2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines

PATS: a Performance Aware Task Scheduler for Runtime Reconfigurable Processors

Lars Bauer, Artjom Grudnitsky, Muhammad Shafique, and Jörg Henkel
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{lars.bauer, artjom.grudnitsky, muhammad.shafique, henkel} @ kit.edu

Abstract—**Multi-tasking is one of the main requirements for complex embedded systems to fulfill user expectations (e.g. flexibility of the system), increase the resource utilization, and thus increase the system efficiency. In general, the flexibility and efficiency can be increased by incorporating a fine-grained reconfigurable fabric (e.g. an embedded FPGA) that is coupled with a general-purpose processor and accelerates the computationally intensive kernels. This work focuses on reconfigurable processors that use a reconfigurable fabric to implement Special Instructions (SIs) that are invoked by the processor and process data-dominant parts. For each SI the decision whether it is executed in hardware or emulated in software can be changed dynamically at runtime.**

In this paper, we present our novel Performance Aware Task Scheduler (PATS) that decides the task schedule at runtime while considering the specific system state of the reconfigurable processor. For instance, if a task *t* has to emulate several SI executions in software because reconfiguring the corresponding hardware implementations is not completed yet, then it might be more efficient to schedule other tasks first, depending on the soft-deadlines of the tasks, until the reconfigurations of that task *t* are completed.

In comparison to other task schedulers (earliest deadline first, rate monotonic scheduling, and round robin), PATS achieves on average a 1.45× better system tardiness (i.e., the sum of cycles by which tasks miss their deadlines). Additionally, PATS reduces the makespan (i.e. the time when all tasks have completed all of their jobs) on average by 1.17× (up to 1.58×). Especially in challenging multi-tasking scenarios with tight deadlines or a small reconfigurable fabric PATS performs significantly better than other task schedulers do.

I. Introduction

Embedded systems with challenging and often-changing computational requirements (as in smart phones) are typically implemented as multi-tasking systems. High computational requirements may be addressed by Application Specific Instruction Set Processors that extend a processor core with so-called Special Instructions (SIs) to expedite the computational intensive kernels of an application in an efficient way [1]. Reconfigurable processors extend this concept by implementing their SIs by means of a reconfigurable fabric. This increases the flexibility, as reconfiguration allows to accelerate different SIs (of potentially different tasks) at different points in time, i.e., changing the set of SIs that are implemented in hardware at runtime.

This work focuses on reconfigurable processors that comprise a standard (fixed) processor pipeline plus an embedded FPGA (eFPGA) [2, 3] to implement SIs. These processors reconfigure accelerators that implement entire SIs [4] or smaller accelerators where potentially multiple accelerators are used to implement an SI [5, 6]. This leads to different *performance levels* at which a task may execute, depending on the number of requested accelerators that are available on the fabric at a certain point in time. When no accelerators are available to expedite an SI, then it executes in software (e.g. using an 'unimplemented instruction'-trap to trigger the software execution). When some accelerators are available, the task's performance level increases due to the parallelism that is exploited by the accelerators. When all requested accelerators are reconfigured, the task operates at its highest performance level.

The accelerators that are beneficial for a task typically vary during execution, for instance whenever the control flow of a task moves from one computational kernel to another that invokes different SIs (demanding different accelerators). Right after moving to a different kernel, the performance level of the task is rather low, as the beneficial accelerators are not reconfigured yet. After some accelerators for the new SIs are reconfigured, the performance level rises. For instance, an H.264 video encoder comprises three different kernels (motion estimation, encoding engine, and de-blocking filter) that need to be executed for each frame. At a frame rate of 25 frames per second (i.e. 40ms per frame), an H.264 encoder reconfigures and executes these 3 different kernels within every 40ms.

This paper presents our performance aware task scheduler (PATS) that is aware of the runtime changing performance levels of tasks (due to their reconfigurations) and exploits them to improve the system performance while still considering the tasks' deadlines. PATS aims at avoiding scheduling tasks when their performance level is low (i.e. when not all of their reconfigurations are completed yet), as they need more cycles to perform the same computations, compared to executing the task after all requested accelerators have been reconfigured. However, when targeting soft real-time systems where tasks have soft deadlines, it is not sufficient to focus on the performance level. To be able to meet deadlines, it might be required to schedule a task even though it does not perform at a high performance level, i.e. both the performance level and the soft deadline have to be considered to determine the schedule.

Paper organization: Section II explains the reconfigurable processor and task model that is used in this work and it illustrates the problem of existing schedulers by means of a case study. Section III reviews related work on multi-tasking for reconfigurable processors and elaborates on the differences to our approach. Section IV explains the concepts and details of our novel task-scheduling approach, Section V evaluates our proposed approach and analyses the results, and Section VI draws conclusions.

II. System Overview and Case Study

A. System Overview

This work employs a typical reconfigurable processor that combines a standard (fixed) processor pipeline with an embedded FPGA (eFPGA) [2, 3] to implement accelerators. Representative examples of this widely used class of reconfigurable processors exist [4-8]. Special Instructions (SIs) are used as an interface between the application and the accelerators. SIs are implemented for computationally intensive operations, e.g. transformations (e.g. DCT, FFT), filtering (e.g. FIR, IIR), or other data manipulations (e.g. encryption, checksums, bit manipulation). One or more accelerators are used to implement the functionality of an SI in hardware. Whenever the application executes an SI

978-0-7695-4699-5/12 $26.00 © 2012 IEEE
DOI 10.1109/FCCM.2012.43

and the required accelerators are not reconfigured, then the decode stage of the processor invokes the “unimplemented instruction” trap and the corresponding trap handler executes the SI functionality without accelerators (i.e. execution takes place in software) [9]. Once the accelerators finished reconfiguration, the operating system marks the SI hardware implementation as ‘available’ in a small lookup table in the decode stage and then the next execution of the SI uses the accelerators on the reconfigurable fabric rather than invoking the trap. The same lookup table could also be used to implement a conditional branch that can select the hardware or software implementation of an SI from the application code [10]. However, that would affect the performance of both SI implementation types (HW and SW), whereas the trap handler does not affect the peak performance (i.e. when an SI executes in hardware). During execution of an SI in hardware, the processor pipeline stalls. The accelerators have access to the same memory hierarchy as the processor to obtain input data and store results [5].

