

TECHNICAL UNIVERSITY OF CRETE, GREECE
SCHOOL OF ELECTRONIC AND COMPUTER ENGINEERING

Implementing a Run-Time System Manager on Partially Reconfigurable FPGA Systems.



George Charitopoulos

Thesis Committee

Professor Dionisios Pnevmatikatos (ECE)

Professor Apostolos Dollas (ECE)

Associate Professor Ioannis Papaefstathiou (ECE)

Chania, October 2015

Abstract

The last few years FPGAs have penetrated the mainstream and have experienced wide usage through the users. Also the concept of reconfigurable computing has benefited numerous application domains, with FPGAs being the stronger representative of that. Partial reconfiguration technology can leverage these systems by swapping in and out task modules in an operating-system fashion. A task can be downloaded upon arrival or when needed, during the system operation. To this direction one of the most important parts of said embedded system is the Run Time System Manager.

Despite the fact that, during recent years, many Run Time System Managers have been proposed, very few of them have been implemented on a realistic FPGA system. Moreover due to the vast collection of platforms utilizing reconfigurable logic and their differences the realization of the RTSM on these machines becomes a highly customized process. Thus the RTSM has to be as generic as possible in order to make it easier for the user to implement our RTSM in different reconfigurable platforms.

In this thesis we present the design and implementation of an RTSM we have crafted that operates and manages multiple reconfigurable platforms created by different vendors (albeit, all using Xilinx Virtex 5, 6, and Zynq series). We present the difficulties and the design choices we had to make for the realization of our RTSM on each different platform. The RTSM is extended in order to be compared with a high level parallelism model, thus displaying the high generic and customizable fashion of our RTSM. Finally we evaluate our designs with different applications and assess the advantages and disadvantages of the applications chosen and their implementations to the respective platform.

Acknowledgements

I would like to thank, first and foremost, my family, who I deeply love, for their love and support, during all these years.

Also I would like to thank Professor Dionysios Pnevmatikatos for accepting me to work and study under his supervision, for his help and his belief that everything will work out, through a specifically hard period of this work. Additionally I would like to thank Professor Apostolos Dollas and Assistant Professor Ioannis Papaefstathiou, for taking the interest in my work and agreeing to evaluate it. Also I would like to thank all the fellow MHL students for keeping the atmosphere in the office happy and interesting at all times.

Last but not least I would like to thank the Theatrical Team of the university as well as the "Fu Manchu" Theatrical Team, which offer the perfect way to clear your head after a tiring day at the office and also my friends in Crete and in Athens for coping with me all these years.

The work of this thesis was supported in the context of the European Commission FP7 FASTER Project.

Publications

1. George Charitopoulos, Kyprianos Papadimitriou, Dionisios Pnevmatikatos, "Run-Time Hardware Task Scheduling for Partially Reconfigurable FPGAs", Poster Tenth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), Fiuggi, Italy, Jul 2014
2. G. Charitopoulos, I. Koidis, K. Papadimitriou, D. Pnevmatikatos, "Realistic Hardware Task Scheduling for Partially Reconfigurable FPGAs", HiPEAC Workshop on Reconfigurable Computing (WRC), Amsterdam, Netherlands, Jan 2015
3. George Charitopoulos, Iosif Koidis, Kyprianos Papadimitriou, and Dionisios Pnevmatikatos, "Hardware Task Scheduling for Partially Reconfigurable FPGAs", ARC 2015, the 11th International Symposium on Applied Reconfigurable Computing, Bochum, Germany, Apr 2015
4. G. Charitopoulos, D. Pnevmatikatos, Marco D. Santambrogio, K. Papadimitriou and D. Pau, "A Run-Time System for Partially Reconfigurable FPGAs: The case of STMicroelectronics SPEAr board", ParaFPGA2015, Parallel Computing with FPGAs: a mini-symposium in conjunction with ParCo2015, Edinburgh, UK, Sep 2015
5. George Charitopoulos, Iosif Koidis, Kyprianos Papadimitriou, and Dionisios Pnevmatikatos, "Run-Time Management of Systems with Partially Reconfigurable FPGAs" Integration VLSI Journal (submitted)

0. PUBLICATIONS

Contents

Publications	v
1 Introduction	1
1.1 Thesis Contributions	1
1.2 Thesis Outline	2
2 Background	5
2.1 Key Features and Functionality of the Scheduling Algorithm	6
2.2 RTSM Input and Execution Flow	9
2.2.1 Tasks with Deadlines	13
2.3 Observations	13
3 Related Work	15
4 The RTSM Extensions and Synthetic Simulation	23
4.1 RTSM Extensions	23
4.2 A Synthetic Workload Simulation	26
4.2.1 Discussion	29
5 Implementations of the RTSM	31
5.1 Partially Reconfigurable Platforms	31
5.2 The XUP-V5 Platform	32
5.2.1 Partial Reconfiguration on the XUP-V5	33
5.2.2 The RTSM on the XUP-V5	34
5.3 The SPEAr Platform	37
5.3.1 The hardware side	38
5.3.2 The software side	41

CONTENTS

5.3.3	The FPGA daughter-board	42
5.3.4	Partial Reconfiguration on the SPEAr	43
5.3.5	The RTSM implementation on SPEAr	49
5.4	Conclusion	51
6	Experimental Results	53
6.1	The XUP-V5 experimental results	53
6.2	The SPEAr experimental results	59
6.3	Conclusion	64
7	A Software-only Environment: the OpenMP case	67
7.1	Framework Description	68
7.2	RTSM vs OpenMP	71
8	Conclusion and Future Work	79
	References	84
9	Appendix A: The ZedBoard implementation	85

List of Figures

2.1	RM2-RR1 does not exist, thus the hardware task laying in RR2 is relocated by first configuring RM1-RR1, and then RM2-RR2.	7
2.2	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 AES+DES module) is available.	8
2.3	RTSM main routine	11
2.4	The RTSM schedule function execution flow.	12
3.1	The architecture proposed by Steiger et al. All the slots are initialized with a dummy task logic.	16
3.2	The VAPRES architecture showing a single RSB with 3 RRs.	17
4.1	Application task-graph annotated with HW and SW execution time and reconfiguration time for each task.	27
4.2	The scheduling outcome of our example, showing features such as relocation, reservation and prefetching. The use of the SW version of task T5 completes faster the overall execution.	29
5.1	The estimated growth of the FPGA market between 2014 and 2020. . . .	32
5.2	The reference architecture provided by Politecnico di Milano.	35
5.3	Full Embedded System Architecture. The red "X" marks modules that have been erased in our architecture.	37
5.4	Circuit diagram for the SPEAr 1310 board.	39
5.5	Master (a) and Slave (b) Interfaces for the AHB-Lite communication bus.	40
5.6	Interconnection scheme of the SPEAr platform.	41

LIST OF FIGURES

5.7	BootROM Architecture.	42
5.8	The architecture used to enable communication between SPEAr and the Virtex-5.	44
5.9	Non-Continuous ICAP Data Loading time diagram.	46
5.10	The architecture of the ICAP component.	47
5.11	The FSM controlling the reconfiguration process.	47
6.1	The Edge Detection application task graph.	54
6.2	Execution flow for two consecutive runs of the Edge Detection application.	58
6.3	Output image of the RayTracer application with all the primitives.	60
6.4	The merged task graph. Edge Detection tasks are executed in SW, RayTracer tasks are executed in HW.	61
6.5	Execution flow of the Edge Detection and RayTracer application.	62
6.6	Resulting images for the RayTracer and Edge Detection applications.	63
7.1	Execution times in <i>nsecs</i> of the Edge Detection application for the small (a,b) and medium (c,d) images.	73
7.2	Execution times in <i>nsecs</i> of the Motion Detection application for the small (a,b) and medium (c,d) images.	74
9.1	System Architecture of the Zynq platform.	86
9.2	Input and output images for the Zynq ZedBoard platform.	88

Chapter 1

Introduction

During recent years several advancements have been made in the FPGA technology, including but not limited to partial reconfiguraiton. This new technology prompted more and more researchers to implement many applications from many different fields on FPGAs taking advantage of PR technology. Also recently a big part of the research was spent on Operating Systems or Run-Time System Managers residing on an FPGA system managing the available resources.

In this thesis we present the work done towards the creation and implementation of an RTSM ported on an actual partially reconfigurable FPGA. In the rest of the chapter we analyze and present the contributions of this thesis and the outline that will be followed.

1.1 Thesis Contributions

In this Master thesis the design and implementation of a complete Run-Time System Manager (RTSM) is presented. The RTSM manages and schedules hardware implementations of tasks. This RTSM could be ported and implemented on the current FPGA technology managing the resources available respecting the current technology restrictions induced by each vendor. During the course of this thesis we will present two implementations of the RTSM on FPGA platforms as well as one implementation done in a SW multi-processor system. The work done in this thesis is an extention of the work done throughout my Bachelor thesis where the original scheduling algorithm, the backbone of the RTSM, was presented.

1. INTRODUCTION

In this thesis we evolve this theoretical scheduling algorithm to a complete Run-Time System Manager (RTSM). The contributions of this thesis are:

- ✓ Extensions to the original algorithm, to provide the following additional options.
 - The ability of the RTSM to manage and schedule not only hardware tasks but software ones, which together constitute a hybrid application.
 - Support for dynamic adaptation of the initial execution times provided by the user.
 - Support to execute multiple applications sharing the already existing Reconfigurable Regions (RRs).
 - Support of complex task graphs, including fork-join operations, if clauses and loop structures.
 - The ability for the user to choose how many times the application will execute, or if the application is a streaming one.
- ✓ Creation of an architecture, that supports for the first time, the realization of Partial Reconfiguration (PR) on the STMicroelectronics SPEAr 1300 development board.
- ✓ Integration and Evaluation of the RTSM on two FPGA boards, using real applications.
 - A Xilinx ML505-V5LX110T evaluation platform.
 - A STMicroelectronics SPEAr 1300 development board coupled with a XCV5LX110 FPGA daughter-board.
- ✓ Adaptation of the created RTSM in order to manage only software tasks, in a multi-processor environment and comparison with the OpenMP API.

1.2 Thesis Outline

In Chapter 2 we provided the background needed in order for the reader to comprehend the original scheduling algorithm, which was adapted in a full RTSM. In Chapter 3 of this thesis we present related work done in the field of RTSM targeting FPGA devices

and their implementations, if any. Also we present all the extensions made to our RTSM compared to the original scheduling algorithm in Chapter 4.

In Chapter 5 we present the implementation and integration of the RTSM in the two FPGA platforms, as well as the architecture allowing the PR feature on the SPEAr board. Consequently, Chapter 6 contains the results derived from the experiments made in these two platforms. In Chapter 7 we present the transformation of the RTSM in order to manage software implemented tasks in a multi-processor system and the comparison with the OpenMP API. Finally Chapter 8 concludes the thesis and provides comments for possible future work. In Appendix A the reader will find another implementation of this RTSM on a ZedBoard.

1. INTRODUCTION

Chapter 2

Background

In this chapter we focus on all the needed information for the in-depth understanding of the thesis. The key part of this thesis is the Run-Time System Manager chosen to implement on the different development boards. The RTSM created was based on the scheduling algorithm presented in the thesis "Hardware task Scheduling for Partially Reconfigurable FPGAs". Using the algorithm presented there we apply extensions and changes in order to implement the RTSM on an actual FPGA platform. In Chapter 4 the changes and extensions made to create the final RTSM are presented.

The Run-Time System Manager

The RTSM, used in our case, is incorporating efficient scheduling mechanisms that balance effectively the execution of hardware (HW) tasks and the use of physical resources. We aim to execute as fast as possible a given application without exhausting the physical resources. Our motivation during the development of RTSM was to find ways to overcome the strict technology restrictions as exemplified by the Xilinx PR flow [1]:

- Static partitioning of the reconfigurable surface in reconfigurable regions (RR).
- 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.

2. BACKGROUND

- 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 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.

The proposed RTSM manages physical resources employing scheduling and placement algorithms to select the appropriate hardware processing element (HW-PE), i.e. a Reconfigurable Region (RR), to load and execute a particular HW tasks, which are implemented as Reconfigurable Modules, stored in a bitstream repository.

2.1 Key Features and Functionality of the Scheduling Algorithm

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) [1]. This limitation was discussed in [2] and [3].

(2) *Task information*: Execution time of 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.

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 [2], and accounts for the increased scheduling flexibility and quality [3].

2.1 Key Features and Functionality of the Scheduling Algorithm

(2) *Reservation list*: When a task cannot be served immediately due to resource (RR) unavailability, it is reserved in a queue for later configuration/execution. A HW task will wait in the queue until an RR 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 reconfiguration 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 of time [4]. 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 figure 2.1. 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 [5]. To fully exploit the benefits of this approach context save techniques are needed [6].

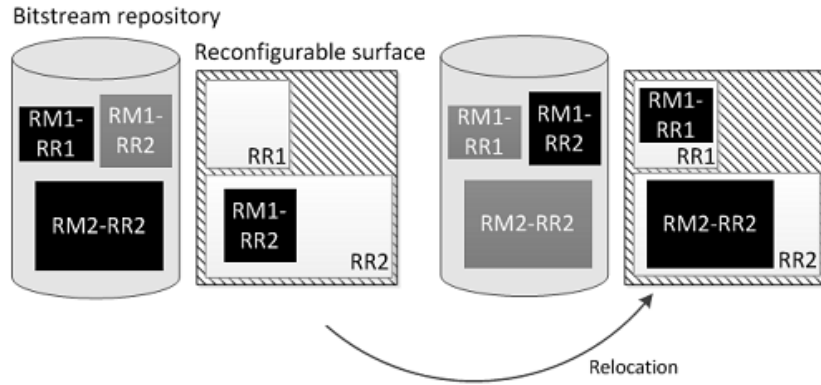


Figure 2.1: RM2-RR1 does not exist, thus the hardware task laying in RR2 is relocated by first configuring RM1-RR1, and then RM2-RR2.

(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

2. BACKGROUND

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 [7].

(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).

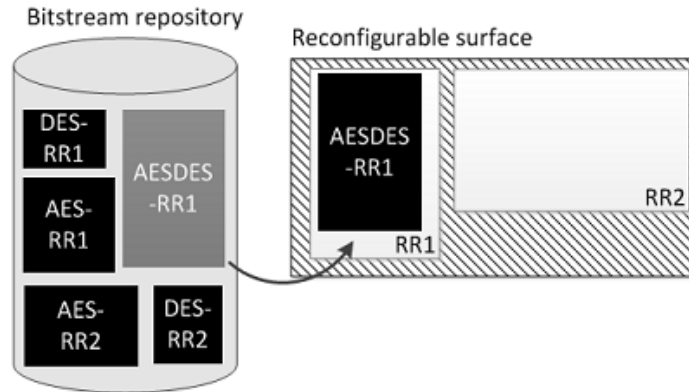


Figure 2.2: Implementations for all crypto modules are available for and can be loaded into any RR, however a large amount of resources will be under-utilized. JHM utilizes more efficiently the RR area, given that the corresponding bitstream (the combined AES+DES module) is available.

(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 figure 2.2, 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 [8].

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 it is important to study if complex techniques and features are actually required to serve efficiently different kind of applications.

2.2 RTSM Input and Execution Flow

In order to understand the functionality and the scheduling choices made by the RTSM it is important to analyze the execution flow followed and the input of the RTSM. First the RTSM needs to have information about the partitioning of the reconfigurable surface in partially reconfigurable HW-PE resources, the availability of SW-PE resources if any, the tasks to be scheduled, the task graph representation describing task dependencies, and the available task mapping, i.e. bitstreams for the different implementations of each hardware task, and executable files for the tasks implemented in software that can be executed by SW-PEs.

For each task it is important for the RTSM to have the HW/SW execution time estimates, as well as, reconfiguration time estimates of each hardware implementation of a task, and optionally the task's deadline. This information is crucial for the RTSM to perform the initial scheduling and provide a "warm start" to the process. Later on when more real-time information will arrive regarding the tasks' execution times, the RTSM will dynamically change his behaviour.

All the above information is given to the RTSM via a .dat file, provided an existing file system, or can be dynamically linked with the RTSM library. That way the RTSM retrieves all the needed information prior to the execution of the application. The RTSM needs to constantly keep track of the state of the reconfigurable regions, tasks, bitstreams etc. To achieve that we use list structures to represent each important aspect, the RR list; the task list; the mappings list ,i.e. available bitstreams and software executable files; and a reservation list for the "newly arrived" tasks that cannot be immediately start there execution, in either a HW-PE or a SW-PE [9].

2. BACKGROUND

In our case since we consider a task graph that describes dependencies between tasks we do not consider random arrival times. However this option is also viable by describing a task on the input file as independent. A task is characterized as arrived in two occasions:

Definition 1. *If a dependent 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 $t_{arr}=x+1$.*

Definition 2. *An independent task's arrival time is the one dictated in the input .dat file.*

The position of the RRs is also described. Since the current FPGA technology supports a 2D representation of the device, the position is described by pairs of x, y coordinates, representing the bottom-left corner of the rectangle the RR 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.

During the application execution, the RTSM alters its scheduling decisions in a dynamic fashion. The RTSM reacts according to dynamic parameters such as the runtime status of each HW/SW task and region, e.g. executing, idle, scheduled for reconfiguration, scheduled for execution, region that is free/reconfigured-but-idle/reconfigured-but-active etc. All this information, about the FPGA condition and the tasks' status, is assessed during run-time and according to that decisions are made.

The RTSM is divided into three main phases of execution. The first phase continuously checks for newly arrived tasks, and invokes the schedule function. After a scheduling decision is reached, the RTSM checks if the task can be served immediately, and then whether the *Reuse Policy* can be invoked. Accordingly it issues a reconfiguration or an execution instruction. During the second phase the RTSM checks if a task has completed execution, and decides which task to tag as "newly arrived" next according to the task graph, in order to schedule it. In the third and final phase the RTSM checks if there are reserved tasks that should begin execution. The RTSM's main routine is shown in figure 2.3.

