w$, 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\(_v\) overwrites all already configured parts of \(t_w\). After finishing the reconfiguration of \(t_w\), we cannot resume the RT\(_v\) phase of the preempted task. Instead, we have to restart RT\(_v\) completely, as already loaded parts of the bitstream are lost.

EDF in the uniprocessor domain minimizes the maximum lateness. Applying EDF for the reconfigurable port scheduling, 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. We deal with an NP-hard scenario in such a case. Furthermore, 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.

6 Conclusion

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 showed 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. We plan to extend our scenario to aperiodic and periodic real-time task sets also having precedence constraints.

References


---

**Algorithm 2 EDF for Reconf. Slot Architectures**

1: if \(d_{\text{new}} < d_c\) then
2: if all slots are in EX then
3: wait for next free slot
4: else if all other slots \(s_i \neq \text{slot}(t_c)\) in EX then
5: add \(t_c\) completely to \(Q\)
6: reconfigure now free slot
7: else add rest of \(t_c\) to \(Q\)
8: Reconfigure next free slot
9: end if
10: else Insert \(t_{\text{new}}\) in queue \(Q\)
11: end if