When a task demands a different set of SIs (e.g. when moving from one kernel to another) it informs the operating system which SIs it requires next and the operating system manages their reconfiguration. During reconfiguration of an accelerator, all other accelerators and the processor pipeline remain operational. Our prototype uses the partial reconfiguration feature [11] of a Xilinx Virtex-4 to reconfigure the accelerators in this way. Reconfiguration takes a non-negligible amount of time (in the range of milliseconds, for instance 0.6-0.7ms for each accelerator on our Virtex-4 prototype). Therefore, it is not beneficial to assign the entire available reconfigurable fabric to the executing task (i.e. the task that is scheduled to execute) because that would require many reconfigurations after a task is preempted and the context is switched, i.e. after scheduling a different task that demands different accelerators. For instance, Linux executes tasks in *time slices* that can last for about 1-100 milliseconds (depending on the system configuration) before the context switches to a different task. For a video input rate of 25 frames per second, a new video frame arrives after each 40ms. Thus, the task’s time slice needs to be noticeable shorter than 40ms. If a task demands eight accelerators then their reconfiguration demands 4.8-5.6ms (0.6-0.7ms per accelerator), i.e. it would last about half of the time of a 10ms time slice to reach the highest performance level. Hence, instead of continuously reconfiguring after each context switch, each task attains a share of the reconfigurable fabric and it can reconfigure the accelerators that it requires to this share. For tasks that execute SIs, the user-determined task priority decides which task obtains which share of the reconfigurable fabric.

The reconfiguration of multiple accelerators is performed sequentially one after each other. This is a technical constraint of the reconfigurable hardware as it provides only one internal configuration access port (ICAP). The reconfiguration sequence of the accelerators requested by one task is determined as described in [12]. As multiple tasks may request SIs (and thus need accelerators), their reconfiguration sequence needs to be determined by the operating system. A weighted round robin approach is used in our system to decide the reconfiguration sequence, i.e. tasks with higher priorities receive their accelerators faster. Other policies could be considered as well, but relying on task priorities for deciding the reconfiguration sequence and for deciding which task obtains which share of the reconfigurable fabric allows the user to specify which tasks shall benefit more from acceleration by the reconfigurable fabric.

B. Task Model

We now define the task model that is targeted by PATS and define measures for evaluation (based on the one presented in [13]) as follows:

- Our system supports **sporadic tasks** (executing once after being released) and **periodic tasks** (executing repeatedly after being released), without focusing on inter-task dependencies (e.g. task graphs).
- Our system supports **preemption**, i.e. the executing task is interrupted after a certain time slice to re-evaluate which task shall execute next (potentially the same task that was interrupted), depending on the scheduling policy. However, we do not support preempting an SI if it started executing in hardware. The longest running hardware implementation of an SI in our benchmarks executes for 139 cycles, thus the delay of the preemption is insignificant.
- **Job *j*:** corresponds to one period of a periodic task.
- **Release Time R_j:** the release time of job *j* corresponds to the time when the job arrives at the system (i.e. the earliest time at which job *j* can be scheduled). Note that a released job can only be scheduled after the preceding jobs of the same task finished execution (i.e. the jobs of a task execute in the same sequence in which they are released; they do not *overtake* each other).
- **Deadline D_j:** the deadline of job *j* denotes the time at which the job should be completed (soft deadline).
- **Completion Time C_j:** the time when job *j* actually finishes. A subsequent job *k* of the same task (i.e. a later period $R_k > R_j$) cannot be scheduled before job *j* finishes (i.e. $R_k > C_j$).
- **Tardiness A_j:** the tardiness of job *j* is defined as the time job *j* finishes too late (i.e. after its deadline) or 0 if it finishes in time, see Eq. 1.
- **System Tardiness ST_t:** The tardiness of the system at time *t* is defined in Eq. 2. It considers all jobs that are released until time *t* and sums up the tardiness of these jobs (if they already completed) or the time that has passed since their deadline (if they did not yet complete but already missed their deadline). Jobs that are already released but that did not yet complete or miss their deadline do not increase the System Tardiness as these jobs may still finish before their deadline.
- **Makespan:** The makespan is defined as the time when all tasks have completed all of their jobs, see Eq. 3.

$$A_j := \max\{0,\ C_j - D_j\} \tag{1}$$

$$ST_t := \sum_{\text{Jobs } j \text{ with } R_j \le t} \begin{cases} A_j & \text{, if } t \ge C_j \\ t - D_j & \text{, if } t < C_j \text{ and } t \ge D_j \\ 0 & \text{, else} \end{cases} \tag{2}$$

$$\mathit{makespan} := \max\{C_j \mid \forall \text{Tasks } T\ \forall \text{Jobs } j \text{ of Task } T\} \tag{3}$$

Each task provides –among others– information about its priority and a deadline. The soft deadline denotes when the task should terminate, or when it should execute the *yield* system call (used by periodic tasks to inform the operating system that the task completed its current job). After a yield is issued, the respective task is blocked until its next job is released. A task can also request different SIs (typically when proceeding from one kernel to another), which will

