INTEGRATION the VLSI journal 57 (2017) 34–44

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Run-time management of systems with partially reconfigurable FPGAs

George Charitopoulos[a,b], Iosif Koidis[a,b], Kyprianos Papadimitriou[a,b], Dionisios Pnevmatikatos[a,b,*]

[a] Institute of Computer Science Foundation for Research and Technology (FORTH-ICS), Heraklion, Greece
[b] School of Electronic and Computer Engineering Technical University of Crete, Chania, Greece

ARTICLE INFO

Keywords:
Run time system
Scheduling
Partial reconfiguration
FPGA
OpenMP

ABSTRACT

Partial reconfiguration (PR) of FPGAs can be used to dynamically extend and adapt the functionality of computing systems by swapping in and out HW tasks. To coordinate the on-demand task execution, we propose and implement a Run-Time System Manager (RTSM) for scheduling software (SW) tasks on available processor(s) and hardware (HW) tasks on any number of reconfigurable regions (RRs) of a partially reconfigurable FPGA. Fed with the initial partitioning of the application into tasks, the corresponding task graph, and the available task mappings, the RTSM controls system operation considering the status of each task and region (e.g. busy, idle, scheduled for reconfiguration/execution, etc). Our RTSM supports task reuse and configuration prefetching to minimize reconfigurations, task movement among regions to efficiently manage the FPGA area, and region reservation for future reconfiguration and execution. We validate the correctness and portability of our RTSM executing an image processing application on two Xilinx-based platforms: ZedBoard and XUPV5. We also perform a more extensive evaluation of its features using a simulation framework, and find that – despite the technology limitations – our approach can give promising results in terms of scheduling quality. Since our RTSM supports also the scheduling of parallel SW tasks, we use it to manage the execution of the entire parallel Edge Detection application on a desktop; we compare the application execution time with that using the OpenMP framework and find that with our RTSM execution is 2.4 times faster than the unoptimized OpenMP version. When processor affinity optimization is enabled for OpenMP, our RTMS and the OpenMP are on par, indicating that the scheduling efficiency of our RTSM is competitive to this state-of-the-art scheduler, while supporting in addition the management of HW tasks.

1. Introduction

Reconfiguration offers the possibility to dynamically adapt the functionality of hardware systems by swapping in and out HW tasks. To coordinate resource management, loading and triggering HW task reconfiguration, and execution in partially reconfigurable systems with FPGAs, efficient and flexible runtime system support is needed [1]. To this end, several scheduling algorithms of various complexities have been proposed [2]. In this paper we propose and implement a Run-Time System Manager (RTSM) incorporating efficient scheduling mechanisms that efficiently manage the execution of HW and SW tasks and the use of physical resources. We aim to execute a given application as fast as possible without exhausting the physical resources. Our motivation during the development of our RTSM was to find ways to design a versatile system under the strict technology restrictions imposed by the Xilinx PR flow and devices [3]:

- Static partitioning of the reconfigurable surface in reconfigurable regions (RRs).
- Reconfigurable regions can only accommodate particular hardware core(s), called reconfigurable modules (RM). The RM–RR binding takes place at compile-time, after sizing and shaping properly the RR.
- An RR can hold one RM only at any point of time, so a second RM cannot be configured into the same RR even if there are enough free logic resources for it.

The proposed RTSM can run on Linux Intel-x86 based systems with a PCIe FPGA board, e.g. XUPV5, or on embedded processors (Microblaze or ARM) within the FPGA, while it can be ported in other systems with different processors and FPGAs. Furthermore, with the appropriate changes it can also run solely on Linux based systems without an FPGA in to manage the available CPU cores only.

* Corresponding author at: School of Electronic and Computer Engineering Technical University of Crete, Chania, Greece.
E-mail addresses: gcharitopoulos@mhl.tuc.gr (G. Charitopoulos), koidis@mhl.tuc.gr (I. Koidis), kpapadim@mhl.tuc.gr (K. Papadimitriou), pnevmati@ics.forth.gr (D. Pnevmatikatos).

http://dx.doi.org/10.1016/j.vlsi.2016.11.008
Received 30 April 2015; Received in revised form 27 October 2016; Accepted 24 November 2016
Available online 28 November 2016

We validated the behavior of RTSM in three different fully functional systems: a ZedBoard Zynq SoC-based platform, an XUPV5 platform, and a desktop PC with a 12-core Intel Xeon E5 processor at 2.2 GHz. This also allowed us to asses the RTSM in three different CPU technologies: ARM, MicroBlaze, and Intel-x86. Also we evaluate extensively the RTSM with complex cases within a simulation framework that observes all the restrictions of partial reconfiguration technology. The present work extends our previous publication [4] and makes with the following contributions:

- A portable RTSM capable of scheduling both HW and SW tasks in PR FPGA-based systems and SW-only tasks with comparable results to the OpenMP API.
- Support for dynamic execution of complex task graphs, with forks, joins, loops and branches so as to minimize the restrictions on the application.
- Support multiple scheduling policies, such as relocation, reuse, configuration prefetching, reservation and Best Fit.
- Extensive evaluation of the proposed RTSM on three different systems and a comparison with industry standard OpenMP API in controlling SW-only tasks.

The paper is structured as follows. In Section 2 we discuss previous work in the field. In Section 3 we present the key concepts and provide details on the RTSM input and operation. Then, in Section 4 we evaluate the performance of RTSM in a simulation environment with complex test cases and in Section 5 we extend our evaluation and validating on real FPGA-based platforms. Finally, Section 6 summarizes our work.

2. Related work

There is an increasing interest in exploiting the advantages of using partial dynamic reconfiguration instead of full reconfiguration. The work in [5] was one of the first to study the use of partial reconfiguration in the high-performance computing domain. In one of the first research works on hardware task scheduling for PR FPGAs, Steiger et al. addressed the problem for the 1D and 2D area models by proposing two heuristics; Horizon and Stuffing [6]. In [7], Marconi et al. inspired by [6] presented a novel 3D total contiguous surface heuristic in order to equip the scheduler with “blocking-awareness” capability. Subsequently, Lu et al. created a scheduling algorithm that considers the data dependencies and communication amongst hardware tasks, and between tasks and external devices [8].

Efficient placement and free space management algorithms are equally important. In [9], Bazargan et al. offer methods and heuristics for fast and effective on-line and off-line placement of templates on reconfigurable computing systems. Compton et al., in a fundamental work in the field of task placement, inspired by the concept of task relocation, proposed run-time partitioning and creation of new RRs in the FPGA [10]. However, the proposed transformations are still well beyond what is currently supported by FPGA technology. Since then, much work has been done towards efficient bitstream relocation [11].

A work that did not consider Compton's paradigm and therefore followed the strict FPGA technology restrictions regarding partial reconfiguration is [12]. There the authors present a novel reduced data movement scheduling (RDMS) algorithm takes data dependency among tasks, hardware task resource utilization, and inter-task communication into account during the scheduling process. However their algorithm is not being tested on an actual FPGA and is evaluated by means of emulation. The authors in [13] studied an approach for hardware task placement and space management focusing on a resource- and configuration-aware floorplacement framework, using an objective function, based on external wirelength [13]. This work targets scheduling at compile-time.

Burns et al., in one of the first efforts to create an operating system (OS) for partially reconfigurable devices, extracted the common requirements for three different applications, and designed a runtime system for managing the dynamic reconfiguration of FPGAs [1]. Ghringer et al. addressed the efficient reconfiguration and execution of tasks in a multiprocessing SoC, under the control of an OS [14,15].

More recently, Agne et al. targeted the issue of a run-time system for reconfigurable systems [16]; in ReconOS the creators provide the user with strict semantics and an OS support. ReconOS incorporates the basic principle of message parsing by creating delegate threads that handle the communication between hardware and software threads. However, throughout the ReconOS description there is no mention of a high-level decision making scheduler.

On the field of programming models that target multi-core heterogeneous architectures a great impact had the OmpSs model presented in [17]. Apart from multi-core architectures, OmpSs can incorporate the use of OpenCL and CUDA kernels. OmpSs differs from similar programming models like OpenMP and MPI in the sense that they do not adopt a fork–join model. Instead, OmpSs has a thread pool were all the threads, to be used throughout the application, are present from the beginning.

The run-time management of hardware tasks in partially reconfigurable devices is interesting and very active [18]. The OpenPR toolchain [19] and the GoAhead frameworks [20] provide a solid base for further research into partial reconfiguration and reconfigurable run-time systems. Also many previous efforts have evaluated scheduling and placement algorithms on actual FPGA-based systems [21,14].