At the end of the execution flow of the main routine, 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

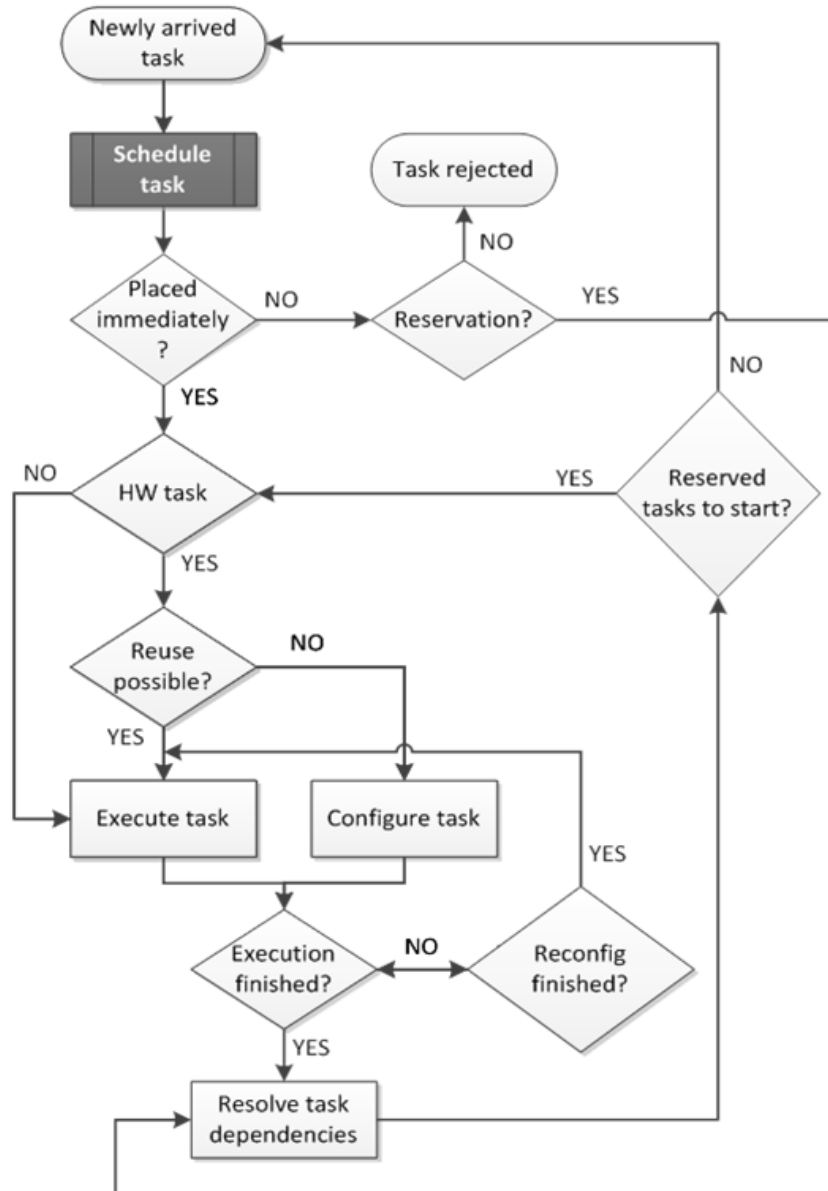


Figure 2.3: RTSM main routine

reconfiguration of a task should finish, and then the RTSM will initiate its execution. Then, given that the time required of the execution of a task is unknown, since we have available only the estimated time, the RTSM is waiting to be informed by the task itself that it completed its execution. Successively the RTSM will resolve the task dependencies. Not until execution of the task is finished, will the RTSM exit the "Execution finished"

2. BACKGROUND

step.

However the key part of this main routine is the scheduling function, the grey box in figure 2.3, which is shown in figure 2.4. In this function the RTSM decides upon the scheduling of the incoming task. To reach a scheduling decision the RTSM follows a complex process, using a set of functions and policies, which are already described. Firstly the RTSM tries to exploit all the available bitstreams, for the scheduled task, in terms of space, *Best Fit in Space Policy*. Then the RTSM tries to find a solution that will provide the task with the best ending time, *Best Fit in Time Policy*. To do so the RTSM creates and maintains new lists for the scheduled task. These lists contain information for the available mappings and the available RRs, for which a mapping exists.

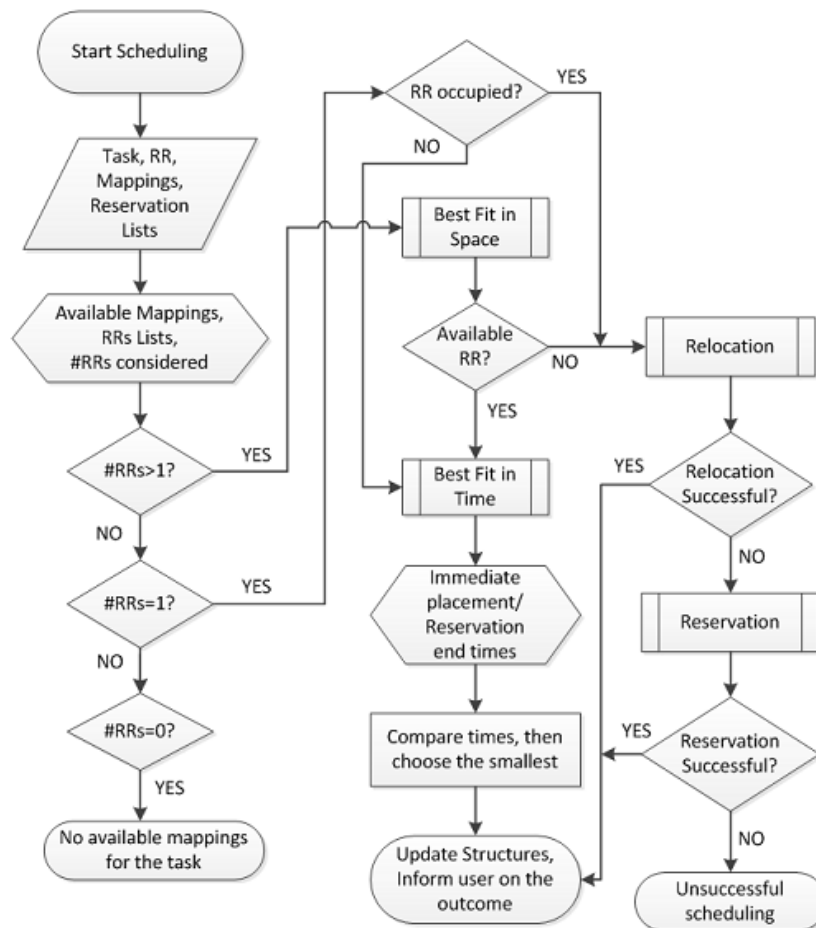


Figure 2.4: The RTSM schedule function execution flow.

If the available RRs list contains more than one RR then a Best Fit Policy decides which RR the task will be placed on, considering the occupied area by the task. The policy will pick the bitstream of the task that best utilizes the area of the corresponding RR. The Best Fit Policy can be summed up in the following statement:

Best Fit in Space Policy. *A newly arrived task will be placed on the RR producing the smallest unused area, provided this RR is free.*

If all the RRs on the available RRs list are occupied, but there are free RRs on the device that have no corresponding mappings for the scheduled task, the scheduler performs *relocation*. By employing relocation the RTSM tries to relocate an already placed task to another RR so as to accommodate the newly arrived task. If this alternative is also unsuccessful the scheduler will attempt to make a reservation for the newly arrived task, i.e. reserve a slot in an RR for the task to be executed at one point in the future.

Even if the scheduler finds a suitable RR for immediate placement, it will also perform the Best Fit in Time Policy in order to check if by reserving the task for later execution and reusing a previously placed core, the newly arrived task will finish its execution at an earlier time, than the one expected by immediately placing on the device. It is important to note that the RTSM besides the RR, treats also the configuration controller as a resource that must be scheduled.

2.2.1 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 the user is notified to execute it in software applying a penalty time to the execution time of the application. Also it must be noted, that the proposed relocation mechanism moves a HW task only if its deadline will be met.

2.3 Observations

At this point it is important to take note of some observations regarding the current version of the RTSM, prior to describing the upgrades that have been made.

2. BACKGROUND

- **Hardware Execution:** At this point the RTSM manages only hardware implementations of the tasks described in the input file and the hardware resources present in an FPGA.
- **Task reconfiguration and execution times:** These inputs are derived through profiling, which can be done by the end user. It can also be computed using theoretical reconfiguration times and the HW bitstream size, while the execution time can be estimated by the compilation tools, or can be provided by the programmer during the design phase of the application. This information is provided to the RTSM in its initialization. However, in practice, a task may execute for more time than its predicted execution time, hence the RTSM should be notified when a task completes execution, so that it may update its scheduling decisions dynamically.
- **Size of RRs and RMs:** These parameters are defined at design-time and have fixed values. Best Fit in Space reacts based on these parameters.

Chapter 3

Related Work

In this chapter the work done in the literature regarding several Run-Time System Managers and their implementation or lack thereof on FPGA partially reconfigurable devices is described. Some of the analysis will be spent on related works done about scheduling algorithms for hardware tasks.

With the introduction of Partial Reconfiguration (PR) FPGAs overcame the high overhead which comes along with Full Reconfiguration, which could negatively impact the overall performance of an application. In [10] an effort was made to include PR on High-Performance Reconfigurable Computing (HPRC), by investigating the performance potential of PR on HPRCs from both theoretical and experimental perspectives. Based on their experiments El Araby et al. conclude that hardware virtualization and multi-tasking using PR could prove beneficial.

This study concluded the question of whether hardware task multi-tasking could be used on a partially reconfigurable platform, allowing the community to revisit old ideas of Operating Systems targeting reconfigurable devices.

In [2] J. Burns et al. based on three different applications of full dynamic reconfiguration extract a set of common requirements. Based on their analysis they present the design of an extensible run-time system manager (RTSM) that controls the dynamic reconfiguration of the FPGAs, giving not only architectural insight on how the RTSM would interact with the hardware but also describing the higher-level needed software support. This work, despite the fact that does not consider PR, is one of the definitive works for the beginning of designing RTSMs/OS targeting FPGAs.

3. RELATED WORK

In order to make the development of an RTSM/OS targeting a partially reconfigurable FPGAs easier several aspects of dynamic reconfiguration had to be simplified for the end user. In [1] the authors describe architectural enhancements to Xilinx FPGAs that provide better support for the creation of dynamically reconfigurable designs. Along with the enhancements the authors tried to develop better PR tools that could allow the user to create reconfigurable designs with less overall effort. Also several improvements were made on Xilinx's PlanAhead tool to better automate the sizing of the Partially Reconfigurable Regions (PRRs).

An important work towards the realization of a RTSM that could target partially reconfigurable FPGAs was presented by C. Steiger et al. at [9] with a preliminary study being made in [11]. In these papers the researchers discuss design issues for reconfigurable hardware operating systems. Also an attempt is made in creating a runtime system for guarantee-based scheduling of hard real-time tasks. In their design Steiger et al. partition the reconfigurable surface in vertical slots, which can accommodate hardware tasks according to their width. The proposed architecture presented is shown in figure 3.1.

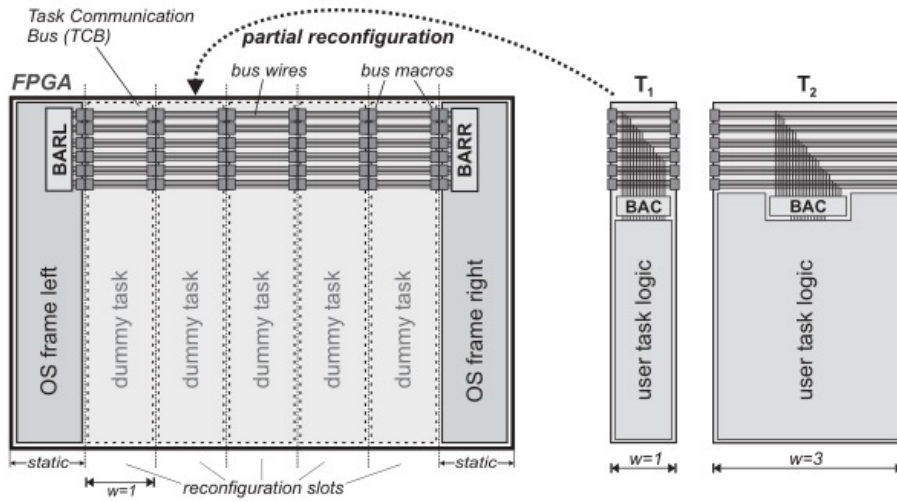


Figure 3.1: The architecture proposed by Steiger et al. All the slots are initialized with a dummy task logic.

In [9] C. Steiger also introduced the idea of a scheduler for hardware tasks, i.e. an algorithm that would manage the placement and beginning of execution of hardware tasks

on the FPGA. Despite the fact that the results obtained by simulation are promising both schedulers developed cannot be used with current FPGAs due to strict technology restrictions.

Despite the fact that the enhancements offered by PR have been proved and also the ability to create RTSMs to assist with the execution of task-based application has been realized by several authors the PR technology has not yet penetrated the mainstream amongst HPC users. That is due to the challenges involved in creating a PR design.

In [12] the authors present VAPRES, a PR base architecture that provides a customizable and flexible platform for PR system and application design. VAPRES's key architectural contributions include customizable PR regions (PRRs) needed for PR, and seamless hardware module replacement. Alongside with VAPRES in [13] the authors present a communication protocol, for communication between PRRs called SCORES. The reconfigurable architecture presented includes a static region and Reconfigurable Streaming Blocks (RSB) that includes numerous Reconfigurable Regions.

The architecture presented offers the user a scalable and parametric streams-based communication interface between RRs. The work considers a reconfiguration prefetching mechanism. This work could be benefited by the inclusion of an RTSM to manage the user applications running on the current architecture. The complete VAPRES architecture is depicted in figure 3.2.

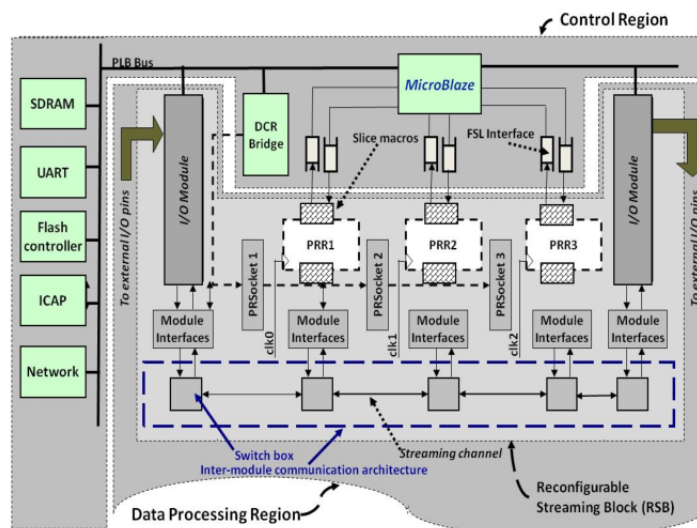


Figure 3.2: The VAPRES architecture showing a single RSB with 3 RRs.

3. RELATED WORK

In the field of complete works another interesting work was presented in [14] by R. Pellizzoni et al. The authors present a real-time computing architecture that can integrate hardware and software task execution in a user-transparent manner, and can support real-time QoS adaptation by means of partial reconfiguration of FPGA devices. The main contribution of this work is allowing task to perform context-switch operations between hardware and software execution, enabling thus run-time acceleration in a dynamic way between different computational intensive tasks. Similar to other works referenced, the authors here also underline the need of OS support in order to conduct the scheduling and switching of the tasks. However the scheduler employed in this work is quite simplistic since it considers that provided enough resources exist and the system is not currently in a heavy load state, tasks that have a hardware configuration will be executed in hardware.

Another work that abides to technology restrictions is presented in [3]. In this paper the authors introduce a Run-Time System Manager (RTSM) that is able to map multiple applications on the underlying architecture and execute them concurrently. Also the authors try to overcome portability issues by creating an RTSM that is not architecture-dependent thus allowing its migration to other FPGAs. Also as with [12] this work presents a ready-to-use application-independent architecture that can execute user application efficiently without much effort by the user.

This work provided the reference architecture-independent design on which our RTSM was placed. We choose to implement our RTSM on this design due to its simplicity. Also a deciding factor was that the RTSM's scheduler deployed by the authors does not base its scheduling decisions on sophisticated placement and time-sharing heuristics and it could be easily replaced by our own RTSM.

Since the design of an RTSM was beginning to be possible the research community focused on several other aspects that are related to partial reconfiguration, a key aspect of it being the reconfiguration overhead. Since it will be added to the overall application execution time studies were made on how the floorplanning should be made, in order to achieve better reconfiguration times.

First attempts were made in creating dedicated platforms that could perform better reconfiguration times. In [15] the authors present a networked lightweight and partially reconfigurable platform assisted by a remote bitstreams server. We propose a software and hardware architecture as well as a new data-link level network protocol implementation dedicated to dynamic and partial reconfiguration of FPGAs. With their design the

authors achieved a x10 speed up in reconfiguration speed (Mb/s.MHz) against the then Xilinx proposed partial reconfiguration architecture.

In [8] the authors present a technique which can be incorporated into the existing tool flow that overcomes the need for manual floorplanning for PR designs. It takes into account overheads generated due to PR as well as the architecture of the latest FPGAs. This results in a floorplan that is efficient for PR systems, where reconfiguration time and area should be minimized.

A work towards module reuse in order to reduce unused/wasted resources and reconfiguration time is presented in [16]. In this paper the authors present a Dynamic Resource Manager that provides efficient hardware resource management. By applying module reuse check that would prevent PR and avoid the resulting reconfiguration overhead. The DRM presented in this work is coupled with the SCORES and VAPRES architectures constitute a whole FPGA design that can execute and manage applications. The result of this work lacks as well as others in a sophisticated scheduling algorithm that could take complicated decisions during runtime.

Dimitris Theodoropoulos et al. in [17], presented a runtime framework for reconfigurable heterogeneous embedded MPSoCs, which allows rapid application development and efficient tasks allocation to all available processing elements. Their work considers a task-based programming model that requires pragma annotations to resolve task dependencies. Their approach as many others works towards assisting the user to easily map an application on a reconfigurable architecture and benefit from hardware acceleration. However despite the fact that this work targets heterogeneous embedded MPSoCs, the authors treat only MicroBlaze microprocessors as software processing elements, and they do not take into account partial reconfiguration technology and hardware processing elements.

In [18] K. Papadimitriou et al. discuss problems in using reconfigurable technology and suggests some research directions. A detailed analysis of two systems equipped with reconfigurable platforms is discussed and the authors explore the capabilities offered to the end-user with the employment of partial reconfiguration in his designs. Main aim of this work was to draw attention in the importance of transparency in the acceleration of certain kernels inside an application. This can be achieved with the automatic configuration of bitstreams into the FPGA co-processor. This work delves in the ways partial reconfiguration can become easy for the end-user and sets the need of a RTSM

3. RELATED WORK

that would take control of it without needing the user's in-depth partial reconfiguration knowledge.

Another quite interesting and thorough work is presented in [19] by I. Beretta et al. In this work the authors present a design flow able to efficiently map multiple multi-core applications on a dynamically reconfigurable SoC. In the context of this work the authors use the term "core" to refer in different tasks comprising an application. Their work heavily relies on core reuse among different applications and also on the *Joint Hardware Module* ability discussed also in this work in Chapter 2. The methodology proposed maps the different cores in reconfigurable slots trying to minimize the reconfigurations needed and the communication overhead between different reconfigurable slots.

The authors present two mappers, one intended to be used during design time and the other during run-time when an new application is being introduced. The run-time mapper heavily relies on configuration re-use, trying to take advantage of existing bitstreams, provided the application does not need a new core. When a new application introduced in the system the run-time mapper tries to create a new reconfigurable slot configuration using essentially the design time mapper algorithm. This work also tries to include configuration relocation in order to take advantage of all the slots present on the FPGA. The flow presented here maps efficiently the use applications off-line, as well as during run-time, by the means of new bitstream creation, if deemed necessary. A RTSM controlling the reconfiguration process, as well as core communication could greatly enhance the proposed flow. Finally this mapping flow has not been implemented on an actual device, but evaluated via simulation with real-world case studies and synthetic benchmarks.

In [20] the authors present a scheduler that is tailored for Bi-Dimensional Reconfigurable Architectures. The scheduler proposed is a reconfiguration-aware heuristic scheduler, which exploits configuration prefetching, module reuse and antifragmentation techniques. Despite the fact the mentioned antifragmentation techniques are in violation of certain PR technology restrictions, the scheduler presented is quite sophisticated.

Reconfiguration has also been used in reconfigurable processors that, beside the standard processor pipeline include an embedded FPGA to implement Special Instructions (SIs). These processors reconfigure accelerators that implement entire SIs [21] or smaller accelerators where potentially multiple accelerators are used to implement an SI. In order to efficiently manage these reconfigurable processors and the implemented SIs several

schedulers have been presented towards that end. In [22] L. Bauer et. al present a performance aware task scheduler, that is able to measure the acceleration a task will get from the currently present SIs on the reconfigurable surface, and schedule the task accordingly. The philosophy of this scheduler comes really close to ours, since both of them consider not the deadline of the task only, but also the performance acquired by the hardware existing on the device.

