Hardware Task Scheduling for Partially Reconfigurable FPGAs

George Charitopoulos[1,2], Iosif Koidis[1,2], Kyprianos Papadimitriou[1,2], and Dionisios Pnevmatikatos[1,2]
[1] Institute of Computer Science, Foundation for Research & Technology – Hellas
[2] School of ECE, Technical University of Crete
gcharitopoulos@isc.tuc.gr, {koidis, kpapadim, pnevmati}@ics.forth.gr

***Abstract*—Partial reconfiguration (PR) of FPGAs can 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, implement and validate a practical run time system manager (RTSM) for scheduling SW tasks on available processor(s) and HW tasks on any number of reconfigurable regions of a partially reconfigurable FPGA. Our approach is a practical one, in the sense that we take into account all the technology restrictions imposed by the FPGA vendor. The RTSM is fed with the initial partitioning of the application into tasks, the corresponding task graph, and the available task mappings, and reacts according to dynamic parameters such as the runtime status of each task and region, e.g. busy executing, idle, scheduled for reconfiguration/execution task, free, reconfigured, active region etc. The RTSM employs a task reuse policy to minimize reconfigurations, moves tasks among regions to manage efficiently the FPGA area, reserves tasks for future reconfiguration and execution, and supports configuration prefetching. To validate its correctness we use it to control an image processing application running on a ZedBoard platform. We evaluate its features with a simulation framework, and we find that despite the practical limitations, our approach can give promising results in terms of quality of scheduling.**

***Index Terms*—Run-Time System, Scheduling, FPGA, Partial Reconfiguration.**

I. Introduction

Reconfiguration can dynamically adapt the functionality of hardware systems by swapping in and out HW tasks. Partial reconfiguration in particular is attractive in contrast to full reconfiguration for its benefits in flexibility and speed. To coordinate the execution of HW tasks and select the proper resource for loading a HW task in systems with partially reconfigurable FPGAs, we need runtime system support [9]. The runtime system, undertakes decisions for triggering task reconfiguration and execution while managing the reconfigurable area. To this end, several scheduling algorithms of various complexities have been proposed [23].

In this work we propose a Run-Time System Manager (RTSM) incorporates scheduling mechanisms that balance effectively the loading of HW tasks and the use of physical resources. We aim to execute as fast as possible a given application, without exhausting the physical resources. Furthermore, the RTSM is cautious not to occupy resources unnecessarily if allocating them does not contribute to speeding-up the execution. The RTSM can also control the execution of SW tasks, taking them into account in the scheduling decisions.

Our motivation during the development of the RTSM was to find ways to overcome the strict technology restrictions imposed by the Xilinx PR flow [13]. Such restrictions are:

- Static partitioning of the reconfigurable surface of the device in reconfigurable regions (RR).
- Each reconfigurable region can accommodate certain hardware core(s) only, called reconfigurable modules (RM). This 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, thus a second RM cannot be configured into the same RR even if there is enough space - in terms of logic resources - for it. This is illustrated in Figure 1.

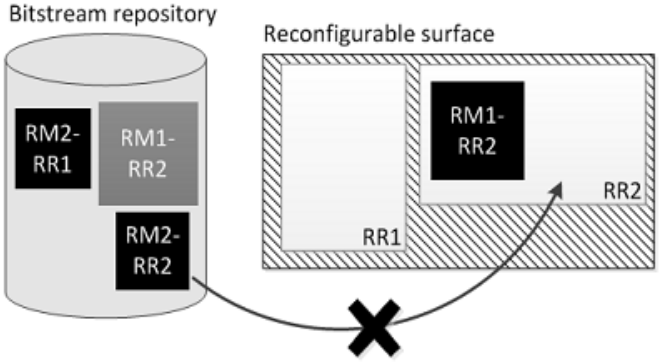


Figure 1. Despite that a partial bitstream of RM2 for RR2 exists, and RR2 has enough physical resources for accommodating it, this operation is prohibited to avoid erasing the entire RR2, thus displacing RM1. For the time being, RM2 can be placed only into the empty RR1.

To develop and test the RTSM we created a simulation framework that incorporates current technology restrictions. We validated the behavior of RTSM on a fully-functional system with a ZedBoard platform executing an edge detection application [11]. Our RTSM can run either on a linux x86 PC or on an embedded processor inside the FPGA, i.e. Microblaze, ARM etc. For the transition from the desktop implementation to the embedded one, no changes were needed in the RTSM core code.

To evaluate the performance of our RTSM we performed experiments in a simulation environment. The RTSM is rather generic and although we demonstrate its functionality on an FPGA of a certain vendor, it can be used to support PR FPGAs of other vendors. The main contributions of the this work are:

- An RTSM with portable functionality in its main core, capable to control HW and SW tasks in PR FPGA-based systems;
- dynamic execution of complex task graphs, with forks and joins, loops and branches;
- combination of different scheduling policies, such as relocation, reuse, configuration prefetching, reservation and Best Fit.