What seems to be missing are complete solutions that take into consideration all current technology restrictions. In [14], the actual overhead of the scheduler compared to the execution time of each task is not calculated and also the reconfiguration time measured is the theoretical one, while the application execution is presented in a theoretical way. The run-time manager presented in [21] is able to map multiple applications on the underlying PR hardware and execute them concurrently and takes all restrictions in consideration; however the mechanics of the scheduling algorithm are simple and the overhead considerable.

3. The Run-time System Manager

Our proposed RTSM manages physical resources employing placement and scheduling algorithms to select the appropriate hardware processing element (HW-PE), i.e. a Reconfigurable Region (RR), to load and execute a particular HW task, or activate a software processing element (SW-PE) for executing the SW version of a task. HW tasks are implemented as Reconfigurable Modules, stored in a bitstream repository.

3.1. Key concepts and functionality

During initialization, the RTSM is fed with input, which forms the basic guidelines according to which the RTSM takes runtime decisions:

(1) *Device pre-partitioning and Task mapping*: The designer should pre-partition the reconfigurable surface at compile-time, and implement each HW task by mapping it to certain RR(s) [3]. This limitation was discussed in [1,21].
(2) *Task graph*: The RTSM should be aware of the execution order of tasks and their dependencies; this is provided with a task graph. Our RTSM supports complex graphs with properties like forks and joins, branches and loops, for which the number of iterations is unknown at compile-time.
(3) *Task information*: Execution time of SW and HW tasks, and reconfiguration time of HW tasks should be known to the RTSM; they can be measured at compile-time through profiling. A task's execution time might deviate from the estimated or profiled execution time so the RTSM should react adapting its scheduling

decisions.

(4) *Schedulable resources*: Our RTSM currently considers as schedulable resources the RRs and the SW-PE present on the design, and the configuration controller since ICAP can service only one reconfiguration request at a time. We do not consider communication in the scheduling, as the task requests are based on the control flow of the application. We assume that the communication cost for each task is included in the task execution time estimate, which is considered by our scheduler in deciding whether the HW or the SW version of the task will be used.

The RTSM supports the following features:

(1) *Multiple bitstreams per task*: A HW task can have multiple mappings, each implemented as a different RM. All versions would implement the same functionality, but each may target a different RR, thus increasing placement choices, and/or be differently optimized, e.g. in terms of performance, power, etc. A similar approach is used in [1], and accounts for the increased scheduling flexibility and quality [21].

(2) *Reservation list*: When a task cannot be served immediately due to resource unavailability (either RR or SW-PE), it is reserved in a queue for later configuration/execution. A HW task will wait in the queue until an RR is available, or it is assigned to a SW-PE (provided that a software implementation of the task is available).

(3) *Reuse policy*: Before loading a HW task into the FPGA, the RTSM checks whether it already resides in an RR and can be reused. This prevents redundant reconfigurations of the same task reducing the overhead. If an already configured HW task cannot be used, e.g. it is busy processing other data, the RTSM may find it beneficial to load this task's bitstream to another RR, if such a binding exists.

(4) *Configuration prefetching*: Allows for the configuration of a HW task into an RR ahead time [22]. It is activated only if the configuration port is available.

(5) *Relocation*: A HW task residing in an RR can be "moved" by loading a new bitstream implementing the same functionality to another RR, as illustrated in Fig. 1. Two RMs are being scheduled for configuration into two RRs; RM1 is already configured in RR2. RM2 should also execute, so it is waiting to be configured, but its RR is unavailable. The proposed relocation mechanism first moves the HW task by configuring the RM1 to RR1, and then configures the RM2 to the now empty RR2. This differs from the previously proposed relocation mechanism [10]. To fully exploit the benefits of this approach context save techniques are needed [23].

(6) *Best Fit in Space* (*BFS*): It prevents the RTSM from injecting small HW tasks into large RRs, even if the corresponding RM–RR binding exists, as this would leave many logic resources unused. BFS minimizes the area overhead incurred by unused logic into a used RR, pointing to similar directions with studies on sizing efficiently the regions and the respective reconfigurable modules [24].

(7) *Best Fit in Time* (*BFT*): Before an immediate placement of a task is decided, the BFT checks if reserving it for later start time would result in a better overall execution time. This can happen due to reuse policy: when HW tasks are called more than once, e.g. in loops. For example, consider a HW task that is to be scheduled and already exists in an RR due to a previous request. Scheduling decision evaluates which action amongst reservation, immediate placement, or relocation, will result in the earliest completion time of this task. For instance, BFT might invoke reconfiguration of a HW task into a new RR, even though this HW task (equal functionality, but different bitstream) already resides in another RR (but it is busy executing or has been already scheduled for execution).

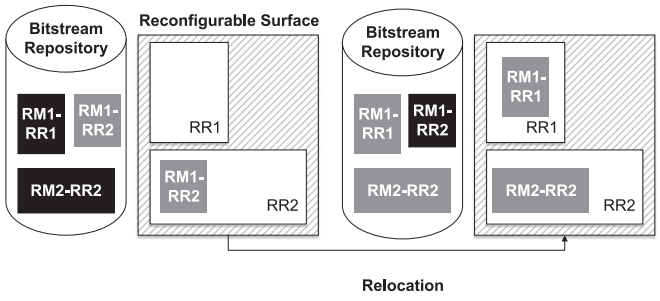

Fig. 1. Relocation: RM2–RR1 does not exist, thus the hardware task laying in RR2 is relocated by first configuring RM1–RR1, and then RM2–RR2.

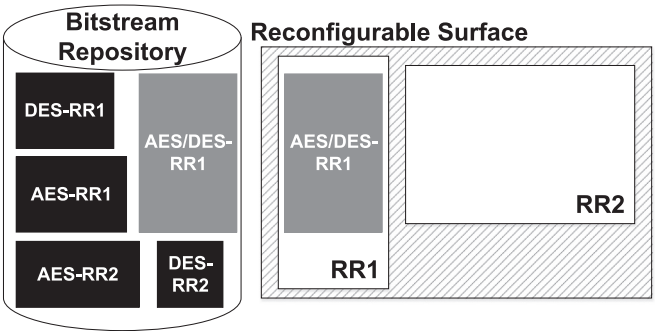

Fig. 2. Implementations for all crypto modules are available for and can be loaded into any RR, however a large amount of resources will be under-utilized. JHM utilizes more efficiently the RR area, given that the corresponding bitstream (the combined AESDES module) is available.

(8) *Joint Hardware Modules* (*JHM*): It is possible to create a bitstream implementing at least two HW tasks, thus allowing more than one tasks to be placed onto the same RR. JHM, illustrated in Fig. 2, exploits this ability by giving priority to such bitstreams, which can result in better space utilization and reduced number of reconfigurations. A similar concept was presented in [25].

We incorporated the above features in the RTSM, and tested them within a simulation framework presented in the following section. The combination of BFT with the Reservation list and their reaction with the Reuse policy constitute an interesting feature, leading the scheduler to hybrid decisions that potentially benefit an application. To this end, we believe that it is important to study if complex techniques and features are actually required to efficiently serve different kinds of applications.

3.2. RTSM input and execution flow

The input to the RTSM is the partitioning of the FPGA into partially reconfigurable HW-PE resources, the availability of SW-PE resources, the tasks to be scheduled, the task graph representation describing the task dependencies, and the available task mappings, i.e. bitstreams for the different implementations of each hardware task, and tasks implemented in software that can be served by a SW-PE. Additionally, the RTSM needs the reconfiguration and execution times of each task, and optionally the task deadlines. This information is used to update the RTSM structures and perform the initial scheduling.

The above are provided in an input file or can be dynamically linked with the RTSM library, so as the RTSM retrieves them prior to application execution. We use list structures to represent the reconfigurable regions (RR list); the tasks to be executed (task list); the bitstreams for each task (mappings list); and reservations for "newly arrived" tasks waiting for free space (reservation list). Since we do not consider random arrival times of tasks, we provide a definition by which a task is characterized as "arrived task": *If a task has completed its execution at time t=x, then the next in sequence dependent task as retrieved from the task graph has an arrival time* t_{arr}=x+1.

