Technical University of Crete

School of Electronic and Computer Engineering

Microprocessor & Hardware Laboratory

Master Thesis

Run Time System Implementation for Concurrent H/W S/W Task Execution on FPGA Platforms

IOSIF KOIDIS

ADVISOR: Professor Dionisios Pnevmatikatos

COMMITTEE: Professor Apostolos Dollas
Assistant Professor Vasilios Samoladas

JUNE 2015

Abstract

In the recent years, technology have made possible to fit a larger number of components on a single chip, and allowed us to realize larger, more complex chips. The large transistor budget can be used to create heterogeneous systems, generally called Multiprocessor Systems-on-Chip (MPSoC). It also allowed the creation of larger FPGA devices, integrating ample amounts of programmable logic, memories, programmable DSP/arithmetic units. Partial reconfiguration (PR) of FPGAs can be used to dynamically extend and adapt the functionality of computing systems, swapping in and out HW tasks. In this thesis we describe the integration process of a Run Time System Manager (RTSM) able to map multiple applications on the underlying architecture, which may consist of microprocessors as software processing elements and Partially Reconfigurable areas as hardware processing elements, and execute them concurrently.

In this thesis we describe the integration process of a Run Time System Manager (RTSM) able to map multiple applications on the underlying architecture and execute them concurrently.

The RTSM is able to schedule application tasks either on available processor core(s), or at the FPGA hardware resources using partial reconfiguration. The choice is made dynamically based on availability and a gain function VERIFY. We integrate and the RTSM on two different system architectures and corresponding platforms in order to demonstrate the RTSM portability and a real time application is used in order to validate its correctness and potential. The two aforementioned embedded platforms are the Xilinx XUPV5 board which hosts a Virtex 5 LX 110T device and the Zedboard platform which hosts a • Zynq®-7000 All Programmable SoC XC7Z020-CLG484-1.

Acknowledgements

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

I would like to thank my advisor Professor Dionisios Pnevmatikatos, for giving me more chances that I would ever give myself in his place.

Publications

1. 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
2. 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
3. George Charitopoulos, Iosif Koidis, Kyprianos Papadimitriou, and Dionisios Pnevmatikatos, “Run-Time Management of Systems with Partially Reconfigurable FPGAs” Integration VLSI Journal (submitted)

Contents

Table of Figures

1. Introduction

Fine-grain, reconfigurable devices have been available for years in the form of FPGA chips, recent consumer electronic devices integrate an increasing number of processing cores, accelerators to satisfy performance and energy requirements. Many of these devices support the dynamic modification of their programming while they are operating (Dynamic Reconfiguration). However, this ability remains mostly unused as FPGA devices are mostly used in a fixed functionality, "asic-replacement" manner. This is due to the increased complexity in the design and verification of a changing system. In addition to design requirements, the process of creating (partially) reconfigurable designs is less widespread and the corresponding tools are less friendly to the designers.

These accelerators also need to adapt to the new requirements, reconfigurable logic allows the definition of new functions to be implemented in dynamically instantiated hardware units, combining adaptivity with hardware speed and efficiency. Also recently emerging SoC devices enable the development of embedded systems where software tasks, running on a CPU, can coexist with hardware tasks running on a reconfigurable device (FPGA). However, designing a changing hardware system is both challenging and time consuming.

All of the above add a lot of complexity to the application designer who has to be an accustomed to different system architectures of the fabric in which he just wants to use in order to execute his application. The need for a complete methodology that will enable designers to easily implement and verify applications on platforms with one or more general-purpose processors and multiple acceleration modules implemented in the latest reconfigurable technology is needed.

In order to address the issue of verifying static and dynamic aspects of a reconfigurable design while minimizing run-time overheads on speed, area and power consumption, and provide a powerful and efficient run-time system for managing the various aspects of parallelism and adaptivity, our approach is a combination of a Run Time System Manager along with various system architectures embedded or not, provide an abstraction layer enabling the application designer to use complex system architectures which utilize many software processors along with hardware processing elements, by providing only the available task mapping and the application task graph.

The starting point of the can be a C application, whose initial decomposition could be described with any of the existing formalisms (e.g. OpenMP) potentially annotated with pragmas to specify additional information provided by the designer. The corresponding task graph is then derived, where every hardware task is to be treated as a region-reconfigurable module or a microreconfigurable module. Both static and dynamic reconfiguration is considered, and devices are assumed as partially reconfigurable, either in a single-FPGA or a multi-FPGA system. Also hardware/software partitioning is performed to determine which tasks can be executed in software instead. Finally, a methodology to schedule the resulting specification on the same reconfigurable architecture is applied. With respect to the above, one goal is to further reduce the dynamic hardware generation and reconfiguration overhead, especially by optimizing the applications and improving the locality of the parameter values that trigger a reconfiguration each time they change. In the past, several alternative configuration architectures were presented, such as multi-context, relocation/defragmentation and pipeline reconfigurable FPGAs. Configuration caching techniques try to keep configurations that will be needed in the future on chip. With respect to these techniques, our approach is compatible with these architectures, since we can bring different uses of a configuration closer in time with loop transformations.

2. Related Work

During the last decade technology has made possible the use of Multiprocessor Systems-on-Chip (MPSoC) also, the advances on silicon technology as well as the advances in FPGA design has made possible to accommodate a big number of microprocessors along with a big number of Partial Reconfigurable areas on the same chip, giving the application designers a very robust infrastructure on which they can develop or use in order to speed up their applications. However the effort and knowledge needed to tame this sheer power proves to be a drawback towards these type of architectures and platforms been more commonly used.

Back from 2005 Weyin Fu and Catherine Compton (1) described an architecture for reconfigurable computing with Reconfigurable Accelerators acting as co-processors. The potential of such system platforms was early identified but also the drawbacks that keep it from penetrating the mainstream were identified also. This work proposed a lot of concepts implemented today to reconfigurable computing such as hardware virtualization, automatic creation of hardware and software implementations of the same tasks, High – Level – Synthesis as also the need for Run Time System Schedulers in order for the OS's to be able to use the available reconfigurable kernels. But except from ideas that where futuristic at the time but today we see a lot of work towards that direction, the authors made some unrealistic assumptions about Partial Reconfiguration technology that sound futuristic even with current technology standards.

Dimitris Theodoropoulos et al (2), presented runtime framework for reconfigurable heterogeneous embedded MPSoCs, which allows rapid application development and efficient tasks allocation to all available processing elements. Their work adopts a task-based programming model that requires the developer to simply pragma-annotate the application code, in order to dynamically resolve task dependencies. They we target MPSoCs that are based on a shared memory model, they consider a structure with one master and several slave processors that communicate via an interconnection module, like buses or NoCs. In **Figure 1** we can see their runtime structure and its mapping on a shared-memory MPSoC architecture. Rounded numbers indicate the required runtime steps, in order to assign a task to a slave core; steps 1-3b are executed by the master runtime to generate a new task and add it to the shared resources, steps 4-6 are executed by the slave runtime to start a specific task, and steps 7-8 update the shared resources when a task is complete.

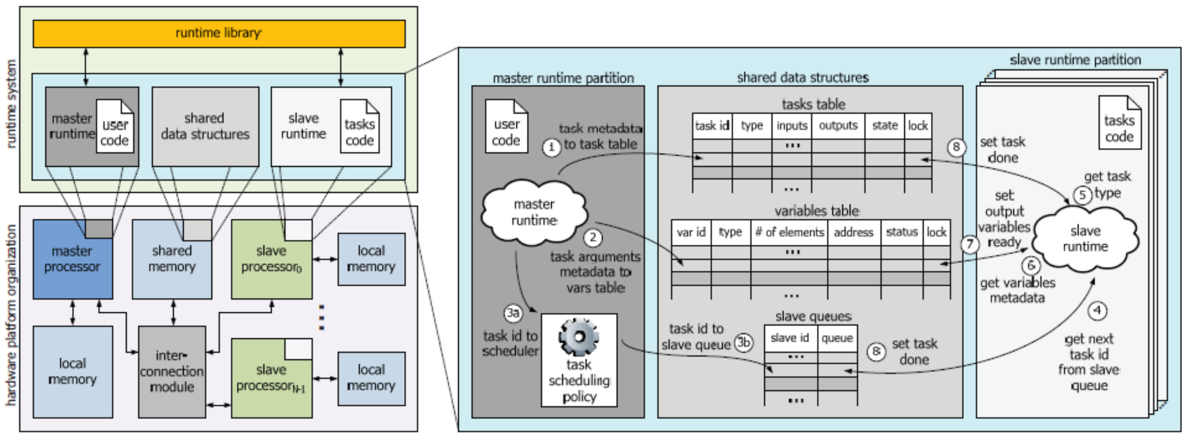

Figure 1 – The runtime structure

They implemented a multi-core system which we can see in **Figure 2**, they used a Microblaze soft-core processor as master processor (MMB) to host the master runtime paritition. The microprocessor has 8 KBytes of instruction and data cache respectively, while a local 64-Kbyte non-cached memory was also available. They also made use of the xilkernel as the host kernel for the microprocessor. They connected seven Microblazes (SMB) as slave processors, where each one of them also has its 8 Kbytes instruction and data cache, and a local 128-KByte noncached memory. The entire system was implemented and mapped onto an ML605 development board with a Virtex6 FPGA.

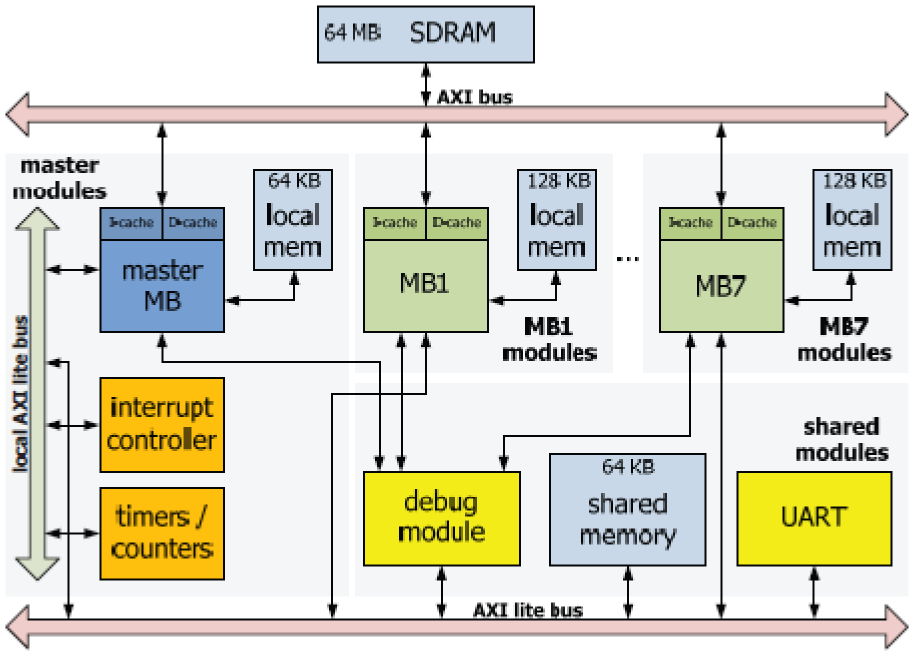

Figure 2 - Experimental system with seven slave MBs

Their work targets heterogeneous embedded MPSoCs, but they do only target microprocessors as software processing elements, and they do not take into account partial reconfiguration technology and hardware processing elements. Also the OS support is also mandatory fro they task scheduler to operate.

Kyprianos Papadimitriou et al (3) presented an implementation that targets hardware virtualization, and more specifically as they state reconfigurable hardware is accessed in a transparent manner by the host, a set of reconfigurable IP cores residing in a repository on a desktop computer are available for configuring the FPGA when needed. They implemented device drivers and APIs needed by the host computer to interface with the FPGA platform, and also provided runtime system support, a software component for supporting execution of application workloads in OS based systems. It undertakes low-level operations so as to offload the programmer from manually configuring the FPGA and controlling the datastransfers. They tested the system with two FPGA-based platforms equipped with a Virtex-II and a Virtex-5 respectively. Both platforms were plugged on the PCI slot of a PC running a linux operating system. In **Figure 3** we can see implemented software components for hardware – software communication and in **Figure 4** we can see a hardware module (accelerator) placed in the FPGA device.

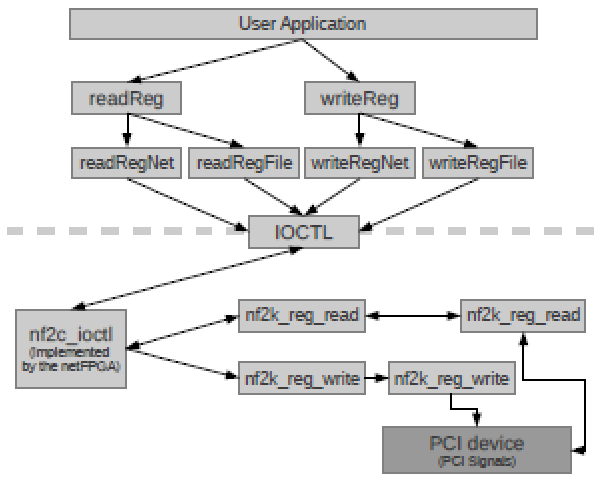

Figure 3 – Software Components

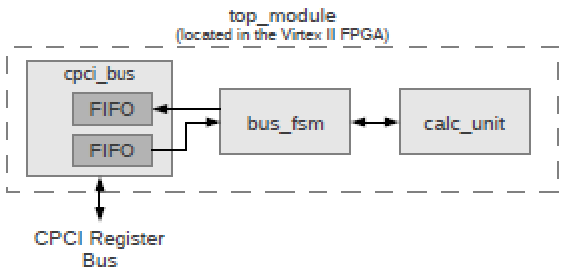

Figure 4 – Hardware accelerator Architecture

This work implements two functional systems with an FPGA acting as coprocessor to the host CPU and supports two basic operations, FPGA reconfiguration and data transfer between the host and the FPGA. The whole process is transparent to the user as the latter is not aware when the host software initiates the injection of a bitstream to the FPGA. On the downside the system targets only hardware processing elements through partial reconfiguration and again depends on OS support.

Another very interesting work is the OmpSs (4), a programming model based on OpenMP and StarSs, that can also incorporate the use of OpenCL or CUDA kernels. They evaluate the proposal on different architectures, SMP, GPUs, and hybrid SMP/GPU environments, showing the wide usefulness of the approach. While OpenMP has a fork-join model, OmpSs defines a thread-pool model where all the threads exist from the beginning of the execution. Of these threads, only one, the master thread, starts executing the user code while the other threads remain ready to execute work when available. The pool is still considered a team of threads in the OpenMP sense, with the difference that teams now consist of two kind of threads, the master thread and the worker threads. OmpSs is extended with a set of constructs to specify data dependences and to specify heterogeneous devices, is based on program annotations taking the best features from the tasks of OpenMP, StarSs dependence analysis and automatic generation of data transfers, and the expression of SIMD operations in OpenCL kernel codes. OmpSs is a proposal to improve programming on multicore processors and GPUs and they proved that productivity is improved and a performance similar or better than existing environments based on CUDA/OpenCL and OpenMP is achieved. The results presented, show that OmpSs outperforms OpenCL/CUDA and OpenMP implementations of the same benchmarks, in those architectures. In addition, it achieves a unified way of programming them in different environments.

Finally Rodolfo Pellizzoni and Marco Caccamo (5) presented a real-time computing architecture that can integrate hardware and software executions in a transparent manner, and can support real-time QoS adaptation by means of partial reconfiguration of FPGA devices. Tasks are allowed to migrate seamlessly from CPU to FPGA and vice versa to support dynamic QoS adaptation and cope with dynamic workloads. The main objective of their work is to devise a real-time computing infrastructure that can integrate hardware and software execution in a transparent manner, and can support real-time QoS adaptation by means of partial reconfiguration of modern FPGA devices. In particular, flexibility of execution is achieved by supporting the seamless migration of tasks from software to hardware and vice versa at run-time. They also consider that every task in the system may have multiple configurations, like we do, which may vary from the obvious software and hardware implementation of the same task but also different hardware implementations which different speed, area of power dissipation characteristics. In their work they need OS support like all of the related work we presented but also their scheduling strategy is a bit more simplistic, since they assume that when the system is not under heavy load all tasks with a HW configuration will be run in hardware. If the load on the system increases and the reconfigurable area is filled, some tasks will have to be moved to SW. In **Figure 5** we can see the block diagram of their proposed architecture. The implementation took place on a Xilinx Virtex-4 FPGA. A CPU is used to run the operating system and software configurations. The Internal Configuration Access Port (ICAP) controller provides reconfiguration capabilities. An interrupt controller receives interrupt requests from the ICAP controller, HW tasks and I/O devices and forwards them to the CPU. A timer module is used to generate a periodic timing signal (every 1 ms in our prototype), which is routed to each HW task and the interrupt controller and used as the base clock resolution by the OS. They state that a Virtex-4 LX200 device can support up to 48 regions. Each hardware task consists of two modules: the user logic and the interface module. The user logic implements the actual task logic. The interface module provides transparency and relocation service and connects the task to an on-chip system bus. The bus is divided in two segments: the CPU segment connects the CPU and all peripherals to an external RAM chip called the system RAM, used to hold OS and software configurations instructions and data. The task bus segment connects all hardware tasks to an external RAM chip called the shared RAM, used to hold shared memory data.

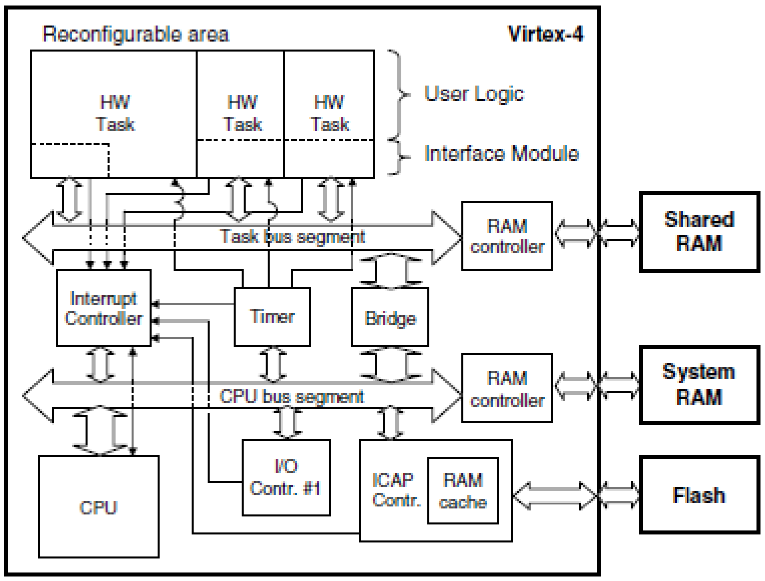

Figure 5 – Architecture block diagram

The actual implementation took place on a Xilinx ML401 development board. The board employed a small Virtex-4 LX25 FPGA which can accommodates up to 8 reconfigurable regions. The MicroBlaze CPU, bus segments, peripherals and all hardware tasks were clocked at 100Mhz; the system RAM used an 8 Mbytes synchronous SRAM chip while the shared RAM used a 64Mbytes, 266Mhz DRAM chip. To obtain maximum software performance, the changed the standard MicroBlaze CPU configuration to use a 8Kbytes instruction and 16Kbytes data cache together with a barrel shifter. For validation they developed several real-time symmetric cipher tasks including cipher-block chaining versions of DES, TDES and AES.