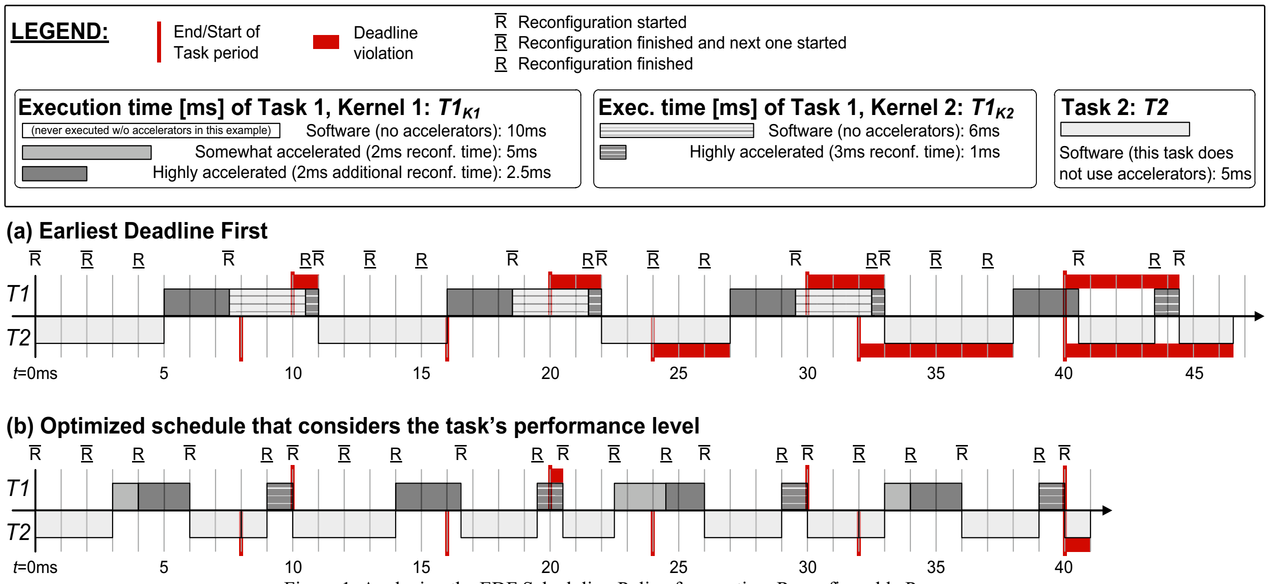

Figure 1. Analyzing the EDF Scheduling Policy for run-time Reconfigurable Processors

also preempt the task and call the task scheduler. If a task does not terminate, execute a yield, or requests new SIs, it executes until its time slice completes. After completing one time slice, the task is preempted and the task scheduler decides which task to execute next (potentially the same).

C. Motivation: A Scenario in Detail

Conventional task schedulers for non-reconfigurable processors assume that the execution time of a task (or a job of a task) is mostly constant (potentially fluctuating dependent on the input data). This is not the case for reconfigurable processors due to the changing performance levels of the tasks. Conventional schedulers base their decisions on task-related parameters (deadline, priority etc.) rather than on the system-state (like the number of available accelerators etc.). The well-known Earliest Deadline First (EDF) scheduler is an optimal scheduler for non-reconfigurable systems, i.e. if any scheduling approach finds a feasible solution to meet deadlines then EDF finds it as well. However, EDF does not consider cases in which the performance levels of tasks vary at runtime. To meet all deadlines (or to minimize the overall deadline misses) in a soft real-time system it may be necessary to schedule a task *T1* later than a task *T2* even though *T1* might have the earlier deadline (EDF would always schedule *T1* first). Executing *T1* at time t_i might lead to reduced system performance compared to later execution at time t_{i+j}, when adequate accelerators are not available at t_i, but complete reconfiguration at t_{i+j}.

We will now present a detailed case study regarding this observation to motivate the differences between a scheduler for non-reconfigurable processors and one that considers the specifics of reconfigurable processors. The scenario in Figure 1 shows the execution of two periodic tasks *T1* and *T2*, where *T1* consists of two subsequently executed kernels $T1_{K1}$ and $T1_{K2}$. Both tasks start executing at time t=0ms and their periods are $p(T1)$=10ms and $p(T2)$=8ms, respectively. The periods also denote the deadlines of the periodic tasks. For simplicity, let us assume only *T1* benefits from accelerators in this scenario. Thus, its performance level depends on the availability of accelerators during execution. The performance levels of all kernels and the reconfiguration times of the accelerators are summarized in the legend of Figure 1. As both kernels of *T1* benefit from accelerators, their execution demands frequent reconfigurations, as it is typical for complex tasks (for instance an H.264 video encoder as motivated in Section I).

When analyzing the EDF schedule in Figure 1(a), it becomes apparent that *T2* starts executing first since it has the earlier deadline. *T1* can already start reconfiguring its accelerators,[1] i.e. at t=2ms and 4ms its performance improves by 2× and 4×, respectively. However, *T2* continues executing according to the EDF policy. There are several drawbacks regarding *T1* in this scenario:

1. The high performance level of *T1* after all its accelerators are reconfigured is not exploited, i.e. the accelerators complete reconfiguration but they remain idle until *T2* finishes its job.
2. Due to the delayed execution of *T1* it is no longer possible to meet its deadline (*T1* completes its first job at t=11ms; deadline misses highlighted as red bars).
3. Due to the delayed start-/completion time of $T1_{K1}$, the reconfigurations for $T1_{K2}$ are started rather late (at t=7.5ms) which leads to a reduced performance level for $T1_{K2}$.

When comparing this first period with a scheduler that is optimized for reconfigurable systems (shown in Figure 1b), it becomes obvious that exploiting the high performance level of $T1_{K1}$ after its reconfigurations are completed, addresses the above-mentioned drawbacks. The optimized scheduler also starts *T2* at t=0ms (like EDF), because *T1* reconfigures its accelerators at that time and has a correspondingly low performance level. At t=3ms the scheduler switches the execution context from *T2* to *T1* where *T1* executes for 1ms at 2× performance (corresponds to 2ms non-accelerated performance level) and then 2ms at 4× (corresponds to 8ms non-accelerated). Depending on the actual scheduling policy the scheduler could also switch from *T2* to *T1* at t=2ms (where the performance level increases to 2×) or at t=4ms (where the performance level increases to

[1] this is for instance possible by using the partial reconfiguration flow [11] from Xilinx (as done in our prototype)

4×). Because the system switches to *T1* before *T2* finishes its first job, the execution of $T1_{K1}$ already finishes at *t*=6ms, which allows initializing the reconfigurations for $T1_{K2}$ earlier. As soon as *T1* triggers these reconfigurations, its performance level becomes low because the available accelerators are no longer required and the demanded accelerators are not yet available. Therefore, the scheduler switches back to *T2*, which still manages completing its job before the deadline.