On the field of programming models that target multi-core heterogeneous architectures a great impact had the OmpSs model presented in [23]. Apart from multi-core architectures the 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 where all the threads, to be used throughout the application, are present from the beginning.

An important work was done by A. Agne et al. in [24]. Agne collaborated with a large group of researchers targeting the issue of a run-time system for reconfigurable systems, more importantly M. Platzner and B. Plattner, who were working alongside C. Steiger by the time [9] was published. 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. Communication is handled by implementing several FSMs for synchronization and the incorporation of a OS interface. Also important fact regarding the ReconOS architecture is the intermediate OS kernel level, adding to the complexity of the overall, but at the same time subtracting from the complexity the user has to face in order to use the architecture. Finally throughout the ReconOS description there is no mention of a high-level and decision making scheduler. In [25] E. Lubbers and M. Platzner present an implementation of the cooperative multitasking technique for thread management, targeted for the ReconOS architecture. In this publication the authors consider the case where a thread can call a `yield()` function thus signaling its intent to suspend its execution. By allowing threads decide that the authors consider that a simple multitasking scheduling algorithm can be modified and implemented for the hardware threads managed by ReconOS.

3. RELATED WORK

Chapter 4

The RTSM Extensions and Synthetic Simulation

The first step in our work was to extend the functionality of the scheduling algorithm, in order to be able to manage more complex applications, provide support with cutting-edge new technology advancements, i.e. MicroReconfiguration and finally to support resource sharing application run concurrently on the same platform. Here we describe these changes and offer a short evaluation of their performance using synthetic simulations.

4.1 RTSM Extensions

The first extension was to modify the scheduler and the input file in order to include apart from hardware implementations of a task, i.e. partial bitstreams, software implementations also, i.e. executable files, or elf images. This way we offer the end user a versatility in the creation and implementation of the tasks comprising an application. However it is understandable that a software implementation running on a SW-PE of a task will be marginally slower than the hardware one.

The scheduler according to the policies applied and the task's deadline, if any, in several cases will actually choose to initiate the execution of a software version of a task. Furthermore the complexity of the task may deem its hardware implementation to expensive or simply impossible, thus making the software version the only one available. The software implementation of a task is signaled in the input file with a dedicated character after the declaration of a mapping, the executable file can be residing in the

4. THE RTSM EXTENSIONS AND SYNTHETIC SIMULATION

external memory, or the code can be included on the RTSM core code and become a function call when the appropriate task needs to be executed.

Another major extension is the ability of the RTSM to manage complex task graphs. In the original version we considered only linear dependencies of tasks that in order to complete their execution need to perform many iterations. This is similar to a fork-join operation, however the implementation of this functionality in the RTSM's core code was quite simplistic and was a part of the task's characteristics.

Now the RTSM can support complex task graph with fork-join operation between different tasks. Inside the input file a new field is added specifying task dependencies that can now extend the linear ones. Also we consider loop structures that include a subset of the application's tasks. Another extension towards task types is the inclusion of branch-tasks. These tasks according to their outcome, which is compared to a threshold or a fixed value, dictate the next task to be executed, between a choice of two. These tasks depict accurately the if-then-else structure in algorithms.

Regarding the execution times of the application the user now is being offered several alternatives. The user can choose to execute the application task graph only one time, or N-times by specifying the number in the input file, or finally the user can choose to execute the application infinite times. With the final choice the application is considered a streaming application and it is terminated when an interrupt arrives from the user.

The "warm start" scheduling decisions reside on the input file and specifically the estimation of SW/HW execution and reconfiguration times of the different tasks. These estimations however can be marginally different from the real execution times of the task, which can be data dependent, or can vary due to data transfer latencies between the external memory and the RR. Thus it is important to update and change dynamically the time-dependent decisions made by the RTSM. Towards that end the RTSM upon beginning the execution of a task starts a timer that will record the real execution time of the task, either SW or HW. This is done for all the tasks that are concurrently run onto the FPGA. The same is done for the reconfiguration times of the hardware tasks. All the real time measurements are stored in the SW/HW execution and reconfiguration time fields in the tasks list.

The constant update serves useful information and dynamic behavior to the RTSM regarding which implementation of a task will be chosen for execution. The hardware implementation of Task A may be proven to be slower than the software one, so if the

executed application is a streaming one the RTSM will choose the software version of Task A during the second iteration. The constant measurement and documentation will result in average execution and reconfiguration times which will enable the RTSM to perform better scheduling decisions in the future.

Alongside with the afore-mentioned enhancements to the RTSM, a new feature was also supported. In [26] the researchers present the concept of micro-reconfiguration and the complete data path and execution flow needed to be followed by the user in order to create micro-reconfigurable bitstreams. The main concept of micro-reconfiguration is to change specific aspects of the reconfigurable module, e.g. a value in a LUT array, or the contents of a BRAM cell, in order to alter the functionality of the module without reconfiguring again the whole bitstream. Towards supporting this function the RTSM was altered, in order to consider two cases of micro-reconfiguration.

First the micro-reconfiguration was considered as a user-driven instruction issued sometime during the execution of the application. This instruction was perceived as an interrupt by the RTSM and treated accordingly. The format of the user interrupt is the letter "M", followed by the value of the changed attribute and then followed by the task to be micro-reconfigured, e.g. **M020T9** this issues a micro-reconfiguration request for task 9 to set the micro-reconfigured attribute to value 20. A prerequisite for such operation is that the issued micro-reconfiguration regards a task that has a micro-reconfigurable bitstream available. The interrupt could be sent to the RTSM via various ways depending what is the communication between the user and the RTSM, e.g. the user can enter the micro-reconfiguration instruction through a console that then sends it to the RTSM via RS-232.

To achieve micro-reconfiguration some new global flags were added on the RTSM's core code, that would be set depending on the user's micro-reconfiguration instruction. Those flags are the micro-reconfiguration flag (`micro_flag`), i.e. if a MR request has been sent. The micro-reconfiguration value flag (`micro_value`) signaling the new value/difference in the reconfigurable module, e.g. a new value in a LUT array. Finally the mapping in which micro-reconfiguration should occur (`micro_mapping`) because different mappings of the same task, due to placement and routing differences of the design, can occur to different micro-reconfigurable bitstreams.

The RTSM distinguishes two cases of micro-reconfiguration interrupt, regarding to their arrival time.

4. THE RTSM EXTENSIONS AND SYNTHETIC SIMULATION

1. The first case considers the arrival of the micro-reconfiguration request prior to the scheduling and/or placement of the corresponding task. In this case the RTSM sets a field in the task's structure named "Micro" equal to 1. This field denotes that once the reconfiguration of the task finishes the micro-reconfiguration will immediately follow. During the scheduling of this task the micro-reconfiguration overhead is also considered when applying the **Best Fit in Space** policy.
2. The second case considers the arrival of the micro-reconfiguration request during the execution of the corresponding task. In this case the RTSM sets a field in the task's structure named "Re-execute" equal to 1. This field notifies the scheduler that this task will first micro-reconfigure, the soonest time the ICAP resource is idle and re-execute with the new value in the micro-reconfigurable attribute. This re-execution flag is considered throughout, any scheduling decisions made for other tasks before the task re-executes, thus forcing the RTSM to dynamically adapt to the new state of the task's execution and the FPGA's state.

Finally it is possible for the user to combine multiple applications in a single task graph and provide the tasks and mappings information for each one on the same input file. The RTSM considers the two applications as one and thus we can achieve a resource sharing scheduling. However the option to add a priority weight to the numerous applications is not added in the RTSM's code. The user can achieve maximum bitstream re-usage and thus performance, if the applications have some common tasks, nevertheless it must be clear for which application a common task is being executed for, so that the RTSM can send the correct input data to the RR/SW-PE.

4.2 A Synthetic Workload Simulation

In order to evaluate the performance and the scheduling decisions of the extended version of the RTSM we created a discrete time event based simulation, with a synthetic workload application. We will provide the RTSM with a synthetic workload as input. The task graph of the synthetic application is shown in figure 4.1.

In this figure the HW/SW execution times and reconfiguration time is expressed 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

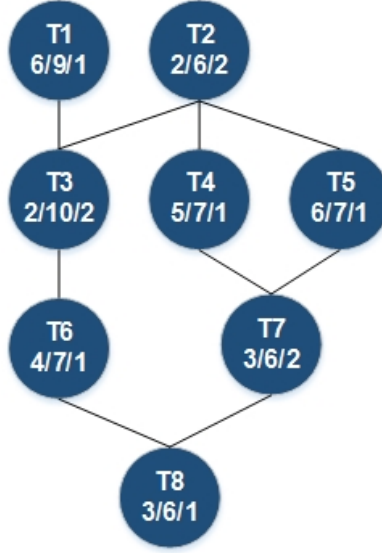


Figure 4.1: Application task-graph annotated with HW and SW execution time and reconfiguration time for each task.

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 used by the Best Fit in Space function. In Table 4.1 the characteristics for each RR and SW-PE present on the device. Each resource is characterized by a number depicting, whether it is a SW-PE or an RR and then if it is an RR the number of slices it contains, expressed in $[x,y]$.

Resource Name	Type	Size $[x,y]$ (slices)
RR 1	HW	$[22, 29]$
RR 2	HW	$[12, 37]$
SW-PE 1	SW	-

Table 4.1: Summary of the logic resources used by the HW cores.

In Table 4.2 the characteristics of each RM-RR binding are shown, the table regards only the hardware implementations of the applications tasks'. The task mappings shown are those that will drive the RTSMthe options of RTSM for making the best scheduling decision for a given task, together with the estimated execution and reconfiguration times.

4. THE RTSM EXTENSIONS AND SYNTHETIC SIMULATION

For example, on one hand T1 has two hardware implementations for both RR1 and RR2, thus providing the RTSM with more options, on the other hand T2 has only one RM-RR binding, making the scheduling options limited.

We assume that every task has a software implementation as well, in order to study how the RTSM reacts in exploiting both hardware and software resources, always to the advantage of the overall application execution time. It is important to note that the software implementation of a task has a longer execution time than the hardware one.

Tasks	Mapping Characteristics		
	RR	Width (slices)	Height (slices)
T1	1	13	24
	2	10	20
T2	2	11	31
T3	2	12	30
T4	1	14	24
	2	12	22]
T5	1	21	25
T6	2	12	30
T7	1	15	26
	2	10	20
T8	1	13	24
	2	11	22

Table 4.2: Summary of the logic resources used by the HW cores.

According to these inputs the RTSM will perform the optimal run-time scheduling according to its policies. The scheduling result is shown in figure 4.2. This is a simple one-time execution of the application and most of the RTSM features are activated. The Relocation alternative is activated in order to accommodate task T2, which as seen has limited placement options. Since Best Fit in Space (BFS) function has placed task T1 on RR2, in order to place task T2 on the FPGA, the RTSM first performs relocation and reconfiguration of task T1 on RR1, and then reconfigures T2 to the now empty RR2. Additionally, the decision to execute task T5 on SW-PE is due to the Best Fit in Time (BFT) function as a later reservation of task T5 on a RR gave a longer completion time.

4.2 A Synthetic Workload Simulation

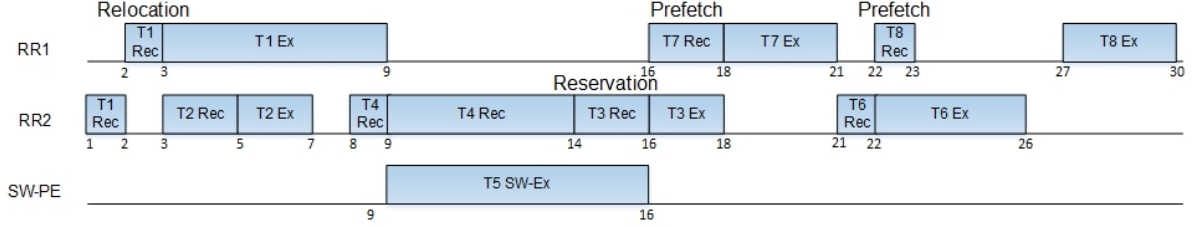


Figure 4.2: The scheduling outcome of our example, showing features such as relocation, reservation and prefetching. The use of the SW version of task T5 completes faster the overall execution.

Also we can see the use of 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. Note 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, there is a high level of inner task parallelism between tasks of the same level but also from different levels, i.e. tasks T1, T2, T4, and T5, and we observe the use of configuration prefetching for tasks T7 and T8.

4.2.1 Discussion

In this chapter we presented the extensions done to the previous version of the RTSM scheduling algorithm. Now the RTSM is more versatile and offers more options to the user on how to model and implement the application, by allowing software task implementations to co-exist with hardware ones. Also the user can have more than one application run on the FPGA.

Also an example was given in order to show the scheduling policies and abilities of our RTSM. In this example we observed the following:

- The task graph is complex enough to demonstrate fork and join operations. We assumed multiple bitstreams per task to show the flexibility of RTSM.
- We demonstrated almost all RTSM features: relocation, reservation, prefetching, BFT and BFS. The reuse policy was not demonstrated as no task is repeated, nor

4. THE RTSM EXTENSIONS AND SYNTHETIC SIMULATION

does the task graph contain a loop. We also did not demonstrate the use of JHM, as we did not assume availability of such bitstreams.

- In figure 4, 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.
- We assumed that the SW-PE execution takes more time than the combined hardware execution and reconfiguration operation, prompting the scheduler to choose the hardware accelerator to program in the FPGA.
- Finally, we assumed that HW task execution time is 2-3 times larger than the reconfiguration time, to model fast reconfiguration times or coarser grain tasks.

Chapter 5

Implementations of the RTSM

This chapter focuses on the two implementations of the presented RTSM made in two different partially reconfigurable platforms. The implementations were carried out with respect to the platforms' technology and design restrictions, while of course keeping the high quality of resource management (HW and SW) offered by the RTSM.

5.1 Partially Reconfigurable Platforms

Partial reconfiguration has started to penetrate the mainstream and a lot of research effort is spent in making it more user-friendly. During recent years boards with embedded FPGAs have documented a big growth in their revenue. This is due to the huge advantages of FPGAs and programmable logic compared to ASICs, adding to that advantages the low-cost and the Partial Reconfiguration ability, then it is understandable why the expected estimated growth of the FPGA market is 8.1% as seen in figure 5.1. The largest industries in the FPGA market are Xilinx, Altera and Actel. However more and more industries are embedding FPGA to their development boards, like STMicroelectronics, Terasic and Texas Instruments.

The platforms used in this work utilize and combine a partially reconfigurable (PR) FPGA with a hardcore or softcore software processor(s). The FPGA can be either integrated on the same board as the processor and the peripherals (leds, switches, buttons, RS-232 port, etc.) or can be connected to the board as a slave daughter-board. Despite the fact that the internal FPGA architecture is quite similar amongst manufacturers, the

5. IMPLEMENTATIONS OF THE RTSM

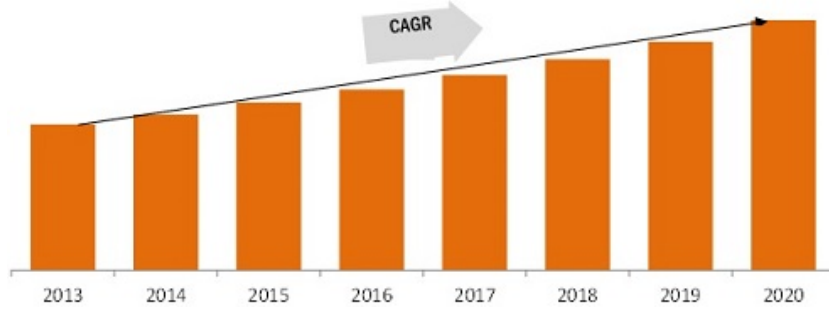


Figure 5.1: The estimated growth of the FPGA market between 2014 and 2020.

architecture of each of the boards that utilize an FPGA can have many differences and technology restrictions.

The RTSM presented was designed with respect to most common FPGA technology restrictions but also with the intention of being easily portable amongst different PR platforms, hence the reason of two implementations on marginally different platforms in terms of FPGA integration and overall design architecture. In this chapter we present the implementations of the RTSM made on two PR platforms, the well-known, Xilinx XUP-V5 and the SPEAr 1310 development board, provided by STMicroelectronics.

5.2 The XUP-V5 Platform

The XUP-V5 LX110T evaluation platform is a general purpose evaluation and development platform with on-board memory and standard connectivity interfaces. It also features a Virtex-5 LX110T FPGA. The work presented here has been done in collaboration with [27], this work focuses more on the hardware side of the problem and how the architecture was implemented. In this chapter the focus is on the changes that had to be made in order to port the RTSM, from a practical point of view.

On this platform the first attempt for porting the RTSM was made. Despite the fact that the FPGA included in this platform is a medium one, in terms of resources, the vast features it includes make it ideal for hardware acceleration supported by an OS. Some of the key features included on the device are:

- Compact Flash configuration controller.

- 64-bit wide 256Mbyte DDR2 small outline DIMM module.
- On-board 32-bit ZBT synchronous SRAM.
- 10/100/1000 tri-speed Ethernet.
- USB host and peripheral controllers.
- RS-232 port, 16x2 character LCD, and many other I/O devices and ports

To implement the RTSM on a XUP-V5 platform several decision have to be made. These decisions regarded the invocation of execution and reconfiguration of the hardware tasks and where the RTSM would be placed on the device. Also if the RTSM would be placed outside of the platform the connection and transfer of messages and results between the RTSM and the platform would have to be specified.

5.2.1 Partial Reconfiguration on the XUP-V5

The partial reconfiguration ability is well established on the XUP-V5 platform. Since the board offers many on-board memories, but also Compact Flash (CF) memory controllers, the user can create the partial bitstreams and store them in a CF memory which directly communicates with the board.

The easiest way to achieve this communication is by utilizing the Xilinx Kernel built-in functions implemented by the MicroBlaze softcore microprocessor. The MicroBlaze is a microprocessor that is included in the Xilinx software as an IP building block. The Xilinx IPs are building blocks that can be implemented on Xilinx targeted devices. The MicroBlaze is a Soft microprocessor core designed for Xilinx FPGAs from Xilinx. As a soft-core processor, MicroBlaze is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs.

The user can deploy on the FPGA fabric as many MicroBlaze processors as he wants, provided there are sufficient resources. Also the FPGA can accommodate reconfigurable regions, which are the areas in which partial reconfiguration can take place. The partial reconfiguration is initiated by the MicroBlaze processor, which communicates with the Internal Configuration Access Port (ICAP). The ICAP controller then write the reconfiguration data to the Configuration Memory, which in turn changes the logic included in the specified RR accordingly.

5. IMPLEMENTATIONS OF THE RTSM

The communication between the RRs and the ICAP controller is done through the fast Processor Local Bus, a two-way bus with high bandwidth capabilities used for communication between the MicroBlaze processor and several peripherals (RS-232, ICAP, RRs, etc.).

5.2.2 The RTSM on the XUP-V5

So far the choices for where the RTSM would be placed are plenty. However two only stand out, the true embedded solution, with the RTSM running on an embedded Microblaze, and the host PC solution with the RTSM running on a host PC communicating with the FPGA via the RS-232.

In the course of this work both solutions were implemented. An architecture provided by Polytecnico Di Milano (PDM) [3] was used as reference. The basic work was done with this architecture which was mildly modified in order to port the RTSM. The initial architecture's diagram is shown in figure 5.2. The architecture's main components were:

- **Two MicroBlaze SW processors.**
 - The first MB acts as the basic Run-Time System Manager (this RTSM was implemented by PDM) and serves Memory Requests by the RRs.
 - The second MB acts as a SW-PE, Reconfiguration Manager, and reads Data from the Compact Flash.
- **DDR2 SDRAM Memory** accessed by both MB processors.
- **Two Reconfigurable Regions (RRs).** Their reconfiguration is controlled by MB2 through the ICAP interface and their memory accesses are handled by MB1, through interrupts.
- **Timers and counters**
- **A Mailbox** module for communication between the two processors.
- Two **PLB** buses for communication between the two processors and the numerous peripherals.