3. XUPV5 Implementation

This document presents the implementation of the Run-Time System Manager (RTSM) to the embedded platform architecture provided to us by Politecnico Di Milano (PDM). We present the new features of the RTSM and also its implementation in order to be used with the embedded platform. The different implementation are presented and the specific implementation details are discussed.

The rest of the document is organized as follows. In Section RTSM TUC we describe the RTSM Functionality which is Platform Independent, section PDM Architecture the initial system architecture from Politecnico di Milano, in which we intergraded the our RTSM, section RTSM Integration describes the different RTSM target hardware platforms in which we intergraded the RTSM, and finally in section System Extensions - New Features we present the new features that we added to the system architecture.

3.1 TUC RTSM

The Run Time System Manager is responsible for scheduling, software and hardware tasks, on a reconfigurable architecture system, based on information acquired during system and application initialization but also during runtime. Such an architecture system we can see in **Figure 6**.

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

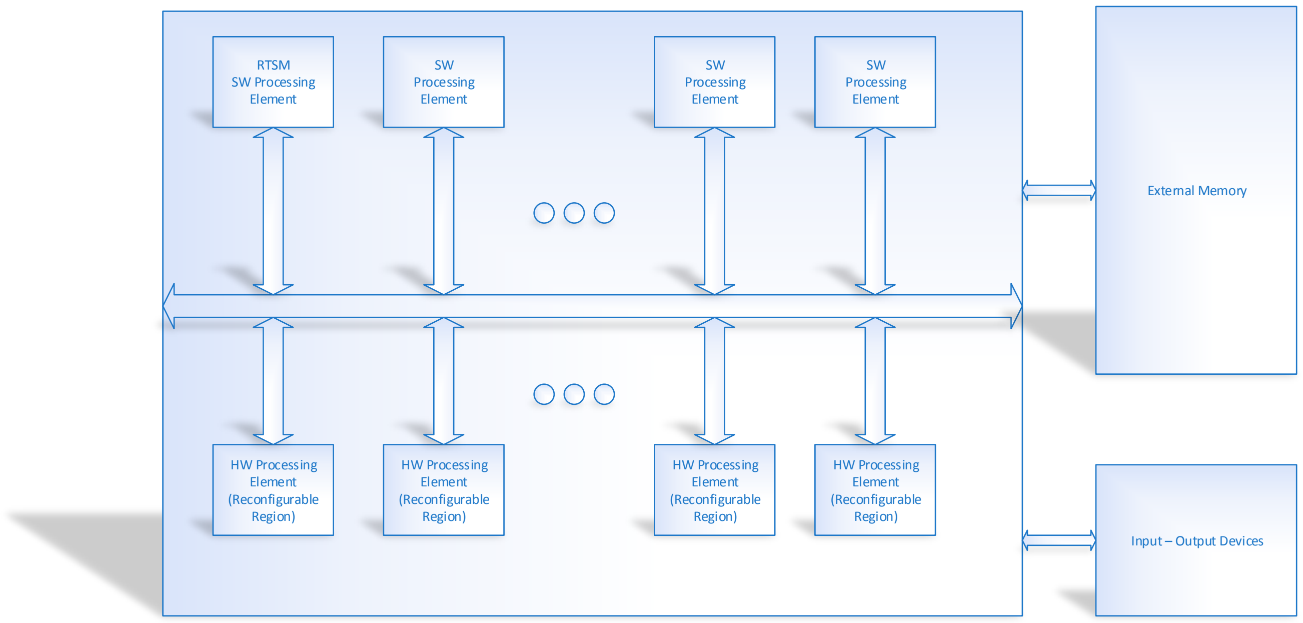
RTSM SW Processing Element
SW Processing Element
SW Processing Element
SW Processing Element
External Memory
HW Processing Element (Reconfigurable Region)
HW Processing Element (Reconfigurable Region)
HW Processing Element (Reconfigurable Region)
HW Processing Element (Reconfigurable Region)
Input – Output Devices

Figure 6 – General System Architecture

3.1.1 Key Concepts and Functionality

The RTSM gets as input the initial partitioning of the FPGA device, the tasks to be scheduled, the task graph representation and the different mappings of these tasks (i.e. different bitstreams corresponding to different implementations of one task). It also requires the execution and reconfiguration times of the tasks. At each tick of time, the RTSM checks if there is a newly arrived task to be scheduled onto the device, then notifies the user (used for monitoring purposes only) as to if this task was directly placed onto the device, or a different scheduling alternative was chosen. After that, the RTSM checks if a task has completed its execution on the device. Finally, it checks if there are reserved tasks that must be loaded to the FPGA to start their execution.

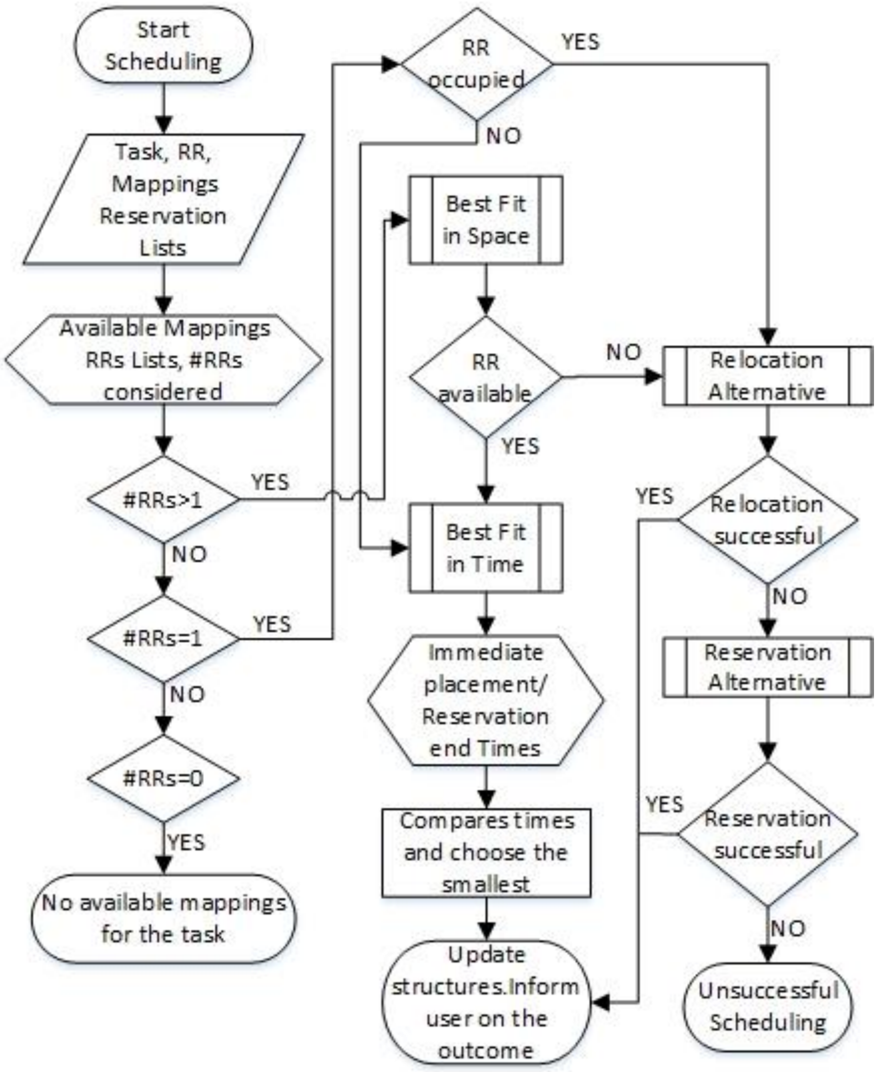
Start Scheduling
Task, RR, Mappings Reservation Lists
Available Mappings RRs Lists, #RRs considered
#RRs>1
YES
NO
#RRs=1
YES
NO
#RRs=0
YES
No available mappings for the task
RR occupied
YES
NO
Best Fit in Space
RR available
NO
YES
Best Fit in Time
Immediate placement/ Reservation end Times
Compares times and choose the smallest
Update structures.Inform user on the outcome
Relocation Alternative
Relocation successful
YES
NO
Reservation Alternative
Reservation successful
YES
NO
Unsuccessful Scheduling

Figure 7 - Operation flow of the RTSM

One of the main functions of the RTSM is the scheduling function. Its scope is to successfully place the newly arrived task (arrival of task is defined explicitly by the task graph, i.e. a new task starts when the previous task with which it has a dependency has completed its execution either because the execution time has elapsed, or, a signal by the task is raised to indicate task completion) on the device or schedule it for a later start. In order to schedule the task on the reconfigurable device, the RTSM first creates a list of the available mappings for the newly arrived task. According to the PRRs these mappings have been created for, a list of PRRs is also created; this list contains the PRRs the task has mappings for. If this list contains more than one PRR, a Best Fit (BF) policy decides the PRR in which the task will be loaded. The function that implements the Best Fit policy also checks whether the PRR is free at this moment. The Best Fit policy will place the newly arrived task on the PRR, minimizing the unused area in the PRR. **Figure 7** shows the flow chart of the scheduling operation.

During initialization, the RTSM is fed with basic 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).
(2) *Task graph:* The RTSM should know the execution order of tasks and their dependencies; this is provided with a task graph. RTSM supports complex graphs with properties like forks and joins, branches and loops for which the number of iterations is unknown at compile-time.
(3) *Task information:* Execution time of SW and HW tasks, and reconfiguration time of HW tasks should be known to the RTSM; they can be measured at compile-time through profiling. A task's execution time might deviate from the estimated or profiled execution time so the RTSM should react adapting its scheduling decisions.

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 (increasing placement choices) and/or be differently optimized, e.g. in terms of performance, power, etc.
(2) *Reservation list:* If a task cannot be served immediately due to resource unavailability, it is reserved in a queue for later configuration/execution. A HW task will wait in the queue until an RR is available, or it is assigned to the SW-PE.

(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, etc.), the RTSM may find it beneficial to load this task's bitstream to another RR is such a binding exists.

(4) *Configuration prefetching:* Allows the configuration of a HW task into an RR ahead of time. 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.

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

(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

(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.

3.1.2 RTSM Input and Execution Flow

The input to the RTSM is the partitioning of the FPGA in partially reconfigurable HW-PE resources, the availability of SW-PE resources, the tasks to be scheduled, the task graph representation describing task dependencies, and the available task mappings, i.e. bitstreams for the different implementations of each hardware task, and tasks implemented in software that can be served by a SW-PE. Additionally, the RTSM needs the reconfiguration and execution times of each task, and optionally the task deadlines. This information is used to update the RTSM structures and perform the initial schedulingThe RTSM uses lists to represent the reconfigurable regions (RR list); the tasks to be executed (task list); the bitstreams for each task (mappings list); and reservations for "newly arrived" tasks waiting for free space (reservation list). During application execution the RTSM changes its scheduling decisions dynamically. Specifically, it reacts according to dynamic parameters such as the runtime status of each HW/SW task and region, e.g. busy executing, idle, scheduled for reconfiguration, scheduled for execution, free/reconfigured-idle/reconfigured-active region etc. The approach is dynamic, i.e. at each point of time the RTSM reacts according to the FPGA condition and task status.

The RTSM is structured in two phases. In the first phase the RTSM continuously checks for newly arrived tasks. After the scheduling decision is made, the RTSM checks if the task can be served immediately and then whether the *Reuse Policy* can be invoked, and accordingly issues an execution or reconfiguration instruction. Then, it checks if a task has completed execution, and decides which task to schedule next according to the task graph. Finally, the RTSM checks if there are reserved tasks that should start executing.

The second phase is the main scheduling function of the RTSM. In order to reach a scheduling decision for a task, the RTSM follows a complex process, employing a variety of functions and policies. The RTSM tries to first exploit the available bitstreams in terms of space and then tries to find a solution that will provide the task with the best ending time. The RTSM creates a list of the available mappings and of the available RRs, for which a mapping exists, for the newly arrived task. If the later list contains more than one RR, a Best Fit policy decides which RR the task will be placed on, considering the area occupied by the task. If no free RR could be found on the available RRs list, but there are free RRs that have no corresponding mappings for the task, the scheduler performs *Relocation.* With this step the scheduler tries to relocate a previously placed task to another RR so as to accommodate the newly arrived task. If this step is also unsuccessful, the scheduler will attempt to make a reservation for the newly arrived task, thus execute it at a later time.

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

3.2 PDM Architecture

We received from the Politecnico di Milano an embedded Architecture based on the XUP V5 Board which is an evaluation platform of the Xilinx V5LX110T FPGA.

The system architecture consists of:

- **Two (2) MicroBlaze SW Processors**
    - 1st MB acts as a basic Runtime System Manager, and serves the Memory Requests from the Reconfigurable Areas
    - 2nd MB acts as SW Processing Element, Reconfiguration Manager, and reads data from the Compact Flash
- **Compact Flash, SysACE Controller,** on the CF are stored the partial bitstreams, of all the Hardware Tasks, the input image, and there are stored also the output images from every stage of the Edge Detection Process.
- **DDR2 SDRAM Memory,** which both MB processors can access, through the PLB interface
- **RS232_Uart Interface,** as standard output
- **Two PRRs,** whose reconfiguration is handled by the 2nd MB processor through the ICAP interface and their memory access requests are handled by the 1st MB processor, through interrupts.
- **A 32bit Timer/Counter,** used for timing purposes
- **A MAILBOX module,** in order for the two MB SW Processors to exchange information one another
- **Two PLB buses,** one for each MB processor, in order for each processor to be able to communicate with its attached peripherals.
- **A Hardware ICAP,** for reconfiguration of the Reconfigurable Regions.

We can see a general diagram of the architecture in **Figure 7** as presented in (6).

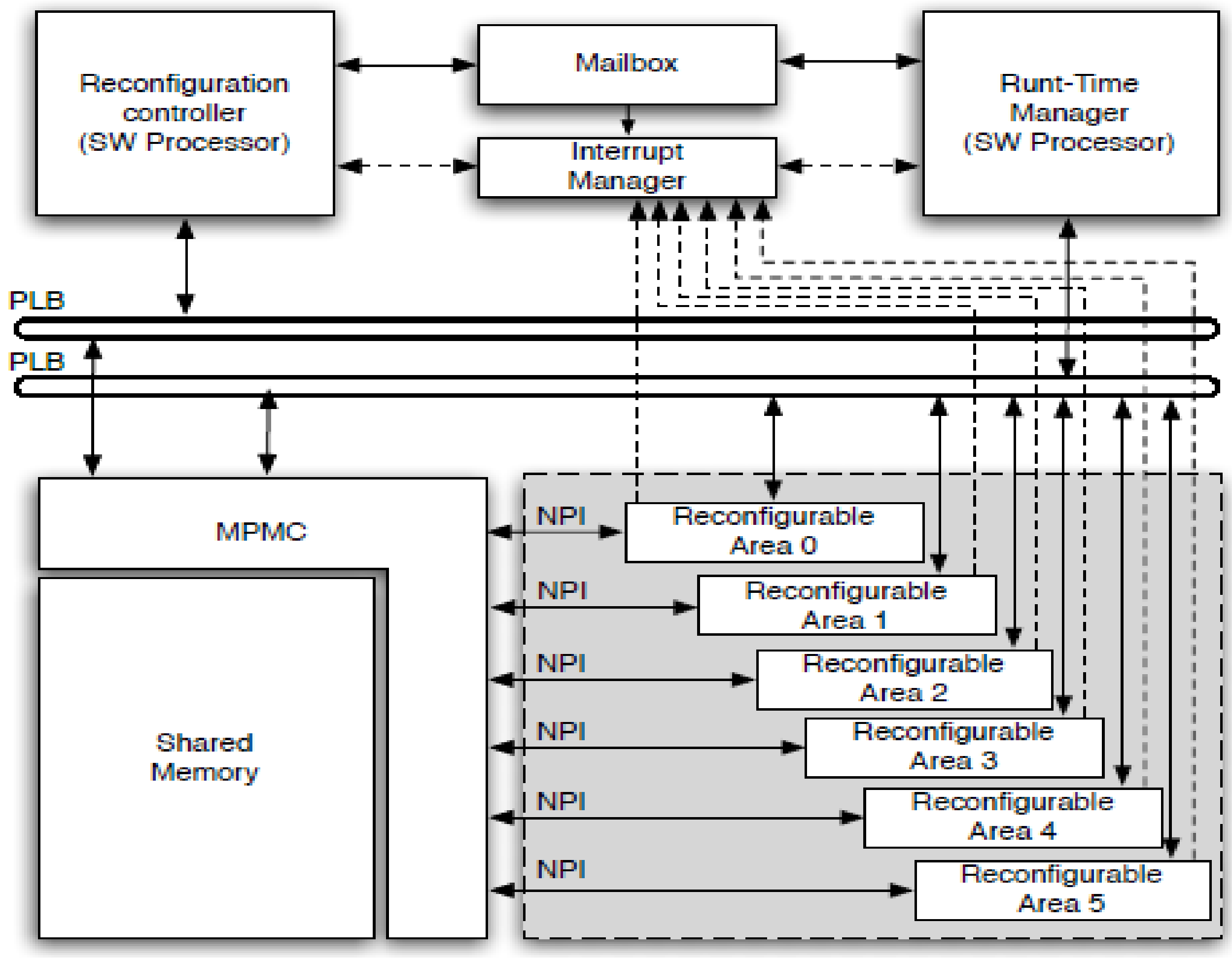

Figure 3 - PDM Target Architecture

On power up the Run-Time Manager Processor initializes the system, configures the use of interrupts, sets up the Timer/Counter, Mailbox etc.

After system initialization the Edge Detection application is initialized. The Edge Detection application consists basically of four tasks that have to be executed on an input image. Namely those tasks are Grayscale, Gaussian Blur, Edge Detection and Threshold. Input and output from and to the SD card is done by software and the corresponding tasks are treated as software (SW) tasks.

Application initialization is hardcoded and doesn't use information from other sources e.g. an XML file. It consists of tasks, SW/HW implementations, mapping and processing elements registration. A static dependency matrix is used in order for the RTM to know the dependencies between tasks.

Based on the above mentioned dependency matrix the RTM issues the commands for the PE/Reconfiguration Controller Processor to execute the relative tasks.