As EDF is not exploiting the different performance levels of tasks in reconfigurable processors, it performs bad, as can be noticed by the deadline misses that are highlighted by the red bars shown in Figure 1. For instance, at *t*=33ms the fourth job of *T2* starts executing which actually should have finished already at *t*=32ms. At *t*=40.5ms EDF preempts the execution of *T1* which hides the reconfiguration latency between completing $T1_{K1}$ and starting $T1_{K2}$. This is possible because *T1* and *T2* have the same deadline in that case (*t*=40ms). As EDF does not specify which task shall execute first when the deadlines are identical, the task that leads to the best performance is scheduled here. Altogether, EDF leads to seven deadline misses in the scenario shown in Figure 1, summing up to a system tardiness of 26ms (highlighted by the red bars). The optimized scheduler that considers the advantages of reconfigurable processors only leads to two deadline misses, summing up to a system tardiness of 1.5ms.

III. Related Work

Reconfigurable processors have been investigated and developed in academia and industry for several years. Surveys that present general classifications as well as recent trends are available [14, 15]. Recently, an overview of reconfigurable processors with support for multi-tasking was presented in [16]. It categorizes existing approaches into those with *implicit*, *explicit*, and *no* architectural support for multi-tasking and raises several research questions, which shows the actuality of this topic.

Many approaches for reconfigurable processors with multi-tasking support focus on implementing entire tasks either in software or in hardware. ReconOS [17, 18] provides an infrastructure that allows tasks to communicate independent of whether their communication partner executes in hardware or in software. Additionally, ReconOS investigated the reconfiguration and placement for the hardware tasks. Noguera et al. [19] focus on task graphs, where each task executes in hardware or in software, independent of the other tasks. However, they are limited to a non-preemptive multitasking with a statically predetermined decision which task executes in hardware and which in software. OS4RS [20] proposes a hierarchical approach to assign tasks dynamically to computing resources (processor or reconfigurable fabric). Stitt et al. [21] investigated an approach for online synthesis to create hardware implementations of threads on the fly. They focus on a single multi-threaded task that executes on multiple processors and accelerators in parallel. However, the time-consuming online synthesis limits the amount of adaptation, as it prohibits a frequent reevaluating of the hardware/software partitioning. FUSE [22] and HTI [23] introduce abstractions for the reconfigurable fabric to enable programming of FPGA fabric and CPU resources in a consistent manner. They focus on providing a general infrastructure rather than a scheduling algorithm that is optimized for reconfigurable processors. Tang et al. [24] present a scheduler that targets architectures with heterogeneous processing elements including reconfigurable FPGAs. They schedule periodic tasks offline and extend the resulting schedule at runtime to integrate sporadic tasks. However, they only support non-preemptive (i.e. a task always executes until completion) tasks.

In summary, all of these approaches are limited to implementing an entire task in hardware or in software and thus their flexibility is restricted. This affects the efficiency, as the control flow dominant and/or computationally non-intensive parts of an application operate more efficiently when executed on the processor, whereas data-flow dominant parts that are computationally intensive operate more efficiently when executed on the reconfigurable hardware.

Instead of implementing entire tasks in hardware, many reconfigurable processors (e.g. [4, 25, 5, 8, 7]) use software tasks that are accelerated by Special Instructions (SIs). For each SI, the decision is made whether it shall be implemented using a hardware accelerator or whether it shall execute in software. This SI-specific HW/SW decision provides more flexibility for multi-tasking as it allows more different performance levels for each task (compared to implementing entire tasks either in HW or in SW). Wu et al. [26] present such an approach with several SIs for different tasks for the coarse-grained reconfigurable ADRES processor. However, they are limited to a compile-time prepared arrangement of all SI implementations and reconfiguration decisions for a specific multi-tasking scenario, which also significantly limits the adaptation and therefore does not exploit the inherent potential of reconfigurable processors. Huynh et al. [27] focus on non-preemptive tasks and use an offline-prepared task execution sequence. However, in multi-tasking scenarios, the tasks execute in a time-multiplexed manner rather than one after each other, which raises the question for task scheduling at runtime.

HybridOS [28] focuses on access methods between a task and its accelerator to simplify programming such tasks, but it provides no specific support or evaluation for multi-tasking. Proteus [7] focuses on the ability to preempt SI executions and on sharing the accelerators among different tasks but does not propose a new scheduler. Santambrogio et al. [29] design an operating system that provides support for reconfiguration management. They have no support for multi-tasking, though their system is evaluated with two different tasks in two different simulations, running exactly one task per simulation. Huang et al. [30] present a system where a task can be accelerated by loosely-coupled reconfigurable coprocessors. The decision whether that coprocessor shall be reconfigured is determined when the task starts execution and it cannot be changed afterwards. Kahrisma [8] presents a reconfigurable multi-core processor where several tasks execute at the same time (one per core, i.e. simultaneous multi-tasking). They do not use any task scheduler but focus on distributing the reconfigurable fabric among the tasks.

To summarize: none of the discussed approaches presents a task scheduler that is optimized for reconfigurable processors executing sporadic or periodic preemptible tasks. In this paper, we present our Performance Aware Task Scheduler (PATS) that considers the changing performance levels of tasks for the task scheduling decision.

IV. Performance Aware Task Scheduler (PATS)

To consider changing performance levels of tasks, we introduce a metric for *task efficiency* as defined in Eq 4. For task *T* it considers the accelerators *T.reqAcc(K)* that are requested to expedite Kernel *K* and it considers the accelerators *T.attAcc(t)* that are already attained (i.e. finished reconfiguration) at time *t*. Depending on the attained accelerators, the execution latency *S.latency()* of an SI *S* changes. The efficiency metric in Eq. 4 divides the latency of an SI when using the accelerators that are requested (but potentially not reconfigured yet) by the latency when using the accelerators that are actually available at time *t*. The highest efficiency of an SI is '1', i.e. all requested accelerators are available. When not all requested accelerators are available yet, then the SI executes slower (larger latency), thus the efficiency is smaller than '1'. The metric in Eq. 4 averages these efficiencies over all SIs that are executed in a kernel to determine the efficiency of the task. The efficiency of a kernel that does not demand any SIs is set to '1', because it cannot improve its performance level further.

$$TaskEfficiency(T,K,t) :=$$

$$\begin{cases} \dfrac{\displaystyle\sum_{\substack{\text{SIs } S \text{ invoked} \\ \text{in Kernel } K}} \frac{S.latency(T.reqAcc(K))}{S.latency(T.attAcc(t))}}{\left|\text{SIs } S \text{ invoked in Kernel } K\right|}, & \text{if } \left|\begin{matrix}\text{SIs } S \text{ invoked} \\ \text{in Kernel } K\end{matrix}\right| > 0 \\ 1, & \text{else} \end{cases} \quad (4)$$

To consider task efficiencies, PATS uses three different queues to manage the tasks. Each task (except the currently executing task) is placed in one of these queues.