The remainder of the paper is organized as follows. Section II discusses previous works studying run-time systems for PR FPGAs. Section III presents the key concepts, and provides details on the input to feed the RTSM and on its operation. Section IV is devoted to performance evaluation in a simulation environment and validation on a real FPGA-based system. Finally, Section V summarizes the paper.

II. Related Work

Throughout the years several scheduling algorithms and Run Time Managers have been proposed. One of the first works on hardware task scheduling was done by C. Steiger et al [1]. In this work Steiger formulated the scheduling problem for the 1D and 2D area models, and proposed two heuristics, the Horizon and the Stuffing techniques.

Many scheduling algorithms for managing hardware tasks were proposed by Marconi et al. One of the first works presented by Marconi and Lu [2] - inspired by Steiger's [1] and Chen's and Hsiung's works [3] - create a scheduling and placement algorithm for partially reconfigurable devices.

In [4], Marconi et al. presented a novel 3D total contiguous surface heuristic in order to equip the scheduler with "blocking-awareness" capability. Subsequently, Lu and Marconi created the first scheduling algorithm that considers the data dependencies and communication amongst hardware tasks, and between tasks and external devices [5].

Also, in the field of scheduling algorithms Montone et al. present a different approach in hardware task placement and space management focusing on a resource- and configuration-aware floorplacement framework, using an objective function, based on external wirelength [8]. However this work is done before running the actual application.

In addition to efficient scheduling algorithms, it is equally important to create efficient placement algorithms. The fast and effective placement of hardware tasks on the device, in addition to efficient free space management, are very important attributes, not only for a good online placement algorithm, but also for a good scheduler.

A very important work in this field presented, by Bazargan et al, offered the first methods and heuristics for fast and effective online and offline placement of templates on reconfigurable computing systems [6]. Also fundamental in the field of task placement was [7], in which, inspired by the concept of task relocation, Compton et al. propose several techniques, most important of them being the ability to perform run-time partitioning and creation of a new RR on the device, that optimize the existing techniques. However, the proposed transformations are still beyond supported FPGA technology. Since Compton's studies, much work has been done towards efficient bitstream relocation [21].

Finally one of the most thorough works in the field of creating a complete run time manager has been shown in [11]. They introduce a run time manager (RTM) that is able to map multiple applications on the underlying hardware and execute them concurrently.

Burns et al. made one of the first efforts in the creation of an operating system (OS) inhabited in a partially reconfigurable device [9], where, based on three different applications, they extracted a set of common requirements, and designed a runtime system for managing the dynamic reconfiguration of FPGAs. [17] and [18] discuss a similar topic, that of efficient reconfiguration and execution of tasks in a multiprocessing system-on-chip, under the control of an OS. Along these lines, several studies have shown the advantages of using partial reconfiguration, in contrast with full reconfiguration, such as [10].

As listed above, many researchers have proposed and created scheduling algorithms, placement algorithms, for managing hardware tasks on a partially reconfigurable FPGA based operating system, and complete runtime systems inhabited in partially reconfigurable devices [14]. However, very few of them have been evaluated using an actual partially reconfigurable device, [11], [17].

What seems to be missing in the current state of the art is scheduling algorithms that take in consideration all the current technology restrictions. In [17] 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, as well as how the application would be executed is presented in a theoretical way. In [11] all restrictions are taken in consideration, however the mechanics of the scheduling algorithm are simple and the overhead considerable.

Finally it is important to note that nearly all of the works mentioned take advantage of the transformations proposed in [7], not yet available in the current state of the art, in order to create scheduling algorithms allowing them to bypass certain technology restrictions regarding the PR process.

III. The Run-Time System Manager

The RTSM manages physical resources employing scheduling and placing algorithms to select the appropriate HW Processing Element (PE), i.e. a Reconfigurable Region, for loading and executing each HW task. HW tasks are implemented as Reconfigurable Modules, stored in a bitstream repository. The RTSM can also activate software-processing elementt (SW-PE) to execute the SW version of a task, if any available.

A. Key Concepts

The RTSM handles hybrid applications consisting of both HW and SW tasks. The most challenging case concerns applications with partially reconfigurable tasks. Below we discuss the functionalities we considered in our RTSM:

(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) [12]. This limitation was discussed in [9], and was followed-up in later works [11].

(2) Task graph: The RTSM should be aware of the execution order of tasks and their dependencies; this is provided with a task graph, which essentially represents the application itself. For example, with representation "1,2" the RTSM understands that task 1 should first complete execution before marking task 2 as "arrived". The RTSM can also understand complex graphs with properties like forks and joins, branches and loops. The above form the basic guidelines to the RTSM according to which it takes runtime decisions.