The complete list of tasks in the order that they are executed follows:

1. **Read Image (SW Task)**, the PE reads the input image from the Compact Flash and stores it on a predefined address in the DDR2 SDRAM
2. **Reconfiguration of PRR0 with the Grayscale bitstream** (gr.bit). PE/Reconfiguration Controller Processor through the ICAP interface
3. **Execute Grayscale (HW Task)**, PRR0 starts the execution of grayscale task on the input image stored in DDR2 SDRAM and writes the results on also on a predefined address in the DDR2 SDRAM, memory requests from the downloaded bitstream (reads & writes) are served by the Run-Time Manager Processor
4. **Write Grayscale Image (SW Task)**, the PE writes the grayscaled image to the Compact Flash from a predefined address in the DDR2 SDRAM
5. **Reconfiguration of PRR1 with the Gaussian Blur bitstream** (gb.bit). PE/Reconfiguration Controller Processor through the ICAP interface
6. **Execute Gaussian Blur (HW Task)**, PRR1 starts the execution of Gaussian blur task on the grayscaled image stored in DDR2 SDRAM and writes the results on also on a predefined address in the DDR2 SDRAM, memory requests from the downloaded bitstream (reads & writes) are served by the Run-Time Manager Processor
7. **Write Blurred Image (SW Task)**, the PE writes the blurred image to the Compact Flash from a predefined address in the DDR2 SDRAM
8. **Reconfiguration of PRR0 with the Edge Detection bitstream** (ed.bit). PE/Reconfiguration Controller Processor through the ICAP interface
9. **Execute Edge Detection (HW Task)**, PRR0 starts the execution of Edge Detection task on the blurred image stored in DDR2 SDRAM and writes the results on also on a predefined address in the DDR2 SDRAM, memory requests from the downloaded bitstream (reads & writes) are served by the Run-Time Manager Processor
10. **Write Edge Detected Image (SW Task)**, the PE writes the edge detected image to the Compact Flash from a predefined address in the DDR2 SDRAM
11. **Reconfiguration of PRR1 with the Threshold bitstream** (th.bit). PE/Reconfiguration Controller Processor through the ICAP interface
12. **Execute Threshold (HW Task)**, PRR1 starts the execution of Threshold task on the edge detected image stored in DDR2 SDRAM and writes the results on also on a predefined address in the DDR2

SDRAM, memory requests from the downloaded bitstream (reads & writes) are served by the Run-Time Manager Processor

13. **Write Output Image (SW Task)**, the PE writes the edge detected image to the Compact Flash from a predefined address in the DDR2 SDRAM

All the "start" commands are issued by the Run-Time Manager Processor to the PE/Reconfiguration Controller Processor through MAILBOX and some predefined addresses in the DDR2 SDRAM where we write the task names and attributes. The "end task" signals from the PE/Reconfiguration Controller Processor to the Run-Time Manager Processor are issued through MAILBOX for the SW tasks and Reconfigurations, and by Interrupts for the HW tasks.

As we mentioned above the partial reconfiguration of the FPGA is handled by the PE/Reconfiguration Controller Microblaze Processor so is actually a software task, but one that is added and handled by the RTSM and is not part of the application task graph.

3.3 RTSM Integration

3.3.1 Desktop Based RTSM

In order for us to intergrade our Runtime System Manager in the system architecture that was described above we decided, to follow a step by step approach. As a first step we decided to implement a Desktop Embedded Platform, were the Desktop - Host Computer would read the configuration file that was produced by the XML parser and would schedule the tasks based on information from it. The embedded platform would receive task issue commands (SW/HW/Reconfiguration) from the Host Computer, would execute them and then respond to the host computer with the appropriate task completion signal. In order to implement the communication between the two (Host Computer – Embedded Platform) we used the RS232 UART connection between them, which until now was used only as the standard output of the embedded device.

In order to implement this UART communication between the Desktop – Host computer and the embedded platform, we removed all the task scheduling functionality from the Run-Time Manager Processor, and in its place we implemented the UART Communication protocol between XUPV5 board and the host processor that runs the RTSM. The rest of the processors functionality and in particular the interrupt service routines that served the Reconfigurable Areas remained as is. In **Figure 9** we can see the Host Computer – Embedded Platform Architecture.

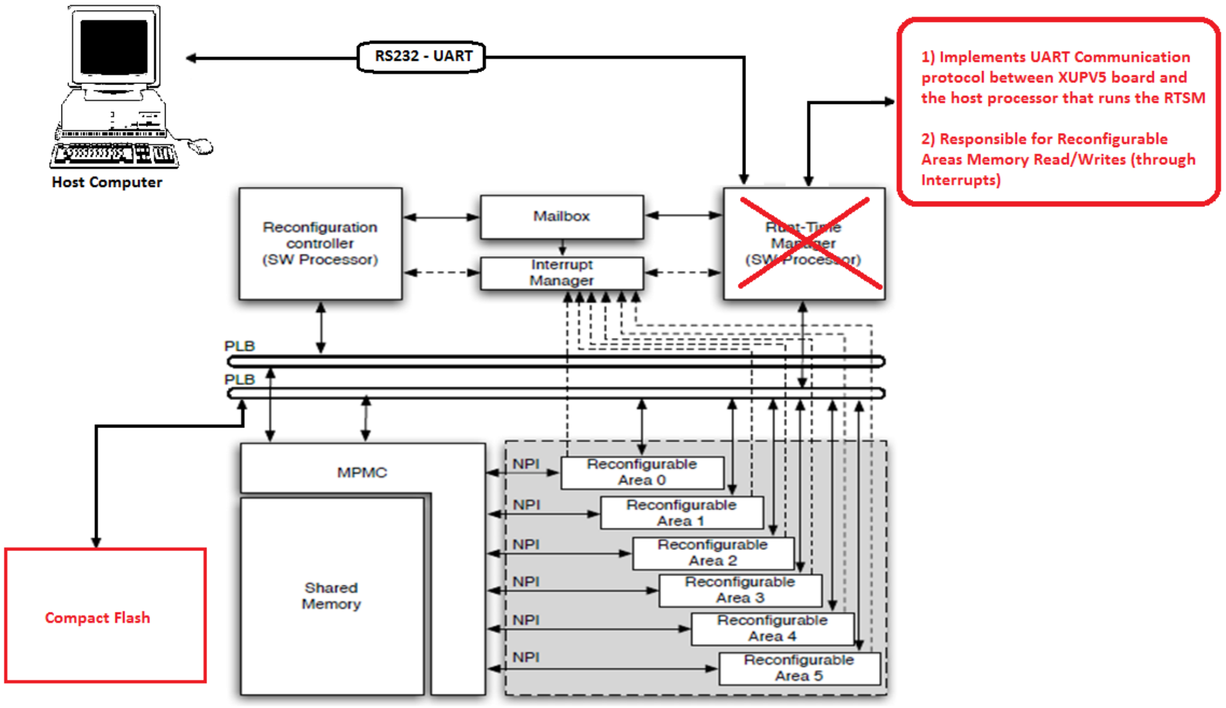
Host Computer
RS232 - UART
1) Implements UART Communication protocol between XUPV5 board and the host processor that runs the RTSM
2) Responsible for Reconfigurable Areas Memory Read/Writes (through Interrupts)
Reconfiguration controller (SW Processor)
Mailbox
Interrupt Manager
Run-Time Manager (SW Processor)
PLB
PLB
MPMC
Shared Memory
Compact Flash
NPI
Reconfigurable Area 0
Reconfigurable Area 1
Reconfigurable Area 2
Reconfigurable Area 3
Reconfigurable Area 4
Reconfigurable Area 5

Figure 9 - Desktop System Architecture

The communication protocol that we implemented is based on short character codes exchanged between the Host Computer and the XUPV5 embedded platform.

3.3.1.1 RTSM to Embedded Platform SW commands

- SW_ noOperation → **S0**
- SW_reconfigure → **S1**
  - **C1** (Greyscale RecArea0)
  - **C2** (Gaussian Blur RecArea1)
  - **C3** (Edge Detection RecArea0)
  - **C4** (Thresshold RecArea1)
- SW_imageRead → **S2**
- SW_imageWrite → **S3**
  - **W1** (Write Gray)
  - **W2** (Write Gauss)
  - **W3** (Write Edge)
  - **W4** (Write Image)

3.3.1.2 Embedded Platform to RTSM SW task termination Responses

- RECONF_TERM → **RT**
- SW_TERM → **ST**

3.3.1.3 RTSM to Embedded Platform HW commands

- GreyScale → **H1**
- Gaussian Blur → **H2**
- Edge Detection → **H3**
- Threshold → **H4**

3.3.1.4 Embedded Platform to RTSM HW task termination Responses

- HW_TERM (RecArea0) → **HT2**
- HW_TERM (RecArea1) → **HT3**

For beeter understanding if for example the Host Computer wants to initiate let's say the software **task in order to write the edge detected image to the Compact Flash**, would have to send the command S3W3 to the embedded platform through the RS232 UART interface. And after the completion of the SW task the embedded platform again through the RS232 UART interface would send the **ST** command to the Host Computer.

3.3.2 Full Embedded RTSM

Next step was to port the RTSM to the system architecture that we described earlier in this document, and was provided to us by PDM. This system architecture key elements as we mentioned above consist of two Microblaze software processors, the Run-Time Manager Processor and the PE/Reconfiguration Controller Processor, the DDR2 memory and two Reconfigurable Regions.

One of the issues that quickly emerged while porting the RTSM from Technical University of Crete, which was built on a desktop workstation with GNU/Linux OS, was the fact that the Run-Time Manager Processor didn't have access to the compact flash, since the SysACE controller is connected with the PLB of the PE/Reconfiguration Controller Processor, and no PLB-bridge existed. Hence there was no way for the Run-Time Manager Processor to access and read the configuration file which was produced by the XML parser.

The solution was that the PE/Reconfiguration Controller Processor was to load the system configuration file from the compact flash during initialization and copy it on o predefined memory address in the DDR2 SDRAM. After this initialization step was completed the PE/Reconfiguration Controller Processor would send a new message to the Run-Time Manager Processor, by MAILBOX, that the initialization is over and that it is safe to parse the configuration file from the previously determined memory location at this point.

On the same time the initialization of the Run-Time Manager Processor, comes to a stop until it receives the “initialization end” command, by MAILBOX, from the PE/Reconfiguration Controller Processor.

After the Run-Time Manager Processor receives the “initialization end” signal from the PE/Reconfiguration Controller Processor, the RTSM parses the contents of the memory location where the application configuration settings exist, and according to those takes place the initialization of the run-time scheduler. During this process emerged the need to add a new terminating character to the configuration file, since now that the file resides in memory there is no EOF. For this reason the “X” was chosen as a terminating character for the configuration file, and the XML parser was changed accordingly in order to comply. In **Figure 10** we can see the new Full Embedded System Architecture.

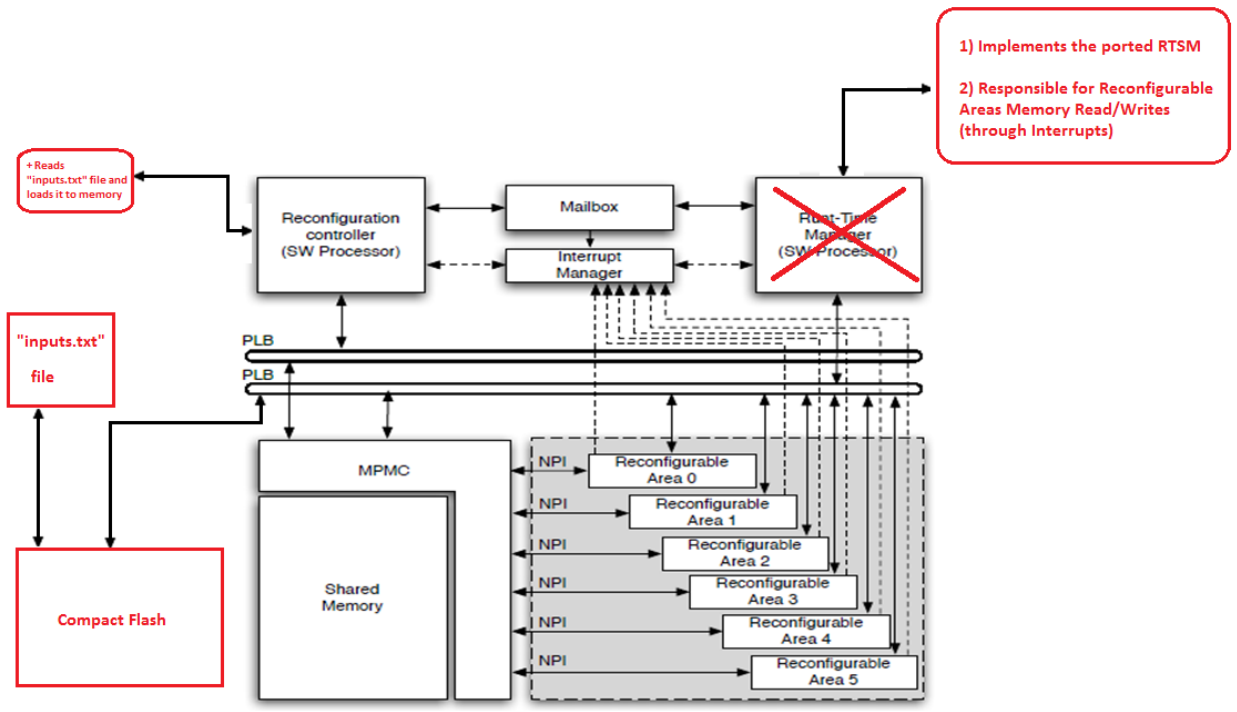
1) Implements the ported RTSM
2) Responsible for Reconfigurable Areas Memory Read/Writes (through Interrupts)
+Reads "inputs.txt" file and loads it to memory
Reconfiguration controller (SW Processor)
Mailbox
Interrupt Manager
~~Run-Time Manager (SW Processor)~~
"inputs.txt" file
PLB
PLB
MPMC
Shared Memory
Compact Flash
NPI
Reconfigurable Area 0
Reconfigurable Area 1
Reconfigurable Area 2
Reconfigurable Area 3
Reconfigurable Area 4
Reconfigurable Area 5

Figure 10 - Full Embedded System Architecture

As we have mentioned above the demo application treats the read and write image tasks from and to the Compact Flash as software tasks which are part of the users application and not only as input/output functions utilized merely for debugging purposes.

3.4 System Extensions - New Features

In the previous chapter we discussed the process of integrating the RTSM from TUC with the system architecture of Politecnico di Milano. The obvious advantage of this is that the task scheduling isn't static anymore and the RTSM can be utilized in order for as to execute more complex applications faster and more efficient.

Some other system extensions or new features that we developed on top of this system architecture are: new software mappings for all the hardware tasks, new hardware mappings and finally the implementation and integration of microreconfiguration option for the hardware threshold module.

3.4.1 New S/W and H/W mappings

As a new addition to the application are the software mappings for the existing hardware tasks namely Greyscale, Gaussian Blur, Edge Detection and Threshold. Each set of data is stored in a preconfigured memory address, so it makes it possible to use only hardware, only software or every combination of both on the same set of data. As it is obvious except for the new software functions also new mapping and new commands between the two processors were also implemented in order for them to be functional.

The initial system architecture included only one hardware mapping for each task for a specific reconfigurable region, we produced hardware mappings for every reconfigurable region so the RTSM has more than one mapping for each task and can utilize the system's resources in the most efficient way in terms of time and resources.

3.4.2 Microreconfiguration

Dynamic Circuit Specialization (DCS) is an optimization technique developed by the Ghent University, and is used to implement a parameterized hardware design. A design is said to be parameterized if some of the inputs (parameters) are infrequently changing compared to the other inputs. Instead of implementing these parameter inputs as regular inputs, in the DCS approach these inputs are implemented as constants and the design is optimized for these constants. When the parameter values change, the design is re-optimized for the new constant values by reconfiguring the FPGA.

The DCS tool flow consists of two stages: the generic stage and the specialization stage. In the generic stage, the design with parameterized inputs described in a Hardware Description Language (HDL) is processed to yield a Partial Parameterized Configuration (PPC), which contains the bitstream expressed in the form of Boolean functions. In the specialization stage, the Boolean functions are evaluated for a specific parameter value by the Configuration Manager to generate a specialized bitstream. Usually the configuration manager is implemented on an embedded processor. The embedded processor is responsible to swap the specialized bitstream into the configuration memory using the HWICAP. Thus it forms a reconfiguration called Micro-reconfiguration. This technique can be implemented using an automatic tool flow called TLUT tool flow (7).

The edge detection application aims to identify points in an image at which the brightness changes sharply, or more formally, where discontinuities exist. In order to detect edges in an image, we have to load the image from the file system to memory (SW task), and execute at the following order, Gaussian Blur (SW or HW task) running a matrix over the image in order to "smooth" it, Edge Detection (SW or HW task) running a matrix over the image in order to identify areas with high contrast, Thresholding (SW or HW task) examining if a pixel value after edge detection is over or under a limit that categorizes pixels on an edge or not, and finally write down the edge detected image to the file system again (SW task). During the edge detection process the resulting images after each stage where also saved to the file system so some other software tasks resulted. As we can see the edge detection application is a perfect candidate in order to demonstrate the RTSM capabilities and operation, and since it chosen as a demo application for the RTSM we had to find a way to include also the micro-reconfiguration capabilities of the system. From the beginning the threshold level was an obvious choice since it gave us the opportunity to change the pixel threshold level each time without the need for different bitstreams. Hence it was the ideal option in order to demonstrate micro-reconfiguration along with the other system features in a logical and meaningful way.

3.4.2.1 Parameterized Threshold module

The Threshold module contains a state machine with 18 different states. We identified the threshold level value to be infrequently changing signal and it is user configurable. Hence the threshold value can be parameterized. However, for the parameterization we make use of existing "Initialize (INIT)" and "Set Threshold (SET_THRESHOLD)" states only, since these two states are important in setting up the threshold level values that are used during edge detection process.