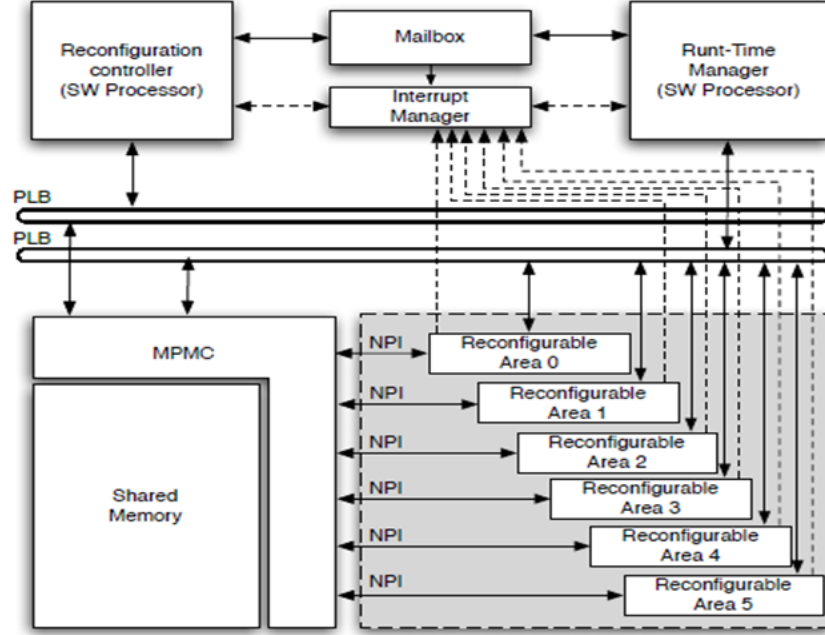


Figure 5.2: The reference architecture provided by Politecnico di Milano.

In order to use our RTSM in order to control the reconfiguration and execution of hardware tasks on this architecture several changes were made. The first case considered is that where the RTSM is running on a host PC communicating with the FPGA via the RS-232 port. Since the RTSM will run on the host computer then the MB1's implemented by PDM manager should be bypassed, also in our RTSM we had to implement the RS-232 communication protocol, in order to send coded execution and reconfiguration instructions for each task. The coded instructions for each operation from the RTSM to the FPGA are:

- **SW_{*i*}**: Each code *i* signals the beginning of either a software task or a software reconfiguration carried out by MB2.
- **H_{*i*}**: Each code *i* signals the invocation of a hardware task.

Also since the RTSM takes decision regarding task placement based on estimated values of execution and reconfiguration times. Thus we need to create from the FPGA side a communication protocol that would signal the RTSM with the ending of execution and reconfiguration for each task. The codes send from the RTSM to the FPGA and

5. IMPLEMENTATIONS OF THE RTSM

from the FPGA to the RTSM can be modified and extended by the user depending on the needs of the application. The interrupts sent from the FPGA to the RTSM are:

- **ST:** This interrupt signals the termination of a software task, executed on MB2.
- **RT:** This interrupt signals the termination of a reconfiguration.
- **HT 1 or 2:** This interrupt signals the termination of hardware execution of a task. This interrupt is followed by a number signaling which RR has ended its task execution.

Now MB 1 has only the communication protocol with the host PC and the RTSM, sends the appropriate reconfiguration commands to MB 2 and handles interrupts from the RRs. The second architecture was build with the RTSM running on the device, thus implementing a full embedded solution. In order for the RTSM to run on the MicroBlaze processor we needed to migrate the core-code to the Software Development Kit, a program by Xilinx, which allows the user to create designs and write code for execution on the MicroBlaze micro-processor.

The migration of the code even though the programming language is the same, C and MicroBlaze-C are almost identical, presented some difficulties. These difficulties did not concern the code of the RTSM but rather the available memory of the MicroBlaze, which could not accommodate the whole core code of the RTSM.

A change in the processor's memory was not possible due to the many designing issues it arose. In order to tackle this problem we decided to disable some RTSM features. This was done in accordance with the application implemented depending on which features it would benefit from. More on this choice will be presented on Chapter 6, where the experimental results of the different implementations will be presented.

Now that the RTSM run on MB1 the communication between MB1 and MB2 had to be established via the already present Mailbox. When MB1 signaled a software execution or a reconfiguration instruction on MB2 it writes the same coded instructions on the Mailbox. As soon as the instruction was sent and the operation on the other side begins, the RTSM continues its execution. Once the operation finishes an interrupt arrives from MB2 and once the RTSM handles it, continues scheduling according to what operation has finished. The new architecture is presented in figure 5.3. The changes made in the architecture to support the features mentioned are shown in red.

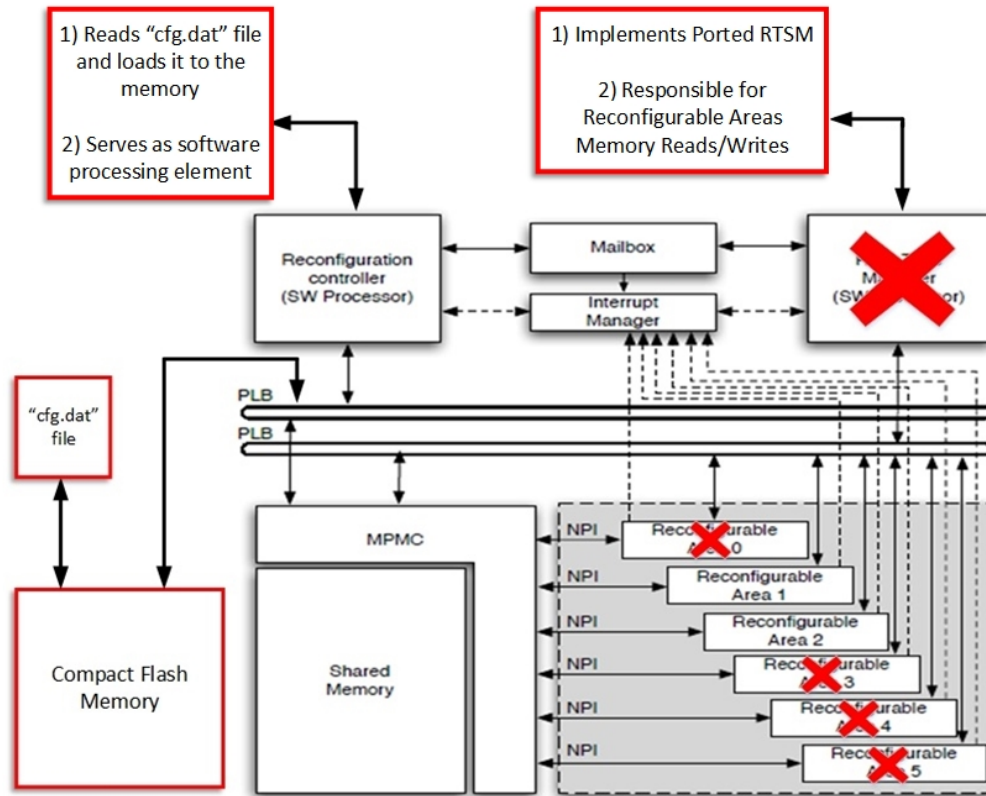


Figure 5.3: Full Embedded System Architecture. The red "X" marks modules that have been erased in our architecture.

To achieve the implementation and integration of the RTSM on one of the most known partially reconfigurable FPGA systems, the XUP-V5, we followed a sample architecture and customized it in order to fit our purposes. Also small changes had to be made in the core code of the RTSM as to customize the initiation of HW/SW execution and reconfiguration. Two separate architectures were created, whose only difference was where the RTSM would be placed, MB2 or host PC.

5.3 The SPEAr Platform

In this part first the hardware part of the SPEAr 1310 development board is presented and then we present the needed software in order to use the device.

5. IMPLEMENTATIONS OF THE RTSM

5.3.1 The hardware side

The SPEAr 1310 is a prototyping platform developed by STMicroelectronics. The board has been developed by STM with the purpose of being used for feasibility studies. In particular to evaluate software systems or application developed for embedded platforms and how these systems or applications could be benefitted by hardware acceleration. The SPEAr1310 is a member of the SPEAr family of embedded MPUs for network devices. It offers an unprecedented combination of processing performance and aggressive power reduction control for next-generation communication appliances. The SPEAr1310 is based on ARM's new multi-core technology (Cortex-A9 SMP/AMP) and it is manufactured with ST's 55nm HCMOS low power silicon process.

SPEAr1310 targets cost and power sensitive networking applications for the home and small business as well as telecom infrastructure equipment, with lowest overall leakage under real operating conditions. The device integrates ARM's latest generation ARMv7 CPU cores, ST's proven C3 security coprocessor, and advanced connectivity interfaces and controllers.

So far its primary target are Research & Development projects that create customized architectures of systems and/or applications. The goal of these R&D projects so far has been the exploration of possible uses for the SPEAr board, creating feedback for the ease of use of the accompanying OS and finally enabling new features not yet developed by STM. This work has worked towards the later, first enable PR on the SPEAr board coupled with the FPGA daughter-board and then implement the RTSM that could manage all the resources HW and/or SW present on the platform.

The SPEAr board circuit diagram is shown in figure 5.4. As seen the board utilizes an ARM dual-core Cortex-A9@666MHZ processor with two levels of caching:

- 32+32 KB L1 Instructions/Data cache per core with parity check
- Shared 512 KB L2 cache (ECC protected) with parity check

The board also contains a 256MB DDR3 serving as Central Memory. The memory communicates with the ARM processors via an AXI 64 bits bus. Having the ability to support many peripherals the board is equipped with a variety of interfaces allowing them to be connected to the main board through standard interfaces like USB, Ethernet

or UART. With the use of the EXPansion Interface (EXPI), it is possible for the user to plug in dedicated boards that can expand the SOC with customizable hardware.

This connection's communication is driven by the AHB-Lite protocol. The AMBA AHB-Lite protocol is used to address the requirements of high-performance synthesizable designs. It is a bus interface that supports a single bus master (SPEAr board) and provides high-bandwidth operation. The AHB-Lite protocol implements features required for high-performance systems like:

- burst transfers
- single-clock edge operation
- non-tristate implementation
- wide data bus configurations.

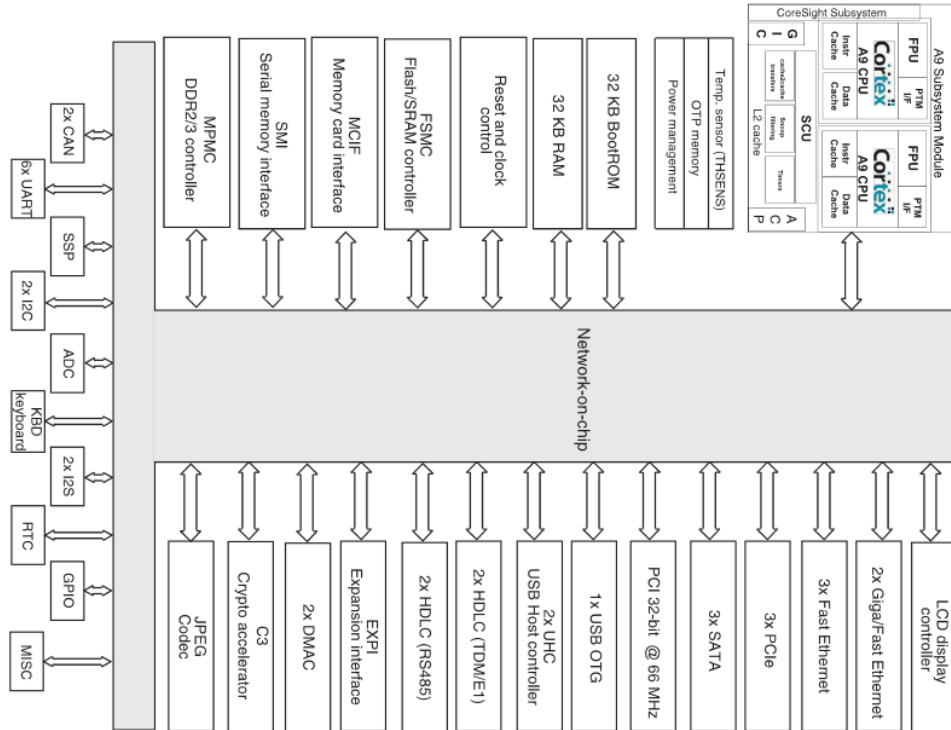


Figure 5.4: Circuit diagram for the SPEAr 1310 board.

5. IMPLEMENTATIONS OF THE RTSM

In order to understand further the AHB-Lite bus we show the diagrams for the Master and the Slave figure 5.5 a and b respectively [28]. The AHB-Lite master provides addresses and control information to initiate read and write operations with the slave. Then the slave responds to the requests.

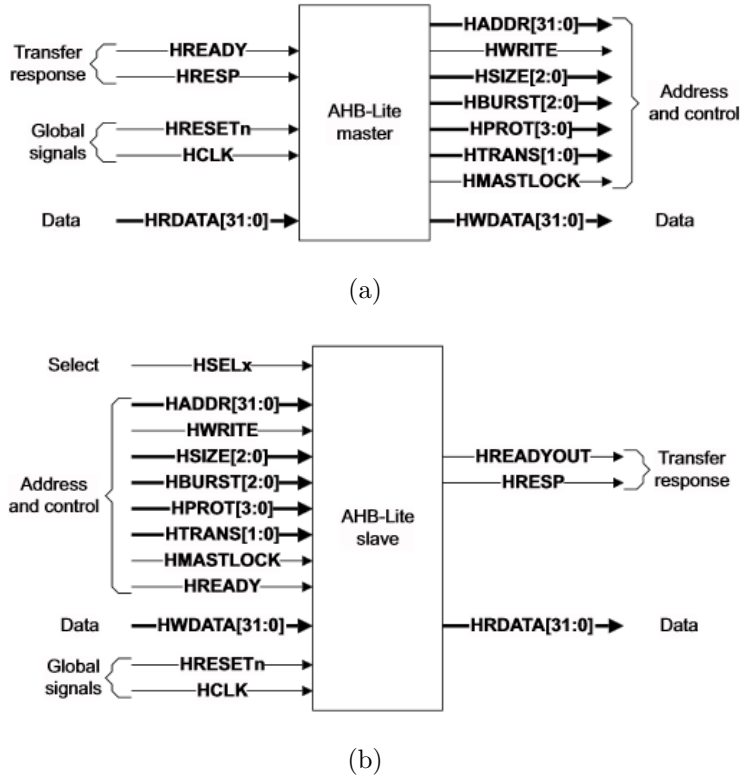


Figure 5.5: Master (a) and Slave (b) Interfaces for the AHB-Lite communication bus.

The interconnection scheme of the system is shown in figure 5.6. The AXI protocol defines double data channel and the read and write can be done simultaneously. However the AHB has only one data channel and the read and write operation is impossible to be executed in parallel, thus making performances of AHB bus lower than the AXI one. Moreover, the internal interconnection crossbars, which convert the AHB bus to the AXI one and vice-versa, introduce a physical limitation that also has an impact on the performance. This further delay is because of the conversion crossbar and AHB bus increasing transmission latency. This may create bottleneck effects when multiple transmissions are taking place.

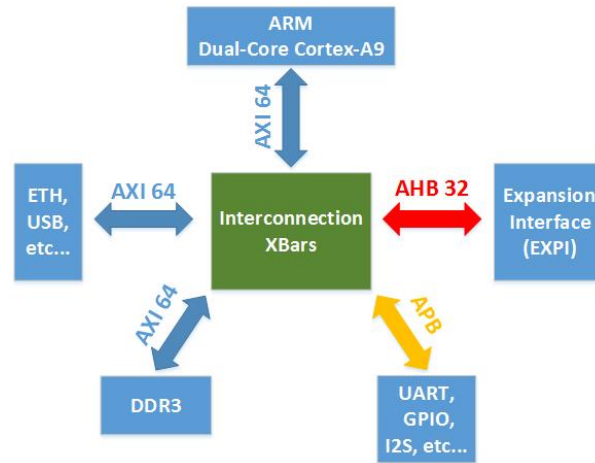


Figure 5.6: Interconnection scheme of the SPEAr platform.

5.3.2 The software side

The SPEAr board is an embedded platform that works in combination with an OS. The OS is a customised Linux distribution by STM called STLinux. The SPEAr boot process is divided into four different phases:

- On power-on of the board the Boot-ROM, which is hard coded on the silicon (eROM) starts. The eROM locates the XLoader and transfers the control to it.
- XLoader initializes the DDR memory and loads to U-boot to execute it.
- U-boot prepares the environment to boot Linux (loads the filesystem, retrieves the Linux image from the memory, etc.).
- Finally Linux is booted and takes over system's control.

A graphical representation on how the booting sequence of the SPEAr board works and specifically the Boot-ROM is shown in figure 5.7. The Boot-ROM is embedded in the silicon (SoC) and it is not part of the Linux Support Package loaded in the SPEAr's memory. The XLoader is a small firmware, loaded in the second stage from the eSRAM. The main actions performed by it are:

- Initialization of the DDR and PLLs.

5. IMPLEMENTATIONS OF THE RTSM

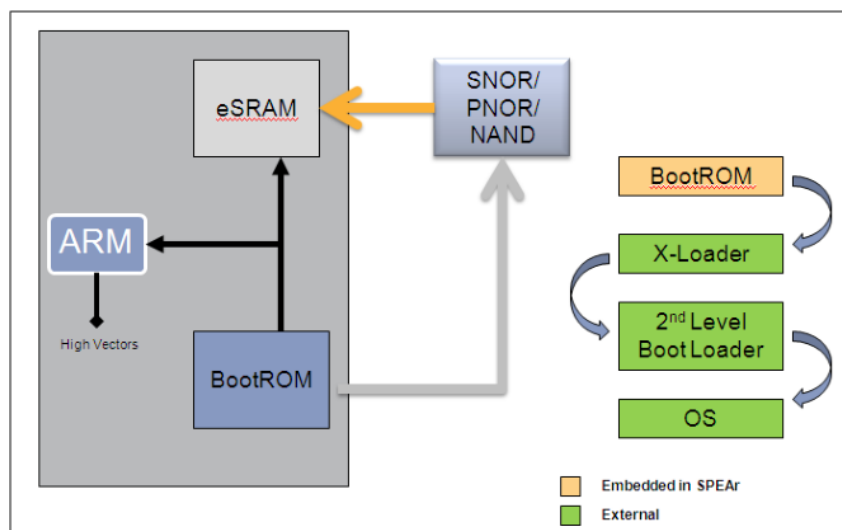


Figure 5.7: BootROM Architecture.

- Loading the U-boot from the NAND, NOR memory depending on the boot type selection made by the user.

The U-boot is an open source boot monitor, used right after powering up an embedded system. It can be used to either monitor the system for development and debug purposes, with the use of simple command input via a .txt file or through the U-boot console, or to boot an OS with the setting of environment variables, loading the filesystem and locating the Linux image.

Finally the STLinux OS is a customization of the known Linux operating system for the STM embedded micro-processor platforms like SPEAr. The STLinux contains all the necessary drivers in order to for the user to have complete control of the SPEAr platform. The STLinux communicates with a host computer via the UART port, Ethernet or USB, by opening a command prompt.

5.3.3 The FPGA daughter-board

In order to expand the SPEAr board with customizable hardware STM ships is with the EVALSP13XXFPGA. This is an expansion board based on an Xilinx[®] Virtex-5 series LX FPGA. The daughter-board is used for development and verification of its reconfigurable array system. The daughter-board is taking power through the EXPI connection with the

SPEAr. Also more expansion cards can be connected to the other end of the daughter-board.