(3) Multiple bitstreams per task: A HW task can have multiple mappings, i.e. several versions of the same task with each one implemented as a different RM. Although all versions produce the same functionality, each one can target a different RR, and/or can be synthesized with different characteristics, e.g. in terms of number of pipeline stages, power consumption, performance, etc. A similar approach is described in [9], and accounts for the flexibility of the scheduling policy to potentially increase its quality [11].

(4) Reservation list: When a task cannot be loaded immediately due to unavailability or resources, it is reserved in a queue for later configuration/execution. The authors of [1] proposed keeping a task in a queue only if its deadline would be met, otherwise it is rejected and removed from the queue. Our RTSM keeps a HW task in the queue until it is configured successfully into an RR or assigned to the SW-PE.

(5) Reuse policy: Before loading a HW task to the FPGA, we check whether it already resides in any RR. This prevents reloading the task, thus reducing reconfiguration overhead. If the already configured HW task cannot be used, e.g. it is busy running on other data or has been scheduled already for execution, it might be necessary to load a bitstream for this HW task to another RR (given that such binding exists).

(6) Best Fit in Space (BFS): This algorithm prevents the RTSM from injecting small HW tasks into large RRs, as this would leave many logic resources unused, even if the corresponding RM-RR binding exists. This aims at minimizing the area overhead incurred by unused logic into a used RR, pointing to similar directions with studies on floorplanning for sizing efficiently the regions and the respective reconfigurable modules [19]. Another interesting work at the compile-time side examines the impact of the module size on the internal fragmentation and finds that it can lead to large overhead [16].

(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, results to better ending time. This can happen due to the reuse policy: such case occurs when HW tasks are called more than once, such as in loops and branches. For example, consider a HW task that is to be scheduled, which already exists in an RR due to a previous request. Scheduling decision relies on which action (reservation, immediate placement and relocation) will result in the earliest ending time of the task. For instance, BFT might invoke reconfiguration of a HW task into a new RR, even though this HW task already resided in another RR (but it is busy executing or has been already scheduled for execution). BFT decisions rely on static parameters, i.e. initial partitioning, task mappings, reconfiguration and execution time, and on dynamic parameters, i.e. status of regions and tasks.

(8) Joint Hardware Modules (JHM): It is possible to implement at least two HW tasks in the same bitstream, thus allowing more than one tasks fetched into the same RR. JHM, illustrated in Figure 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 [22].

(9) Configuration prefetching: This technique allows loading a HW task into an RR, ahead of time [20]. It is activated under two conditions; first, if the reuse policy was invoked and thus the Internal Configuration Access Port (ICAP) is not used, and second, if a task has just been configured and there is no other task waiting to be configured.

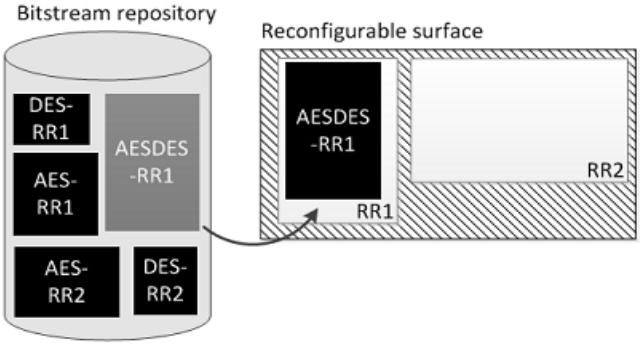


Figure 2. Each crypto module can be loaded into any RR, however a large amount of resources will be under-utilized. JHM utilizes more efficiently the RR area, given that the corresponding bitstream exists.

(10) Relocation: A HW task residing in an RR can be "moved" by loading a new bitstream implementing the same functionality to a new RR. This is illustrated in Figure 3. Two RMs are being scheduled for configuration into two RRs. RM1 is already configured in RR2. Also, RM2 should execute, so it is waiting to be configured, but its RR is not available. The proposed relocation mechanism moves first 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 [21]. To fully exploit the benefits of this approach context save techniques are needed [15].

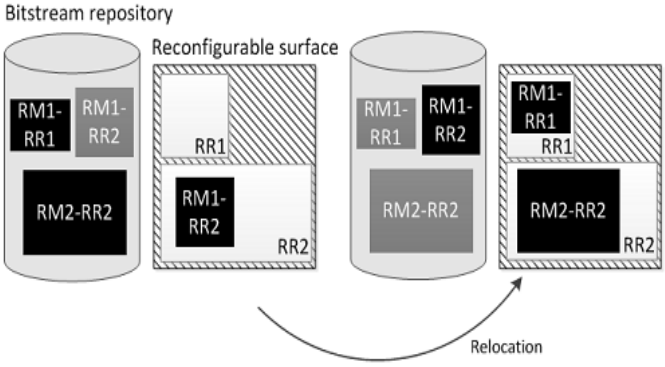


Figure 3. RM2-RR1 does not exist, thus the hardware task laying in RR2 is relocated by first configuring RM1-RR1, and then RM2-RR2.