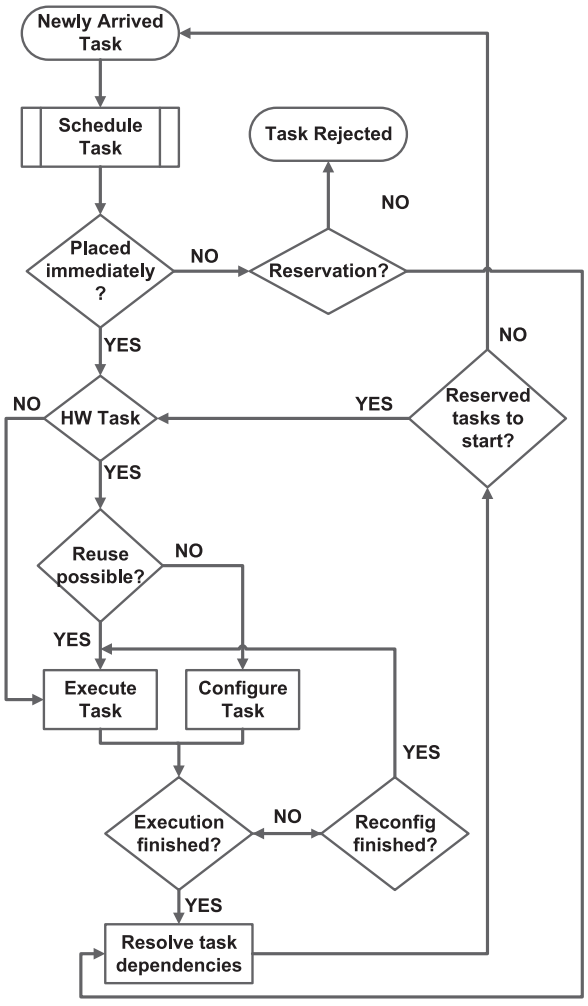

Fig. 3. RTSM main routine.

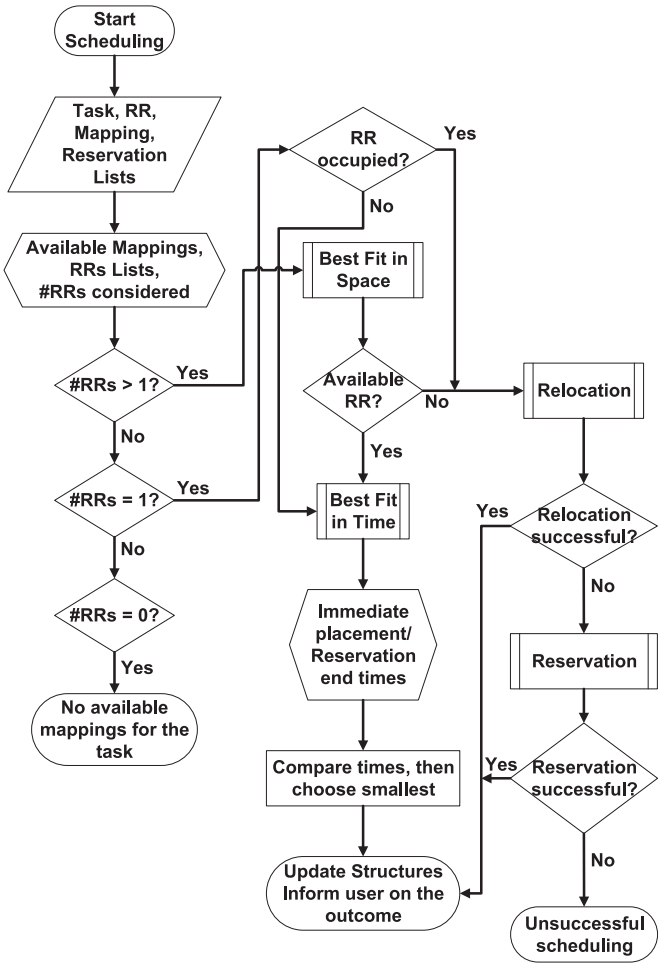

Fig. 4. RTSM "Schedule Task" function.

The RTSM gets as input the 2D placement of the RRs on the FPGA, identified by (i) a pair of [x, y] coordinates and (ii) its size measured in slices. For example, an RR with coordinates [2,3] corresponds to the point within the reconfigurable area on which the bottom-left corner of the rectangle is placed. Regarding the size, for example, an RR with size 4×5 specifies the number of slices covered by the RR in x and y dimensions respectively. The hardware tasks should be available in a bitstream repository, each of which corresponds to an RM–RR binding.

During application execution, the RTSM alters its scheduling decisions dynamically. Specifically, it reacts according to dynamic parameters such as the runtime status of each HW/SW task and region, e.g. busy executing, idle, scheduled for reconfiguration, scheduled for execution, region that is free, reconfigured-but-idle, reconfigured-and-active, etc. In this way, the RTSM reacts dynamically according to the FPGA condition and task status.

The RTSM's main routine is depicted in Fig. 3. Each rectangle in the figure refers to a more abstract action taking place at this moment, i.e. the rectangle labeled "Execute Task" refers to the actions the RTSM has to take in order to initiate the new execution of a task. A rhombus indicates an if-like statement depending on the state of a task or the specifics of a scheduling decision. Finally the ellipses on the graph indicate input/output points from/to the user.

We have structured the RTSM into two parts. In the first part the RTSM continuously checks for a newly arrived task, and calls the schedule function. After the scheduling decision is made, the RTSM checks if the task can be served immediately, and then whether the reuse policy can be invoked, and accordingly it issues an execution or reconfiguration instruction. Then, it checks if a task has completed execution, and decides which task to schedule next according to the task graph. Finally, the RTSM checks if there are reserved tasks that should start executing.

At the end of Fig. 3, we observe a bidirectional connection between the "Execution finished" and "Reconfiguration finished" states. This concerns the case where reuse is not possible, and RTSM has initiated a configuration process of a task, i.e. "Configure task" step. It reflects the fact that RTSM handles the reconfiguration and execution of a task as two completely different processes, i.e. first reconfiguration of a task should finish, and then the RTSM will initiate its execution. Then, given that the time required for the execution of a task is unknown, the RTSM is waiting to be informed by the task itself that it completed its execution, and successively the RTSM will resolve the task dependencies.

In Fig. 4 we depict the control flow of the "Schedule Task" function. If the scheduler cannot immediately find an available placement nor a future reservation for the task, we label it as "rejected" and inform the user. This is done in order to be in correspondence with other significant works on the field such as, [6–8], which make the assumption that a task can be rejected if no suitable placement is found on the FPGA. However, it is possible to configure the scheduler with enough resources so that a task can always be reserved for a future time and never be rejected.

If there are more than one RR free on the device and mappings exist for those RRs, a Best Fit policy decides which RR the task will be placed on, considering the area occupied by the task. Our scheduler will pick the bitstream of the task that best utilizes the area of the corresponding RR, i.e. it places the newly arrived task on the RR producing the

smallest unused area, provided this RR is free.

If there is only one free RR on the device and no mapping of the scheduled task exists mapping it to it, the scheduler performs relocation (box #RRs=1). With this step the scheduler tries to relocate a previously placed task to another RR so as to accommodate the newly arrived task. If this step also fails, the scheduler attempts to make a reservation for the newly arrived task, thus execute it at a later time.

Even if the scheduler finds a suitable RR for immediate placement, it will also perform a Best Fit in Time (BFT) in order to check if by reserving the task for later execution and reusing a previously placed core, the incoming task will finish its execution at an earlier time. It is important to note that the RTSM besides the RR and SW-PEs treats also the configuration controller as a scheduled resource.

3.3. Tasks with deadlines

Several works consider an abstract concept of tasks with deadlines. The objective of this approach is to achieve a task scheduling before the task's deadline. If no scheduling can happen the task is considered as rejected. These kinds of tasks do not impose any consequences to the application. Our RTSM can consider a task's deadline prior to taking scheduling and placement decisions. If no alternative – either via relocation, reservation, and execution – can meet the deadline, the task is rejected. In our case task rejection means that the task is deferred for SW execution, specifically on the host PC's CPU. However this case is not encountered in our experiments.

3.4. Observations

Task reconfiguration and execution times: These inputs are supplied by the designer, and can be derived through profiling or it can be computed using theoretical reconfiguration times and the HW bitstream size. The execution time can be estimated by the compilation tools, or can be provided by the programmer during the application design phase. This information is provided to the RTSM in its initialization. In practice though, a task's execution time can be smaller or larger than the predicted one, and the RTSM is notified on task completion, so that it may update its scheduling decisions dynamically.

Best Fit in Size (*BFS*): The BFS feature aims at placing a newly arrived task onto the RR producing the smallest unused area. Without BFS, the size of an RM does not pose any restriction for loading it into an RR, given that such a bitstream exists. The programmer can disable this feature. In all our current experiments we enabled it.

Size of RRs and RMs: These parameters are defined at design-time and have fixed values. BFS reacts based on these parameters.

4. Software simulation framework, testbed and results

To demonstrate the aforementioned features, we deployed the RTSM in a simulation environment and used it to control synthetic workloads.