The board currently used contains a Virtex-5 LX110 FPGA with a Flash PROM XCF32P, which provides flexible updating and revision of the designs. The programming of the FPGA is done through a standard JTAG cable and the Xilinx iMPACT program. The Virtex series are partially reconfigurable enabled by Xilinx. A most common way to achieve PR on a Xilinx FPGA is by accessing the Internal Configuration Access Port (ICAP). In the common case a Virtex-5 packed by Xilinx has a number of I/O peripherals such as, SD card Reader, UART interface, LEDs and switches, etc. The Virtex-5, packaged by STMicroelectronics, I/O options are only through the EXPI to the SPEAr board. The board has 417 I/O signals directed to and from the SPEAr board. The communication of the FPGA and the SPEAr is done through the AHB-Lite bus which was explained.

5.3.4 Partial Reconfiguration on the SPEAr

In order to achieve PR on the Virtex-5 daughter-board the designer has to overcome the issue of data transfer between the SPEAr and the FPGA. In order to achieve that a DMA engine implemented in hardware and provided by STMicroelectronics was used. The DMA implements a two way channel between the SPEAr and the FPGA boards [29]. Specifically the data are being transferred to hardware cores implementing hardware functions, residing in the reconfigurable fabric of the FPGA. The DMA interface interprets the AHB-Lite incoming and outgoing signals and according to the operation requested by the master does one of the following:

- Stores the incoming data, placed on the signal HWDATA by the Bus Master, on FIFOs, for them to be used later by the hardware accelerator.
- Writes the outgoing data to the HRDATA signal, which goes to the Bus Master. Then the AHB-Lite writes the data to special addresses on the SPEAr's Central Memory, specifically marked by the designer as the DMA registers.

The Virtex series are partially reconfigurable enabled by Xilinx. A most common way to achieve PR on a Xilinx FPGA is by accessing the Internal Configuration Access Port

5. IMPLEMENTATIONS OF THE RTSM

(ICAP). However as stated already the daughter-board's only mean of communication is through the EXPI. This sets a huge limitation for the designer, who has to transfer the reconfiguration data from an external memory connected to the SPEAr, e.g. a USB stick, to the FPGA. In order to do that we create another core that the DMA can send data to and in this core we implement the ICAP controller and the needed hardware to control it. In the architecture presented in figure 5.8 we see that the DMA transfers data to and from the two Reconfigurable Regions present on the FPGA and the ICAP component.

In order to make a DMA transfer we use a software driver created by Politecnico di Milano. The software driver is loaded through the bash console. The driver provides the user with simple functions that initialize and perform DMA transactions, the driver is implemented in C-language. The software functions implemented by the driver concerning the DMA are:

- *fstart_transfer*: Used to initiate the transfer and also informs the DMA controller which RR/core the data are being transferred to or from.
- *fwait_dma*: Used to inform the OS when the dma transfer has ended.
- *fwrite_buffer* and *fread_buffer*: Used from the user write and read the buffers accommodating the dma transfer data.

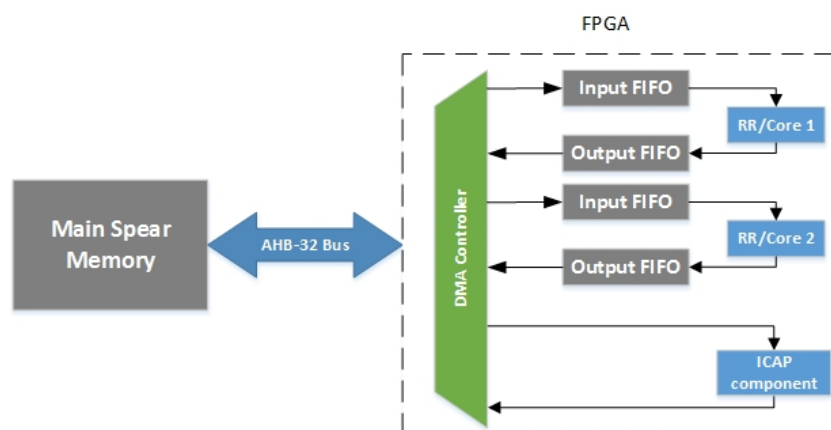


Figure 5.8: The architecture used to enable communication between SPEAr and the Virtex-5.

The DMA component is implemented using VHDL language. The DMA component sends the data to input and output data FIFOs for each core. For each core to understand whether the DMA request refers to it the DMA implements a handshake master/slave protocol, by simplifying the AHB-Lite. Each core has its own signals that signal whether the data present on the FIFOs are valid or not. The user through the aid of the driver's dma functions initiates a transfer, by setting an integer parameter on the *fstart_transfer* arguments, which is the selection of the core the transfer refers to. This by the means of the AHB-Lite protocol sets the HSELx signal to a certain value and then with the DMA component the interpretation of this signal controls the multiplexer that sets all the signals of the handshake protocol for all the RRs/cores.

Xilinx offers many ways to achieve partial reconfiguration in their marketed FPGAs. By offering many primitives each one with unique reconfiguration capabilities and timing constraints and diagrams. The most common primitive is the ICAP controller mentioned earlier, which is located on the fabric's side, provides the user logic with access to the Virtex-5 configuration interface.

There are two ways to access the ICAP. The first and easy option is to include in the hardware design a MicroBlaze micro-processor. The MicroBlaze processor and the ICAP controller are connected to PLB bus, then with the use of Xilinx kernel software functions and MicroBlaze-C language (a C customization) reconfiguration data can be sent through the fast PLB bus. The ICAP primitive and the MicroBlaze micro-processor are instantiated with the use of the Xilinx Platform Studio (XPS).

The second more complex option of accessing the ICAP is via an FSM, which will control directly all the input and output signals to the primitive. In [30] all the timing constraints and control signals for the ICAP are presented. figure 5.9 depicts the sequence for transferring reconfiguration data to the ICAP is described, which can be implemented by a simple FSM. In this case the ICAP primitive is instantiated directly inside the VHDL source code.

In order to explain the way reconfiguration works in the ICAP primitive we will briefly explain figure 5.9. Signal RDWR_B is active low and is driven by the user, setting the ICAP ready to read data from signal DATA. The signal CS_B is also controlled by the user, once the signal is drive low for the first time the first 32-bit reconfiguration will be written by the ICAP on the second rising edge of the CCLK, after that one word is written on each rising edge. If the CS_B signal is asserted then whether or not new data

5. IMPLEMENTATIONS OF THE RTSM

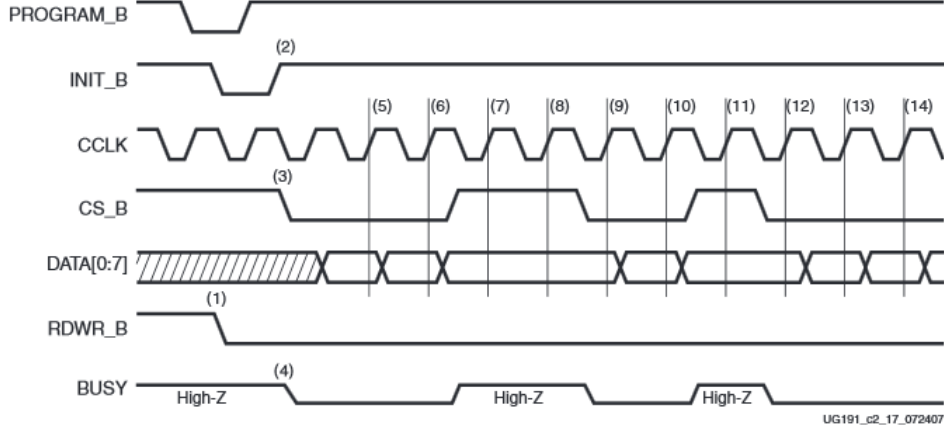


Figure 5.9: Non-Continuous ICAP Data Loading time diagram.

arrive the ICAP will not write them. After a "pause", i.e. CS_B signal is '1', a word is written at each rising edge of the clock. It is important to note that the incoming words must appear to the ICAP on the descending edge of the clock in order to be written on the immediately following rising edge.

The use of the XPS is well documented and Xilinx has done a great effort to make the PR flow easy for the user to follow in order to design his own partially reconfigurable designs. However in our case due to the wide range of different clocks, i.e. ARM666MHz, AHB-Lite166MHz, ICAP and the MicroBlaze clocks, huge synchronization and clock skew issues presented. So the exploitation of the easy-to-use MicroBlaze processor was deemed impossible. Thus the choice of access to the ICAP controller was chosen to be the direct access via FSM to the ICAP. A FIFO solely dedicated to storing the reconfiguration was created inside the larger ICAP component. This FIFO would store 15 words at a time, once this limit was reached these 15 words were sent to the ICAP, after applying the bitswapping technique mentioned on [30].

The choice to make transfer packets of 15 words was made due to a limitation found at the later stages of designing the architecture on the software side of the DMA. Despite the fact that the creators of the driver mention that the DMA transfer can transfer up to 1KB of data in a single burst, the case was that this burst was divided again in 15 32-bit words packets. This created again several synchronization issues with the ICAP and the adding and removing of the DMA and the ICAP FIFOs. The architecture of the ICAP component we created is shown in figure 5.10.

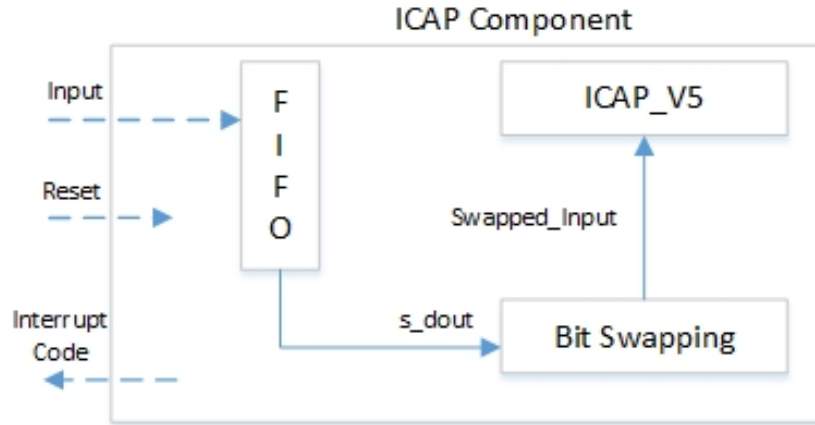


Figure 5.10: The architecture of the ICAP component.

To control the behavior of the ICAP component we created another FSM which works in communication with the FSM controlling solely the ICAP primitive. This FSM depending on current state of the system, decides the next, and also controls the CS.B and RDWR.B signals, responsible for the correct reconfiguration of the device. The FSM's diagram is shown in figure 5.11.

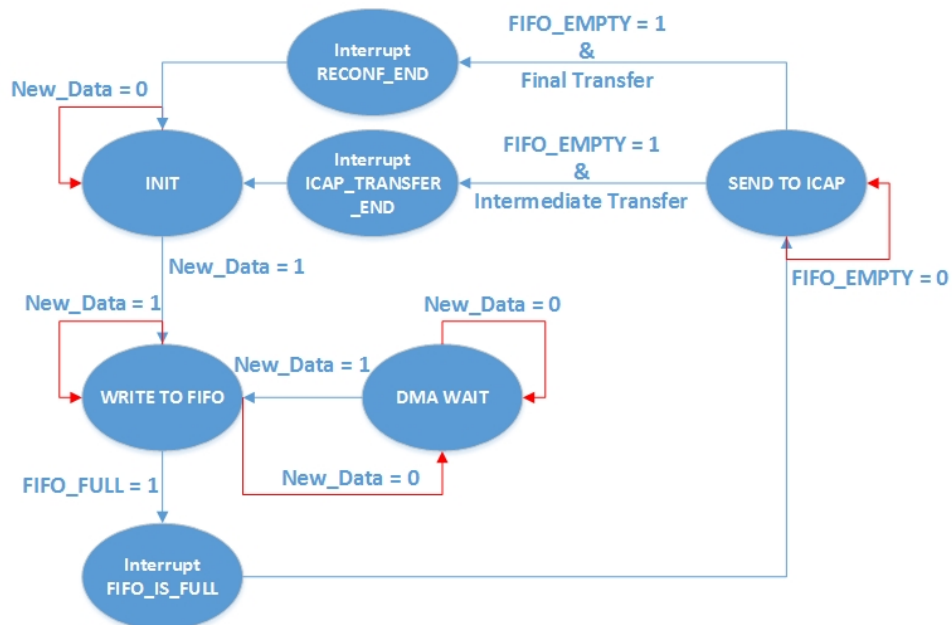


Figure 5.11: The FSM controlling the reconfiguration process.

5. IMPLEMENTATIONS OF THE RTSM

During the initialization state, the system waits for new data to appear on the DMA FIFO, specifically the data have to be about the ICAP core, i.e. the `fstart_transfer` to have the appropriate core selection, in this architecture we have set the ICAP core to correspond to the selection 3 by the user. Once new data arrive, the FSM begins to write new data to the intermediate FIFO, upon completion the ICAP component produces an interrupt, sent directly to SPEAr and managed by the function that controls the software side of the reconfiguration process. If for any reason the DMA transfer stops, the FSM enters a wait for new data state, until new data appear again in the DMA FIFO.

Then the FSM begins the reconfiguration phase by sending data to the ICAP controller, as soon as all the data, residing on the intermediate FIFO have been transferred, the ICAP component produces another interrupt. The value of this interrupt depends on whether or not this reconfiguration data transfer is the last or not. If this is the last transfer, the interrupt produced informs the SPEAr and by extension the reconfiguration function for the reconfiguration ending point, if not, the interrupt denotes the ending of an intermediate data transfer. While the reconfiguration process is active, this FSM prohibits other cores on the FPGA to perform a data transfer through the DMA, this is necessary to ensure the validity of the reconfiguration process. It is strictly denoted in the Xilinx PR flow that while a reconfigurable region undergoes partial reconfiguration the I/O ports must be decoupled with the use of tri-state buffers. Finally, when the reconfiguration ends the ICAP component resets the newly reconfigured region, as dictated by the Xilinx user guides. The interrupt codes for the three new interrupts used during the reconfiguration process are:

- Interrupt `FIFO_IS_FULL=15`
- Interrupt `ICAP_TRANSFER_END=13`
- Interrupt `RECONFIGURATION_END=12`

By implementing all the above we have created a component that when coupled with the DMA engine, enables the Partial Reconfiguration utility on the EVALSP13XXFPGA expansion card for the SPEAr board. Also the method for accessing the ICAP implemented in this work, allows to take full usage of the ICAP bandwidth. However this

advantage is greatly hindered by the really slow DMA transfers, necessary for transferring the reconfiguration data, i.e. the partial bitstream, from the USB memory to the FPGA.

We also implemented a software program/function that opens the partial bitstream file and begins 15 words transfers to the FPGA using the software driver's functions explained. It also monitors the interrupt signals send to the SPEAr by the FPGA regarding the state of the reconfiguration process.

5.3.5 The RTSM implementation on SPEAr

Now that the PR utility was achieved in the SPEAr board, it is able also for the RTSM to be implemented. The RTSM as said ultimately controls the hardware and software resources residing on a board. Another resources that needs management in our case is the DMA engine and the ICAP primitive. In addition when the RTSM decides on the execution of a hardware task, it needs to take control of when the data transfers will occur.

Also we have to consider that the user may also include software implementations of task in the target application. Considering now this the RTSM has to manage hardware and software tasks that all need access not only to the FPGA but also the ARM processors:

- A software task SW1 that executes on an ARM processor residing on SPEAr.
- A hardware task HW1 that needs to be partially reconfigured on the FPGA using the DMA engine.
- A hardware task HW2 that performs data transfers of inputs and results to and from the FPGA using the DMA engine.

As seen it is impossible with two cores to perform all these operations simultaneously, since the SPEAr board offers one dual-core ARM processor. Also we have to consider the core the RTSM will run on and the various processes created by the STLinux OS. This problem urged us to set aside the direct mapping and scheduling of threads to software processing elements according to the RTSM's decision and use the time-sharing scheduling of processes done by the STLinux OS. With that way the access to the DMA

5. IMPLEMENTATIONS OF THE RTSM

engine could be more fair instead of having several threads waiting in order to gain access to it.

A simple DMA transfer code is shown in Algorithm 1:

Algorithm 1 A DMA transfer.

```
fstart_transfer(0, WRITE, size_out, 1);  
fwait_dma();  
fread_buffer (buffer_out, size_out, 0);
```

Considering a scenario where, HW1 executes the first instruction in this algorithm and then its timeshare on the ARM core is over followed by HW2, we can observe that the DMA transfer code has to be considered as critical piece of code and be accessed by a task holding a semaphore on that piece of code. This semaphore must be obtained by the task regardless of whether this transfer is done for reconfiguration purposes or execution ones. Hence Algorithm 1 after considering that new tasks have to be executed as separate processes and that the access to the DMA has to be done with the use of a semaphore changes to Algorithm 2, which shows the spawning of a new task by the RTSM. As said during reconfiguration all transfers stop.

Algorithm 2 Spawn of a new task

Input:

Scheduling structure

```
sem_wait(sem);    //P operation  
DMA_Reconfiguration_Transfers;  
sem_post(sem);    //V operation  
char *args[] = ".executable_Of_Taski", 1, (char*)0;  
child_pid1 = fork();  
if child_pid1 < 0 then  
    printf("Fork error");  
else[child_pid1 = 0]  
    execvp(".executableOfTaski", args);
```

The *execvp()* function changes the executable run by the forked process, the arguments passed to the *execvp()* function through the *args[]* array specify the executable according

to the task being scheduled. The user creates as many executables as needed for the RR available, only if there is a region specific software code, e.g. in our case the software code for a hardware task residing and making DMA transfers is different for RR1 and RR2 since the arguments in the *fstart_transfer()* are differentiated. If we are considering a data transfer for a HW task then we refer to Algorithm 3.

Algorithm 3 Data DMA transfer

```
sem_wait(sem);    //P operation
fstart_transfer(0, WRITE, size_out, 1);
fwait_dma();
fread_buffer (buffer_out, size_out, 0);
sem_post(sem);    //V operation
```

5.4 Conclusion

In this chapter the two implementations in different reconfigurable platforms, of the RTSM were presented. In order to successfully port the RTSM in the two boards we had to make modifications in the reconfiguration or the task execution processes invoked by the RTSM. Also a novel architecture had to be conceived in order to achieve for the first time Partial Reconfiguration on the SPEAr board. In the next Chapter the evaluation of the correctness of both designs with real applications will be presented as well as the results and timings obtained by the experiments made.

5. IMPLEMENTATIONS OF THE RTSM

Chapter 6

Experimental Results

In this chapter the results obtained by the experiments on the two partially reconfigurable platforms, are presented. The experiments were done with the use of real-life image processing applications. The applications used were an Edge Detection and an Image RayTracer applications.

6.1 The XUP-V5 experimental results

The performance of the RTSM on the XUP-V5 platform was evaluated through experiments with a real application implemented in hardware. The application chosen was an Edge Detection application. The Edge Detection application consisted of 9 tasks.

Its task graph is rather simple and tasks have a linear dependency. In specific, tasks execute in 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. Gray Scale (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. Figure 6.1 depicts the task graph of the application, showing only task dependencies but not how it actually executes over time. It also depicts the (implicit) SW tasks that trigger the reconfiguration process.