The above features are incorporated in our RTSM, and have been tested within a simulation framework presented in the following Section. The combination of Best Fit in Time 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. In fact this is application-dependent, i.e. depends on whether the application has such characteristics that will allow for speeding-up its execution when using the above feature. To this direction, we believe it is important to study the extent to which complex techniques and features contribute are required needed to serve efficiently different kind of applications.

B. RTSM Input and Execution Flow

The RTSM gets as input the partitioning of the FPGA; the tasks to be scheduled; the task graph representation carrying the task dependencies; and the different task mappings, i.e. different bitstreams corresponding to different implementations of the same hardware task. Also information regarding the software tasks (if any) can also be available. We do not consider multiple SW-PEs; instead, if a software task is available, once the RTSM calls it, it is executed in the CPU. Additionally, the RTSM requires as input the execution and reconfiguration times for each task, and optionally the task deadlines. Based on this information the RTSM performs the initial scheduling.

The above can be provided in a file -given a file system exists- 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 define upon which, a task is characterized "arrived" or "newly arrived" as:

Definition 1: If a task has completed its execution at time $t=x$, then the descendant task, as retrieved from the task graph has an arrival time $t_{arr}=x+1$.

We tested the RTSM within a simulation framework. 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 4x5, 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.

At each point of time, the RTSM checks for a newly arrived task, and calls the schedule function. If the task can be served immediately, the RTSM checks if it is already configured so as to reuse it, and 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.

The main function of RTSM is the schedule function. In order to reach a scheduling decision for a task, the RTSM follows a complex process, employing a variety of functions and policies. The RTSM first creates a list of the available mappings for the newly arrived task. If the task has a mapping to a particular RR, the latter is added to the RR list for this task. If this list contains more than one RR, 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 no suitable RR could be found after applying the *Best Fit* policy, the scheduler performs *Relocation*. 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 is also unsuccessful, the scheduler will attempt 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 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, e.g. HWICAP, as a resource that must be scheduled.

C. Tasks with Deadlines

For applications with task deadlines, the RTSM can consider them prior to taking scheduling and placement decisions. If no alternative -either via relocation or reservation- can meet the deadline, the task is either rejected or executed in a software processing element. Here, we should note that the proposed relocation moves a HW task only if this still allows for its deadline to be met, i.e. relocation of a HW task is prevented if a deadline is violated.

This feature is disabled in our current work, as we focus on applications that require all tasks to be served without deadlines. This is done because the application we implemented could be run in a streaming fashion, thus rendering the deadline feature useless.

D. Additional Observations

Task reconfiguration and execution times: Static inputs derived through profiling. Profiling is done by the end user and can refer to theoretical reconfiguration times calculated before the execution or an estimated calculation of the execution during the design phase of the application.

They are given to the RTSM during its initialization to perform the initial task scheduling. In the real use-case a task might not complete within the predefined execution time, thus the RTSM should be signaled once a task completes execution. Therefore, the RTSM should be able to adjust its scheduling decisions dynamically based on the new status of the system.

BFS: The BFS opts for placing a newly arrived task (implemented as bitstream) on the RR producing the smallest unused area. Without BFS, the size of an RM does not pose any restriction for loading it into a RR, given that such a

bitstream exists. The programmer can disable this feature; however in all our experiments it is enabled.

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

IV. Experimental Testbed and Results

We studied the RTSM within a simulation framework, and then tested its correctness by controlling an edge detection application developed on a Xilinx ZYNQ™-7000 SOC, 512 MB DDR3 and an SD port for input/output.

A. Simulation Framework

To demonstrate all concepts described in Section IV we feed the RTSM with a synthetic workload. The RTSM gets as input the task graph, parameters for each task, the available RRs, SW implementations of the tasks, if any present, and the task mappings.

Figure 4 shows the task graph that the RTSM will manage. Also in the same 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 consist of two RRs and one SW-PE, as well as the FPGA configuration port, which is also treated as a resource to be scheduled. Also, the RTSM accepts as input the width and height of each RR; these are mainly used by the *Best Fit in Space* function.

Table I has the available task mappings. These drive the options of RTSM for making the best scheduling decision for a given task, e.g. T2 can be loaded only in RR2. If a task has only one RR-RM binding, options are limited.

In the experiment performed we assume that every task has a software implementation, in order to study how the RTSM exploits both hardware and software resources residing on an FPGA, always to the advantage of the overall application execution time. It is important to note that the software implementation of a task has a much bigger execution time than the hardware one.

The scheduling result of the experiment presented is shown in Figure 5. In this experiment we observe the majority of the features supported by our RTSM. We observe that Relocation is activated to accommodate task T2. Since Best Fit in Space function has placed task T1 on RR2, in order to place task T2 on the FPGA we have to first relocation and reconfiguration of task T1 on RR1 and then reconfigure T2 to the now empty RR2.