4.1. Simulation framework

Fig. 5 shows the task graph we use as case study to demonstrate RTSM's behavior. In this figure we express the HW/SW execution times and reconfiguration time in arbitrary time units in order to make the understanding easier. The task graph has one instance in which three tasks have more than one dependency, i.e. T3, T7 and T8, which results in join operations. Also, in T2 there are fork operations. The available resources are two RRs and one SW-PE, as well as the FPGA configuration port, which is also treated as a resource to be scheduled. The RTSM accepts as input the width and height of each RR; these are used by the Best Fit in Space function.

Table 1 shows the available task mappings that drive the options of RTSM for making the best scheduling decision for a given task, e.g. T1 can be loaded either in RR1 or RR2. If a task has only one RR–RM binding, e.g. T2, options are limited. We assume that every task has a software implementation as well in order to study how the RTSM reacts when exploiting both hardware and software resources, which aims to reduce the overall application execution time. It is important to note here that in all experiments we assumed that the software implementation of a task has a longer execution time than its hardware counterpart; this is shown in the annotations of Fig. 5.

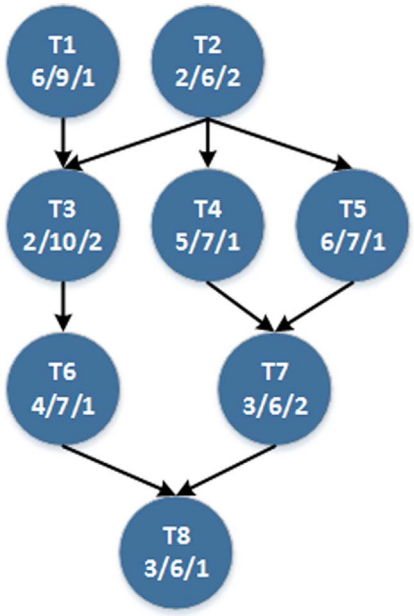

Fig. 5. Annotated application task-graph: the notation h/s/r indicates with HW, SW execution time and reconfiguration time for each task. For example for T1 the respective values are 6/9/1.

Table 1
RM–RR bindings and required space.

Tasks	Mapping characteristics		
	#RR	**Width**	**Height**
Task 1	1,2	1	2
Task 2	2	1	3
Task 3	2	1	3
Task 4	1,2	1	2
Task 5	1	2	2
Task 6	2	1	2
Task 7	1,2	1	2
Task 8	1,2	1	2

4.2. Simulation results

In Fig. 6 we illustrate the scheduling result of the previous experiment. In this run, most of the RTSM features were activated. Relocation is activated quite early in order to accommodate task T2. Since the Best Fit in Space (BFS) function has placed initially task T1 onto RR2, in order to accommodate task T2 on the FPGA, task T1 should be relocated given that an alternative binding exists. In fact, the RTSM "moves" T1 from RR2, by configuring a bitstream implementing the T1's functionality into RR1; this way the RR2 becomes available and the RTSM then loads the task T2 into RR. Furthermore, it appears that the RTSM decides to execute the software version of task T5. This is due to the Best Fit in Time (BFT) function, as reserving the hardware version of the task for later configuration onto an RR would result in longer execution time of the application.

We can also see that the reservation feature is activated in the

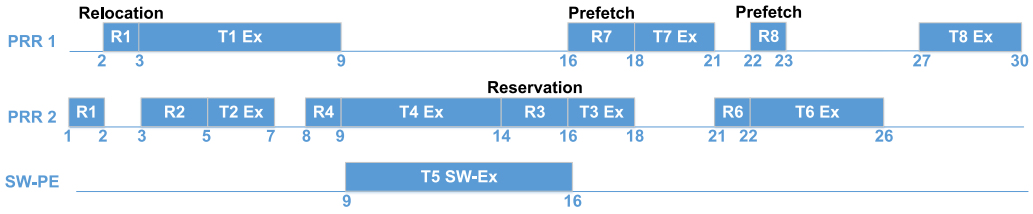

Fig. 6. The scheduling outcome of our example, showing features such as relocation, reservation and prefetching. The use of the SW version of task T5 contributes to completing faster the execution.

decision taken for task T3. The arrival time point of task T3 is t=10, and since the only available mapping concerns RR2, the only option is to reserve task T3 for later configuration on RR2. It is worth noting that the scheduler does not relocate task T4 to RR1, because T4 is near completion of its execution (otherwise it would restart its execution).

Finally, Fig. 6 shows that there is a high level of inner task parallelism between tasks of the same level but also from different levels, i.e. between T1–T2, and T4–T5, and the prefetching of tasks T7 and T8, with the execution of tasks T3 and T6 respectively.

4.3. Discussion analysis

This experiment offered a first basis for our RTSM evaluation. It focused on a quite simple PR design with 2 RRs and 1 SW-PE. Several complex operations are demonstrated here despite not being implemented in the actual system. The assumptions made for this experiment were:

- We assume a simple reconfigurable device following the 2D area model.
- Resource utilization and timing is assumed to be in arbitrary units. So a task can have a HW implementation occupying 2 units of width and 1 unit of height on the device and its execution time being 4 time units.
- We assume multiple bitstreams for each task, each with different resource utilization. We do not consider Joint Hardware Modules in this experiment.
- All tasks are unique and no loops are presented, thus reuse of hardware implementations is not demonstrated.
- We assumed that the SW-PE execution takes more time than the combined hardware execution and reconfiguration operation. In general, this favors the selection of hardware accelerators.
- To model more accurately a task's execution we include the communication time requirements of a task in the task's execution time.
- Finally, to model faster reconfiguration times, we assumed that execution time of a HW task is larger than its corresponding reconfiguration time.

The experiment presented allows us to make the following observations:

- The task graph is complex enough to demonstrate fork and join operations.
- We demonstrated most of the RTSM features: relocation, reservation, prefetching, BFT and BFS.
- Despite the fact that the RTSM favors hardware execution in the above case we saw that the scheduler opted for the software version of a task, which was due to the reduced overall execution time.
- From Fig. 6 we obtain that relocation takes place from the very beginning of the scheduling. This evidences that our approach is dynamic, i.e. at each point of time the RTSM reacts according to the FPGA condition and task status.

5. Experimental framework, testbed and results

We validated the correctness of the RTSM in two FPGA platforms by controlling the execution of an Edge Detection application. Its task graph is rather simple and tasks have a linear dependency. Thus, tasks execute is succession, with each task just passing the result of its processing to the next one. The task graph consists of 9 tasks: 5 of them are SW, i.e. 1 read image task and 4 write image tasks, while all other tasks are implemented in HW, i.e. Grayscale (GS), Gaussian Blur (GB), Edge Detection (ED) and Threshold (TH). The tasks have a linear dependency, i.e. read image passes the intermediate results to GS; GS to write image task; write image to GB; GB to the next write image task, and so on.

We are aware that the write tasks greatly affect our applications performance. However, in order to compare our results with [21] we decided to keep the same application graph. Also inner-RR communication is not implemented and thus the intermediate images need to be written in an external memory before being used again.

Fig. 7 depicts the task graph of the application, showing only task dependencies but not how the application actually executes over time. It also depicts the (implicit) SW tasks that trigger the reconfiguration process.

We used two platforms: a XUPV5 platform carrying a Virtex-5LX110T FPGA and a Zedboard platform carrying a Xilinx ZYNQ™-7000 SOC. Both are equipped with a DDR memory and an SD port for input/output. The XUPV5 architecture design and the Edge Detection hardware modules were provided by the authors in [21]. In both

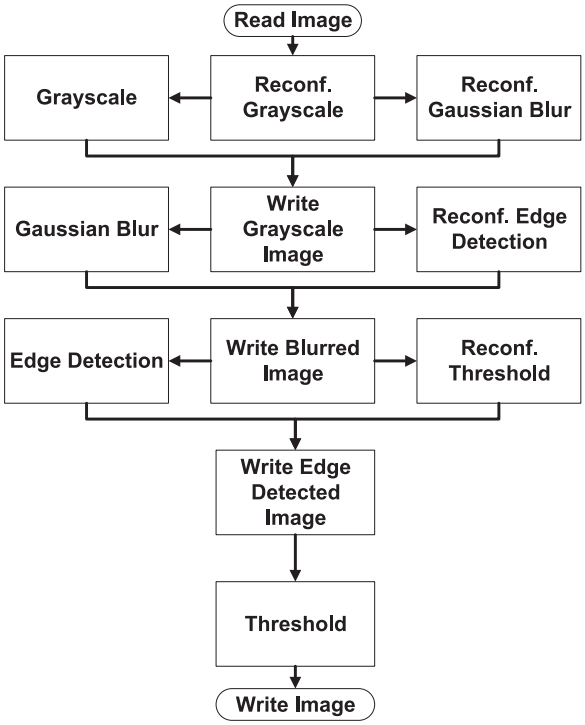

Fig. 7. The implemented Edge Detection application task graph.