NRQ: Not Released Queue – these tasks cannot be scheduled, as the previous job (if any) has completed and the next job is not released yet.

LEQ: Low Efficiency Queue – these tasks can be scheduled but they would run at reduced efficiency due to not yet reconfigured accelerators.

FEQ: Full Efficiency Queue – these tasks can be scheduled and all requested accelerators are available.

The task scheduler places the currently executing task in the appropriate queue when control is passed to another task (context switch). Whenever an accelerator finishes reconfiguration, a task that was previously in LEQ might need to be moved to FEQ if that accelerator was the last accelerator that the task intended to reconfigure. Moving the tasks from LEQ to FEQ is managed by the handler that is responsible for triggering the next reconfiguration.

The main steps of PATS are i) to select a set of tasks that shall be considered for the scheduling decision and ii) to select one of these tasks while considering the task efficiencies and the task deadlines. The pseudo code for PATS is shown in Figure 2 and will be explained in the following. PATS first inserts the currently executing task *CT* into the appropriate queue (lines 1-8). This simplifies the algorithm, because *CT* does not need to be handled as a special case (as it otherwise would not be in any of the queues).

PATS then pre-selects a subset of all executable tasks (from LEQ and FEQ) as candidates to be scheduled next (lines 10-20). All tasks from FEQ are considered as candidates, because they run at their full efficiency. Additionally, those tasks from LEQ are considered that have a negative

PATS Task scheduler: Called due to one of the following reasons: a) Time Slice finished, b) The task finished its period ('yield' system call), c) The task requested a different set of accelerators ('SI request' system call), or d) a new job (period) was released from one task
INPUT: *CT* – currently executing task
OUTPUT: *next_task* – task to be scheduled next

```
1.  // Insert CT into the appropriate task list
2.  if (CT.yielded) {
3.      NRQ.insert(CT);
4.  } else if (CT.RequestedAccelerators = CT.AttainedAccelerators) {
5.      FEQ.insert(CT);
6.  } else
7.      LEQ.insert(CT);
8.  }
9.
10. // Identify candidate tasks to be scheduled
11. if (FEQ = ∅) {
12.     candidate_tasks ← LEQ;
13. } else {
14.     candidate_tasks ← FEQ;
15.     for each task T in LEQ {
16.         if (T.slack < 0) candidate_tasks.insert(T);
17.     }
18. }
19. if (candidate_tasks = ∅)
20.     return NULL; // nothing to be scheduled
21.
22. // Select one of the candidates
23. next_task ← NULL;
24. for all tasks T in candidate_tasks {
25.     // Calculate Efficiency of T
26.     if (T.numberOfRequestedSIs=0) {
27.         efficiency ← 1;
28.     } else {
29.         efficiency ← 0;
30.         for all SIs S that are requested by T {
31.             efficiency ← efficiency +
                    S.getLatency(T.requestedAccelerators) /
                    S.getLatency(T.attainedAccelerators);
32.         }
33.         efficiency ← efficiency / T.numberOfRequestedSIs;
34.     }
35.
36.     // Additionally consider the relative slack
37.     score ← α × efficiency – (T.slack / T.averageExecutionTime);
38.     // Remember the task with the best score
39.     if (next_task = NULL || next_task_score < score) {
40.         next_task ← T;
41.         next_task_score ← score;
42.     }
43. }
44.
45. // remove selected task from its queue
46. if (next_task.RequestedAccelerators =
        next_task.AttainedAccelerators) {
47.     FEQ.remove(next_task);
48. } else {
49.     LEQ.remove(next_task);
50. }
51. return next_task;
```

Figure 2. Pseudo code for our Performance Aware Task Scheduling approach: PATS

slack (see line 16). The *slack* denotes the remaining amount of cycles that the task is expected to execute until the end of its period or until it terminates. The average task execution time is estimated by averaging the execution times of the previous periods of this task. A negative slack denotes that the task can no longer meet its deadline (which is acceptable as it is a soft deadline, but should be avoided), as the estimated execution time is longer than the time until the deadline. Considering these tasks allows reducing the system tardiness (i.e., the sum of cycles by which tasks miss their deadlines) at the cost of scheduling tasks that do not perform at their full efficiency (as motivated in Section II). If there are no tasks performing at their full efficiency, then all tasks in LEQ are considered as candidates.

TABLE I. PARAMETERS USED FOR BENCHMARKING

Configuration Parameter	Values
Processor Frequency [MHz]	100
SI Frequency [MHz]	100
Reconfiguration Bandwidth [MB/s]	66
Accelerator reconfiguration time [ms]	0.6 – 0.7
Scheduler time slice [ms]	4[a]
Number of Reconfigurable Containers [RCs]	8 – 20
Scheduling Policies	EDF, RMS, RR, PATS
Number of evaluated Multi-tasking Scenarios	10
Number of Tasks per Multi-tasking Scenario	2 – 6
Task Deadlines[b]	Relaxed, Normal, Tight
Number of total Simulations	360

a: Such a rather short time slice is required to execute video encoders that target 25 frames per second (40ms per frame) or even higher frame rates

b: The deadlines are specific for the different applications; 'relaxed' deadlines are only violated in some simulations; 'tight' deadlines are nearly always violated

After pre-selecting a list of candidates, PATS selects one task out of this list (lines 22-43 in Figure 2). The efficiency of all tasks in the candidate list are computed (lines 25-34) according to Eq. 4. Therefore, PATS iterates over all SIs *S* that are requested by the task (i.e. invoked in the kernel that the task executes) and calculates the average latency of executing *S* using the accelerators that are available at time *t* in comparison to the latency of *S* after all reconfigurations are completed (lines 30-33). For calculating the task efficiency, PATS needs to know the SI latencies that depend on the number of accelerators. These latencies are prepared for each task at compile time and provided to the operating system when the task starts.