Additionally, the decision of executing task T5 on the SWPE is due to the Best Fit in Time function as a later reservation of task T5 on a RR gave a later ending time.

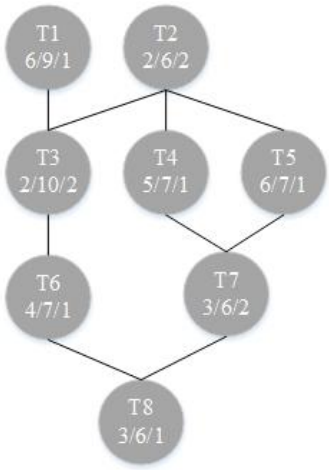
Also we can see the use of the Reservation on the decision taken for task T3. The arrival time of task T3 is on t=10 and since the only available bitstream binds task T3 to RR2, there is no other option but to reserve task T3 for later execution on RR2. It is worth noting that the scheduler does not relocate task T4 to RR1, because T4 is near completing its execution (otherwise it would restart its execution).

Finally, there is a high level of inner task parallelism between tasks of the same but also from different levels, task T1, T4, and T5. Finally we can see again the effective use of the reconfiguration prefetching on tasks T7 and T8.

TABLE I. RM-RR Bindings

Tasks	Mapping Characteristics		
	#RR	***Width***	***Height***
T1	1,2	1	2
T2	2	1	3
T3	2	1	3
T4	1,2	1	2
T5	1	2	2
T6	2	1	2
T7	1,2	1	2
T8	1,2	1	2

Figure 4. The task graph of the application. The numbers inside the circles



indicate HW Exec. Time/SW Exec. Time/Reconfiguration Time.

B. Discussion Analysis

In both experiments we use the same task graph but we consider different number and type of resources, and different task mappings per experiment. We stress the following observations:

- The chosen task graph is complex enough and demonstrates fork and join operations. In addition we assume that multiple bitstreams per task are available, which accounts for flexibility of RTSM's choices. Almost all features were demonstrated; relocation, reservation, prefetching, *BFT* and *BFS*. The reuse policy was not demonstrated due to that there is no task repeated, neither a loop exists in the task graph. Also we didn't demonstrate the use of JHM, due to that we didn't assume availability of such bitstreams
- In Figure 5, 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.

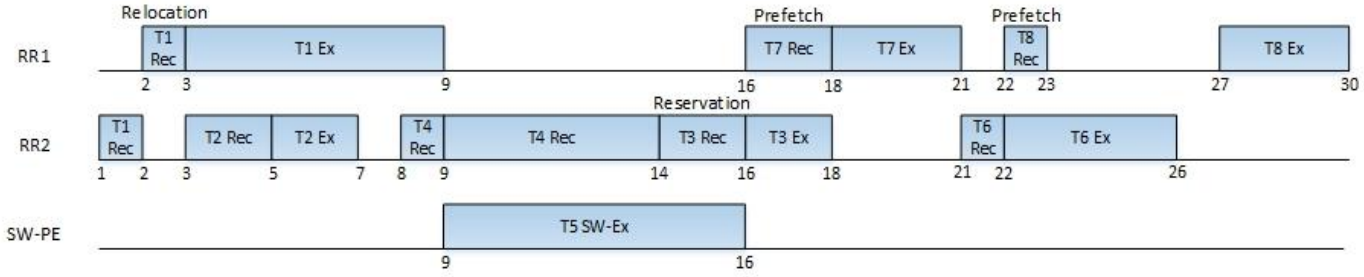


Figure 5. The scheduling outcome of the example is shown. The usage of prefetching, as well as, the relocation and reservation alternatives is shown.

Important Notes:
Hardware execution times compared to software: The decision to execute a task in the SW-PE will take 2-3 time units more than the combined hardware execution and reconfiguration time. With this we prompt the scheduler to choose the hardware accelerator to program in the FPGA.
Execution time compared to reconfiguration: The execution time is 2-3 times greater than the reconfiguration time, this way we try to simulate as better as possible the fast reconfiguration times compared to execution ones.

C. Validating the RTSM with a Real-world Application

In order for us to implement a complex environment that would allow us to demonstrate the RTSM with a real world application, we used the Zedboard platform, which utilizes a Xilinx ZYNQ™-7000 SOC, 512 MB DDR3 and an SD port for input/output.

At Figure 6 we can see schematic of the system architecture, which utilizes the two embedded ARM CortexA9 processors, two reconfigurable regions with one DMA engine each, and the DDR3 memory. At boot time the system is programed by the on-board flash memory, the first stage boot loader is executed, which initializes the CPU0 (ARM) processor with the RTSM object code, the CPU1 (ARM) with the process element object code (PE) and the PL with the provided bitstream.