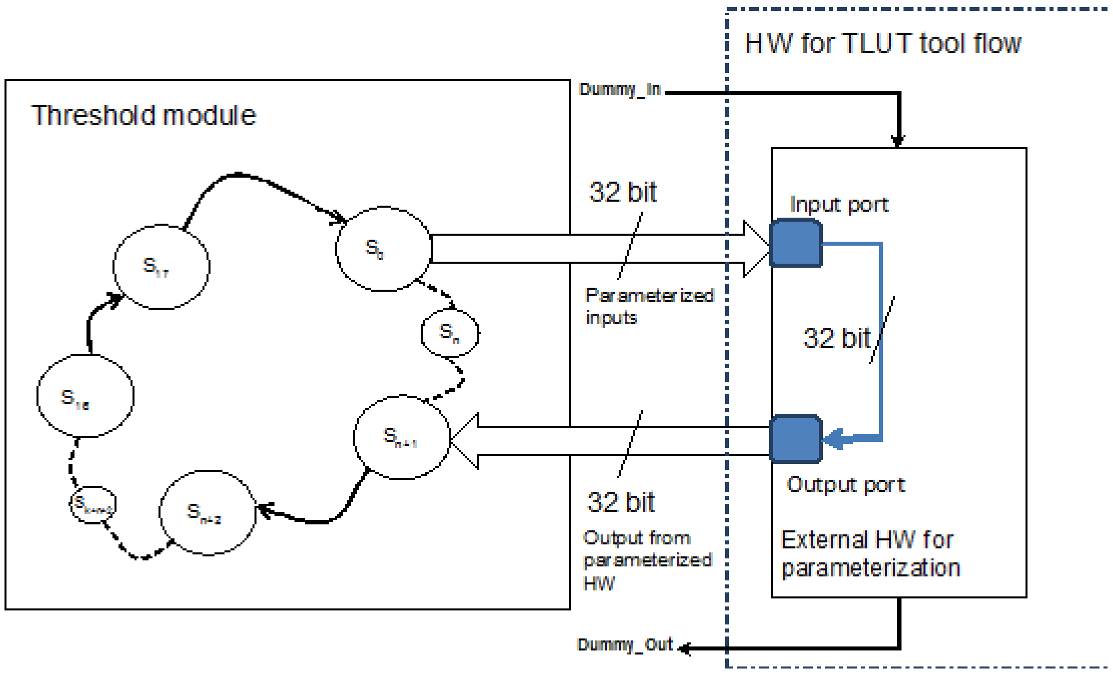

Figure 11 - Parameterized Threshold module

We have created an external hardware module, which has an input and an output of 32-bit width as shown in **Figure 11**. There also exist a dummy input and a dummy output port, which is required to avoid the synthesis optimization.

The RTSM contains a soft-core MicroBlaze processor which is suitable for the implementation of the micro-reconfiguration's configuration manager. Thus there is no need for another MicroBlaze that is dedicated only for micro-reconfiguration purposes. Hence the configuration manager is now also an integrated part of the RTSM. The RTSM uses the PLB bus and can also be used to connect the micro-reconfigurable modules.

Since micro-reconfiguration only requires, knowledge of the TLUTs locations and a MicroBlaze with access to the HWICAP in order to execute the micro-reconfiguration function itself, the MicroBlaze Process Element CPU1 was the obvious choice since only a new software function had to be implemented and the RTSM is going to deal with micro-reconfiguration the same way as every other software function. On **Figure 3**, CPU1 is described as "Reconfiguration controller (SW Processor), and is in charge of executing software tasks, partial reconfiguration, reads/writes data from and to the Compact Flash and microreconfiguration as a new feature. In order for us to add the micro-reconfiguration capability to the system, additions were required to both the RTSM (Run Time System Manager – MicroBlaze_0) and the SW PE (Software Processing Element – MicroBlaze_1).

3.4.2.2 *RTSM* Additions

In the RTSM some new global flags were added, that would inform the system manager that micro-reconfiguration should occur. Those flags are the micro-reconfiguration flag (micro_flag) the micro-reconfiguration value flag (micro_value) and the mapping in which micro-reconfiguration should occur (micro_mapping).

In our scheduler we recognize two conditions in which micro-reconfiguration is needed. The first is when the user decides to send a micro-reconfiguration instruction before the task is scheduled or even the dependencies regarding it have been solved. In this scenario the user having knowledge about his application sends a micro-reconfiguration instruction early during the execution of the application. In this case the scheduler will take in consideration only the mapping in which the micro-reconfiguration will occur and first schedules the "plain" reconfiguration of the task and then the micro-reconfiguration.

The second condition is when the user decides to send a micro-reconfiguration instruction while the task is running on the device. In this case the RTSM recognizes that the task mentioned is already being executed so it sets a re-execution flag that will signal that the specific task needs to re-execute after having performed micro-reconfiguration on the area. All the above is done provided that the mapping that is already placed on the device is the one that is micro-reconfigurable. Otherwise the scheduler performs the steps mentioned in the first condition, first reconfiguration and then micro-reconfiguration.

Also some new drivers were added for the RTSM to be able to write the correct micro-reconfiguration parameters to the DDR and to successfully issue the micro-reconfiguration order to the Software Processing Element to execute.

For the time being the micro-reconfiguration command and value is given from the user through UART, but the system approach enables a software or hardware function to invoke it also the same way as the user does, by setting the micro-reconfiguration flag, and value.

3.4.2.3 Software Processing Element Additions

As stated above the micro-reconfiguration process is treated by the system, as another software function. The fact that only the Software Processing Element (MicroBlaze_1) has control over the HWICAP eliminates the possibility of two different processes trying to access at the same time, since only one software process can run on the microprocessor at a given time.

After a micro-reconfiguration command is issued by the RTSM the PE is accessing the corresponding DDR locations and reads the value that should be micro-reconfigured. Then the correct TLUT values are evaluated and written to the corresponding TLUT locations which are extracted at pre-compiled time using the information provided by the placed & routed hardware design.

3.4.2.4 Micro-reconfiguration as a part of Modular reconfiguration (Partial reconfiguration)

The system architecture consists of two Partial Reconfigurable Areas:

1) recArea0
2) recArea1

The resource details of the reconfigurable partition areas are tabulated in **Table 1**.

The following are the Reconfigurable Modules which are partially reconfigured within the above stated reconfigurable areas:

1) Grayscale (recArea0)
2) Gaussian Blur (recArea1)
3) Edge Detection (recArea0)
4) Threshold (recArea1)

The FPGA physical resource utilization of the two reconfigurable partitions are tabulated in

Table 2 and **Table 3**.

Table 1: Reconfigurable partitions

Reconfigurable Partition Area name	**Boundaries**	**Number of LUTs**	**Number of BRAMs**
recArea0	SLICE_X16Y62:SLICE_X37Y79 DSP48_X0Y26:DSP48_X0Y31 RAMB36_X1Y13:RAMB36_X1Y15	1584	3
recArea1	SLICE_X8Y129:SLICE_X41Y139 DSP48_X0Y52:DSP48_X0Y55 RAMB36_X1Y26:RAMB36_X1Y27	1496	2

Table 2: Reconfigurable partition recArea0 resources

Physical Resource Estimates recArea0			
Site Type	Available	Required	% Util
LUT	1584	726	46
FD_LD	1584	640	41
SLICEL	270	134	50
SLICEM	126	63	50
DSP48E	6	2	34
RAMBFIFO36	3	0	0

Table 3: Reconfigurable partition recArea1 resources

Physical Resource Estimates recArea1			
Site Type	Available	Required	% Util
LUT	1496	502	34
FD_LD	1496	445	30
SLICEL	264	96	37
SLICEM	110	40	37
DSP48E	4	1	25
RAMBFIFO36	2	0	0

The reconfigurable modules can be subjected to the TLUT tool flow which creates TLUTs within the modules thus creating parameterized partially reconfigurable modules. Once these modules are partially reconfigured, the micro-reconfiguration can be performed to change the truth table entries of the TLUTs within the partial reconfigurable modules.

3.4.2.5 HWICAP control

The micro-reconfiguration module is also a part of the RTSM, since the MicroBlaze (MicroBlaze_1) in the RTSM can also be used for the configuration management of the micro-reconfiguration. There is no need of handshaking between the micro-reconfiguration configuration manager and the RTSM to access the HWICAP.

We make use of the HWICAP that was initialized by the RTSM for the partial reconfiguration. We have made micro-reconfiguration function “mreconfigure( );” as a separate function which contains all the necessary functions to perform the micro-reconfiguration along with the Boolean function of parameters evaluation, setting the parameters and making use of the “*XhwIcap_setClb_bits*" driver function.

We have added a check to make sure that the configuration manager of the micro-reconfiguration subsystem releases access to the HWICAP after the micro-reconfiguration is completed.

3.4.2.6 Microreconfiguration Implementation

We implemented the parameterized Threshold module of which the threshold value levels are parameters. These parameters can be set by the user using an UART interface. For every change in threshold value by the user, the truth table entries of the TLUTs are reconfigured. The resource size of the parameterized Threshold module is tabulated in **Table 4**.

There are 33 (32 Output Ports + 1 Dummy_Out) TLUTs within the parameterized Threshold module. The 32-bit output ports are the output from the 32 TLUTs and there is one extra TLUT for the “Dummy_Out”.

Table 4: TLUTs and LUTs size of the parameterized Threshold module

Parameterized module	Number of TLUTs	Total LUTs
Threshold	33	568

The micro-reconfiguration comes at the cost of its overheads. There are three important overheads we need to consider:

1) *Reconfiguration time*: This is the time it takes to reconfigure the truth table entries of the TLUT.
2) *Evaluation time*: This the time it takes to evaluate all the Boolean functions of the parameters in a parameterized design.
3) *Boolean functions - Memory Size*: This is the memory occupied by the Boolean functions in a BRAM for micro-reconfiguration.

Table 5 lists all the overhead values of the micro-reconfiguration.

Table 5: Overheads of the micro-reconfiguration

Time to reconfigure 32 TLUTs (ms)	Boolean functions evaluation time (ms)	Size of the Boolean functions (KB)
3.84	0.24	6.4

A comparison of LUT resources between conventional implementation and DCS implementation of the Threshold module is tabulated in **Table 6**. We observe that there was a gain of approximately 6 % of the LUT resources by using the DCS implementation and it also provides an additional option of changing the Threshold level values at the hardware during run time.

Table 6: Implementation comparison

Parameterized module	Total LUTs by Conventional implementation	Total LUTs by DCS implementation	Gain
Threshold	600	568	≈6%

We used the TLUT tool flow to implement the parameterized Threshold module of the RTSM in which the threshold level values are infrequently changing / user configurable parameters. We interfaced the parameterized Threshold module with the RTSM. For every change in parameters, the micro-reconfiguration is performed to reconfigure the threshold level values at the hardware level. We found that there is an area gain of ≈6% compared to the conventional implementation of the Threshold module. The same approach can be used to implement the other parameterized partially reconfigurable modules.

4. ZedBoard Architecture Implementation

ZedBoard is a low-cost development board for the Xilinx Zynq™-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 7**.

Table 7: Z-7020 Characteristics

Processor Core	Dual ARM® Cortex™-A9 MPCore™ with CoreSight™
Processor Extensions	NEON™ & Single / Double Precision Floating Point for each processor
L1 Cache	32 KB Instruction, 32 KB Data per processor

L2 Cache	512 KB
On-Chip Memory	256 KB
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 KB
DSP Slices	220

In **Figure 12** we can see a schematic representation of the Zynq-7000 All Programmable SoC and the interconnections between the Programmable System (PS) and the Programmable Logic (PL).

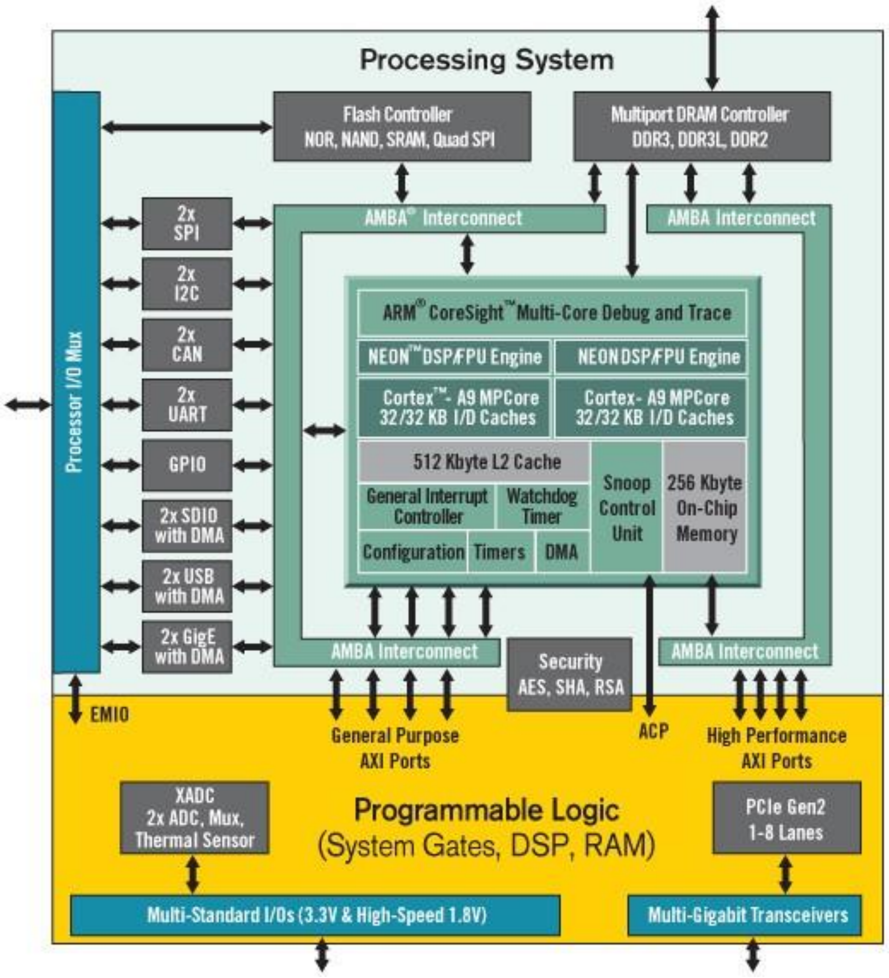
Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
AMBA® Interconnect
AMBA Interconnect
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure 12 - Zynq-7000 All Programmable SoC

In **Figure 13** we can see a Zedboard Platform and in **Figure 14** we can see all the interconnections on the Zedboard between the Xilinx Zynq™-7000 SoC and its board peripherals.

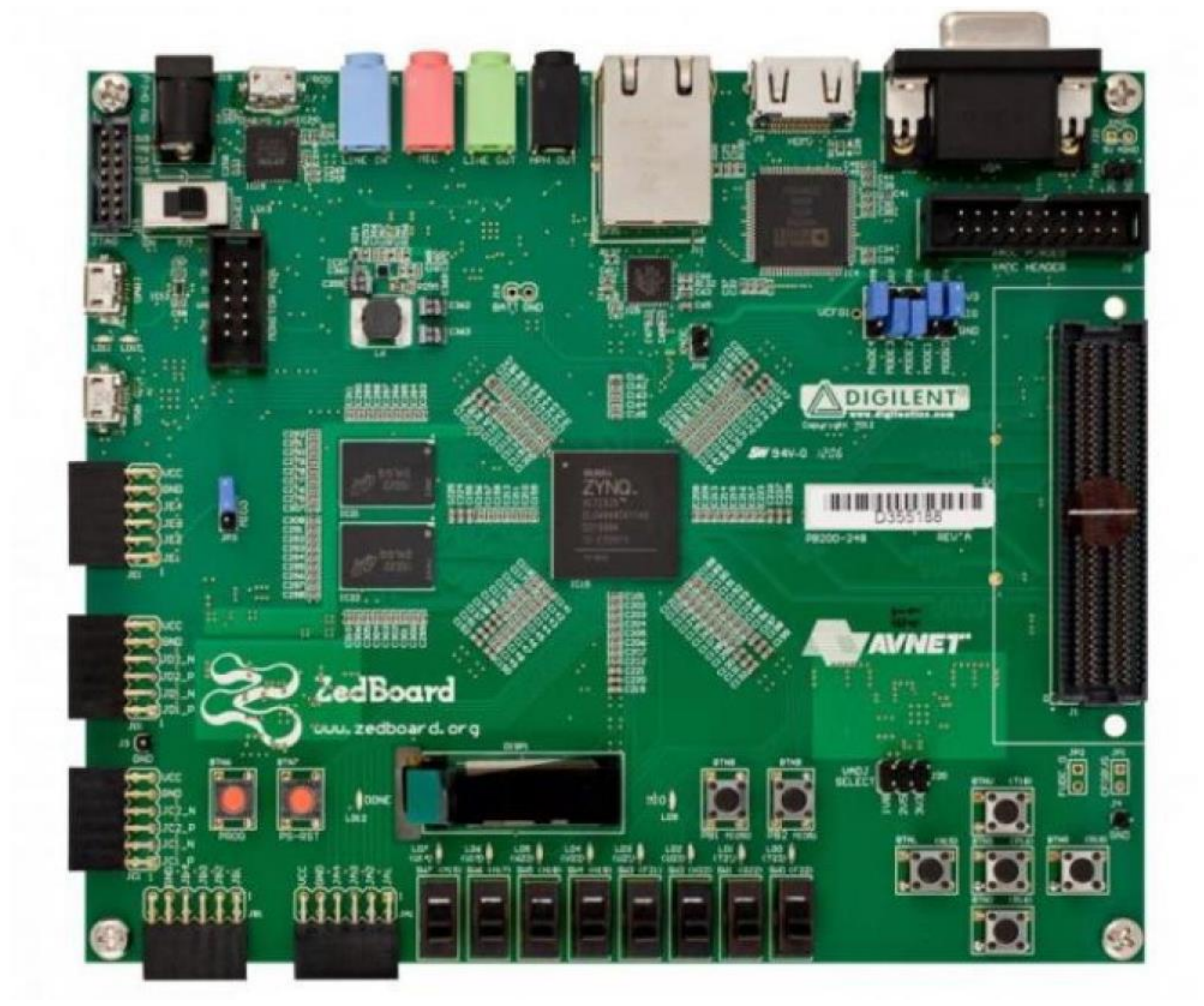

Figure 13 – The ZedBoard Platform

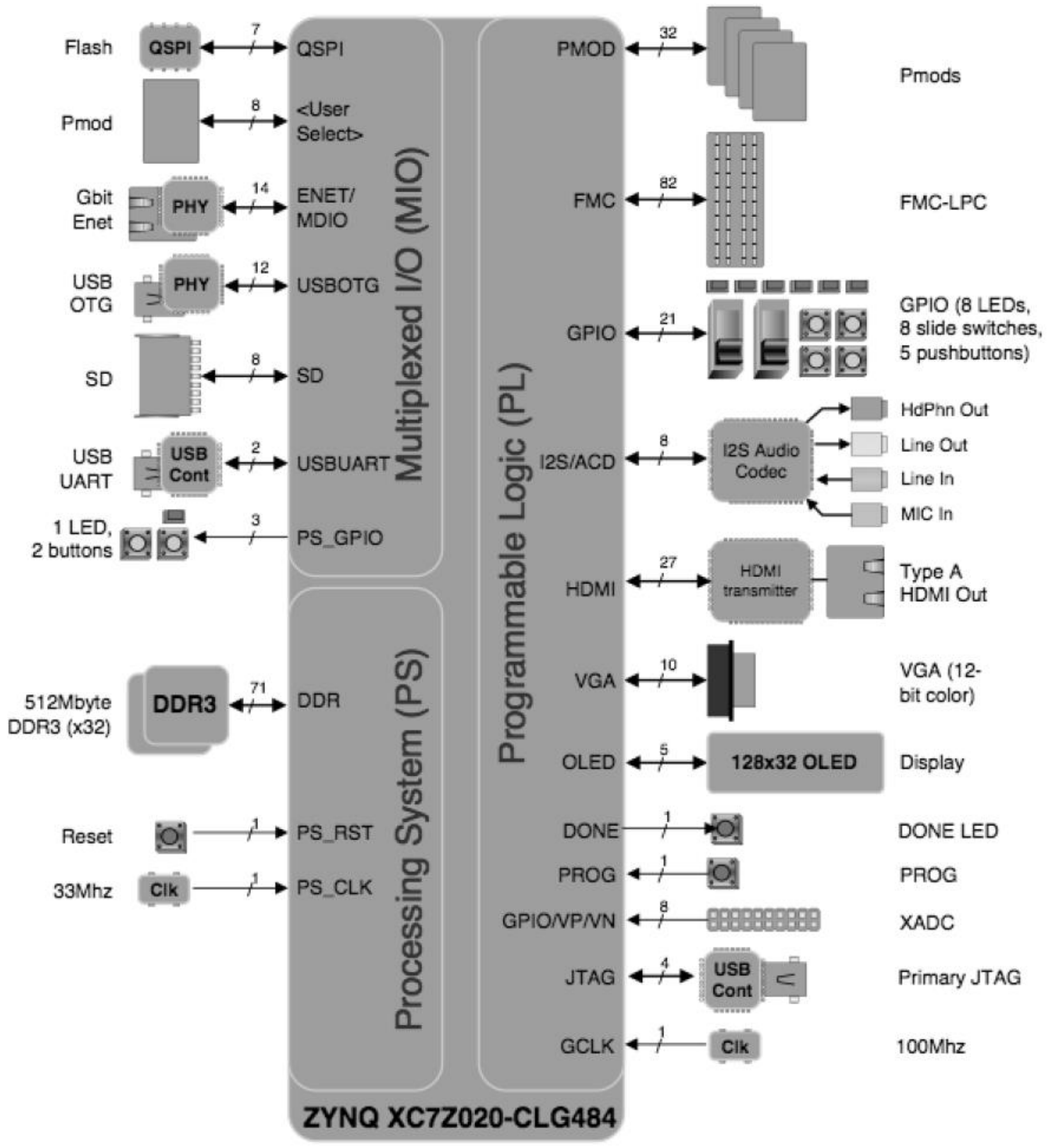

Figure 14 – ZedBoard platform Layout

4.1 Architecture

We validated the behavior of RTSM on a fully functional system on a ZedBoard platform executing an edge detection application. **Figure 15** 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.