Only focusing on the task efficiency would lead to a very good utilization of the reconfigurable fabric but might lead to a large amount of deadline misses, which should be avoided in soft-deadline systems. Therefore, in addition to the task efficiency, PATS also considers the slack of the tasks for its scheduling decision. For each task in the candidate list, PATS calculates a score that is used to decide which task (highest score) shall be scheduled.

The score is initialized with the calculated efficiency and then modified, depending on the slack of the task (see line 37 in Figure 2). The score of tasks with a positive slack (i.e. they can still meet their deadline) is reduced, whereas the score of tasks with a negative slack is increased. To be able to compare the slacks of the different tasks with each other, they are put in perspective to the estimated task execution time (i.e., by how much percent of the estimated task execution time will the task miss its deadline). To combine the task efficiency and the relative slack into a score, the efficiency is weighted, because it is a number between 0 and 1, whereas the relative slack is a number in cycles. For all task sets in the experiments, a constant value of $\alpha=10$ is used. Potentially this value could be specific for each task and adapted at runtime (e.g. depending on the tardiness of a task), but even with a constant parameter, our PATS outperforms the other task schedulers as shown in the results. The task with the largest score is identified (lines 38-42), removed from its queue (lines 45-50), and returned as the scheduling decision (line 51).

The computational complexity of PATS to decide which task is scheduled next is $\mathcal{O}(|Tasks| \times |SIs|)$, due to calculating the task efficiency for potentially all SIs of potentially all tasks. The number of SIs in a task is typically small (never more than 10 in the benchmark applications). The number of tasks that can be accelerated is limited by the size of the reconfigurable fabric (note that the efficiency of tasks that are not accelerated is '1' by definition, see line 27 in Figure 2). Therefore, executing PATS after every time slice is computationally unproblematic.

V. EVALUATION AND RESULTS

A. EXPERIMENTAL SETUP

In addition to the processor and system-overview provided in Section II.A, we now present details on the experimental setup. To evaluate many different multi-tasking scenarios running on a reconfigurable processor, we integrated the proposed task scheduler into our in-house cycle-accurate SystemC simulator for reconfigurable processors. Using this simulator allows us to evaluate different scenarios and architectural parameters with a faster turnaround time in comparison to hardware prototyping. However, important parameters of our simulator (e.g. reconfiguration time, context switching time etc.) are configured according to measurements on our Virtex-4 LX 160 based hardware prototype and summarized in Table I.

The prototype is based on a LEON 2 processor [31] with MMU, caches, and DDR-RAM. The reconfigurable fabric is connected to the pipeline and can be reconfigured at runtime while maintaining all other components functional [11]. The SIs in the reconfigurable fabric obtain their input data from the register file, the memory hierarchy (connected to the cache), and an on-chip scratchpad memory that can be accessed by two 128-bit ports (similar to Tensilica's LX processor family [32]). We run Linux 2.6.21.1 on the hardware prototype and measure a context switching time between 6.4 and 6.85 µs. Reconfiguring one accelerator demands between 0.6 and 0.7ms. An accelerator demands on average 112 Virtex-4 Slices and can be configured into a reconfigurable container (RC, a region on the reconfigurable fabric for partial reconfiguration). Our prototype comprises 10 RCs. For our simulations, the size of the reconfigurable fabric is varied between 8 and 18 RCs. For 18 RCs, the reconfigurable processor using PATS performs 14.3× faster than the identical processor without SIs, which shows the general performance advantage of reconfigurable processors.

For comparing the quality of PATS with other task schedulers, we have implemented Earliest Deadline First (EDF), Rate Monotonic Scheduling (RMS), and Round Robin (RR). They represent different concepts of task scheduling approaches that focus on the completion time (EDF), priorities (RMS), and fairness (RR). We have generated 10 different multitasking scenarios where each scenario executes between 2 and 6 SI-accelerated periodic tasks. Each task has an individual deadline and three different settings for these deadlines are evaluated: relaxed, normal, and tight. Relaxed deadlines are chosen such that they are only violated in rather few simulations. Tight deadlines represent the case that it is practically impossible to meet all deadlines and we evaluate how the different task schedulers perform in these different scenarios.

Table II shows the tasks with the amount of SIs and accelerators used for benchmarking. All accelerators for the SIs were synthesized, placed, and routed for all RCs to obtain their reconfiguration times. The task with the highest complexity is the H.264 video encoder that is accelerated using 9 different SIs that are composed of 10 different ac-

TABLE II. PROPERTIES OF THE TASKS USED FOR BENCHMARKS

Task	Number of SIs	Number of different Accelerator Types[c]
Video Encoding: H.264	9	10
Image Decoding: JPEG	4	5
Image Processing: SUSAN	3	7
Audio Encoding: ADPCM	1	2
Error Detection Code: CRC	1	1
Hash Algorithm: SHA	1	1

[c:] Multiple instances per accelerator type can be used to expedite SI execution; each accelerator demands on average 112 Slices on our Virtex-4 prototype; reconfiguring one accelerator demands 0.6-0.7ms

celerator types. Three different kernels (motion estimation, encoding engine, and deblocking filter) are executed per encoded video frame. The most SIs are used in the encoding engine, i.e. DCT, inverse DCT, Hadamard Transformation (HT), inverse HT, motion compensation, and intra prediction. The other applications are taken from the MiBench suite.

B. *RESULTS*

As the workload consists of periodic tasks, the ability of the scheduler to minimize system tardiness (i.e., the sum of cycles by which tasks miss their deadlines, see definitions in Section II) is of interest. Figure 3 shows the system tardiness averaged over all 360 multi-tasking scenarios that are described in Section V.A. In average, PATS achieves a 1.45× better (i.e. lower) system tardiness than the other three schedulers. In comparison to EDF, RMS, and RR, the system tardiness is in average 1.29, 1.92, and 1.14 times better, respectively.