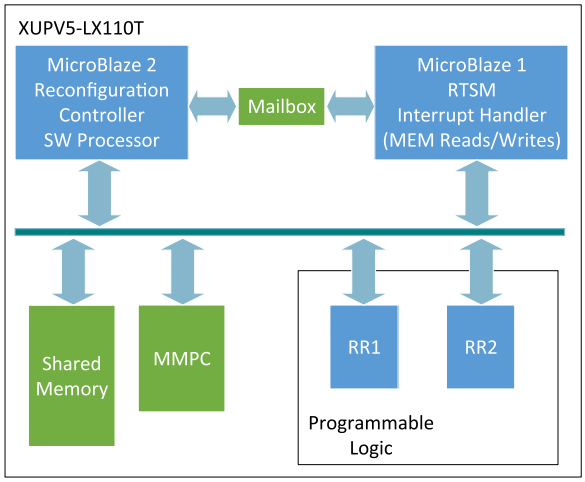

Fig. 8. The modified system we implemented in the FPGA of the XUPV5. MicroBlaze#1 holds the RTSM and handles the interrupts and memory read/writes. MicroBlaze#2 is responsible for controlling the reconfiguration process and executing the SW tasks.

platforms we created two HW-PEs, i.e. two Reconfigurable Regions.

Finally, we have validated the RTSM with a software-only system, i.e. a desktop PC with an x86 multi-core Intel CPU, in which the RTSM schedules the task execution amongst the different cores. The main idea was to compare the schedule overhead induced by our RTSM for the achievement of parallel execution in software-only environments, with OpenMP, which is an industry standard API for parallel execution.

In the remaining section we present the three implementations and comment on the portability of our RTSM. Then, we compare our work with the Open MP API.

5.1. The RTSM on the XUPV5 platform

To evaluate our RTSM we used a XUPV5 FPGA-based system provided by the authors in [21]. This system implements two MicroBlaze soft processors, and two RRs as the HW-PEs. The RTSM runs in MicroBlaze#1 to manage the physical resources, such as deciding whether a task should execute in hardware or software, in which RR, and so on. Depending on the decision outcome, MicroBlaze#1 would signal MicroBlaze#2; the latter is responsible for invoking the reconfiguration process and executing the SW tasks. Fig. 8 illustrates the system architecture initially built by the authors in [21], along with our extensions.

We described previously that the RTSM considers only estimations of the execution and reconfiguration times of the tasks, and based on them it produces the initial scheduling decisions. However, the estimated values differ considerably from the actual ones, so the RTSM should wait for a completion message from MicroBlaze#2 in order to proceed. This signal can be either a reconfiguration completion or an execution completion. Once the signal is activated, the RTSM resolves the dependencies and proceeds with the next task in the graph.

An image, either the first one or any of the intermediate ones, resides in the DDR and the application accesses it via interrupts on the Microblaze#1 processor by sending one interrupt per pixel. Then, Microblaze#1 sends the processed pixel to the corresponding RR. This mechanism is very slow, and the total execution of the application takes roughly 2 min per image. We have verified that the main cause for the long execution time is indeed the pixel fetching mechanism.

With the experiments above we demonstrated the capability of RTSM to control the execution of applications on hybrid systems with partially reconfigurable HW-PE and SW-PE. Due to the simple nature of the current application, features like reservation or relocation were not shown. We evaluated the RTSM performance by measuring the time intervals with timestamps in the MicroBlaze code. The system was

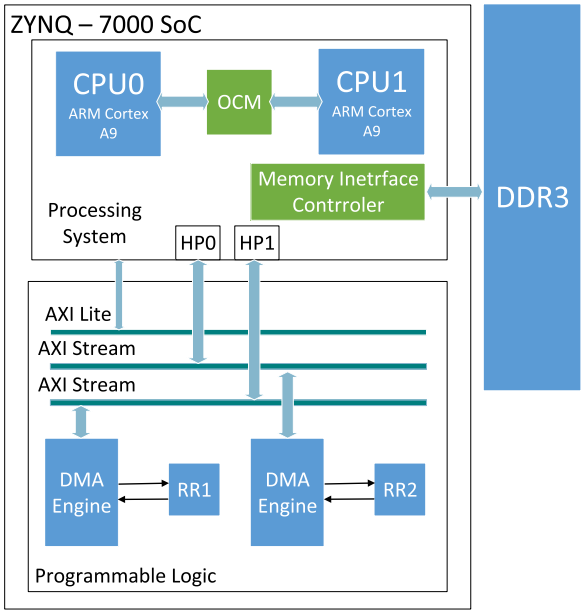

Fig. 9. System architecture of the Zynq platform.

clocked at 100 MHz. The average time of the schedule function was measured equal to 4.7 ms, which compared to the total execution time produces negligible overhead.

To the best of our knowledge, it is first time that a run-time manager targeting SW/HW systems with PR FPGAs is designed and implemented in a way that respects all current technology restrictions, and at the same time aims at minimizing the total execution time.

5.2. The RTSM on the ZedBoard platform

After the first implementation we transferred the design to the ZedBoard platform. One of our main concerns was to perform the execution of the application faster, and evaluate again the overhead of our scheduler. In order to do that we focused on the data fetching mechanism and devised it, as it was the main source of the delay in the execution at the XUPV5 platform. In the new platform we had as basis the AXI stream bus and passed the processed data with DMA operations. Fig. 9 illustrates the system architecture. It features two ARM Cortex-A9 cores, two reconfigurable regions acting as the HW-PEs each of which is connected to a DMA engine, and a DDR3 memory. The RTSM resides in CPU0, while CPU1 acts as the SW-PE executing SW tasks.

During start-up, the system is initialized by the on-board flash memory; the boot loader initializes CPU0 with the RTSM code, sets-up CPU1 as the Processing Element (PE), and loads the initial bitstream in the programmable logic. Then, the RTSM loads the application description, i.e. task graph, task information, task mappings, etc., from the SD card. It also transfers the partial bitstreams from the SD to the main system memory (DDR3). During normal operation the RTSM takes scheduling decisions and issues tasks, on the SW-PE, i.e. CPU1, and the two HW-PE, i.e. RR1 and RR2.

Our target application is again the edge detection that consists mainly of four filter kernels executing in a sequence. For each task we have implemented both a HW version (as partial bitstream) and a SW version. It is upon the RTSM to decide which version to use based on the run-time availability of HW- and SW-PEs. The input image is loaded from the SD card, while intermediate images resulting after processing each task and the final output image, are also written back to SD card. The transfer from and to the SD card is carried out by SW tasks executed in CPU1, which also controls partial reconfiguration of HW tasks.

The two CPUs communicate with each other through the on-chip shared memory using two memory locations, one for each communication direction, using a simple 2-flag handshake protocol (set and acknowledge/clear). One flag shows the PE status where a "−1″ indicates that the PE is idle and the RTSM can issue any SW task to it; when the PE is busy, this flag indicates the task assigned, e.g. "2" for SW_imageRead, "3" for SW_imageWrite, etc. Upon the completion of task execution, the PE notifies the RTSM by using the other flag, which indicates the type of the completed task.

In the system architecture of Fig. 9, the two RRs (RR1 and RR2) are connected onto the processing ARM cores through an AXI_Lite bus running at 75 MHz, and through a DMA engine, each one having read and write channels on a dedicated AXI_Stream running at 150 MHz. The AXI_Stream is connected to the processor High Performance ports (HP0 and HP1) that provide access to DDR memory. In order to execute a HW task, the RTSM issues a reconfiguration command to CPU1, which in turn configures the FPGA with the corresponding bitstream through the PCAP configuration port. Once partial reconfiguration completes, RTSM initiates HW task execution, programs the appropriate values, and triggers the corresponding DMA engines and kernel filters. All HW mappings of the kernels were implemented with the Xilinx HLS tool that also creates automatically the SW drivers for the SW/HW communication over the AXI_Lite bus. When a kernel completes execution, it generates an interrupt to the RTSM, which then updates its structures and proceeds to a new scheduling decision.

The RTSM operation can be broken down into different phases shown in Table 2. The table lists the total time spent on an ARM A9 Cortex CPU for each distinct phase, both in clock cycles and μs. The times presented in Table 2 refer to the cumulative time spent in each phase and not the average time for each phase.