Now that the application task graph is available we need to decide on the codes that will be generated by the RTSM in order to signal the execution of each task and the re-

6. EXPERIMENTAL RESULTS

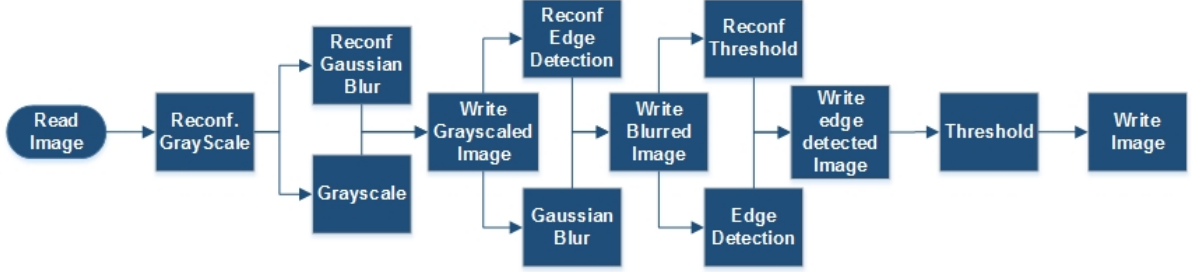


Figure 6.1: The Edge Detection application task graph.

configuration of the partial bitstreams. Also the type and number of available bitstreams has to be decided. The implementation of the hardware tasks for the application is presented in [27]. The specifics of the hardware tasks implementation will not be mentioned in this thesis.

The design implemented included 2 reconfigurable regions. The bitstreams created are the following:

1. Grayscale.bit corresponds to RR1.
2. GaussianBlur.bit corresponds to RR2.
3. EdgeDetection.bit corresponds to RR1.
4. Threshold.bit corresponds to RR2.

Also every hardware task has a software implementation in order to be able to execute on the SW-PE. Having constructed the task graph and determined all the operations the FPGA will have to execute, we can decide on the coded instruction the RTSM will send. These instructions will be send either through the RS-232 or through the MailBox. In this case we had to create 4 instructions for hardware execution and 13 for software tasks, either execution or reconfiguration, since reconfiguration is executed on the SW-PE MB2. The created instructions are:

- **H1, H2, H3, H4:** signals the beginning of execution for the four hardware tasks.
- **S1C_i:** signals the reconfiguration of the i_{th} hardware task.
- **S2_i:** signals the execution of the read image software task.

- **S3W_i**: signals the intermediate write image software tasks.
- **S4H_i**: signals the execution of the software implementation of the i_{th} hardware task.

After deciding on the codes we created an extra function in the RTSM code that depending on the scheduling decision reached will decide on the instruction to be sent on the MailBox or the RS-232. The new function is a large **switch-case** structure that decides on the instruction. The issue encountered was that the **switch-case** structure is expensive in terms of MicroBlaze memory.

The issues regarding the MicroBlaze memory mentioned in Chapter 4, were dealt by first executing the application in a simulation environment. As we said it was a design choice to not change the available memory of the MicroBlaze processor. So we decided to disable some features of the RTSM that we observed in the simulation that are not used during the execution of the Edge Detection application.

The disabled features was the Relocation Alternative. Firstly this feature was not used due to the mappings created for each task. Additionally the Relocation is not yet fully implemented in partially reconfigurable platforms [5]. Having this feature disabled the size of the RTSM code was reduced and able to fit in the available memory.

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 MB1 processor by sending one interrupt per pixel. Then, MB1 sends the processed pixel to the corresponding RR. This mechanism is very slow, and the total execution of the application takes roughly 2 minutes per image. We have verified that the main cause for the long execution time is indeed the pixel fetching mechanism.

With the above application 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 clocked at 100MHz.

To have a clear image of the RTSM overhead we break down the operation of the RTSM into several phases and measure each one separately. The phases determined are:

- RTSM initialization: Includes the fetching of the `cfg.dat` file from the CF memory and the initialization of the structures.

6. EXPERIMENTAL RESULTS

- Schedule decision: Time spent for the scheduler to reach a schedule decision.
- Initiate Command: Time spent for either HW or SW execution or reconfiguration command initiation.
- List Update: Time spent for the RTSM to update its lists after the completion of an operation (HW/SW task execution, reconfiguration).

During the execution of the RTSM each phase is executed more than one time. In the target application during the simple test, i.e. one execution of the whole application, each phase is reached 9 times. In Table 6.1 we can see apart from the aggregate time of each phase, the average time to reach and execute each phase of the RTSM just one time.

Table 6.1: RTSM phases and aggregate overhead.

RTSM Phases	# times the phase is reached	Elapsed Time (usec)	Average Time (usec)
RTSM initialiaza-tion.	1	39,761	39,761
Schedule decision	9	4,726	525
Issue Command (HW/SW Exec or Reconfiguration)	13	138,119	10,624
Lists Update (HW task completion)	4	3,152	788
Lists Update (SW task completion)	5	4,175	835
Lists Update (Re-configuration completion)	4	153,397	38,349

We can see that the longest average time is the initialization stage, which is normal since the input file has to be transferred from the CF memory to the MicroBlaze memory. The scheduling decision time is taken very quickly, measured at an aggregate of 4,726 microseconds. The largest times are also observed in the list updating after a reconfiguration completion and a command issue code segments. The later time is understandably large, since the MB1 has in two out of three commands go through the MailBox and wait for the MB2 to acknowledge the receiving of the command.

For the reconfiguration completion time we have to examine the steps taken after. First the RTSM issues a HW task execution command and then depending on whether the prefetching mechanism takes place it issues also a reconfiguraiton command. So in this reconfiguration completion time we have two Issue Command times, thus making the true reconfiguration completion time: 17,101 microseconds.

After evaluating and confirming the correctness of the architecture we continued experimenting in order to unlock and enable several other RTSM features. Another test regarded the enabling of the micro-reconfiguration feature. A detailed analysis of how the micro-reconfiguration was implemented and included on the current architecture can be found in [27]. We verified the ability of the RTSM to handle micro-reconfiguration instructions with issuing an instruction to change the Threshold level midst the Threshold's task execution. The default threshold level is 20 and the micro-reconfiguraiton instruction set it to 15, this resulted in the re-execution of the threshold task and the creation of two different output images, one with threshold level 20 and one with 15.

The final experiment run was to verify the ability of the RTSM to dynamically change its behavior, when running the application multiple times if that is needed. In order to achieve that we set the number of times the application will be executed as 4. A simple assumption we make when creating the RTSM's input file is that the hardware implementation of a task will execute faster than the software one. Table 6.3 depicts the estimated execution times for hardware and software execution for the Edge Detection application.

Tasks	Execution Estimated Time	
	HW (us)	SW (us)
Gray Scale	887	4,435
Gaussian Blur	904	4,520
Edge Detection	782	3,910
Threshold	476	2,380

Table 6.2: Estimated HW/SW execution times given as input to the RTSM.

This assumption is based on the common assessment that hardware execution will be faster than software, however in this application due to the way individual pixels are fetched to the RR from the CF memory, the hardware execution of a task is considerably slower than the software one, this being the case for all the hardware tasks of the application.

6. EXPERIMENTAL RESULTS

Running the application multiple times results in a dynamic change of the execution times. After the first time the application is executed the RTSM notes a considerable difference between the estimated and the actual execution time of the hardware version. During the second time the RTSM will schedule the application, it will be also noticeable that the estimated software execution time is faster than the actual hardware one, this will prompt the software execution of the task. After the second completion of the application's execution the RTSM will also have a measurement of the actual software execution time for each task. In the Edge Detection's application case this software execution time is smaller than the hardware, so for the next 2 executions the RTSM will continue issuing the execution of the software version of the tasks, averaging the new measured times after each execution. The execution flow for two consecutive executions of the application is shown in figure 6.2, it is clear the change during the second run of the execution.

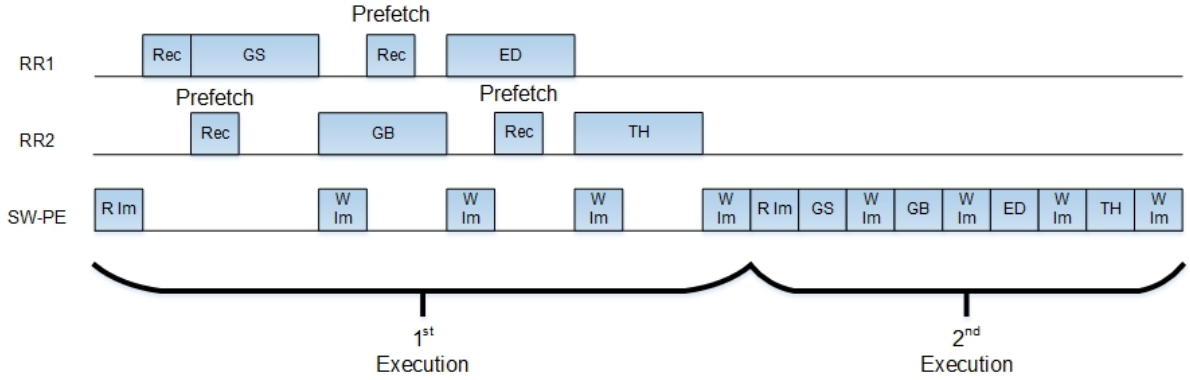


Figure 6.2: Execution flow for two consecutive runs of the Edge Detection application.

Considering a streaming application, if at any point and for any reason, e.g. a increase in the device's MicroBlaze temperature that resulted in a lower clock frequency, the measured software execution time becomes bigger than the hardware one the RTSM will switch again the decision issuing a hardware task execution. This dynamically changes in the RTSM's behavior is, to the best of our knowledge, the first time it is documented with the use of an actual application.

6.2 The SPEAr experimental results

In order to evaluate the correctness of the PR design created for the STM SPEAr board we choose two different applications from the field of image processing. This experiment will also demonstrate the ability of the RTSM to manage more than one applications sharing the resources on one single platform. The first application is an Edge Detection application, run exclusively on SW, the second is a RayTracer application provided by [29].

The Edge Detection application is a rather simple one, similar to the one used to evaluate the XUP-V5 platform. The only difference is the absence of the intermediate write image tasks. The other tasks consisting the application are the already known, Gray Scale, Gaussian Blur, Edge Detection, Threshold and Write Image tasks. As seen, the application does not offer task parallelism, since every task has as input the output of the previous one, we split the image in 4 parts in order to achieve data parallelism. This way the RTSM can spawn four Gray Scale tasks that can run in parallel fashion on the SPEAr's two ARM cores.

The RayTracer application has a combination of HW and SW execution. In the beginning the application is run on a SW-PE and then switches its execution in HW accelerators able to run on the partially reconfigurable FPGA fabric. The RayTracer application is created by STMicroelectronics and starts from a description of the scene as a composition of 2D/3D geometric primitives. The basic primitives supported by the algorithm are the following: squares, spheres, cylinders, cones, toruses, and polygons.

Each of the mentioned primitives is described by a set of geometric properties, e.g. the position in the scene, the height of the primitive or the rays of the circles composing the primitive. The complete image produced by the RayTracer algorithm is shown in figure 6.3. In the case considered for the experiments made we use only three of the mentioned primitives, the sphere, the torus or ring and the square, because for this primitives we had a HW implementation created by the Vivado Suite with the use of the SW original algorithm. The logic utilized by each of these three HW cores is shown in Table 6.3, the results presented are produced by the PlanAhead Suite, used in our case to create the partial bitstreams needed for the design.

6. EXPERIMENTAL RESULTS

Table 6.3: Summary of the logic resources used by the HW cores.

Core	LUT	FF	DSP	BRAM
Sphere	7,051	4,763	15	2
Ring	5,466	3,372	18	2
Triangle	6,168	3,432	24	4

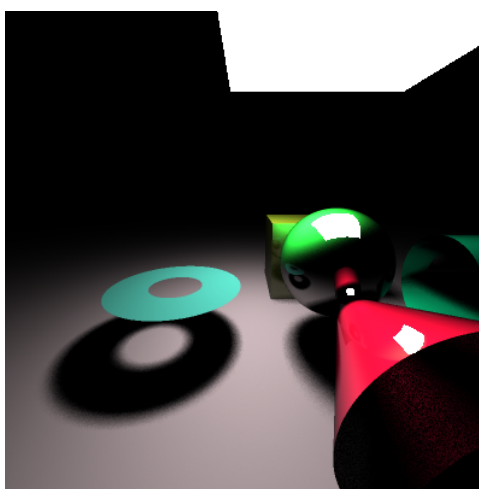


Figure 6.3: Output image of the RayTracer application with all the primitives.

It is clearly observed that the critical hardware resource for creating the partially reconfigurable regions are the DSP needed. Considering also the fact that we want to have equally sized RRs in order for the RTSM to be able to place every primitive's design to every RR. Thus we need 3 RRs with at least 24 DSPs at each, however Virtex 5 LX110 FPGA used by the EVALSP13XXFPGA daughter-board is has only 64 DSPs available, which limits our design to only two RRs.

In order for the RayTracer application to run the user executes an executable in software and applies certain flags in order to denote, which primitive the scene will contain, and where the primitive creation would be run (SW or HW). In the static design each primitive's hardware implementation is statically bound to a certain region on the other side of the DMA engine. So if the user wants to create the sphere primitive, the selection signal, regarding the DMA transfers, would be explicitly be set to the value corresponding to the sphere HW core. In our case we have to consider which primitive we want to create and in which core/region the bitstream of this core is placed. To achieve

that another flag is passed to the execution of the RayTracer, denoting the core/region. So an execution bash command of the RayTracer would look like this:

```
./stRayTracer -s -f settings_sphere.txt -p sphere.ppm -r 2
```

This command executes the sphere primitive, -s flag, with input file the sphere settings file, on RR 2 and creates an image with the name sphere.ppm.

The task synchronization over the DMA engine and the creation of the tasks, especially the tasks of the RayTracer application, which use a separate executable, were done with the use of the Algorithms 2 and 3 shown in Chapter 5. Since the resources of the SPEAr board are quite limited, 2 SW-PE and 2 RRs, all the abilities and fine-grain resource management offered by the RTSM deployed, could not be exploited. Thus the built-in Linux time-slicing scheduler was also used.

In order to execute two applications a merged Task Graph was created. For the RayTracer application a 4 tasks task graph was created, to display the parallel execution on two RRs and the sharing of the DMA engine. The resulting task graph is shown in figure 6.4.

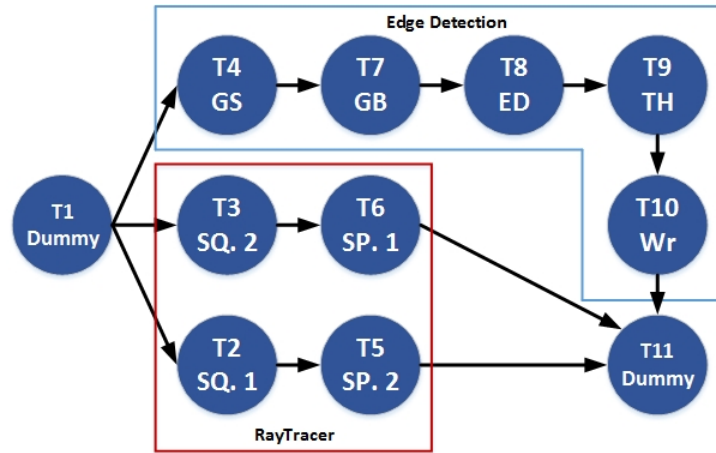


Figure 6.4: The merged task graph. Edge Detection tasks are executed in SW, RayTracer tasks are executed in HW.

The first and last tasks are dummy tasks used to synchronize the beginning and the ending of the two different applications. The RayTracer application tasks are the SP. and SQ., corresponding to either the sphere or the square primitive. Each task creates

6. EXPERIMENTAL RESULTS

an image with the name of the primitive and the region the task was executed in, e.g. sphere.RR1.ppm. To better display the advantages of PR the tasks SP.1 and SP.2 are considered as two different instances of the same sphere primitive task, that have to be run on RR1 and RR2 respectively, thus forcing the reconfiguration process to take place. If not for the differentiation the RTSM would reuse the already present, sphere and square cores, the second time the tasks would require execution, without the use of reconfiguration.

The experiments verified the correctness of the PR architecture implemented. The decision made by the RTSM are in accordance with the ones described in Chapter 4. The execution flow of the application is shown in figure 6.5. Specifically the RTSM performs the prefetching action for the hardware task SP.1 in order to mask the reconfiguration time. Also we can observe that for almost all the software tasks the reservation alternative takes action in order to the RTSM to reach a scheduling decision before the resource becomes available.

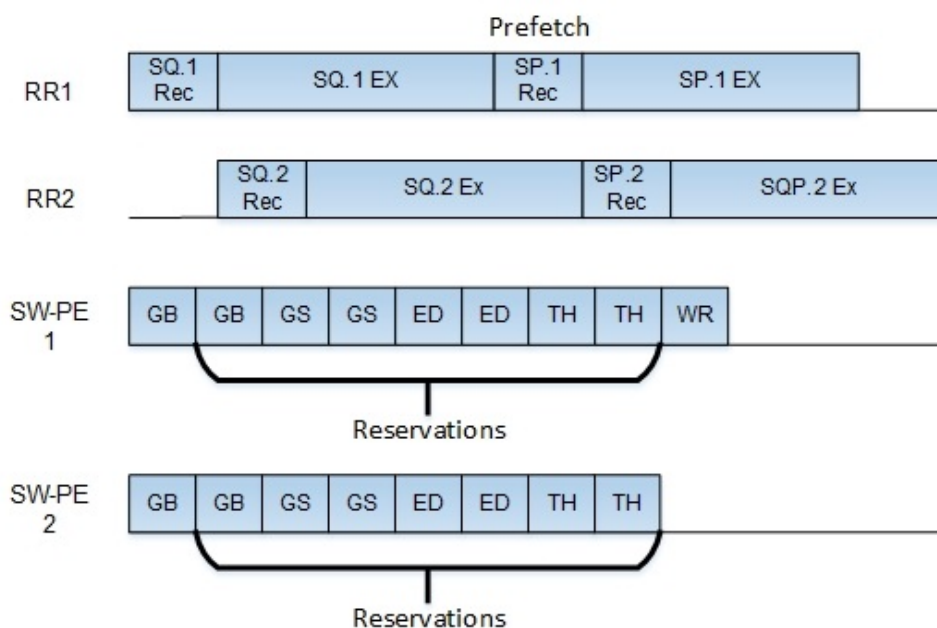


Figure 6.5: Execution flow of the Edge Detection and RayTracer application.

However the figure does not depict the actual time taken to execute the two applications. The overall execution time of the two applications is 2 minutes and 8 seconds.

It is important to note though that the Edge Detection application finishes its execution much earlier, 26.781 ms, and then simply waits for the RayTracer to finish execution. As seen the reconfiguration overhead is masked by the hardware execution of the RayTracer and the SW execution of the Edge Detection, which compared to the overall execution time is negligible. The measured throughput for the reconfiguration process through the slow DMA bus is 1.66MB/s which is considerably slower than the 50MB/s offered by the ICAP. The applications produce three resulting images which are shown in figure 6.6, the images of the square and the sphere are produced two times each with different names.