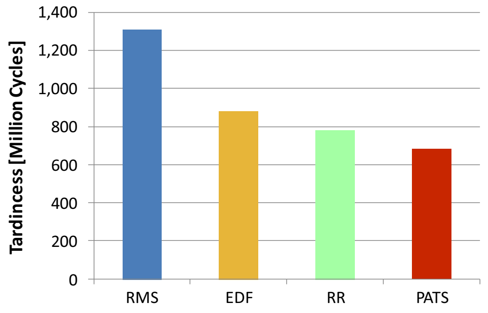

Figure 3. Comparing the system tardiness of the four task schedulers, averaged over all 360 simulations

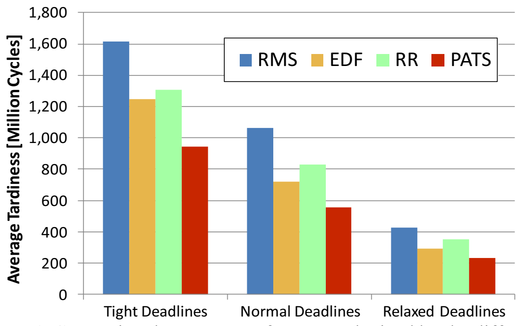

Figure 4. Comparing the system performance obtained by the different task schedulers for different tight task deadlines

To evaluate PATS in more detail, we examine one multi-tasking scenario, where four tasks are executing together (SUSAN, ADPCM, and two instances of the complex H.264 video encoder, see Table II. Figure 4 evaluates this scenario, while applying tight, normal, or relaxed deadlines, i.e. requesting more or less performance from the system and thus from the task scheduler. We have chosen the deadlines for all four tasks independent of each other (but identical for all task schedulers for a fair comparison) such that the 'relaxed deadlines' are typically manageable, whereas 'tight' deadlines' are hardly manageable, to stress the system. It becomes clear by Figure 4 that PATS outperforms the other schedulers in all three cases and that it achieves the largest margins when the deadlines are tight.

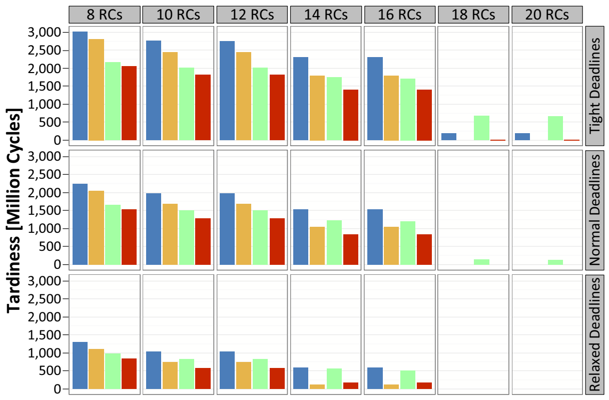

Figure 5. Detailed analysis of the system performance obtained by the different task schedulers when changing the size of the reconfigurable fabric (i.e. number of reconfigurable containers (RCs)) and the deadlines

Figure 5 analyzes this behavior further. The horizontal axis shows the size of the reconfigurable fabric, i.e. the number of RCs. For less than 18 RCs, the H.264 video encoder tasks cannot implement all of their kernels in hardware, which degrades their performance significantly and leads to the deadline misses. Still, our PATS scheduler manages this situation better than the other schedulers do. When more RCs are available, RR still fails to meet the deadlines in some cases, due to its too simplistic scheduling decision. However, when a small reconfigurable fabric is used (8 RCs), then RR achieves a system tardiness that is nearly as good as when using PATS. Still, PATS performs better than all other schedulers in all scenarios, which shows the wide applicability of our algorithm.

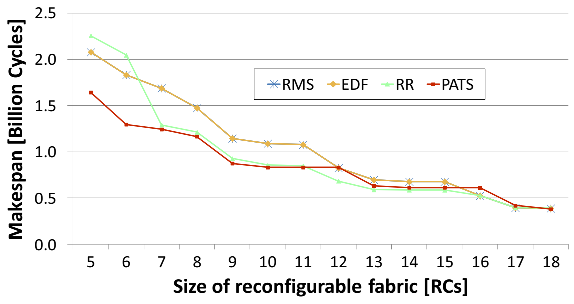

Figure 6. Makespan (i.e. time when all tasks completed) for different task schedulers and different number of reconfigurable containers (RCs)

For benchmarking the makespan (as defined in Section II.B) we use multi-tasking scenarios without deadlines. Deadlines could lead to idle times (i.e. no job is released to be executed at a certain point in time), but the makespan denotes how fast a workload is processed and not whether or not deadlines are met. Figure 6 shows the makespan for the four different task schedulers. Even though PATS does not explicitly consider the makespan, it leads to the fastest completion of all jobs when using 5-11 RCs and is up to

TABLE III. SPEEDUP OF MULTI-TASKING SCENARIO (I.E. REDUCED MAKESPAN) WHEN SCHEDULED BY PATS

Scheduler	min	avg	max
EDF	0.86	1.17	1.41
RMS	0.86	1.17	1.41
RR	0.82	1.05	1.58

1.41× faster than the closest competitor (up to 1.58× faster than RR). For more than 11 RCs, Round Robin leads to the shortest makespan (i.e. fastest execution). Still, PATS is never more than 18% slower than RR. Due to the missing deadline, EDF and RMS perform exactly the same scheduling decision. In comparison to EDF and RMS, PATS leads to an 1.17× shorter makespan. Table III summarizes the different makespan results.

VI. CONCLUSION

This work presents the novel Performance Aware Task Scheduler (PATS) for processors that are accelerated by reconfigurable Special Instructions (SIs). PATS considers the efficiency of a task to determine the scheduling decision. The efficiency depends on the accelerators that a task requests to implement SIs and the accelerators that are available (i.e. reconfigured) at a certain point in time. When more of the requested accelerators are available, then the task executes with a better performance, i.e. at a higher efficiency. Additionally, PATS considers the soft deadlines of tasks to reduce the system tardiness, i.e., the sum of cycles by which tasks miss their deadlines.