The *RTSM initialization* phase is performed only once and includes: (i) fetching of the initialization file from the SD card that describes the tasks, the control flow graph, and the task mappings; (ii) parsing of initialization file; and (iii) initialization of RTSM data structures. The time for the RTSM initialization phase does not include the overheads for loading the file from SD card and for transferring the partial bitstreams to DDR3 memory. The *Schedule phase* refers to the time needed to execute the Schedule function once, in order to take a decision about the task to be executed next, i.e. when and where this task is going to be executed. The *Issue Execution phase* refers to the time required to issue either a reconfiguration or execution task instruction, depending on what the scheduling decision was. Also, depending on whether a HW core is reused, the RTSM checks if configuration prefetching can be performed. The *HW Task completion phase and SW Task completion* phase refers to the phases of updating the RTSM data structures after the completion of a HW and SW task respectively, and to resolve the dependencies in order to set the next task in the graph as "arrived". Finally, the *Reconfiguration task completion and HW task execution issue phase* refers to the interval in which the RTSM receives a reconfiguration completion notification from the PE, issues a task execution command, and checks whether it can perform configuration prefetching of a not yet "arrived" task.

Table 2
RTSM phases and time overhead per phase throughout the execution of edge detection on ZedBoard.

RTSM phases	# Clock cycles	Elapsed time (μs)
RTSM initialization	7707	23.121
Schedule (total of 9 times)	17,346	52.038
Issue execution	5995	17.985
HW Task completion	2493	7.479
Reconfiguration task completion & Hardware task execution issue	1224	3.672
SW Task completion	2748	8.244

Table 3
Execution time of the application and overhead of the RTSM and the reconfiguration process.

Application phases	Elapsed time (ms)
Edge Detection application	129.62
RTSM overhead	0.112
Reconfiguration overhead	5

In Table 3 we report the total RTSM overhead, the reconfiguration overhead, and the execution time of the application itself. In all cases we report the average time of multiple runs. It is obtained that the RTSM overhead is small as compared to the application execution time. Furthermore, the reconfiguration overhead is higher than anticipated due to the fact that we use software routines to perform reconfiguration, and thus we do not take advantage of the maximum PCAP throughput. The theoretical reconfiguration overhead, given that PCAP has a throughput of 400 MB/s, is 0.6 ms. Since we use a software to perform reconfiguration, the throughput is considerably lower at 50 MB/s, which increases the reconfiguration cost to 5 ms. Still, compared to the total execution time, this overhead is negligible. It is important to note that the reconfiguration overhead is added only once; in the rest cases of our case study, configuration prefetching takes place, thus hiding the reconfiguration overhead with HW execution.

Finally, we compare the performance of our system with the one presented in [21]. That work used the same application running on a Xilinx Virtex-5 FPGA and reported a throughput of 18 fps for a 640×480 image. In our case, we used a 1920×1080 image, and we measured a throughput of 7 fps. By converting the two results into pixels per second throughput, our system is faster by a factor of 2.6. Since the two platforms are quite different, a direct comparison is not easy; for example in the Zynq we use the ARM hard processors while the Virtex-5 FPGA supports only soft-core MicroBlaze processors.

Regarding portability, our initial RTSM was developed on an Intel x86 ISA desktop; porting it to the ARM architecture required only: (i) cross-compiling the code and (ii) re-implementation of architecture specific drivers and communication protocols between the RTSM and the Processing Elements.

5.3. Combining complex task graphs with real time measurements

Having ported our RTSM in two different platforms we validated its ability to successfully manage and schedule tasks efficiently, despite several technology restrictions. However our benchmark application (Edge Detection) is quite simple in structure, and it is very I/O intensive reading and writing its intermediate results on the SD card.

In order to obtain a better picture of the overheads incurred in these types of embedded systems we conducted experiments with a benchmark with a more complex task graph, similar to the one described in Section 4. The task graph shown in Fig. 10 was created by a random task graph generator. As we do not have a fully working executable of this benchmark we combine the actual times measured during the execution of the Edge Detection application's tasks on the ZedBoard platform for steps such as configuration, task execution and data transfer, and combine with the measure times for the RTSM scheduling.

The choice to use the execution times measured during the ZedBoard experiments was motivated by the fact that in this platform the tasks are executed by streaming input data to the accelerator. This is in contrast to the slow task execution times recorded in the Virtex-5 experiments.

This resulted in faster execution times that could lead to a better showcase of the reconfiguration and RTSM overheads.

Each task represented in the graph has been assigned the execution and reconfiguration time of an actual task from the Edge Detection

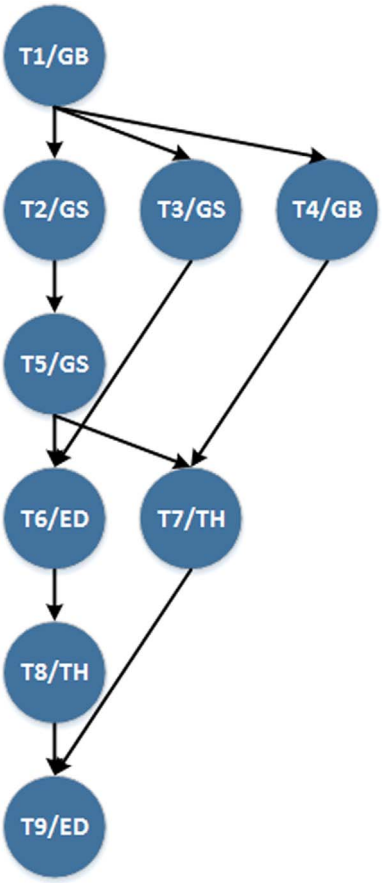

Fig. 10. Application task-graph annotated with the actual Edge Detection tasks assigned to each synthetic task.

application. Tasks 2, 3 and 5 are considered as Grayscale tasks, 1 and 4 as Gaussian Blur, 6 and 9 as Edge Detection and finally 7 and 8 as Threshold.

The hardware/software execution times for each Edge Detection task are shown in Table 4. For the RTSM's overheads we used the ones in Table 2. The reconfiguration time per task is measured at an average of 2.1 ms per task.

The resulting scheduling outcome is visualized in Fig. 11. The figure annotates all the prefetches and reservations made. It is important to note that all reconfigurations (except the first one) impose no overhead, as they are performed in parallel with task execution. Also, even though several prefetches might not be considered useful e.g. prefetch on Task 3, they are not considered as overehead, since the position in the RTSM code that they are being decided does not create an overhead in the overall execution.

Specifically if the decision of the aforementioned prefetches performed is being decided prior to the finishing of the task currently executed on the PRR and thus the RTSM overhead for those lines of code is masked. Also we can observe that due to the *Best Fit in Time* mechanism the RTSM uses for its advantage the fact that several tasks are inherently the same, e.g. Tasks 6 and 9.

The total overhead induced by the scheduling function can be broken down to a number of phases that cannot be executed in parallel with a hardware task. These phases are the initialization phase, the scheduling decisions for tasks 2, 3, 6 and 9, the initiation of the execution for reservations and the prefetching of task 7. Finally to these phases that induce overhead the list updates must also be considered due to their inability to be masked by task execution.

Using the average measured times for these phases from Table 2, we can calculate the total overhead of the RTSM, which sums up to 68.3352 μs. Compared to a total runtime of 200 ms for our synthetic application, we see that the reconfiguration overhead and the RTSM overhead are considered negligible, even with larger and faster applications than the ones used in the previous sections.

Table 4
Hardware and software execution times for each Edge Detection task.

Task	HW execution time (ms)	SW execution time (ms)
Grayscale	32.40	435
Gaussian Blur	32.42	452
Edge Detection	32.42	385
Threshold	32.35	225

5.4. Software scheduling efficiency of our RTSM

Our RTSM, in addition to the HW task scheduling capabilities, also schedules the application's possibly parallel SW tasks. This is a mandatory feature, as current processors are multi-core and software is increasingly parallel/multithreaded. Many optimized runtime systems exist for executing parallel software, and we set to compare the efficiency of our runtime system with that of such a state-of-the-art runtime system.

To do this comparison, we ported out runtime system to an Intel x86 based system, and we considered every available core of our computer as a SW-PE. Also every task can be mapped to all SW-PEs, since there are no limitations to which software core a function can be executed on. It is important to note that in this implementation the RTSM cannot take advantage of all the available cores, due to the fact that the RTSM is running on a dedicated core which is therefore not available as a SW-PE.

We compared our scheduler with OpenMP, a well-known industry framework that supports parallelism in multi-core systems and provides an easy way to write a parallel application. Our intention was to measure the scheduling quality and overhead of our RTSM, and compared it with that of OpenMP, as a typical parallel execution runtime.

With minor changes (annotations) on the task execution initiation code the user can easily modify the RTSM in order to execute tasks in a multi-core system, by binding the execution of task on certain cores. Furthermore, without loss of generality, we assumed that a segment of an application parallelized with an OpenMP pragma is identical to an independent task of the same application during the partitioning phase for our runtime.