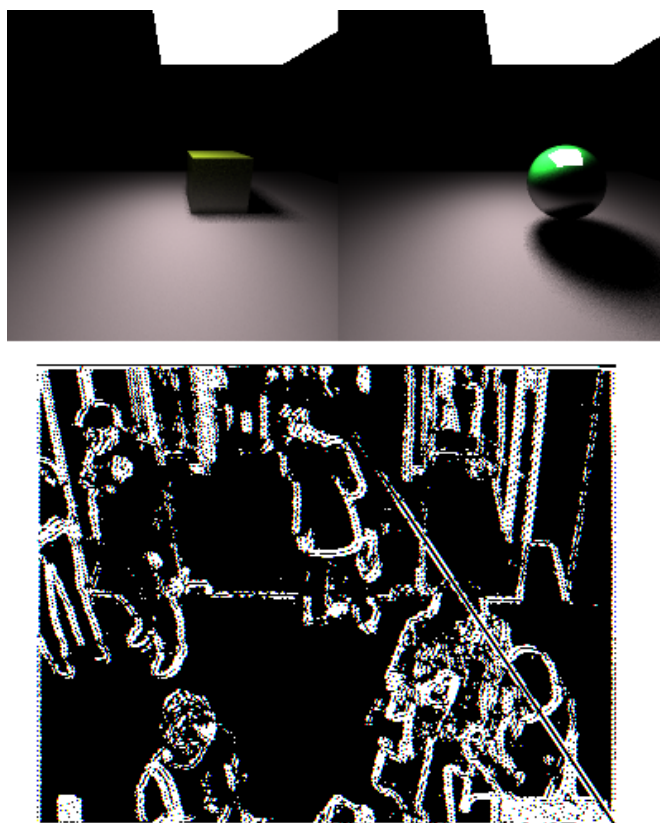


Figure 6.6: Resulting images for the RayTracer and Edge Detection applications.

Comments on the SPEAr experimental results

Compared with the results shown in [29] no speed-up was achieved in the RayTracer application, despite the use of partial reconfiguration. This is due to the huge limitations

6. EXPERIMENTAL RESULTS

invoked by the method for transferring data from the SPEAr to the FPGA and vice versa, which are set by the device's architecture, and the chosen application. However the user can find ways to work around the DMA transfers issue, with the use of large memory blocks on the FPGA storing the resulting or input data and then perform true parallel execution between the RRs.

As commented throughout the biggest drawback of the SPEAr board is the DMA engine, which is extremely slow, compared to how fast a hardware design can compute results, the RTSM's scheduling decision time and the ICAP's reconfiguration throughput. It is important though to consider that the board is a development board, which intended use is for feasibility studies, in particular evaluation of software systems developed for embedded platforms and how they could benefit from external hardware accelerators. Towards these goals the activation of the partial reconfiguration ability on the board can be very beneficial. Also the architecture created is generic and easy to be imported and coupled with R&D projects currently being developed on the platform.

In order to achieve better results with the applicaitons used here and also future R&D projects a change has to be made on the micro-electronic development of the board. Results has shown that the AHB-32 bus is slow, adding to that is the fact that it cannot handle two transfers simultaneously. A faster, two-way bus, handling the DMA transfers would produce better results and parallel solutions than the ones currently offered. Also another solution could be the integration of the FPGA on the SPEAr board itself, without the use of the EXPI interface, and also letting the FPGA board communicate with all the peripherals connected on the main board, following the paradigm of Zynq ZedBoard platforms [31].

6.3 Conclusion

In this Chapter we presented the experimental results taken from implementing and executing a real application, with the use of the extended RTSM presented in Chapter 4, on two different partially reconfigurable platforms. Also in [27] another implementation is presented on a Zynq ZedBoard. A key advantage of the RTSM is the ease in transporting the core code to different platforms, with different technology and design restrictions.

Table 6.4 presents the implementations of the RTSM and the features enabled and tested in each of them. The only features that were not enabled at any of the three

platforms are the Relocation Alternative, the Joint Hardware Modules and the Branches. However all these three features have been evaluated with the use of synthetic workloads in a simulation framework environment. Several features were not activated on the SPEAr 1310 platform not due to inability but due to the RayTracer application used to evaluate our design.

RTSM Features	Reconfigurable Platforms		
	XUP-V5	SPEAr 1310	ZedBoard
Multiple Bitstreams per Task	✓	✓	✓
Reservation	✓	✓	✓
Reuse Policy	✓	✓	✓
Configuration Prefetching	✓	✓	✓
Relocation	✗	✗	✗
Best Fit in Space	✓	✓	✓
Best Fit in Time	✓	✓	✓
Joint Hardware Modules	✗	✗	✗
HW/SW implementations	✓	✗	✓
Fork-Joins	✓	✓	✓
Branches	✗	✗	✗
Loops	✓	✓	✓
Dynamic changes in execution	✓	✗	✓
Micro-reconfiguration	✓	✗	✗

Table 6.4: RTSM features enabled in each of the Partially Reconfigurable Platforms used.

6. EXPERIMENTAL RESULTS

Chapter 7

A Software-only Environment: the OpenMP case

The RTSM created can offer the user, "advance" functionality by making complex decisions considering all the parameter's offered, regarding the tasks', the processing elements' (RR/SW-PE) and the available mappings' characteristics, as well as the current state of the FPGA device.

That way the RTSM can offer a parallel execution scheme of an application. During the course of completing this Master thesis an interesting observation was made. By combining the functionality of the RTSM with the pthreads C-language library, the RTSM could run solely on software environments and desktop computers. The main idea is that since the RTSM can basically manage resources, whatever those resources may be, it should be fairly straightforward for the RTSM to managae cores in a multi/many-processor system.

Multi/Many-core processors are the current norm in everyday computers offering parallel solutions in low-end users. Making the pairing between hardware RRs and SW processing elements residing on an FPGA, and software processor cores on a desktop computer, we are able to transfer the RTSM policies and scheduling to a SW-only environment.

The application the user implements should be, either included in the RTSM core code and the tasks implemented as functions that would be appointed to a thread, or the tasks executable should be linked in a library so that the RTSM could begin their execution. Also it would be more efficient in terms of parallelism that beyond task the

7. A SOFTWARE-ONLY ENVIRONMENT: THE OPENMP CASE

application implemented should have data parallelism also in order to take advantage of as many processing cores as possible.

7.1 Framework Description

Many attempts have been made towards exploitation of data parallelism in software environments. The most important goal of those attempts was to facilitate with the development of such applications. One of the most known standards in the industry is the OpenMP API. OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. The programmer annotates the C code he has written with `#pragmas`, which constitute compiler directives. The compiler analyzes these directions and sets off a fork-join multithreading model. The creation of the numerous threads is transparent to the user making the OpenMP API easy-to-use by even not so experienced users.

The advantages of the OpenMP API are:

- Simple: need not deal with message passing.
- Data layout and decomposition is handled automatically by directives.
- Scalability comparable to MPI on shared-memory systems.
- Incremental parallelism: can work on one part of the program at one time, no dramatic change to code is needed.
- Original (serial) code statements need not, in general, be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs.

OpenMP towards facilitating the end-user in his attempt to easy parallelize applications has made several assumptions and simplifications in the thread creation and memory management process. This introduced several disadvantages as:

- Risk of introducing difficult to debug synchronization bugs and race conditions.

- Currently only runs efficiently in shared-memory multiprocessor platforms. However there are new attempts towards expanding the OpenMP paradigm towards clusters and other distributed shared memory platforms.
- Requires a compiler that supports OpenMP.
- Scalability is limited by memory architecture.
- Lacks fine-grained mechanisms to control thread-processor mapping.

In our RTSM we consider the same thread creation case. However the difference between the RTSM and the OpenMP is the need for the user to incorporate the application code to the RTSM. Also the user needs to perform the data splitting manually during the implementation phase. For example supposing an Edge Detection application the designer should first integrate the different tasks comprising the application in the RTSM and dictate the way the data would be split in order to achieve data parallelism. When the split is done then the user provides the RTSM with the input file, that presents the scheduler with the available cores, the tasks and the task graph of the application, and the RTSM begins mapping the tasks to the available CPUs.

The RTSM provides the function template code in order for the designer to easily integrate the function of his choice. Also this template includes the function-calls needed for the RTSM to explicitly create a thread-processor mapping. This is the main advantage of the RTSM against the OpenMP API. One of the main drawbacks are the integration overhead, which compared to OpenMP `#pragmas`, is fairly more complicated in the RTSM's case.

The overhead consists of the time needed for the user to explore the parallelization capabilities of the desired application and to decide on the data he is going to process in parallel. The RTSM in this SW-only environment begins the execution of tasks using the `pthread_create()` function. Additionally to the files used for implementing the core code of the RTSM, list and structure management, scheduling policies etc., the user needs to include his application code to a separate file called, **application_functions.c**, and link the functions with the existing header file of the RTSM (**functions.h**).

Inside the C file the user will find the function template that will include the task function. The template is shown in Algorithm 4. After that piece of code the designer can add the task's application code.

7. A SOFTWARE-ONLY ENVIRONMENT: THE OPENMP CASE

Algorithm 4 The function template.

Input:

Thread arguments structure containing: Processed data, CPU core number.

Output:

```
2: cpu_set_t cpuset;
   CPU_ZERO(&cpuset);
   CPU_SET(arg_G.core, &cpuset);
   if pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset) < 0 then
4:   perror("pthread_setaffinity_np");
```

The code used by the RTSM to begin execution of a SW task is shown in Algorithm 5. Also the designer has to create a new function that will store in the thread arguments structure the correct data pointers, considering which part of the data will now be processed and which task is about to begin execution, e.g. **setArguments()**.

Algorithm 5 Task execution code.

Input:

Scheduling structure containing information about the task's placement.

Output:

```
3: switch TaskNumber do
   case T1
       switch DataPart do
6:         case D1                                ▷ arg_G1 contains part 1 of the data for Task 1.
           arg_G1 = setArguments(TaskNumber, DataPart);
           rc = pthread_create(&threads[Data_part], &attr, function1, (void*)arg_G1);
9:         break;
           case D2                                ▷ arg_G2 contains part 2 of the data for Task 1.
           arg_G2 = setArguments(TaskNumber, DataPart);
12:        rc = pthread_create(&threads[Data_part], &attr, function1, (void*)arg_G2);
           break;
           case Dn                                ▷ Same instructions for all the tasks/data parts.
15:        case Tn
```

Also the fact that the RTSM needs to be explicitly run on one core uninterrupted limits the available cores, compared to the OpenMP. Finally the ability of the RTSM to map different applications to cores with the use of one input file describing both, is another advantage over the OpenMP, however the integration overhead becomes even greater for the designer.

7.2 RTSM vs OpenMP

The applications chosen in order to compare the RTSM with the OpenMP API were an Edge Detection and a Motion Detection applications. The Edge Detection is similar to the ones mentioned in earlier chapters. The edge detection application offered mainly data parallelism. Task parallelism was explored also using a Motion Detection application. The two applications have a Gray Scale, a Gaussian Blur and a Threshold function in common. Their differences appear in the use of the Edge Laplace and Motion Detection function respectively and the fact that the motion detection function needs two input images.

In order to achieve data parallelism the splitting of the image was done in equal rectangles. After initializing those rectangles, we used the function template to create the different tasks for our applications. The split performed on the processed image was in 4 and 8 parts, increasing the parallelism and the workload of the application. Each Edge Detection phase must run 4 or 8 times before the RTSM considers the task completed.

The experiments were run on a desktop with an Intel[®] Xeon[®] Processor E5 @ 2.2GHz, with 12 cores in total. We used 2 images of different sizes, 512x512 and 1024x512, in order to see the how the execution time changes according to the data. The available cores/SW-PE for both our RTSM and the OpenMP were 4. We applied all the available compiler optimizations (O1, O2, O3) on both implementations, in order to see which one achieves the best execution time. Same experiments with the same parameters were performed for the Motion Detection application as well.

During the course of these experiments we made several measurements to compare our RTSM with the OpenMP API:

7. A SOFTWARE-ONLY ENVIRONMENT: THE OPENMP CASE

- We measure the scheduling overhead and analyzed the different phases of our RTSM.
- We measured and compared the execution time of the whole application.

During the first results the observations made were that by simply writing an OpenMP parallel program with no optimizations, the OS scheduler performs many involuntary context switches. That produced really high application execution times, that did not improve with the compiler optimizations applied.

On the other hand the RTSM created tasks and bound them to SW-PE until the end of their execution, so no context switches were being made. This was due to the usage of the `set_affinity()` function immediately after the creation of a thread.

The different approach of the two implementations regarding thread affinity, resulted in, at most 33x speed-up, of the RTSM over the OpenMP version. Due to this huge difference in execution times a further exploration of the OpenMP API was made. In the OpenMP API exists a flag option that binds the threads created by OpenMP to certain processors, thus eliminating involuntary context switches, this flag was to be set during compilation of the program with an OpenMP compatible compiler.

We applied the affinity optimization before the execution of the OpenMP program with the instruction `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 the Edge Detection application (OpenMP (opt.), RTSM and OpenMP (simple)) are shown in figure 7.1. The measurements made for the comparison of the three versions are mean times after 1000 executions. In figure 7.2 we can see the same measurements for the Motion Detection application. Tables 7.2 and 7.3 present the speed-ups achieved for each of the three different experiments in all the experimental cases.

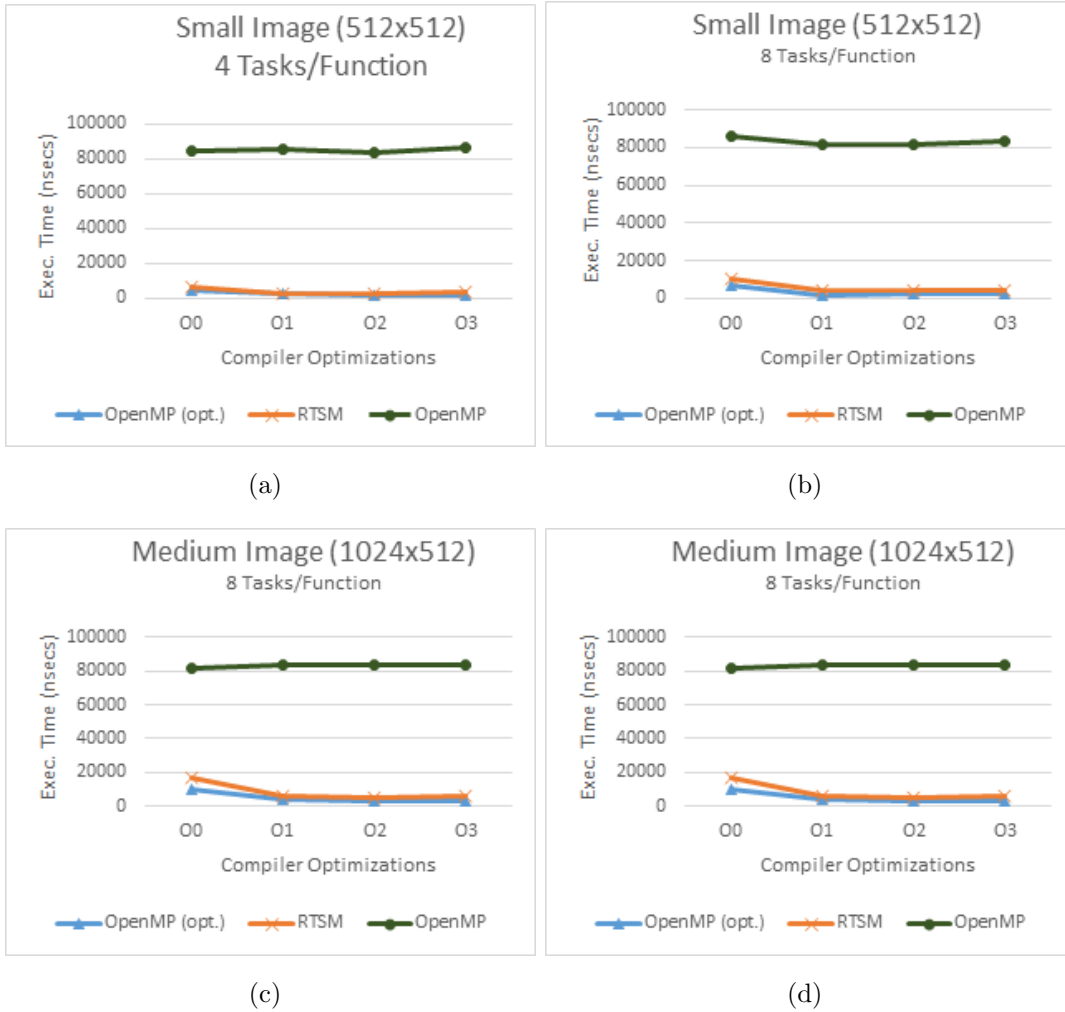


Figure 7.1: Execution times in *nsecs* of the Edge Detection application for the small (a,b) and medium (c,d) images.

7. A SOFTWARE-ONLY ENVIRONMENT: THE OPENMP CASE

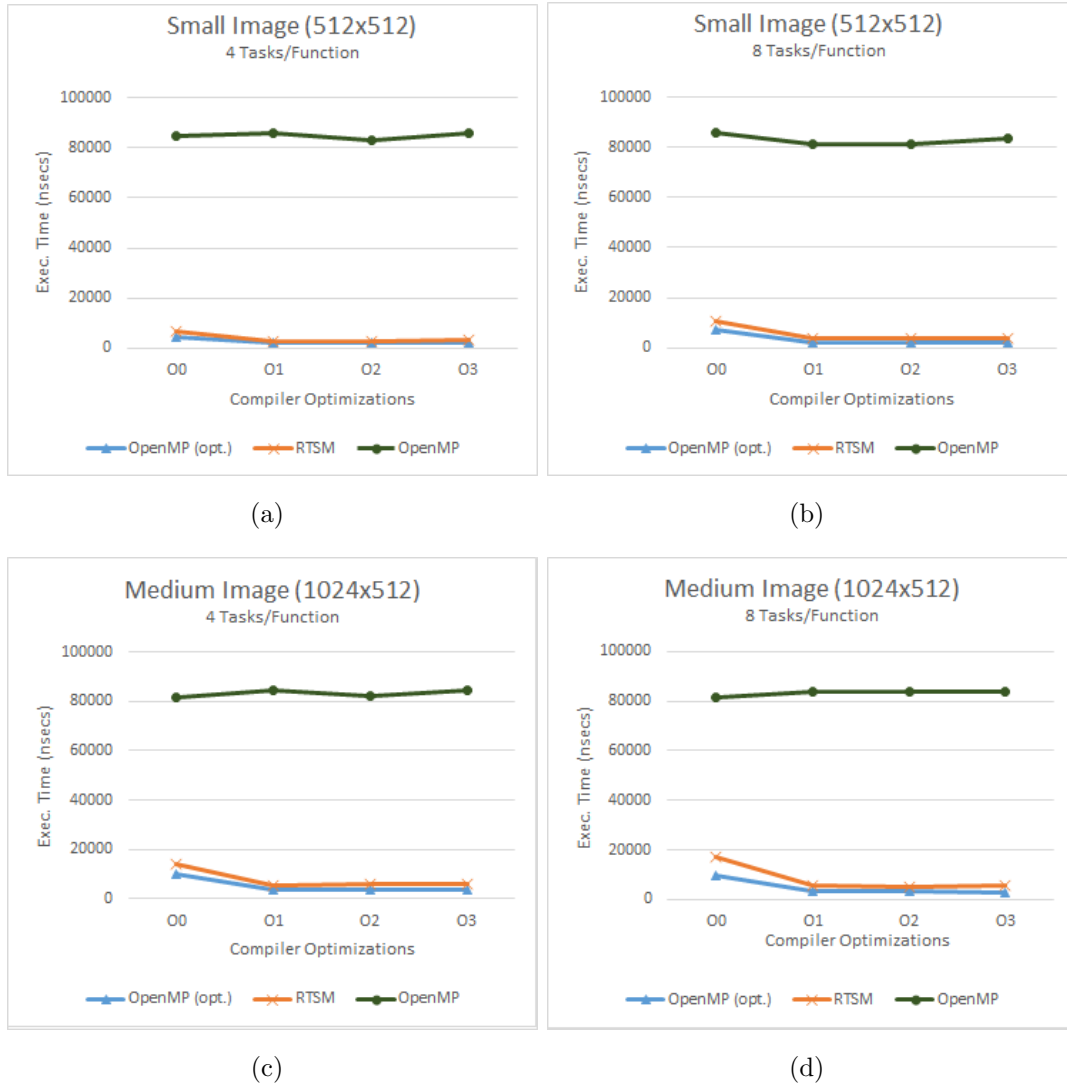


Figure 7.2: Execution times in *nsecs* of the Motion Detection application for the small (a,b) and medium (c,d) images.