During the system initialization process and after the initial programming of all the PEs and RRs, the initialization of the RTSM takes place by a file stored in the SD card which describes the tasks to be executed, the available mappings and the control flow graph. Also at this point all the available partial bitstreams are transferred from the SD card to the DDR3 memory.

After the initialization the RTSM starts making scheduling decisions and task issuing, using the available SW processing elements (CPU1 ARM processor), and HW processing elements (Reconfigurable Regions RR1 and RR2).

The implemented application is an image Edge Detection process that consists of loading the image from the SD executing grayscale, Gaussian blur, edge detection and finally threshold by storing the intermediate produced images, at every step back to the SD card.

All the loading, saving to the SD, and partial reconfiguration tasks are considered as SW tasks and are issued and executed by CPU1. Grayscale, Gaussian blur, edge detection and threshold have both HW and SW mappings and it's up to the RTSM which one is going to be executed based on its directives and the resources available at each time.

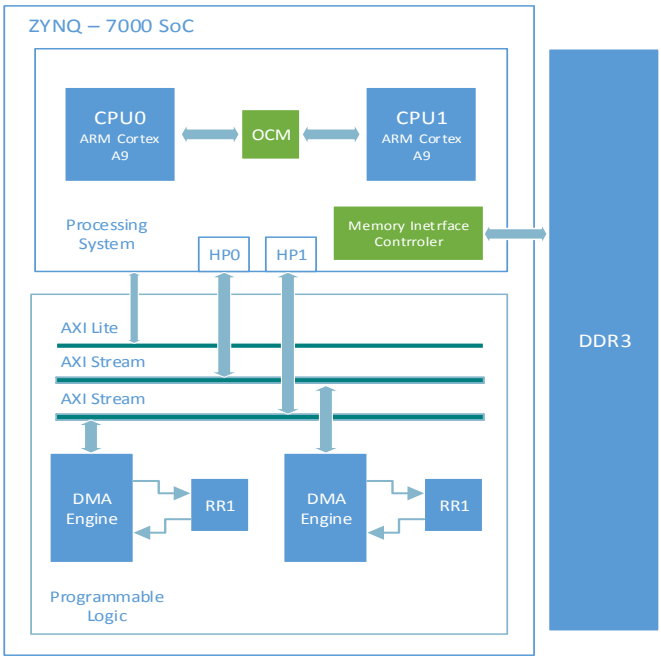


Figure 6. System Architecture Implementation

At Figure 7 we can see the task graph of the afore-mentioned application. For better understanding of the prefetching mechanism, the reconfiguration process is shown and treated as a separate task. It is important to note that this figure depicts only the task dependencies and not how the application will be executed.

The two SW processors communicate each other with the use of on chip memory. Two memory locations are utilized for inter-process communication: one that can only be set by the RTSM and only be acknowledged/cleared by the PE and the other can only be set by the PE and acknowledged/cleared by the RTSM.

More specifically when this flag value is -1 it means that the PE (CPU1) is idle and the RTSM (CPU0) is free to issue any SW task to it, and it does that by changing this flag to an appropriate number based on the issued task e.g. *2 for SW_imageRead, 3 for SW_imageWrite, 4 for SW_greyScale* e.t.c. After the PE finishes the requested task's execution it sends a response towards the RTSM from the other memory

location about the type of the task that ended, and sets again to -1 (clears) the afore-mentioned idle-operation requested flag.