For benchmark, we used was the same Edge Detection application with some minor changes: first we eliminated the intermediate Write Image tasks and then we split the processed image into 4 and 8 parts, increasing the parallelism and the workload of the application. With these modifications, each Edge Detection phase runs 4 or 8 times respectively before the RTSM considers the task completed.

The experiments were run on a desktop with an Intel Xeon Processor E5 at 2.2 GHz, with 12 cores in total. We used 2 images of different input sizes, 512×512 and 1024×512, in order to see how the execution time changes according to the data. There were 4 available cores/SW-PE for both our RTSM and the OpenMP. We applied all the available compiler optimizations (O1, O2 and O3) on both implementations, in order to see which one achieves the best execution time.

In these experiments we collected measurements: (i) to compare our RTSM with the OpenMP API by measuring and comparing the execution time of the entire application, and (ii) to quantify the scheduling overhead and analyze the different phases of our RTSM.

Comparing our RTSM to OpenMP, we found that a simple approach of writing and running an OpenMP parallel program with no optimizations is actually less effective than using our RTSM. The reason is that in OpenMP the OS scheduler may cause many involuntary context switches increasing the application execution time. On the other hand our scheduler created tasks and bound them to SW-PEs until the end of

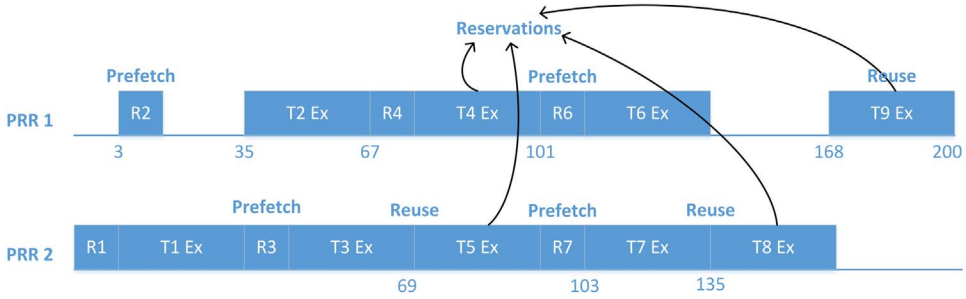

Fig. 11. The scheduling outcome for the task graph of Fig. 10 with the times measured in our ZedBoard experiments.

their execution using the set_affinity() function, so no context switches were introduced.

The different approach of the two implementations regarding thread affinity resulted in at most 33x speed-up of our RTSM over the OpenMP version. Upon closer inspection of the OpenMP API, we found a flag option that binds the threads created by OpenMP to certain processors, thus eliminating involuntary context switches. We applied the affinity optimization before the execution of the OpenMP program with the flag **export KMP_AFFINITY=compact** and then we measured again the overall execution time.

The results produced for both images, the 4 and 8 image splits and the 3 versions of our application (OpenMP, (opt.) RTSM. OpenMP) are shown in Figs. 12a, b and 13a, b. The measurements made for the

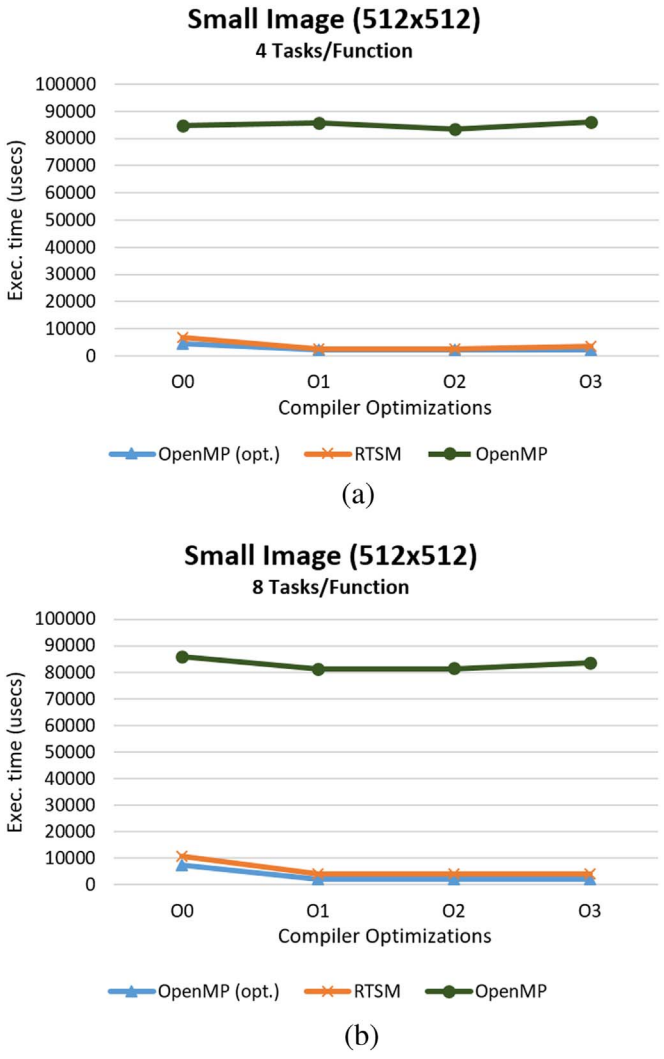

Fig. 12. Execution times of the Edge Detection application for the small image.

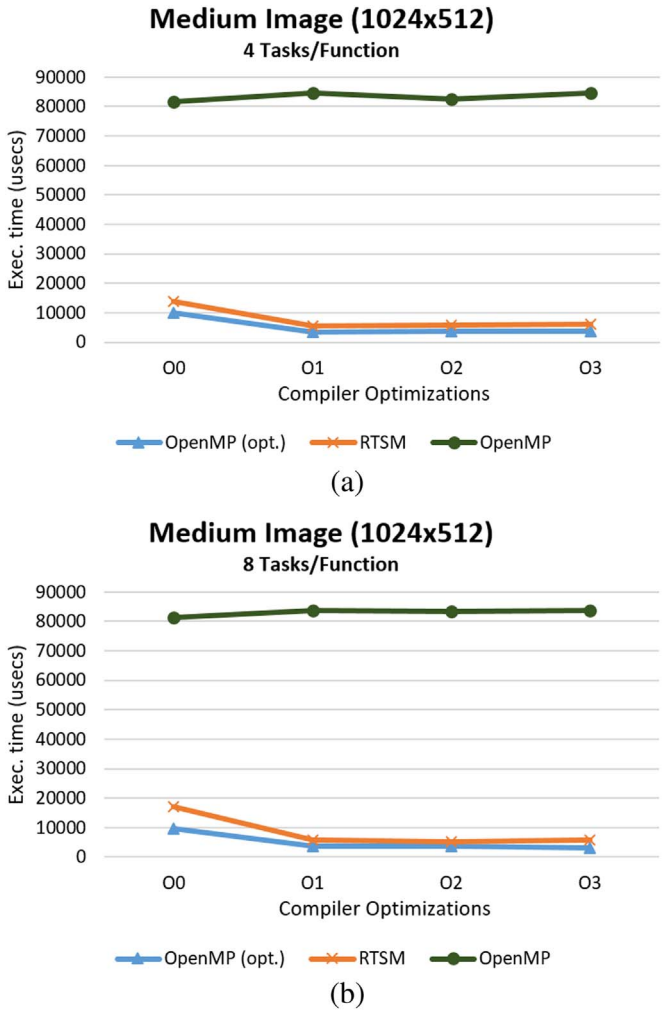

Fig. 13. Execution times of the Edge Detection application for the medium image.

comparison of the three versions are mean times after 1000 executions. As we can observe both the optimized OpenMP and the RTSM version outperform the simple OpenMP, with speed-ups ranging from 5x to 33x. Also the difference between the optimized OpenMP and our RTSM is negligible. Specifically the optimized OpenMP API over our RTSM achieves a 1.1–2.4x speed-up. After this analysis several conclusions can be drawn:

- The OpenMP API by default does not produce optimized versions, even of simple programs.
- The OpenMP KMP affinity optimization, which is used to prevent the involuntary context switches, can offer a speed-up up to 40x on the same application.
- Our RTSM is competitive to the optimized version of the OpenMP.

- If the affinity optimization is not applied, the compiler optimizations do not offer any speed-up to the OpenMP version.

Our RTSM is able to schedule task based applications, while OpenMP performs inner-function parallelism. Considering that the RTSM had an early disadvantage against OpenMP. However the optimized OpenMP version was not considerably slower than our approach and the optimized version offered at most a 3x speed-up.