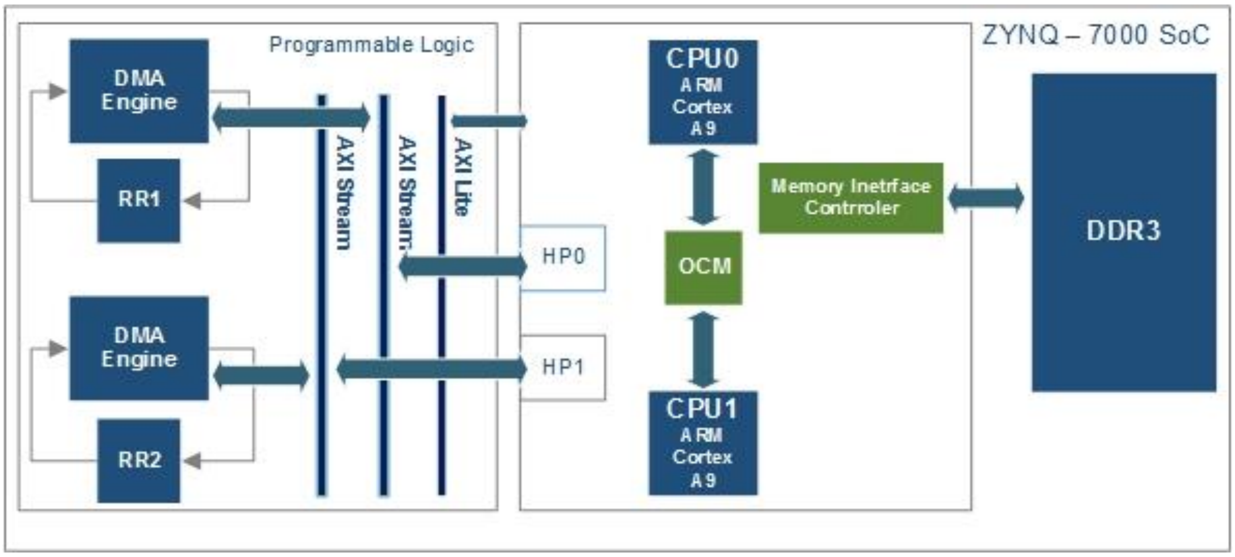

Figure 15 - System architecture of the Zynq platform

In **Figure 16** we can see an overview of the Processing System, system assembly were we can identify the PS components used by our architecture, namely those are:

- The DDR 3 memory
- Two High Performance (HP) ports, HP0 and HP1
- UART1
- USB0
- Timer0
- GPIO
- Core0_nIRQ (PPI)
- Core0_nFIQ (PPI)

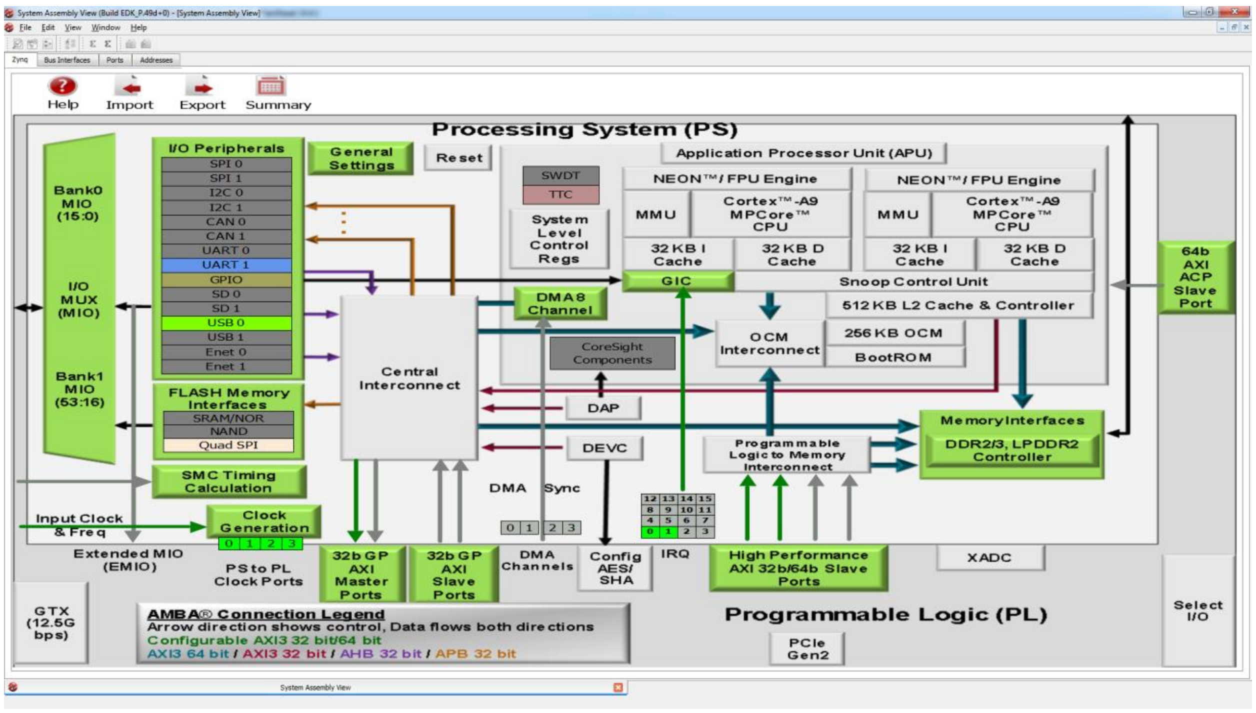

Figure 16 – Processing System Assembly

In **Figure 17** we can see graphical design view of our system Architecture

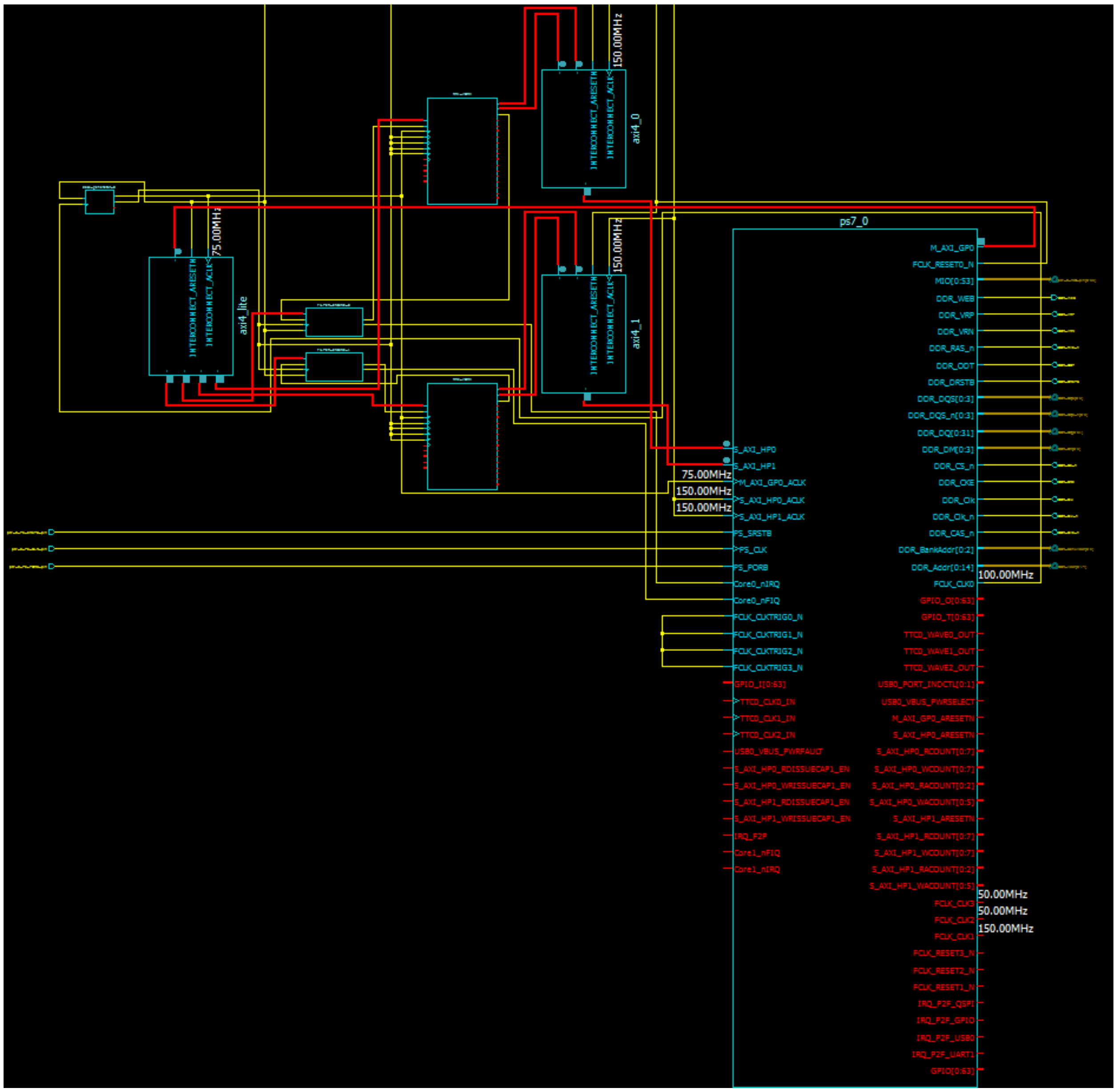

Figure 17 – System Architecture graphical design view

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 edge detection application consists mainly of four filter IP modules executing in sequence: Gray Scale, Gaussian Blur, Edge Detection and Threshold. For these tasks, we have both a HW version as partial bitstreams, and a SW version; the RTSM will decide which version (SW or HW) is better to use based on the run-time availability of HW- and SW-PEs. The input image is loaded from the SD card, while intermediate images resulting after processing each task and the final output image are also written back to SD card. The transfer from and to the SD card is performed by SW tasks running on CPU1, which also controls partial reconfiguration of HW tasks. **Figure 18** depicts the task graph of the application, showing only task dependencies but not how the application will execute over time. We also include the (implicit) SW tasks that perform the reconfiguration process.

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

In the system architecture of **Figure 15**, the two RRs are connected to the processing ARM cores through an AXI_Lite bus running at 75MHz, and through a DMA engine, each one having read and write channels on a dedicated AXI_Stream running at 150 MHz. The AXI_Stream is connected to the processor High Performance ports that provide access to DDR memory. To execute a HW task, the RTSM issues a reconfiguration command to CPU1, which in turn configures the FPGA with the corresponding bitstream through the PCAP configuration port. Once partial reconfiguration completes, RTSM initiates the HW task execution, programs the appropriate values and triggers the corresponding DMA engines and IP filters. All HW mappings of the filter modules were implemented with the Xilinx HLS tool which also creates automatically and the SW drivers for the SW/HW communication over the AXI_Lite bus. When a filter completes its execution, generates an interrupt to the RTSM, which then updates its structures and proceeds to a new scheduling decision.

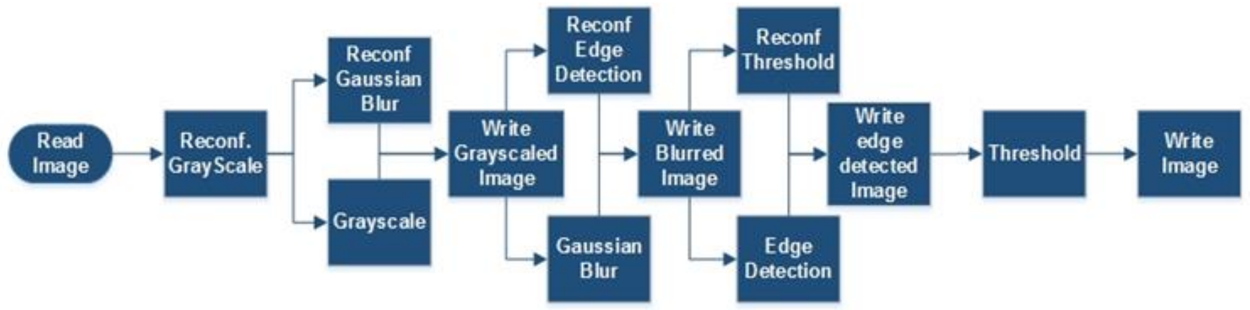

Figure 18 - Edge Detection Application Task Graph

The RTSM operation can be broken down into different phases shown in **Table 8.**

Table 8: Time per each distinct phase of the RTSM, running on ARM A9 at 667MHz.

RTSM phases	#Clock Cycles	Elapsed time (μs)
RTSM Initialization	7,707	23.121
Schedule	17,346	52.038
Issue Execution	5,995	17.985
HW Task completion	2,493	7.479
Reconfiguration task completion & Hardware task execution issue	1,224	3.672
SW Task completion	2,748	8.244

Table 8 also lists the total time spent on an ARM A9 Cortex CPU for each distinct phase, both in clock cycles and μs. The RTSM initialization phase is performed only once and it includes fetching from the flash memory the initialization file that describes the tasks the control flow graph and the task mappings, parsing it, and initializing the RTSM data structures. In the RTSM initialization time we do not include the overheads for loading the file from flash memory and for transferring the partial bitstreams to DDR3 memory. The Schedule phase refers to the time needed to execute the Schedule function in order to take a decision about the task to be executed next, i.e. when and where this task is going to be executed. The Issue Execution phase refers to the time required to issue either a reconfiguration or execution task instruction, depending on what the scheduling decision was. Also, depending on whether a HW core is reused, the RTSM checks if configuration prefetching can be performed. The HW Task completion phase & SW Task completion phase refer to updating the RTSM data structures after the completion of a HW or SW task, and to resolving the dependencies in order to set the next task in the graph as “arrived”. Finally,

the Reconfiguration task completion & HW task execution issue phase refer to the interval in which the RTSM receives a reconfiguration completion notification from the PE, issues a task execution command, and checks whether it can perform configuration prefetching of a not yet “arrived” task.

4.2 Asymmetric multiprocessing (AMP)

The Zynq-7000 AP SoC provides two Cortex-A9 processors that share common memory and peripherals. Asymmetric multiprocessing (AMP) is a mechanism that allows both processors to run their own operating systems or bare-metal applications with the possibility of loosely coupling those applications via shared resources.

Our design runs both Cortex-A9 processors in an AMP configuration, and each CPU runs a bare-metal application within its own standalone environment. Care has been taken to prevent the CPUs from conflicting on shared hardware resources.

The Zynq SoC processing system (PS) includes resources that are private to each CPU and shared by both CPUs. Care must be taken to prevent both CPUs from contending for these shared resources when running the design in an AMP configuration.

Examples of some of the private resources are:

- L1 cache
- Private peripheral interrupts (PPI)
- Memory management unit (MMU)
- Private timers

Examples of some of the shared resources are:

- Interrupt control distributor (ICD)
- DDR memory
- On-chip memory (OCM)
- Global timer
- Snoop control unit (SCU) and L2 cache
- UART0

In our design, CPU0 is the master and controls the shared resources. To keep the complexity of the design to a minimum, the bare-metal application running on CPU1 has been modified to limit access to the shared resources.

OCM is used by both processors to communicate to each other. When compared to DDR memory, OCM provides very high performance and low latency access from both processors. Deterministic access is further assured by disabling cache access to the OCM from both processors.

Actions taken by this design to prevent problems with the shared resources include:

- DDR memory: CPU0 has only been made aware of memory at 0x00100000 to 0x001FFFFF. CPU1 uses memory from 0x00200000 to 0x002FFFFF for its bare-metal application.
- L2 cache: The L2 cache is disabled because it caused problems to the memory access from the hardware modules to the DDR memory.
- ICD: Interrupts from the core in PL are routed to the PPI controller for CPU0. By using the PPI, CPU0 has the freedom to service interrupts without requiring access to the ICD.
- Timers: CPU0 and CPU1 uses their private peripheral timers.
- OCM: Accesses to OCM are handled very carefully by each CPU to prevent contention. Two OCM address locations are used as flags to communicate between the two processors. CPU0 uses the first flag in order to issue an execute command to CPU1. Execute commands vary from input/output functions such as to read an image from the SD card or execute a software function. CPU1 on the other hand clears the first flag after it reads the execute command, executes the appropriate function and after completion responds back to CPU0 by setting an appropriate value to the second flag, which is then cleared by CPU0.