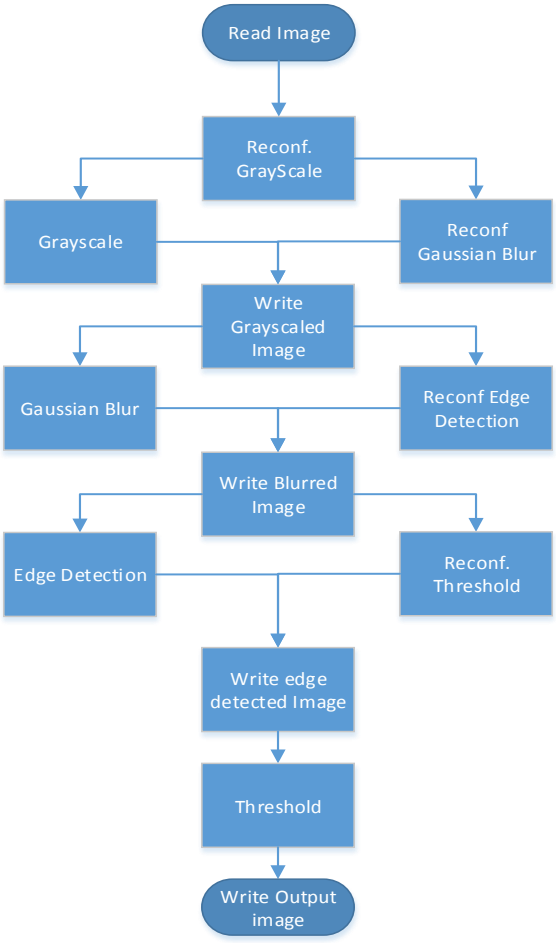


Figure 7. The Task Graph of the Edge Detection Application

The two RRs are connected to the processing system (PS) by AXI_Lite interface for control that runs at a speed of 75MHz and by one DMA engine each with read and write channels on a dedicated AXI_Stream bus that runs at 150 MHz. The AXI_Stream is then connected to the PS's High Performance ports, which give access to the DDR through the PS's Memory interface controller. For every HW task that has to be executed, the RTSM issues a reconfiguration command to CPU1 which programs the corresponding bitstream though the PCAP interface. After the partial reconfiguration completion, the RTSM issues the HW task execution, by programming the appropriate values and starting the corresponding DMA engines and FILTER. At this point we should point out that all the aforementioned HW mappings (FILTERS) were implemented by HLS tools which also create the SW drivers needed for SW-HW communication over the AXI lite bus. When any of the above Filter engines completes its execution, an interrupt is generated toward the RTSM, which then updates its data structures and proceeds with the application.

At Table II we can see the accumulated execution time for various key phases of the RTSM, in clock cycles and μs, both ARM CPUs run at 667 MHz.

TABLE II Various RTSM phases execution time

RTSM phases	Clock Cycles	Required time (μs)
RTSM Initialization	7,707	23.121023
Schedule time	17,346	52.038052
Issue Execution time	5,995	17.985018
HW Task completion time	2,493	7.479007
Reconfiguration task completion & Hardware task issue time	1,224	3.672004
SW Task completion time	2,748	8.244008

The RTSM initialization phase consists of the time needed for the RTSM to parse the initialization file from the SD, which describes the tasks to be executed, the available mappings and the control flow graph, and to initialize all its data structures. At this point we don't take into consideration the time needed for the file to be loaded from the SD card or the time needed for all the partial bitstreams to be loaded from the SD card to the DDR memory.

The *Schedule Time* refers to the time consumed by the RTSM to execute the *Schedule function*, i.e. to reach a decision about the task that is going to be executed next, meaning when and where this task is going to be executed.

Issue Execution time is the time needed by the RTSM in order to issue either a reconfiguration or execution task instruction, depending on what the scheduling decision was. Also depending on whether there has been core re-usage the RTSM checks if configuration prefetching can be performed.

HW Task completion time is the amount of time consumed by the RTSM in order to update its data structures after the completion of a hardware task and to resolve the dependencies in order to set the next task in the graph as "arrived". Similarly *SW Task completion time* is the amount of time spent, for the same reason, when a SW task completes its execution.

Reconfiguration task completion & HW task Execution Issue time refers to the interval in which the RTSM receives a reconfiguration termination command from the PE, issues a task execution command and checks whether it can perform configuration prefetching of a not yet "arrived" task.

After implementing and porting our RTSM with the target Edge Detection application we measured the overall application execution time at 129.62 milliseconds. The scheduling overhead was measured to be 0.112 milliseconds. The theoretical reconfiguration overhead, knowing that the PCAP has a throughput of 400MB/sec, is 0.6 milliseconds, which is taken into consideration once, since all the other times configuration prefetching is performed, during other task HW execution. However due to the fact that we employ a SW processor for performing reconfiguration we know that the throughput is considerably lower, <50MB/sec. Even when calculating the reconfiguration time with the considerably lower throughput the outcome is 5 milliseconds, which is small as compared to the total execution time of application, thus it adds a negligible overhead to it.

We compare our work with the results presented in [11]; with the same Edge Detection application the researchers present a throughput of 18 fps for a 640x480 image, the application was implemented on a Xilinx Virtex5-lx110t. The target image we used was 1920x1080 pixels and the throughput was measured at 7 fps. However by reducing the two results in pixels per seconds we measure a 2.6X speed-up, though we stress that the target platforms are very different, a factor that may well explain the speed difference.

To summarize the RTSM core code did not require any changes and customizations in order to be executed on the ARM processor. We only needed to cross-compile the source-code with the standard ARM C compiler. However we needed to add and implement architecture specific drivers and communication protocols between the RTSM and the various processing elements residing either on the PL or the PS. The target application is a rather simple one and thus features of our RTSM that are utilized by more complex applications, e.g. *Relocation Alternative*, *Reservation Alternative*, are not shown.

V. Conclusion and Future Work

We presented a run-time system combining different mechanisms to control HW and SW tasks in systems with partially reconfigurable FPGAs. In the future we will perform experiments using more complex task graphs (with branches and loops), and applications with task deadlines. Moreover, we will create experiments to demonstrate the reuse policy and the use of Joint Hardware Modules. Also, we are planning to design complex real use-cases that will demonstrate all the features of RTSM and asses them on an FPGA platform. One issue is that there is no standard interface across applications. The designer has to intervene manually to adjust the RTSM to the application needs by respecting some rules. The changes are not major ones and the designer should concentrate on a certain part of the RTSM code only. Next, we will be looking into ways for standardizing this interface.