We have compared our approach with scheduling algorithms that represent different task scheduling concepts, i.e. that focus on the completion time (Earliest Deadline First, EDF), priorities (Rate Monotonic Scheduling, RMS), and fairness (Round Robin, RR). In comparison to these schedulers, PATS achieves a system tardiness that is on average 1.29, 1.92, and 1.14 times better, respectively. Overall, PATS improves the average system tardiness by 1.45×. Additionally, PATS reduces the makespan (i.e. the time when all tasks have completed all of their jobs) up to 1.58×. Especially in challenging multi-tasking scenarios with tight deadlines or a rather small reconfigurable fabric PATS performs significantly better than the other schedulers do.

VII. ACKNOWLEDGMENT

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

REFERENCES

[1] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: The next design discontinuity", in *International Conference on Computer Design (ICCD)*, 2002, pp. 84–90.

[2] S. J. E. Wilton *et al.*, "A synthesizable datapath-oriented embedded FPGA fabric", in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2007, pp. 33–41.

[3] B. Neumann *et al.*, "Design flow for embedded FPGAs based on a flexible architecture template", in *Conference on Design, Automation and Test in Europe (DATE)*, 2008, pp. 56–61.

[4] S. Vassiliadis *et al.*, "The MOLEN polymorphic processor", *IEEE Trans. on Computers (TC)*, vol. 53, no. 11, pp. 1363–1375, 2004.

[5] L. Bauer, M. Shafique, and J. Henkel, "Concepts, architectures, and run-time systems for efficient and adaptive reconfigurable processors", in *Conf. on Adaptive HW and Systems*, 2011, pp. 80–87.

[6] J. Henkel *et al.*, "i-Core: A run-time adaptive processor for embedded multi-core systems", in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, July 2011.

[7] M. Dales, "Managing a reconfigurable processor in a general purpose workstation environment", in *Conference on Design, Automation and Test in Europe (DATE)*, 2003, pp. 980–985.

[8] W. Ahmed *et al.*, "Run-time resource allocation for simultaneous multi-tasking in multi-core reconfigurable processors", in *Symp. on Field-Program. Custom Computing Machines*, 2011, pp. 29–32.

[9] L. Bauer, M. Shafique, and J. Henkel, "A computation- and communication- infrastructure for modular special instructions in a dynamically reconfigurable processor", in *Int'l Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 203–208.

[10] W. Fu and K. Compton, "An execution environment for reconfigurable computing", in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005, pp. 149–158.

[11] P. Lysaght *et al.*, "Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs", in *Int'l Conf. on Field-Program. Logic and Appl. (FPL)*, 2006, pp. 1–6.

[12] L. Bauer *et al.*, "Run-time system for an extensible embedded processor with dynamic instruction set", in *Conference on Design, Automation and Test in Europe (DATE)*, 2008, pp. 752–757.

[13] J. Y.-T. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*. Chapman & Hall/CRC, 2004.

[14] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Publishing Company, Incorporated, 2007.

[15] H. P. Huynh and T. Mitra, "Runtime adaptive extensible embedded processors – a survey", in *International Workshop on Embedded Computer Systems (SAMOS)*, 2009, pp. 215–225.

[16] P. G. Zaykov, G. K. Kuzmanov, and G. N. Gaydadjiev, "Reconfigurable multithreading architectures: A survey", in *Int'l Workshop on Embedded Computer Syst. (SAMOS)*, 2009, pp. 263–274.

[17] E. Lübbers and M. Platzner, "ReconOS: An RTOS supporting hard- and software threads", in *International Conference on Field Programmable Logic and Applications (FPL)*, 2007, pp. 441–446.

[18] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks", *Trans. on Comp. (TC)*, vol. 53, no. 11, pp. 1393–1407, 2004.

[19] J. Noguera and R. M. Badia, "Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling", *Trans. on Embedded Comp. Syst. (TECS)*, vol. 3, pp. 385–406, 2004.

[20] V. Nollet *et al.*, "Hierarchical run-time reconfiguration managed by an operating system for reconfigurable systems", in *Int'l Conf. on Engineering Reconf. Syst. and Algorithms (ERSA)*, 2003, pp. 81–87.

[21] G. Stitt and F. Vahid, "Thread warping: a framework for dynamic synthesis of thread accelerators", in *Int'l Conf. on HW Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 93–98.

[22] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators", in *Int'l Symp. on Field-Program. Custom Comp. Machines (FCCM)*, 2011, pp. 170–177.

[23] D. Andrews *et al.*, "Programming models for hybrid FPGA-cpu computational components: a missing link", *IEEE Micro*, vol. 24, no. 4, pp. 42–53, 2004.

[24] H.-K. Tang, P. Ramanathan, and K. Compton, "Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources", in *International Conference on Parallel Processing (ICPP)*, 2011, pp. 753–762.

[25] J. E. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors", in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2001, pp. 141–150.

[26] K. Wu *et al.*, "Mt-ADRES: multithreading on coarse-grained reconfigurable architecture", in *Int'l Conf. on Reconf. Computing: Architectures, Tools and Applications (ARC)*, 2007, pp. 26–38.

[27] H. P. Huynh and T. Mitra, "Runtime reconfiguration of custom instructions for real-time embedded systems", in *conf. on Design, automation and test in Europe (DATE)*, 2009, pp. 1536–1541.

[28] J. H. Kelm and S. S. Lumetta, "HybridOS: runtime support for reconfigurable accelerators", in *international symposium on Field programmable gate arrays (FPGA)*, 2008, pp. 212–221.

[29] M. D. Santambrogio *et al.*, "Operating system runtime management of partially dynamically reconfigurable embedded systems", in *Workshop on Emb. Syst. for RT Mult. (ESTIMedia)*, 2010, pp. 1–10.

[30] C. Huang and F. Vahid, "Transmuting coprocessors: dynamic loading of FPGA coprocessors", in *Proceedings of the 46th Annual Design Automation Conference (DAC)*, 2009, pp. 848–851.

[31] Aeroflex Gaisler, "Homepage of the Leon processor", http://www.gaisler.com/leonmain.html.

[32] Tensilica Inc., "Xtensa LX2 I/O Bandwidth", http://www.tensilica.com/products/io_bandwidth.htm.