4.2.1 Software

The software can be broken down into three sections:

- First stage boot loader (FSBL)
- Bare-metal application for CPU0
- Bare-metal application for CPU1

4.2.1.1 FSBL

The FSBL always runs on CPU0 and is the first software application that is run after power-on reset of the Processor System (PS). The FSBL is responsible for programming the PL and copies both application executable and linkable format (ELF) files to DDR memory. After loading the applications to DDR memory, the FSBL then starts executing the first application that was loaded. The version of FSBL we are using supports multiple data or ELF files. The current FSBL first looks for a bit file. If a bit file is found, the FSBL writes it to the PL. Next, whether or not a bit file is found, the FSBL loads one application ELF into memory and executes it. After that continues to search for files and loading them into memory until it detects a file that has a load address of 0xFFFFFFF0. Upon detection, the FSBL downloads this last file then jumps to the executable address of the first non-bit or non-boot file found (which is the application for CPU0).

4.2.1.2 Bare-Metal Application Code

Our implementation has both CPU0 and CPU1 running their own bare-metal application code. CPU0 is responsible for initializing shared resources and starting up CPU1. The bare-metal board support package (BSP) includes support for the preprocessor defined constant USE_AMP. This constant prevents the BSP from re-initializing the PS SCU that has previously been initialized by CPU0.

4.2.1.3 CPU0 Application

CPU0's application is located in memory starting at address 0x00100000. The linker script is used to set the starting address. The CPU0 application does the following:

1. Configures the MMU to disable cache for OCM accesses in the address range of 0xFFFF0000 to 0xFFFFFFFF. The address mapping of the OCM is untouched, so OCM exists at addresses 0x00000000 to 0x0002FFFF and addresses 0xFFFF0000 to 0xFFFFFFFF. Only the high 64 KB of OCM is used by the design so cache is disabled on addresses 0xFFFF0000 to 0xFFFFFFFF.
2. Disables L2Cache
3. Sets two memory location in OCM that are used as a semaphore flags.
4. Initializes the PCAP in order to be able to do partial reconfiguration
5. Initializes the VDMA's, as we have said before our design uses two VDMA cores in order for each Reconfigurable Region to be able to read and write data directly to the onboard DDR.
6. Initializes the two Reconfigurable Regions
7. Initializes the PPI interrupt controller and interrupt subsystem.
8. Starts CPU1.
9. Waits for CPU1 to initialize. CPU1 initialization includes also reading the application description file from the SD card and storing it to the DDR memory. CPU0 later uses this information in order to initialize the RTSM.

10. Updates the RTSM structures based on the data read from the application description file located on the SD card.
11. RTSM executes the User Application. Application execution includes
    a. Determining the task sequence
    b. Issuing the necessary commands for SW or HW task execution
    c. Issuing Reconfiguration Instructions
    d. Servicing the interrupts generated from the Reconfigurable Regions

After the PS powers up and the internal boot ROM completes execution, CPU1 is redirected to a small piece of code in OCM at 0xFFFFFE00. This piece of code is a continuous loop that waits for an event, checks address location 0xFFFFFFF0 for a non-zero value, and then continues the loop. If 0xFFFFFFF0 contains a non-zero value, CPU1 jumps to the fetched address.

CPU0 starts CPU1 (both running bare-metal) by writing the value of 0x00200000 to address 0xFFFFFFF0 and then running the Set Event (SEV) command. SEV causes CPU1 to wake up, read the value 0x00200000 from address 0xFFFFFFF0, and then jump to address 0x00200000. The FSBL is responsible for placing the CPU1 ELF at 0x00200000.

4.2.1.4 *CPU1 Application*

The CPU1 application is located in memory starting at address 0x00200000. The linker script is used to set the starting address.

The CPU1 application performs the following:

1. Configures the MMU to disable cache for OCM accesses in the address range of 0xFFFF0000 to 0xFFFFFFFF. The address mapping of the OCM is untouched so OCM exists at addresses 0x00000000 to 0x0002FFFF and addresses 0xFFFF0000 to 0xFFFFFFFF. Only the high 64 KB of OCM is used, so cache is disabled on addresses 0xFFFF0000 to 0xFFFFFFFF.
2. Disables L2Cache
3. Initializes GPIO (We use GPIO in order to access the SD card)
4. Mounts the SD card
5. Initializes the implemented SW tasks
6. Reads the application description file from the SD card, and stores them to predefined location on the DDR memory.

7. Reads the partial bitstreams from the SD card, and stores them to predefined location on the DDR memory.
8. Sets a memory location in OCM that is used as a semaphore flag to inform CPU0 that initialization is complete.
9. Waits for an operation request from CPU0
10. Clears the memory location in OCM that is used as a semaphore flag to issue a task to CPU1
11. Executes the requested SW task
12. Sets a memory location in OCM that is used as a semaphore flag to inform CPU0 that task execution is complete.

The CPU1 application repeats step 9 to step 12 indefinitely.

4.2.1.5 Inter-Processor Communication

The inter-processor communication in our design are two semaphore flags. When the first semaphore, “COMVAL0” is set by CPU0, CPU1 reads it resets it, and executes the appropriate SW task. After the SW task completion CPU1 sets the second semaphore flag “COMVAL1”, which notifies the RTSM which runs on CPU0 that the requested SW task has completed successfully. Then CPU0 acknowledges that by resetting “COMVAL1”.

The OCM memory is chosen because it is a low latency, shared resource. Also, this area of OCM is not cached so the memory accesses are deterministic. If DDR memory were to be used for the semaphores, there would be a higher latency for accesses during cache misses and less deterministic accesses due to background refresh cycles. DDR memory accesses are also bursty in nature so time would be wasted as a write or read burst occurs to access a single 32-bit value.

4.3 H/W Modules (HLS Synthesis)

To better understand the implications of our design choices, we first look at the operation of the Filter Engine IP cores. The cores have an AXI4-Lite interface that is used by the PS to configure and control the operation of the core. First, the IP core needs to be configured with the image dimensions (width and height) of the present video stream. Next, the core is started and after it has processed an entire image, an interrupt is issued. The interrupt handler identifies the interrupt source and notifies the RTSM. In summary, the following functions are provided to operate the core: configure, start, stop, and reset, among others used for checking the filter's state.

Hardware Interface

The Filter Engine IP core has the following hardware interface and connections to the static portion of the system:

- 32-bit AXI4-Lite interface connected to GP0 Master Interface It is used by the CPU to configure main core parameters like the image dimensions at run time.
- 32-bit AXI4-Stream interface connected to VDMA MM2S Master Stream interface.
- 32-bit AXI4-Stream interface connected to VDMA S2MM Slave Stream.
- Interrupt signal connected to ARM_0 PPI (Private Peripheral Interrupt)
- Clock signal connected to the same clock domain as AXI4-Lite and AXI4-Stream interfaces (150 MHz in this design)

All the four filter IP cores where designed and implemented using High Level Synthesis tool from Xilinx Vivado. Each algorithm targeted at the HLS tool must have a notion of a proper memory architecture for achieving performance (8). Line buffers and other memory structures must be part of the C code given to the HLS tool in order to generate the correct design.

After describing the algorithm we want to implement in each reconfigurable module, we add the required constrains in frequency and the desired directives for loop unrolling if required. Then we invoke the synthesis process. After synthesis completion and if the desired constrains are met we proceed to simulate the derived module. For simulation purposes we implemented a testbench which emulates the VDMA functionality, which means that it feeds the module under test with streaming image data read from a real image and writes out the module output to a new "output" image to the hard drive.

After the correct functionality of the fitter IP core is determined we proceed to export it as Pcore for EDK in order to be able to use it later on with our system design.

4.3.1 GrayScale Module

The grayscale IP core has a simple functionality, requires only a single access to the global memory storing the input image. It's input is an image pixel with RGB values and it computes the pixel's grayscale value by the simple equation $\text{Grayscale}_{(\text{value})} = (\text{Red}_{(\text{val})}+\text{Green}_{(\text{val})}+\text{Blue}_{(\text{val})})/3$ the process is straight forward and there is no use of special memory architectures.

In **Figure 19** and **Figure 20** we can see input image to the grayscale module and the image produced respectively.

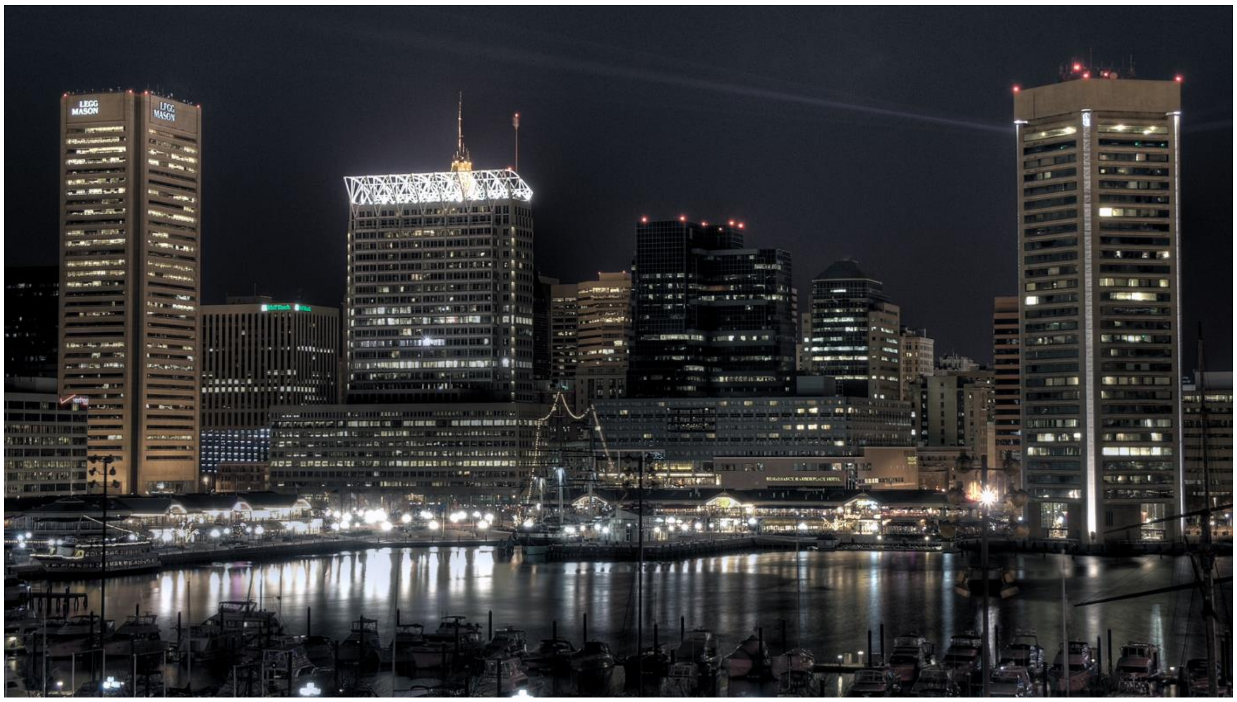

Figure 19 – Grayscale input Image

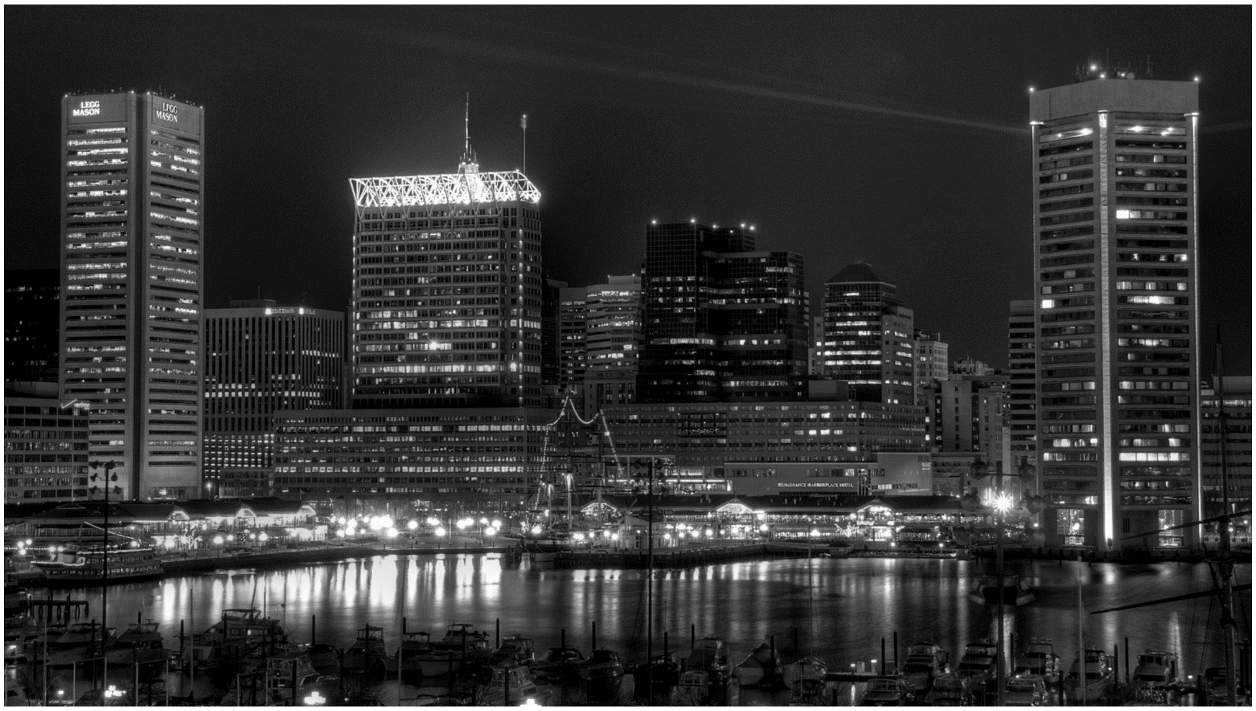

Figure 20 – Grayscaled Image

4.3.2 Gaussian Blur Module

Gaussian Blur module is used in order to smooth the entire image and make it easier to detect true edges because we don't really need "every" edge in the image, only the main features, this is done by the application of a 3x3 mask to the input image. In **Table 9** we can see the matrix used for this purpose. In the context of Gaussian Blur, a proper memory architecture requires only a single access to the global memory storing the input image and a local buffer storage to create the 3 x 3 computation window. Line buffers and other memory structures must be part of the C code given to the HLS tool in order to generate the correct design as we stated before.

Table 9: Gaussian Blur Operator Mask

1	2	1
2	4	2
1	2	1

The input image for the Gaussian Blur filter is the image produced by the gray scale filter which we saw earlier in **Figure 20**. Finally in **Figure 21** we can see the output image of the Gaussian blur filter.

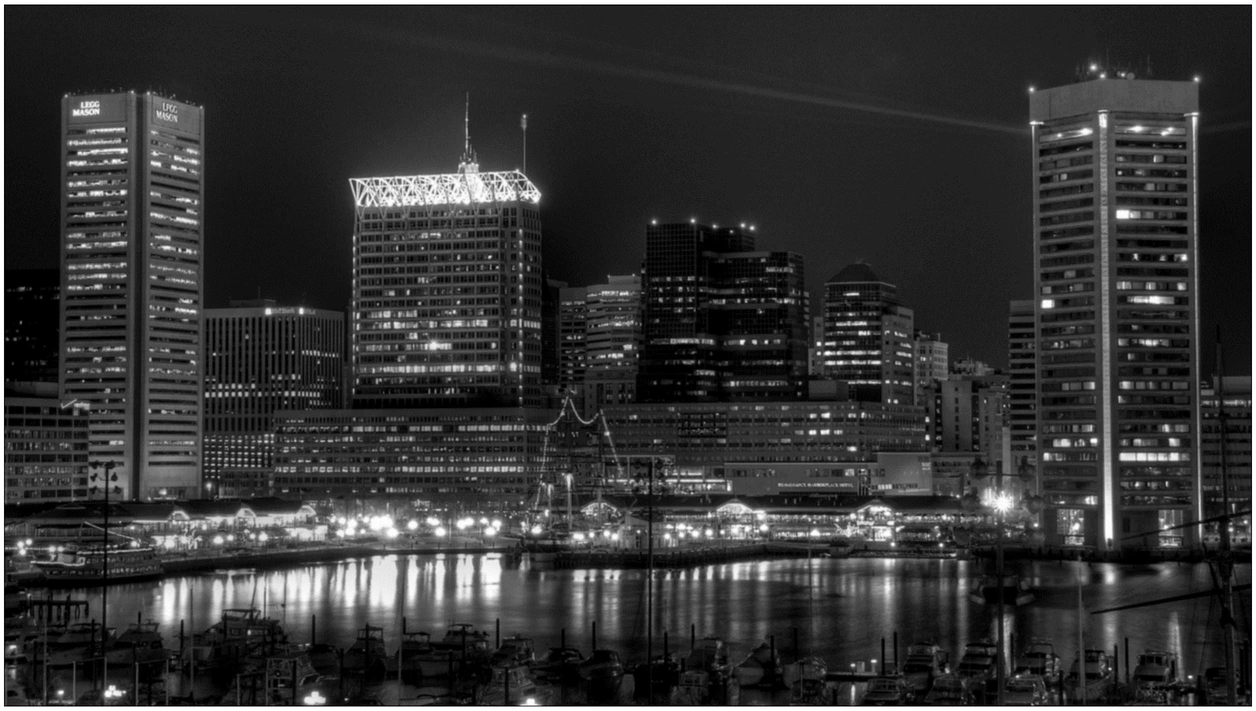

Figure 21 – Gaussian Blurred Image

4.3.3 Laplacian Edge Detection Module

The Laplacian is a 2-D isotropic measure of the 2nd spatial derivative of an image. The Laplacian of an image highlights regions of rapid intensity change and is therefore often used for edge detection. Since the input image is represented as a set of discrete pixels, we have to use a discrete convolution kernel that can approximate the second derivatives in the definition of the Laplacian. A 3x3 mask is again applied to the input image, which for this filter IP is the output of the Gaussian Blur filter we described before and we can see in **Figure 21**. In **Table 10** we can see the matrix used for this purpose. Also in the context of the Laplacian Edge Detector, a proper memory architecture requires only a single access to the global memory storing the input image and a local buffer storage to create the 3 x 3 computation window. Line buffers and other memory structures must be part of the C code given to the HLS tool in order to generate the correct design as we stated before.

Table 10: Edge Detection Operator Mask

0	1	0
1	-4	1
0	1	0

The input image for the Edge Laplace filter is the image produced by the Gaussian blur filter which we saw earlier in **Figure 21**. Finally in **Figure 22** we can see the output image of the Edge Laplace filter.