Acknowledgement

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

Referrences

[1] C. Steiger, H. Walder, and M. Platzner (2004): “Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of RealTime Tasks”. In: IEEE Transactions on computers, Vol. 53, No. 11.

[2] T. Marconi, Y. Lu, K. Bertels, and G. Gaydadjiev: “Online Hardware Task Scheduling and Placement Algorithm on Partially Reconfigurable Devices”. In: Proceedings of International Workshop on Applied Reconfigurable Computing, Architectures, Tools and Applications (ARC 2008), March 26-28, 2008

[3] Chen, Y., Hsiung, P. (2005): “Hardware Task Scheduling and Placement in Operating Systems for Dynamically Reconfigurable SoC”. In: Yang, L.T., Amamiya, M., Liu, Z., Guo, M., Rammig, F.J. (eds.) EUC 2005. LNCS, vol. 3824, pp. 489–498. Springer, Heidelberg.

[4] T. Marconi, Y. Lu, K.L.M. Bertels, G. N. Gaydadjiev (2010): "3D Compaction: a Novel Blocking-aware Algorithm for Online Hardware Task Scheduling and Placement on 2D Partially Reconfigurable Devices". In: Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC), pp. 194-206, Bangkok, Thailand.

[5] Y. Lu, T. Marconi, K.L.M. Bertels, G. N. Gaydadjiev (2010): "A Communication Aware Online Task Scheduling Algorithm for FPGA-based Partially Reconfigurable Systems". In: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines.

[6] K. Bazargan, R. Kastner, and M. Sarrafzadeh (2000): “Fast Template Placement for Reconfigurable Computing Systems”. In: IEEE Design and Test of Computers, vol. 17, no. 1, pp. 68-83.

[7] K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck (2002): “Configuration Relocation and Defragmentation for Run-Time Reconfigurable Systems”. In: IEEE Trans. on VLSI, Vol.10, No.3.

[8] Montone, A., Santambrogio, M. D., Sciuto, D., & Memik, S. O. (2010). Placement and floorplanning in dynamically reconfigurable FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, *3*(4), 24

[9] J. Burns, A. Donlin, J. Hogg, S. Singh, M. de Wit (1997): “A Dynamic Reconfiguration Run-Time System”. In: Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.

[10] E. El-Araby, I. Gonzalez, T. El-Ghazawi (2009): “Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing”. In Journal ACM Transactions on Reconfigurable Technology and Systems Volume 1 Issue 4, January 2009 Article No. 21.

[11] G. Durelli, C. Pilato, A. Cazzaniga, D. Sciuto and M. D. Santambrogio (2012): “Automatic Run-Time Manager Generation for Reconfigurable MPSoC Architectures”. In: 7th International Workshop on Reconfigurable Communication-centric Systems-onChip (ReCoSoC).

[12] A. DeHon (1999): "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)”. In: Proceeding of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays.

[13] 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. FPL 2006: 1-6

[14] L. Bauer, A. Grudnitsky, M. Shafique, and J. Henkel, PATS: A Performance Aware Task Scheduler for Runtime Reconfigurable Processors, in Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) 2012.

[15] A. Morales-Villanueva and A. Gordon-Ross, On-chip Context Save and Restore of Hardware Tasks on Partially Reconfigurable FPGAs, in Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) 2013

[16] D. Koch, C. Beckhoff, Jürgen Teich: Minimizing Internal Fragmentation by Fine-Grained Two-Dimensional Module Placement for Runtime Reconfigurable Systems. FCCM 2009: 251-254

[17] D. Göhringer, M. Hübner, E. Nguepi Zeutebouo, J. Becker: Operating System for Runtime Reconfigurable Multiprocessor Systems. Int. J. Reconfig. Comp. 2011 (2011)

[18] D. Göhringer, S. Werner, M. Hübner, J. Becker: RAMPSoCVM: Runtime Support and Hardware Virtualization for a Runtime Adaptive MPSoC. FPL 2011: 181-184

[19] C. Conger, A. Gordon-Ross, A. D. George: Design Framework for Partial Run-Time FPGA Reconfiguration. ERSA 2008: 122-128

[20] Z. Li, S. Hauck: Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. FPGA 2002: 187-195

[21] T. . Becker, W. Luk, P. Y. K. Cheung: Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. FCCM 2007: 35-44

[22] Vipin, Fahmy, Architecture-Aware Reconfiguration-Centric Floorplanning for Partial Reconfiguration, in Proc. ARC 2012.

[23] Jara-Berrocal, Abelardo, and Ann Gordon-Ross. "Hardware module reuse and runtime assembly for dynamic management of reconfigurable resources." Field-Programmable Technology (FPT), 2011 International Conference on. IEEE, 2011.