Medium Image (1024x512)				
4 Tasks - 4 Cores	O0	O1	O2	O3
Speed-Up OpenMP opt. vs RTSM	1.39	1.52	1.48	1.59
Speed-Up RTSM vs OpenMP	5.83	15.49	14.25	13.85
8 Tasks - 4 Cores				
Speed-Up OpenMP opt. vs RTSM	1.75	1.59	1.44	1.90
Speed-Up RTSM vs OpenMP	4.79	14.50	16.09	14.42

Small Image (512x512)				
4 Tasks - 4 Cores	O0	O1	O2	O3
Speed-Up OpenMP opt. vs RTSM	1.47	1.16	1.22	1.58
Speed-Up RTSM vs OpenMP	12.59	33.58	31.88	24.92
8 Tasks - 4 Cores				
Speed-Up OpenMP opt. vs RTSM	1.46	1.99	1.92	1.88
Speed-Up RTSM vs OpenMP	8.08	20.28	20.63	20.98

Table 7.1: Speed-up values for the Edge Detection.

Medium Image (1024x512)				
4 Tasks - 4 Cores	O0	O1	O2	O3
Speed-Up OpenMP opt. vs RTSM	1.30	1.67	1.70	1.49
Speed-Up RTSM vs OpenMP	5.42	13.85	13.68	13.72
8 Tasks - 4 Cores				
Speed-Up OpenMP opt. vs RTSM	1.49	2.15	2.04	2.06
Speed-Up RTSM vs OpenMP	4.88	11.02	11.03	11.06

Small Image (512x512)				
4 Tasks - 4 Cores	O0	O1	O2	O3
Speed-Up OpenMP opt. vs RTSM	1.39	2.18	2.05	2.18
Speed-Up RTSM vs OpenMP	9.34	20.54	20.62	20.02
8 Tasks - 4 Cores				
Speed-Up OpenMP opt. vs RTSM	1.85	3.07	3.08	3.00
Speed-Up RTSM vs OpenMP	7.64	14.12	14.81	14.73

Table 7.2: Speed-up values for the Motion Detection.

It is observable that the OpenMP (simple) version is marginally slower than the other two implementations of the same application. The only difference between the implementations is, as said the KMP_affinity flag. For the Edge Detection application which offers only data parallelism the optimized OpenMP and the RTSM version outperform the

7. A SOFTWARE-ONLY ENVIRONMENT: THE OPENMP CASE

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 1.1-1.9x speed-up.

The comparison of the three implementations using the Motion Detection application yields similar results as seen. The speed-up of the RTSM version compared to the OpenMP (simple) is ranging from 4.9x to 20.6x. However the speed-up of the optimised OpenMP version over the RTSM is slightly elevated with values from 1.3x to 3.08x. This shows a clear advantage of the RTSM over the simple OpenMP version, also the RTSM can have under circumstances a comparable performance to the affinity optimised OpenMP.

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 forbid the involuntary context switches, performed by the OS, can offer a speed-up up to 40x on the same application.
- Our RTSM is not much slower than 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.

The 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 not optimized OpenMP version was considerably slower than our approach and the optimized version offered at most a 3x speed-up.

By measuring the scheduling overhead of the RTSM after 1000 executions we averaged and calculated the times for each RTSM phase. Results are shown in table 7.1 correspond to the 8-split experiments. The results show that the amount of overhead produced is negligible compared to the overall execution time of the application especially in the compiler optimized experiments.

Table 7.3: RTSM phases and overhead: Medium Image, O0, 8 tasks/function.

RTSM Phases	# times the phase is reached	Avg. Elapsed Time (<i>nsec</i>)
RTSM initialiaza-tion.	1	863
Schedule decision	32	118
Issue Command	32	39
Lists Update (Task completion)	32	98

The RTSM created is not optimized nor created to be run on SW-only environments and offers advanced scheduling functionality on hardware functions. Despite that the core code can overcome many optimizations and changes in order to become faster, when run in a SW-only environment, and comparable to the OpenMP API. Also if the OpenMP is not optimized for processor-mapping then its performance is greatly downgraded. Finally the RTSM as said is able to manage multiple applications unified in one task-graph, whereas the OpenMP is unable to do that.

7. A SOFTWARE-ONLY ENVIRONMENT: THE OPENMP CASE

Chapter 8

Conclusion and Future Work

In this work we presented the extensions and changes made in order to port and implement an RTSM to an actual partially reconfigurable FPGA system. Several of these extensions are towards the creation of a high-level RTSM able to handle HW and SW tasks comprising one or more applications sharing the same Partially Reconfigurable device. Also the extensions made have widen the range of applications the RTSM was able to manage, with the inclusion of loop structures, branches and fork-join operations.

Additionally the dynamic changes in execution times of the tasks offer real dynamically capabilities in our RTSM with which version, either SW or HW, of a task will be executed. The dynamic changes are extremely beneficial to the overall execution time of the application:

- Handles run-time changes of the device characteristics that affect the execution time of a task (HW), or the clock frequency of a SW-PE.
- Handles mistakes by the user during the profiling of HW/SW tasks. The profiling is used to derive the estimation execution times for the RTSM's input file.

It is the first time, to the best of our knowledge, that an actual implemented RTSM on a partially reconfigurable device, is able to handle and forward micro-reconfiguration requests.

After extending the RTSM we implemented and tested it in partially reconfigurable FPGA systems. We made implementations in two systems:

8. CONCLUSION AND FUTURE WORK

- A XUP-V5 FPGA system in collaboration with Politecnico di Milano and [27]. The final design was tested with an Edge Detection application. In this the most features of the RTSM were enabled and verified.
- A SPEAr 1310 platform coupled with a Virtex-5 LX110 FPGA daughter-board provided by STMicroelectronics. It was the first time the PR ability was enabled in this platform, aided by a DMA-based architecture created. The testing was made with the use of a RayTracer and an Edge Detection application. In this platform the parallel management of multiple applications was demonstrated.

A big advantage of the RTSM created is the easy porting throughout reconfigurable systems with different technology restrictions. Also the porting of applications in our RTSM is quite simple.

During the course of this Master a side result was created regarding the ability of the RTSM to manage applications on SW-only environments, e.g. multi-core or many-core systems. After realizing this ability we compared the RTSM with the well-known industry standard OpenMP. The results obtained showed that in the non-optimised version of the same OpenMP program the RTSM displays better core-mapping behavior.

The plans for future expansion of the RTSM include the porting of more complicated applications. Also the Joint Hardware Modules and Relocation alternative have to be realized and tested on an actual FPGA device. Furthermore applications, which include branches have to be also tested. In combination with the work presented in [32] could lead in to interesting results regarding the reconfiguration prefetching and the branch structures in applications.

Finally the RTSM could aid the user prior to the implementation of an application in hardware. The RTSM could offer a handy off-line tool, by aiding the user with the decision of whether to continue with the development and which device is beneficial to use, in terms of logic resources. This combined with the Riverside Optimizing Compiler for Configurable Circuits (ROCCC) [33] could become a handy simulation tool for aiding the decisions of the designer. Also the multi-threaded execution model [34] could provide further insight on the possibilities with coupling the existing RTSM with irregular applications with dynamic workloads.

References

- [1] Lysaght, P., Blodget, B., Mason, J., Young, J., Bridgford, B.: Invited paper: Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas. *Field Programmable Logic and Applications*, 2006. FPL '06. International Conference on (2006) 1–6 [5](#), [6](#), [16](#)
- [2] J. Burns, A. Donlin, J.H.S.S.M.d.W.: A dynamic reconfiguration run-time system. *Field-Programmable Custom Computing Machines*, 1997. Proceedings., The 5th Annual IEEE Symposium on (1997) 66–75 [6](#), [15](#)
- [3] G. Durelli, C. Pilato, A.C.D.S., Santambrogio, M.D.: Automatic run-time manager generation for reconfigurable mp soc architectures. *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2012 7th International Workshop on (2012) 1–8 [6](#), [18](#), [34](#), [87](#)
- [4] Li, Z.: Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *ACM/SIGDA Symposium on Field-Programmable Gate Arrays* (2002) 187–195 [7](#)
- [5] Compton, K., Li, Z., Cooley, J., Knol, S., Hauck, S.: Configuration relocation and defragmentation for run-time reconfigurable computing. *Very Large Scale Integration (VLSI) Systems*, *IEEE Transactions on* (2002) 209–220 [7](#), [55](#)
- [6] Morales-Villanueva, A., Gordon-Ross, A.: On-chip context save and restore of hardware tasks on partially reconfigurable fpgas. *Field-Programmable Custom Computing Machines (FCCM)*, 2013 IEEE 21st Annual International Symposium on (2013) 61–64 [7](#)

REFERENCES

- [7] C. Conger, A. Gordon-Ross, A.D.G.: Design framework for partial run-time fpga reconfiguration. *ERSA* (2008) 122–128 [8](#)
- [8] Vipin, K., Fahmy, S.A.: Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. *Applied Reconfigurable Computing* (2012) 13–25 [8](#), [19](#)
- [9] Steiger, C., Walder, H., Platzner, M.: Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *Computers, IEEE Transactions on* (2004) 1393–1407 [9](#), [16](#), [21](#)
- [10] E. El-Araby, I. Gonzalez, T.E.G.: Exploiting partial runtime reconfiguration for high-performance reconfigurable computing. *Journal ACM Transactions on Reconfigurable Technology and Systems* Volume 1 Issue 4, January 2009 Article No. 21 (2009) [15](#)
- [11] Walder, H., Platzner, M.: A runtime environment for reconfigurable hardware operating systems. *Field Programmable Logic and Applications, 2004. FPL '04. International Conference on* (2004) 831–835 [16](#)
- [12] Jara-Berrocal, A., Gordon-Ross, A.: Vapres: A virtual architecture for partially reconfigurable embedded systems. *ACM Design, Automation and Test in Europe (DATE)* (2010) [17](#), [18](#)
- [13] Jara-Berrocal, A., Gordon-Ross, A.: Scores: A scalable and parametric streams-based communication architecture for modular reconfigurable systems. *ACM Design, Automation and Test in Europe (DATE)* (2009) [17](#)
- [14] Pellizzoni, R., Caccamo, M.: Hybrid hardware-software architecture for reconfigurable real-time systems. *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE* (2008) 273–284 [18](#)
- [15] Bomel, P., Gogniat, G., Diguët, J.P.: A networked, lightweight and partially reconfigurable platform. *Applied Reconfigurable Computing* (2008) 318–323 [18](#)
- [16] Jara-Berrocal, A., Gordon-Ross, A.: Hardware module reuse and runtime assembly for dynamic management of reconfigurable resources. *Field-Programmable Technology (FPT), 2011 International Conference on* (2011) 1–6 [19](#)

-
- [17] Theodoropoulos, D., Pratikakis, P., Pnevmatikatos, D.: Efficient runtime support for embedded mpsoes. *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, 2013 International Conference on (2013) 164–171 [19](#)
 - [18] Papadimitriou, K., Vatsolakis, C., Pnevmatikatos, D.: Invited paper: Acceleration of computationally-intensive kernels in the reconfigurable era. *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2012 7th International Workshop on (2012) 1–5 [19](#)
 - [19] Beretta, I., Rana, V., Atienza, D., Sciuto, D.: A mapping flow for dynamically reconfigurable multi-core system-on-chip design. *Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on* (2011) 1211–1224 [20](#)
 - [20] Redaelli, F., Santambrogio, M.D., Memik, S.O.: An ilp formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures. *Int. J. Reconfig. Comput.* (2009) 7:1–7:12 [20](#)
 - [21] Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.: The molen polymorphic processor. *Computers*, *IEEE Transactions on* (2004) 1363–1375 [20](#)
 - [22] Bauer, L., Grudnitsky, A., Shafique, M., Henkel, J.: Pats: A performance aware task scheduler for runtime reconfigurable processors. *Field-Programmable Custom Computing Machines (FCCM)*, 2012 IEEE 20th Annual International Symposium on (2012) 208–215 [21](#)
 - [23] DURAN, A., AYGUADE, E., BADIA, R.M., LABARTA, J., MARTINELL, L., MARTORELL, X., PLANAS, J.: Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* (2011) 173–193 [21](#)
 - [24] Agne, A., Happe, M., Keller, A., Lubbers, E., Plattner, B., Platzner, M., Plessl, C.: Reconos: An operating system approach for reconfigurable computing. *Micro*, *IEEE* (2014) 60–71 [21](#)
 - [25] Lubbers, E., Platzner, M.: Cooperative multithreading in dynamically reconfigurable systems. *Field Programmable Logic and Applications*, 2009. FPL 2009. *International Conference on* (2009) 551–554 [21](#)

REFERENCES

- [26] Davidson, T., Stroobandt, D.: Data path analysis for dynamic circuit specialisation. 2014 International Conference on ReConFigurable Computing and FPGAs, ReConFig14 (2014) 1–8 [25](#)
- [27] Koidis, I.: Run time system implementation for concurrent hw/sw task execution on fpga platforms. MSc. Thesis (2015) [32](#), [54](#), [57](#), [64](#), [80](#)
- [28] ARM®: Amba ® 3 ahb-lite protocol v1.0 specification. (2006) [40](#)
- [29] Spada, F., Scolari, A., Durelli, G., Cattaneo, R., Santambrogio, M., Sciuto, D., Pnevmatikatos, D., Gaydadjiev, G., Pell, O., Brokalakis, A., Luk, W., Stroobandt, D., Pau, D.: Fpga-based design using the faster toolchain: The case of stm spear development board. Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on (2014) 134–141 [43](#), [59](#), [63](#)
- [30] Xilinx®: Virtex-5 fpga configuration user guide. (2012) [45](#), [46](#)
- [31] Charitopoulos, G., Koidis, I., Papadimitriou, K., Pnevmatikatos, D.: Hardware task scheduling for partially reconfigurable fpgas. Applied Reconfigurable Computing (2015) 487–498 [64](#)
- [32] Papademetriou, K., Dollas, A.: Performance evaluation of a preloading model in dynamically reconfigurable processors. Field Programmable Logic and Applications, 2006. FPL '06. International Conference on (2006) 1–4 [80](#)
- [33] Villarreal, J., Park, A., Najjar, W., Halstead, R.: Designing modular hardware accelerators in c with roccc 2.0. Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on (2010) 127–134 [80](#)
- [34] Halstead, R.J., Najjar, W.A.: Compiled multithreaded data paths on fpgas for dynamic workloads. International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2013, Montreal, QC, Canada, September 29 - October 4, 2013 (2013) 3:1–3:10 [80](#)

Chapter 9

Appendix A: The ZedBoard implementation

In this Appendix the architecture and the results from the third implementation made in a Zynq ZedBoard will be presented. This implementation was done during the course of a Master work done by Iosif Koidis. Due to the fact that the implementation was of this RTSM and for reasons of completeness we are including the architecture and the results of that work in this Appendix.

ZedBoard is a low-cost development board for the Xilinx ZynqTM-7000 All Programmable SoC (AP SoC). We can see the chip characteristics for the Z-7020 part which is utilized on the Zedboard platform below in Table 9.1.

We validated the behavior of RTSM on a fully functional system on a ZedBoard platform executing an edge detection application. Figure 9.1 shows the system architecture that 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. We used CPU0 for the RTSM and CPU1 as the SW-PE, executing SW tasks.

9. APPENDIX A: THE ZEDBOARD IMPLEMENTATION

Table 9.1: **Z-7020 Characteristics.**

Processor Core	Dual ARM® Cortex™-A9 MPCore™ with CoreSight™
Processor Extension	NEON® & Single / Double Precision Floating Point for each processor
L1 Cache	32 KB Instruction, 32 KB Data per processor
L2 Cache	512KB
On-Chip Memory	256KB
Memory Interfaces	DDR3, DDR3L, DDR2, LPDDR2, 2x Quad-SPI, NAND, NOR
Peripherals	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO
Logic Cells	85K Logic Cells
BlockRAM (Mb)	560 LB
DSP Slices	220

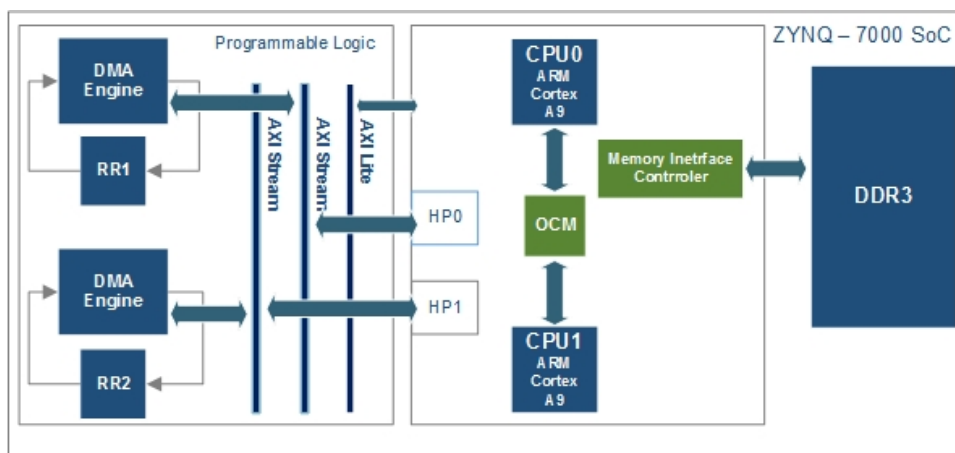


Figure 9.1: System Architecture of the Zynq platform.

At system 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 (task graph, task information, task mappings, etc.) from the SD card. It also transfers the partial bitstreams from the SD to the main (DDR3) memory. 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.

The RTSM operation was again broken down into different phases and the time spent into each phase was measured. The phases analyzed in this platform are the same with the ones, analyzed in the XUP-V5 platform. The results of the time measurements are shown in Table 9.2.

Table 9.2: RTSM phases and aggregate overhead.

RTSM Phases	#Clock Cycles	Elapsed Time (usecs)
RTSM initialiazation.	7,707	23.121
Schedule decision	17,436	52.038
Issue Command (HW/SW Exec or Reconfiguration)	5,995	17.985
Lists Update (HW task completion)	2,493	7.479
Lists Update (SW task completion)	2,748	8.244
Lists Update (Reconfiguration completion)	1,224	3.672

The overall execution time of the application was measured to be 129.62 ms, while the total RTSM overhead derived from Table 2 is 0.112 ms. The theoretical reconfiguration overhead, given that PCAP has a throughput of 400MB/sec, is 0.6 ms. Hardware Task Scheduling for Partially Reconfigurable FPGAs. This is added only once, since in the rest cases of our example configuration prefetching takes place and hides the reconfiguration overhead with HW execution. However, since we use a SW task to perform reconfiguration, the throughput is considerably lower at 50MB/sec, increasing the reconfiguration cost to 5ms. Still, compared to the total execution time of application, this overhead is negligible.

Finally, we compare the performance of our system with the one presented in [3]. That work used the same application running on a Xilinx Virtex-5 FPGA and reported a throughput of 18 fps for a 640x480 image. In our case, we used a 1920x1080 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

9. APPENDIX A: THE ZEDBOARD IMPLEMENTATION



(a)



(b)

Figure 9.2: Input and output images for the Zynq ZedBoard platform.

hard processors while the V5 supports only soft-core MicroBlaze processors. The input and output images for the edge detection application are shown in figure 9.2.

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