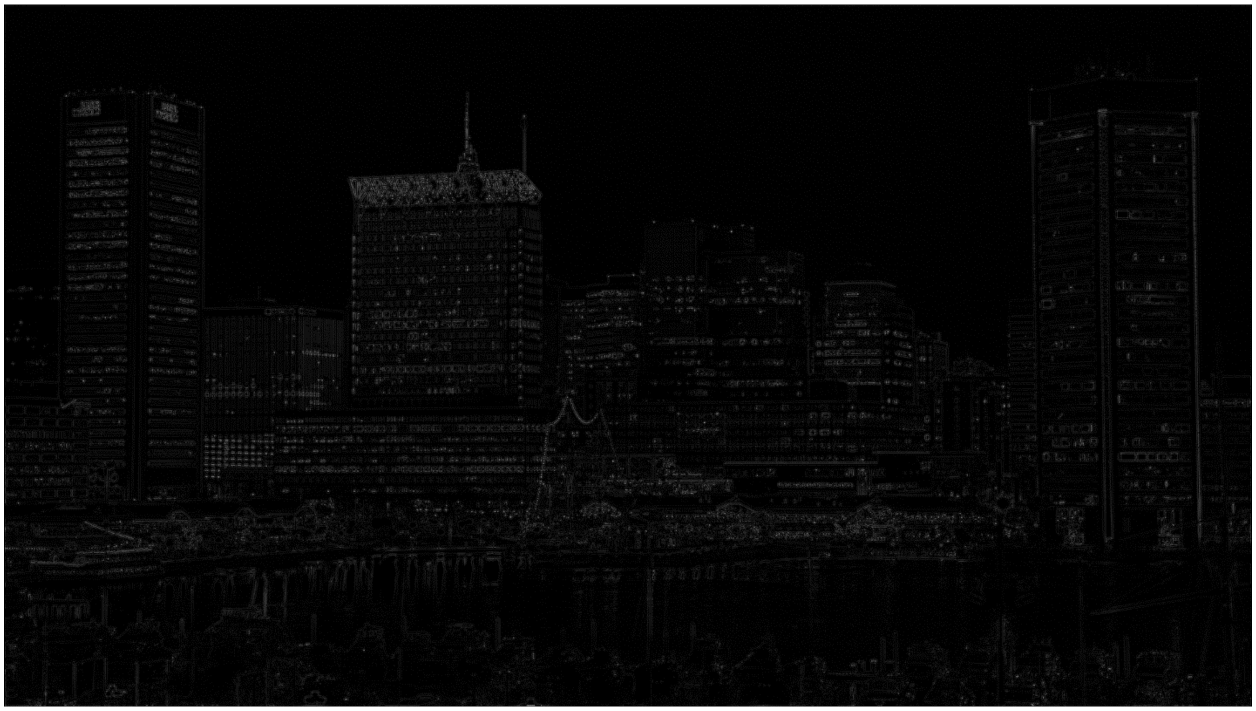

Figure 22 – Output image from Edge Laplace filter

4.3.4 Threshold Module

Finally the last filter IP is the threshold, which checks the value against a preconfigured threshold level, in our case 10 and if the pixel's value is above that threshold it writes the value 255 to the respective pixel position in the output image; if the pixel's value in below the threshold value it writes the value 0 to the respective pixel position in the output image. The threshold filter IP requires only a single access to the global memory storing the input image the same as Grayscale and thereafter there is no use of special memory architectures.

The input image for the Threshold filter is the image produced by the Edge Laplace filter which we saw earlier in **Figure 22**. Finally in **Figure 23** we can see the output image of the Threshold filter IP.

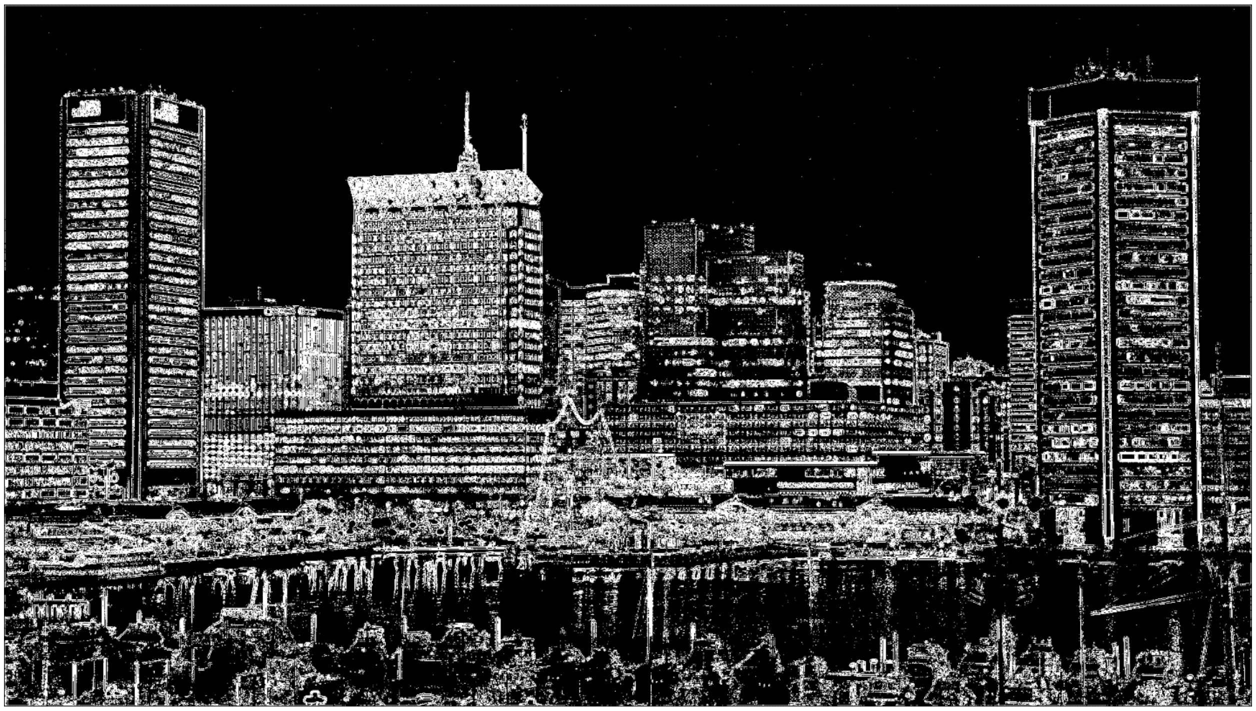

Figure 23 – Threshold filter output image

4.4 Partial Reconfiguration

Xilinx partial reconfiguration (PR) extends the inherent flexibility of the FPGA by allowing specific regions of the FPGA to be reprogrammed with new functionality while applications continue to run in the remainder of the device. Partial reconfiguration offers the following key advantages over traditional full configuration:

- Reduced hardware resource utilization: the designer can fit more logic into an existing device by dynamically time-multiplexing functions of the design
- Increased productivity and scalability: only the modified function needs to be implemented in context with the already-verified remainder of the design
- Enhanced maintainability and reduced system down-time: new functions can be deployed and inserted dynamically while the system is up and running

In order for us to use an IP core for partial reconfiguration, we had to make sure that the hardware interfaces and ports of all RMs are identical with respect to the static portion of the system. In the PR flow, ports are inserted at the boundary of the RP and the static portion of the design. Hence, all RMs (that is, all specific implementations mapped onto the RP) need to share the same ports at the same locations relative to the static logic.

4.4.1 Interface Decoupling

Because the reconfigurable logic is modified while the FPGA device is operating, the static logic connected to outputs of Reconfigurable Modules must ignore data from Reconfigurable Modules during partial reconfiguration. The Reconfigurable Modules will not output valid data until partial reconfiguration is complete and the reconfigured logic is reset. In order to reset the partial reconfigured area after PR completion we use the Reset After Reconfiguration feature that Xilinx has introduced for Virtex-6, 7 series and Zynq-7000 devices.

With the Reset After Reconfiguration feature, the region under reconfiguration is held in a steady state during partial reconfiguration using Global Write Enable (GWE), and a masked Global Set Reset (GSR) event is issued to load in INIT values for all newly-reconfigured logic. Static routes may still freely pass unaffected through the region, and static logic (and all other PR regions) elsewhere in the device will continue to operate normally during Partial Reconfiguration. Partial Reconfiguration with this feature will behave just like the initial configuration of the FPGA, with synchronous elements being released in a known, initialized state.

Common design practices to mitigate this issue are mechanisms such as registering all output signals, handshaking or disabling bus interfaces to avoid invalid transactions. In our case we had to make sure that there is no pending AXI transaction for the Region under partial reconfiguration, which was easy and straightforward since the two Reconfigurable Regions use different AXI buses to connect with the Processing System. The other issue that we had to address, which also troubled us for a long period of time, was that the interrupts from the Reconfigurable Region should be disabled before proceeding with Partial Reconfiguration and re-enabled after completion.

At first we didn’t disable the interrupts from the Reconfigurable Region, which resulted in triggering spurious interrupts during reconfiguration which hang the system. The fact that the incident did not happen at every partial reconfiguration but only when we tried to reconfigure the module that wasn’t produced along with the complete configuration, lead us for a long time to believe that we were facing a hardware issue and not a software one.

The Filter Engine IP cores implement a single generic hardware register interface and use the same memory address map. While this is not a strict requirement, it greatly simplifies the software driver architecture. In this example, the only configurable filter core parameters are the image dimensions which both RMs have in common.

4.4.2 Partial Reconfiguration Design Flow

The Partial Reconfiguration flow uses a variety of Xilinx tools, HLS synthesis is used to produce the filter IPs, Xilinx Platform studio (XPS) is used to produce the system configuration until the synthesis level and then PlanAhead is used to run place and route the design and produce the complete and partial bitstreams, finally the Xilinx SDK is used for software development. Below we list all the required steps for the PR flow.

1. We generate the filter IP cores and their corresponding standalone or bare-metal drivers using the Vivado HLS tool. The generated RTL code can be exported as an XPS Pcore which can then be easily be used by the XPS project.
2. We create a XPS project were we decide what features we are going to use from the Processing System and also the Programmable Logic architecture and the system interconnect.
3. We synthesize the whole system using the first Reconfigurable module for every Reconfigurable Region and then we replace that with the second and resynthesize the new design.
4. We create a PlanAhead PR project targeting the Zedboard platform and import the netlists and constraint files generated in the previous steps, section, except for the filter engines netlist/constraint files. When loading the synthesized design, the Filter Engine module will be treated as black box since there is no netlist associated with it.
5. We define the Filter Engine modules as a Reconfigurable Partitions (RP). Partitions ensure that the logic and routing common to each of the multiple designs is identical.
6. We create as many Reconfigurable Modules (RM) as needed for the above created Reconfigurable Partitions by adding the corresponding netlist/constraint files.
7. We floorplan the Reconfigurable Partitions by setting the physical size of the partitions and the types of resources desired, CLBs (Flip-Flops, LUTs, distributed RAM, Multiplexers, etc.), BRAM, and DSP blocks, plus all associated routing resources. We must floorplan the Reconfigurable Partition in order for it to be able to accommodate the resources required by the Reconfigurable Modules. As a rule of thumb, 20% overhead should be accounted for routing resources (9). A few things that we must point out here is that since we want to use the Reset after Reconfiguration feature of the 7-series FPGAs we must align our partitions vertically to clock region boundaries and also that the left and right edges of Pblock rectangles should be placed between two resource columns (e.g. CLB-CLB, CLB-BRAM or CLB-DSP) and not between two interconnect columns (INT-INT). (10)

8. We first implement the first configuration and after successful implementation, we promote it in order for the implementation results of the static portion of the design to be available and be re-used by the second configuration.
9. We implement the second configuration, making sure that the static logic is being imported from the previous implemented and promoted configuration.
10. We execute the PR Verify Configuration Utility to validate consistency between the implementations of the two configurations.
11. We generate full and partial bitstreams for all configurations.
12. Finally we convert the partial bitstreams to binary format using the PROMGen tool. PROMGen also reports the size of the generated partial binary files which is required when sending the configuration data to the PL via the DevC's DMA engine.

Below in **Figure 24** we can see the PR design flow that was described above and in **Figure 25** we can see the placed design, were the two Reconfigurable Regions are visible in the purple outline.

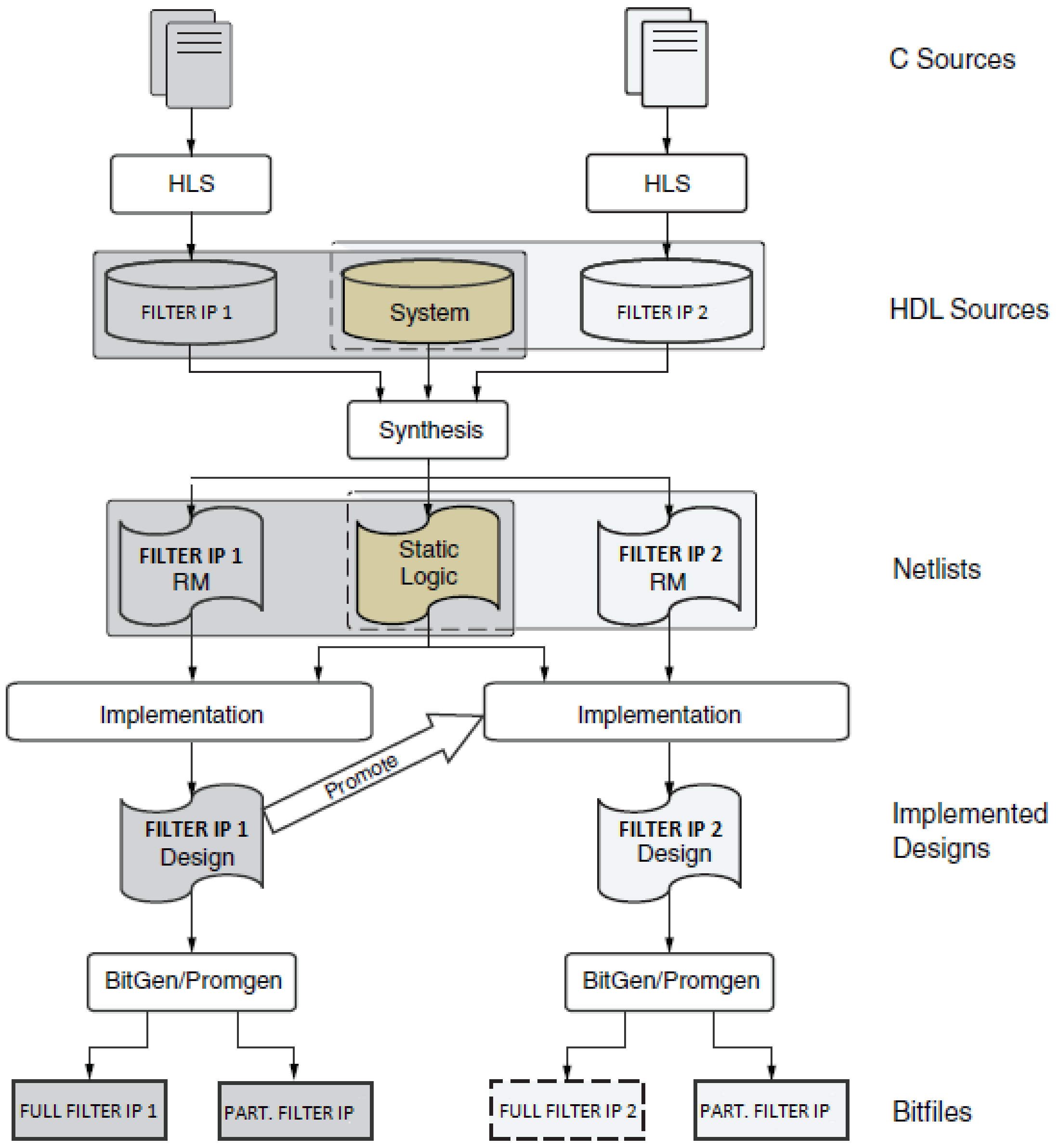

Figure 24 – Partial Reconfiguration design flow

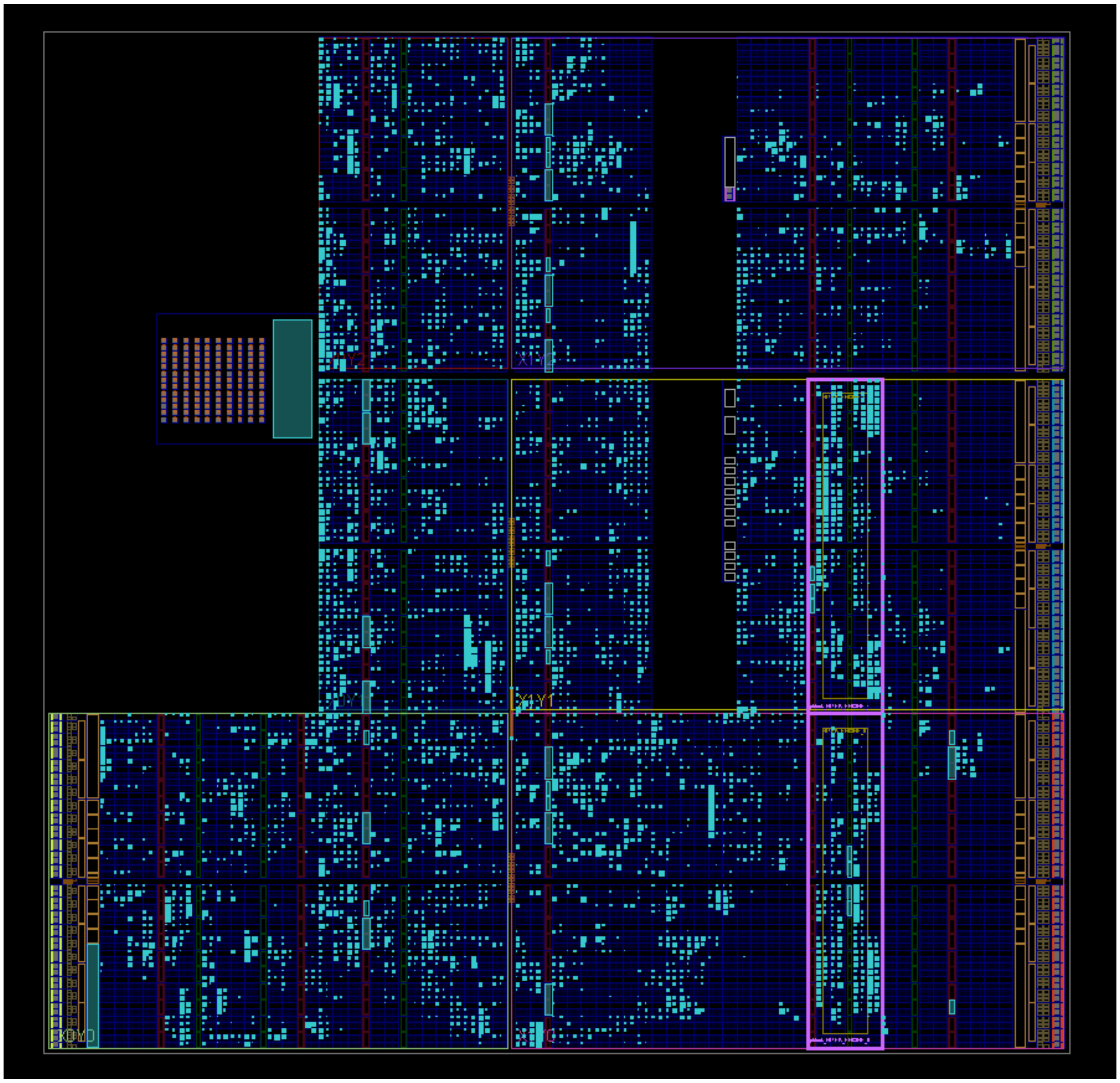

Figure 25 – The placed design on the Zynq SoC

4.5 SD memory Bare Metal Drivers Support

Xilinx provides SD card drivers only through the linux OS. In bare metal mode the supplied drivers support Read Only mode and there is no way, provided by Xilinx at least, to write to the SD card. There for we used an adaptation is largely based on the files and documentation of ChaN's FatFS - Generic FAT File system (11). The configuration uses the Zynq GPIO pins and the Xilinx GPIO driver functions.