Our RTSM was created with portability in mind, and hence it is not fully optimized, nor was created to be run on SW-only environments; instead its main goal was to offer advanced scheduling functionality for hardware functions. Despite this, our experiments show that, even for SW-only environments, it is comparable to the state-of-the-art OpenMP API.

6. Conclusion

We presented a run-time system to efficiently schedule HW and SW tasks in systems with partially reconfigurable FPGAs. Furthermore we showed that our RTSM and scheduling policies can be applied to a desktop environment and compare against the OpenMP API and produce promising and competitive results. Also the version of the RTSM compared was not particularly tailored for software task scheduling, thus leaving space for further development and improvement.

In addition we showed two more experimental frameworks we have created using PR FPGA devices, a XUP-V5 and a Zynq ZedBoard, demonstrating the inherent general use of our RTSM and its ability to be ported in different devices. The PR platforms used throughout this work follow a similar partial reconfiguration model, with an ICAP controlling the reconfigurable regions and an ARM/MicroBlaze microprocessor initiating task execution and reconfiguration. This model has not yet been improved and we believe that our RTSM can be easily ported even in recent platforms like the Ultrascale+. Also the number of RRs is user defined and not bound by the capabilities of our RTSM thus increasing the complexity of the PR design.

Finally we plan to develop complex use-cases, e.g. task graphs with branches and loops that will allow for demonstrating and evaluating all the features of RTSM on actual FPGA platforms. One obstacle to this effort (by us and other researchers) is the lack of standard interface across different applications, and designers have to manually intervene to adjust the RTSM and the task application interfaces, according to the specifics of each platform. Our RTSM relieves the designer from the task invocation complexities; he/she only has to address the task data interface with the core part of the RTSM. Also we plan to have our RTSM consider as a schedulable resource the communication aspect of the tasks.

Acknowledgments

This work was supported by the European Commission in the context of FP7 FASTER project (#287804) and H2020 EXTRA project (#671653).

References

[1] J. Burns, A. Donlin, J. Hogg, S. Singh, M. D. Wit, A dynamic reconfiguration run-time system, in: The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1997. Proceedings, 1997, pp. 66–75 http://dx.doi.org/10.1109/FPGA.1997.624606.

[2] A. Jara-Berrocal, A. Gordon-Ross, Hardware module reuse and runtime assembly for dynamic management of reconfigurable resources, in: 2011 International Conference on Field-Programmable Technology (FPT), 2011, pp. 1–6 http://dx.doi.org/10.1109/FPT.2011.6132721.

[3] P. Lysaght, B. Blodget, J. Mason, J. Young, B. Bridgford, Invited paper: Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas, in: 2006 International Conference on Field Programmable Logic and Applications, 2006, pp. 1–6 http://dx.doi.org/10.1109/FPL.2006.311188.

[4] G. Charitopoulos, I. Koidis, K. Papadimitriou, D. Pnevmatikatos, Hardware Task Scheduling for Partially Reconfigurable FPGAs, Springer International Publishing, Cham, 2015, pp. 487–498 http://dx.doi.org/10.1007/978-3-319-16214-0_45.

[5] E. El-Araby, I. Gonzalez, T. El-Ghazawi, Exploiting partial runtime reconfiguration for high-performance reconfigurable computing, ACM Trans. Reconfigurable Technol. Syst. 1 (4) (2009) 21:1–21:23 URL http://doi.acm.org/10.1145/1462586.1462590.

[6] C. Steiger, H. Walder, M. Platzner, Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks, IEEE Trans. Comput. 53 (2004) 1393–1407. http://dx.doi.org/10.1109/TC.2004.99.

[7] T. Marconi, Y. Lu, K. Bertels, G. Gaydadjiev, 3D Compaction: A Novel Blocking-Aware Algorithm for Online Hardware Task Scheduling and Placement on 2D Partially Reconfigurable Devices, Springer, Berlin, Heidelberg, 2010, pp. 194–206. http://dx.doi.org/10.1007/978-3-642-12133-3_19.

[8] Y. Lu, T. Marconi, K. Bertels, G. Gaydadjiev, A communication aware online task scheduling algorithm for fpga-based partially reconfigurable systems, in: 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2010, pp. 65–68 http://dx.doi.org/10.1109/FCCM.2010.18.

[9] K. Bazargan, R. Kastner, M. Sarrafzadeh, Fast template placement for reconfigurable computing systems, IEEE Des. Test. Comput. 17 (1) (2000) 68–83. http://dx.doi.org/10.1109/54.825678.

[10] K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck, Configuration relocation and defragmentation for run-time reconfigurable computing, IEEE Trans. Very Large Scale Integr. Syst. 10 (3) (2002) 209–220. http://dx.doi.org/10.1109/TVLSI.2002.1043324.

[11] T. Becker, W. Luk, P. Y. K. Cheung, Enhancing relocatability of partial bitstreams for run-time reconfiguration, in: 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007), 2007, pp. 35–44 http://dx.doi.org/10.1109/FCCM.2007.51.

[12] M. Huang, H. Simmler, O. Serres, T. El-Ghazawi, Rdms: A hardware task scheduling algorithm for reconfigurable computing, in: IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009, 2009, pp. 1–8 http://dx.doi.org/10.1109/IPDPS.2009.5161223.

[13] A. Montone, M.D. Santambrogio, D. Sciuto, S.O. Memik, Placement and floorplanning in dynamically reconfigurable fpgas, ACM Trans. Reconfigurable Technol. Syst. 3 (4) (2010) 24:1–24:34. http://dx.doi.org/10.1145/1862648.1862654.

[14] D. Gohringer, M. Hubner, E.N. Zeutebouo, J. Becker, Operating system for runtime reconfigurable multiprocessor systems, Int. J. Reconfigurable Comput. 2011 (2011) 3:1–3:16. http://dx.doi.org/10.1155/2011/121353 http://dx.doi.org/10.1155/2011/121353.

[15] D. Gohringer, S. Werner, M. Hubner, J. Becker, Rampsocvm: Runtime support and hardware virtualization for a runtime adaptive mpsoc, in: 2011 21st International Conference on Field Programmable Logic and Applications, 2011, pp. 181–184 http://dx.doi.org/10.1109/FPL.2011.41.

[16] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, C. Plessl, Reconos: an operating system approach for reconfigurable computing, IEEE Micro 34 (1) (2014) 60–71. http://dx.doi.org/10.1109/MM.2013.110.

[17] A. Duran, E. Ayguade, R.M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, Ompss: a proposal for programming heterogeneous multi-core architectures, Parallel Process. Lett. 21 (02) (2011) 173–193. http://dx.doi.org/10.1142/S0129626411000151 (http://arXiv:10.1142/S0129626411000151arXiv:10.1142/S0129626411000151 URL ⟨http://www.worldscientific.com/doi/abs/10.1142/S0129626411000151⟩).

[18] L. Bauer, A. Grudnitsky, M. Shafique, J. Henkel, Pats: A performance aware task scheduler for runtime reconfigurable processors, in: 2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2012, pp. 208–215 http://dx.doi.org/10.1109/FCCM.2012.43.

[19] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, A. Wood, Openpr: an open-source partial-reconfiguration toolkit for xilinx fpgas, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011, pp. 228–235 http://dx.doi.org/10.1109/IPDPS.2011.146.

[20] C. Beckhoff, D. Koch, J. Torresen, Go ahead: a partial reconfiguration framework, in: 2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2012, pp. 37–44 http://dx.doi.org/10.1109/FCCM.2012.17.

[21] G. Durelli, C. Pilato, A. Cazzaniga, D. Sciuto, M. D. Santambrogio, Automatic run-time manager generation for reconfigurable mpsoc architectures, in: 2012 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012, pp. 1–8. http://dx.doi.org/10.1109/ReCoSoC.2012.6322883.

[22] Z. Li, S. Hauck, Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation, in: Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays, FPGA '02, ACM, New York, NY, USA, 2002, pp. 187–195. http://dx.doi.org/10.1145/503048.503076 URL ⟨http://doi.acm.org/10.1145/503048.503076⟩

[23] A. Morales-Villanueva, A. Gordon-Ross, On-chip context save and restore of hardware tasks on partially reconfigurable fpgas, in: 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2013, pp. 61–64 http://dx.doi.org/10.1109/FCCM.2013.13.

[24] C. Conger, A. Gordon-Ross, A.D. George, Design framework for partial run-time fpga reconfiguration, in: ERSA, 2008, pp. 122–128.

[25] K. Vipin, S.A. Fahmy, Architecture-Aware Reconfiguration-Centric Floorplanning for Partial Reconfiguration, Springer, Berlin, Heidelberg, 2012, pp. 13–25. http://dx.doi.org/10.1007/978-3-642-28365-9_2.