4.5.1 Hardware configuration

In the ZYNQ7 processing system the MIO pins 40-45 are connected to the SD card interface. Since these pins are programmed using the GPIO interface, it is mandatory that the SD0 peripheral in the Zynq Block Design (accessing the same pins) is not selected. On the other hand, the GPIO peripheral needs to be selected.

Since now the SD0 peripheral in the Zynq Block Design is not selected, we can no longer boot the Zedboard Platform from the SD card which is the default and the most common way of booting, searching for an alternative we used the onboard flash memory for booting.

This change only required to choose the appropriate booting mode from the onboard jumpers, in **Figure 26** we can see the jumper position in order to boot from the SD card, and in **Figure 27** we see the jumper new jumper position in order to boot from the flash memory.

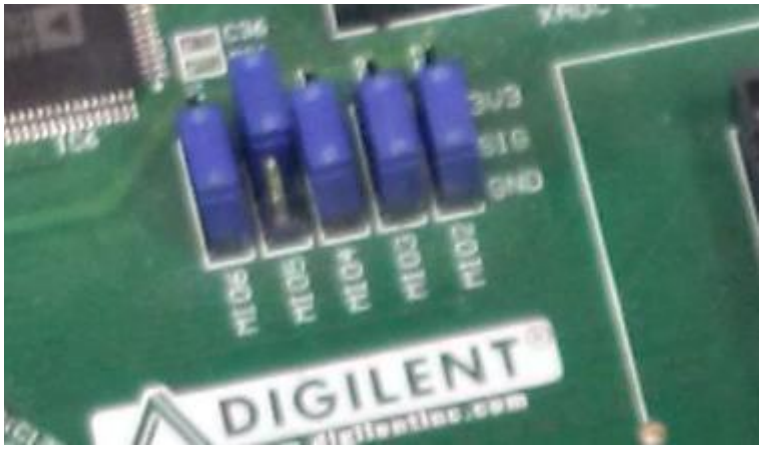

Figure 26 - Jumper position for SD card booting

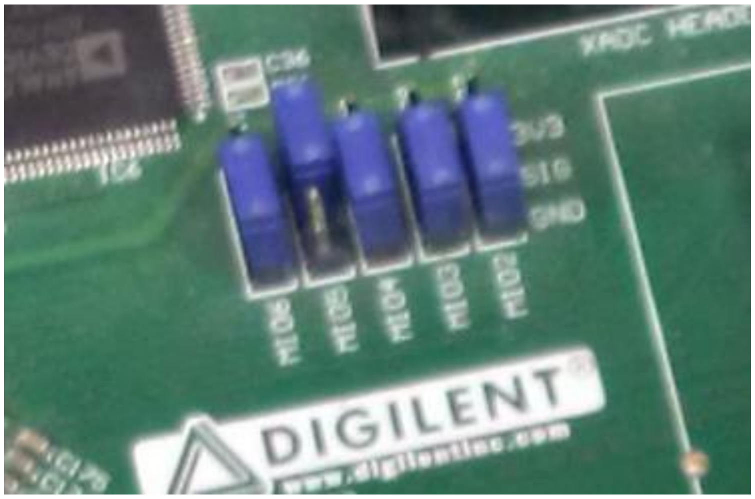

Figure 27 - Jumper position for Flash Memory booting

5. Implementation Results

We conducted experiments that they allowed us to demonstrate a different behavior in the progress of application execution and functionality of the RTSM on the implemented embedded architectures. The platform we used for our experiments is the XUPV5 equipped with a Virtex-5 FPGA, with 2 MicroBlaze processors. And the ZedBoard equipped with the Zynq 7Z020 SoC, with the dual core ARMv7 Cortex-A9.

XUPV5 Implementation Results

With the above we demonstrated the capability of RTSM to control the execution of applications on hybrid systems with partially reconfigurable PE and SW-PE. Below, we evaluate the RTSM performance and the overhead of different actions triggered by the RTSM running in MicroBlaze@100MHz by measuring the time intervals with timestamps. In all experiments MicloBlaze cache was disabled and there was no OS running. This basic architecture was provided by the PDM partners as created with the FASTER tool-chain.

In particular, we break down the RTSM into different phases: initialization, schedule decision, issue command, and update of lists after an operation has completed. Each phase can occur multiple times during RTSM execution, so in ***Table 11*** we show the aggregate time of each phase for the edge detection application which consists of 9 tasks.

Table 11 - RTSM phases and aggregate overhead per each phase throughout the execution of Edge Detection on XUPV5 platform RTSM runs on MicroBlaze@100MHz of XUPV5 platform

RTSM phases	**Avg Time (usec)**
RTSM initialization (read *cfg.dat*, initialization of structures)	39671
Total time spent in schedule function (called every time the RTSM takes a decision)	4726
Total time to issue the commands (can be either a reconfiguration; or a HW task execution; or a SW task execution)	138119
Total time needed to update the lists, after the execution of HW tasks completed.	788
Total time needed to update the lists, after the reconfigurations completed.	153397
Total time needed to update the lists, after the execution of SW tasks has completed.	835
Total RTSM execution time	**337539**

Table 12 - Number of times the schedule function ran to execute the task graph of Edge Detection, and time consumed for 1 run of schedule function, for MicroBlaze@100MHz of XUPV5 platform.

#times schedule function is called	9
Time spent to execute once (1) the schedule function	525 usec
Total time spent in schedule function	4726 usec

In **Figure 28** and in **Table 13** we can see the resource utilization for the complete design architecture for the XUPV5 platform. As we can see uses roughly a 30% of the FPGA resources.

The minimum period achieved by the design is 9.728ns resulting to a maximum frequency of 102.796MHz, which exceeds the required 100 MHz frequency, required by the MicroBlaze microprocessor in order for the design to work.

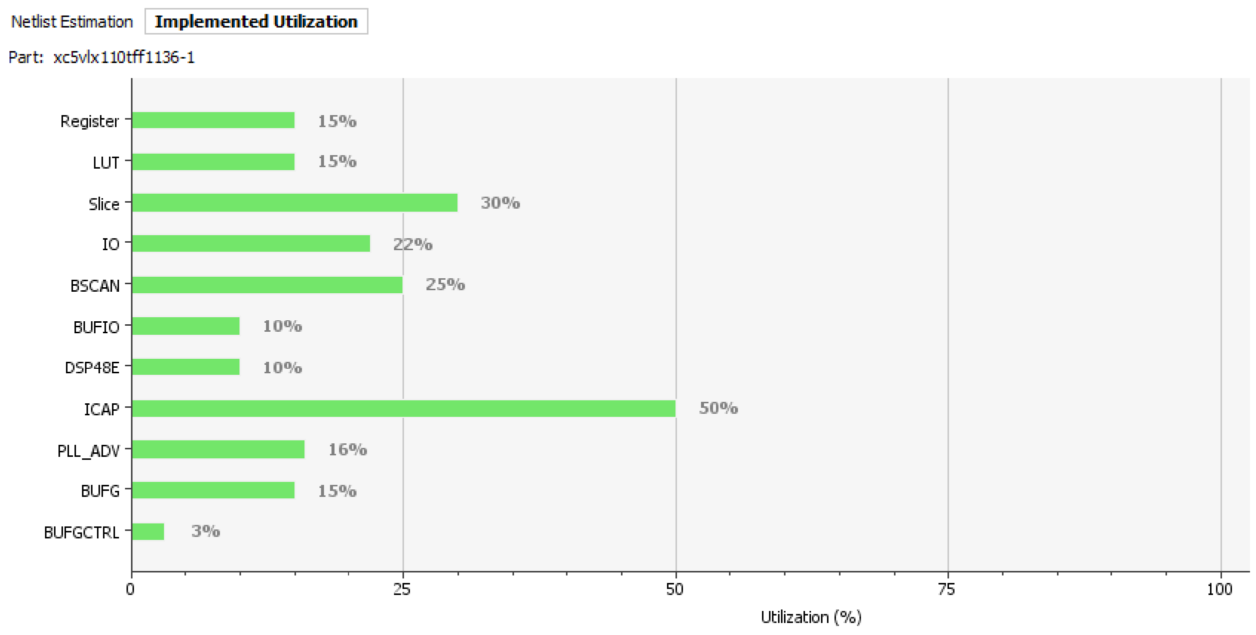

Figure 28 – XUPV5 Implementation Utilization

Table 13 - XUPV5 Resource Utilization

Resource	**Utilization**	**Available**	**Utilization (%)**
Register	10435	69120	15%
LUT	10374	69120	15%
Slice	5267	17280	30%
IO	147	640	22%
BSCAN	1	4	25%
BUFIO	8	80	10%
DSP48E	7	64	10%
ICAP	1	2	50%
PLL_ADV	1	6	16%
BUFG	5	32	15%
BUFGCTRL	1	32	3%

In Table 14, Table 15,

Table 16 and **Table 17** we can see the utilization of the two Reconfigurable Regions by the four Reconfigurable Modules. Partial reconfiguration practice advices that the rule of thumb, or to add an extra 20% on the resources needed by the biggest reconfigurable module, should be used for the resource allocation during the floor planning of the Reconfigurable Regions.

As we can see the maximum resource utilization does not exceed the 69%.

Table 14 - Reconfigurable Region 0, Grayscale Module

Site Type	**Available**	**Required**	**% Util**
LUT	1584	1012	64
FD_LD	1584	512	33
SLICEL	270	173	65
SLICEM	126	81	65
DSP48E	6	0	0
RAMBFIFO36	3	0	0

Table 15 - Reconfigurable Region 0, Edge Detection Module

Site Type	**Available**	**Required**	**% Util**
LUT	1584	1012	64
FD_LD	1584	512	33
SLICEL	270	173	65
SLICEM	126	81	65
DSP48E	6	0	0
RAMBFIFO36	3	0	0

Table 16 - Reconfigurable Region 1, Gaussian Blur Module

Site Type	Available	Required	% Util
LUT	1496	1012	68
FD_LD	1496	512	35
SLICEL	264	179	68
SLICEM	110	75	69
DSP48E	4	0	0
RAMBFIFO36	2	0	0

Table 17 - Reconfigurable Region 1, Threshold Module with MicroReconfiguration

Site Type	Available	Required	% Util
LUT	1496	1012	68
FD_LD	1496	512	35
SLICEL	264	179	68
SLICEM	110	75	69
DSP48E	4	0	0
RAMBFIFO36	2	0	0

ZedBoard Implementation Results

We collected measurements by breaking down the RTSM operation into different phases: initialization, schedule decision, issue command, and update of lists after an operation has completed. Each phase occurs multiple times during RTSM execution, and ***Table 18*** contains the aggregate time of each phase. In all cases we report the average time of multiple runs.

Table 18 - RTSM phases and aggregate overhead per each phase throughout the execution of Edge Detection on Xilinx ZedBoard. RTSM runs on ARM@666MHz of ZedBoard

RTSM phases	**Avg Time (usec)**
RTSM initialization (read *cfg.dat*, initialization of structures)	23.121
Total time spent in schedule function (called every time the RTSM takes a decision)	52.038
Total time to issue the commands (can be either a reconfiguration; or a HW task execution; or a SW task execution)	17.985
Total time needed to update the lists, after the execution of HW tasks completed	7.479
Total time needed to update the lists, after the reconfigurations completed	3.672
Total time needed to update the lists, after the execution of SW tasks has completed.	8.244
Total RTSM execution time	**112.539**

Table 19 - Number of total times the schedule function ran to execute the graph of Edge Detection and time consumed for 1 run of schedule function, for ARM@666MHz on ZedBoard

#times schedule function is called	9
Time spent to execute once (1) the schedule function	5.782 usec
Total time spent in schedule function	52.038 usec

In **Figure 29** and in **Table 20** we can see the resource utilization for the complete design architecture for the Zedboard platform. As we can see uses roughly a 35% of the FPGA resources.

The minimum period achieved by the design for the control path is 11.975ns, resulting to a maximum frequency of 83.507 MHz, which exceeds the required 75 MHz frequency, required by the AXI-Lite bus. And the minimum period achieved by the design for the data path is 6.650ns, resulting to a maximum frequency of 150.376 MHz, which exceeds the required 150 MHz frequency, required by the AXI-Stream bus.

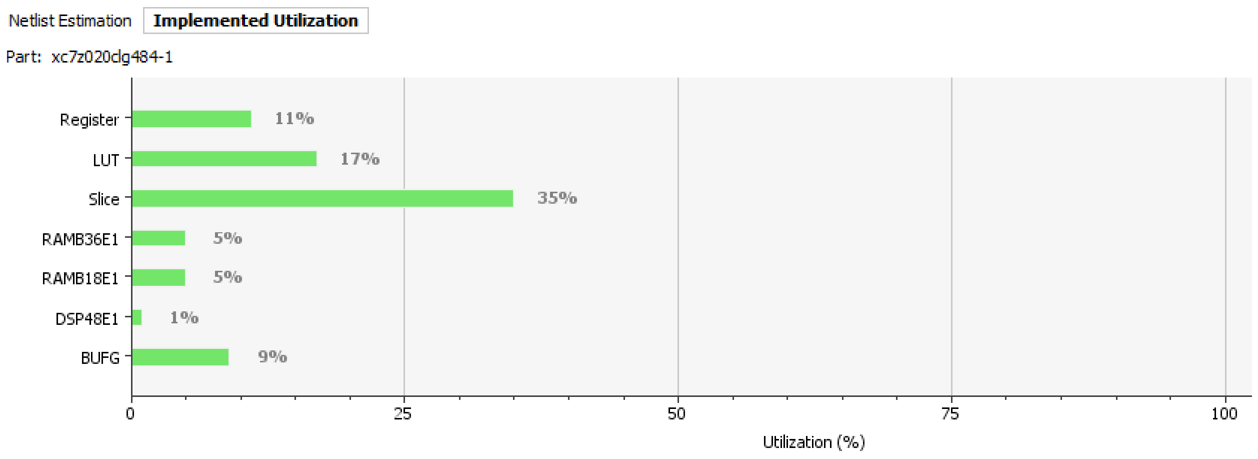

Figure 29 – Zedboard Implementation Utilization

Table 20 – ZedBoard Resource Utilization

Resource	Utilization	Available	Utilization (%)
Register	12010	106400	11%
LUT	9333	53200	17%
Slice	4752	13300	35%
RAMB36E1	16	280	5%
RAMB18E1	15	280	5%
DSP48E1	4	220	1%
BUFG	3	32	9%

In Table 21, Table 22,

Table 23 and **Table 24** we can see the utilization of the two Reconfigurable Regions by the four Reconfigurable Modules. As stated also before, partial reconfiguration practice advices that the rule of thumb, or to add an extra 20% on the resources needed by the biggest reconfigurable module, should be used for the resource allocation during the floor planning of the Reconfigurable Regions.

As we can see the maximum resource utilization does not exceed the 49%, and maybe this would seem like a waste of resources to someone, but the fact is that this was the smallest rectangle shape that could be floorplanned on the Zynq SoC that could also service the needs for resources by the reconfigurable modules. This happens because of the restrictions imposed by the chip technology and architecture which do not allow the edges of a reconfigurable region to be everywhere inside the fabric.

Table 21 - Reconfigurable Region 0, Grayscale Module

Site Type	Available	Required	% Util
LUT	1600	661	42
FD_LD	3200	494	16
SLICEL	200	78	39
SLICEM	200	88	44
DSP48E1	20	4	20
FIFO18E1	10	0	0
RAMB18E1	10	0	0
RAMBFIFO36E1	10	0	0

Table 22 - Reconfigurable Region 0, Edge Detection Module

Site Type	Available	Required	% Util
LUT	1600	746	47
FD_LD	3200	700	22
SLICEL	200	90	45
SLICEM	200	97	49
DSP48E1	20	0	0
FIFO18E1	10	1	10

RAMB18E1	10	2	20
RAMBFIFO36E1	10	0	0

Table 23 - Reconfigurable Region 1, Gaussian Blur Module

Site Type	Available	Required	% Util
LUT	1600	744	47
FD_LD	3200	726	23
SLICEL	200	90	45
SLICEM	200	97	49
DSP48E1	20	0	0
FIFO18E1	10	1	10
RAMB18E1	10	2	20
RAMBFIFO36E1	10	0	0

Table 24 – Reconfigurable Region 1, Threshold Module

Site Type	Available	Required	% Util
LUT	1600	581	37
FD_LD	3200	453	15
SLICEL	200	69	35
SLICEM	200	77	39
DSP48E1	20	0	0
FIFO18E1	10	0	0
RAMB18E1	10	0	0
RAMBFIFO36E1	10	0	0

6. Conclusions

Creating a changing hardware system is a challenging process requiring considerable effort in specification, design, implementation, verification, as well as support from a run-time system.

In this thesis, we presented the integration and porting of the Run Time System Manager to various platforms embedded and hybrid, PC – embedded. The RTSM is built in a way that enhanced its portability the initial RTSM was developed on an x86 ISA desktop, and porting it to the MicroBlaze and ARM architectures 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. The main difficulty of the porting process was on addressing the quirks of each particular platform especially with the actual PR process and the handling of interrupts.

If we should summarize the main difficulties that we came across during the system development, we would have to say that:

The most challenging part for the XUPV5 board was to overcome the limitation imposed by the restricted amount of memory of the MicroBlaze microprocessor that should execute the RTSM. The fact that the RTSM design would require a big amount of memory allocations during runtime because it was at first developed for o Desktop system where there memory restrictions are not an issue, resulted very often to inexplicable system failures which proved to be caused from Heap and Stack overflows.

The ZedBoard platform since we implemented the system architecture from scratch was a more complex process as one would expect. Mostly the difficulties that we came across, mainly had to do with the lack of know how about the ZedBoard platform use in Bare-metal mode. As also the lack of correct documentation from Xilinx in certain issues.

- Very early in our design process we came across some choices about different DMA engines available from Xilinx as IP cores, and the decision on which was the appropriate for our system architecture
- A big issue that bothered us for a long time was the fact that Xilinx didn’t offer write support for SD card bare metal drivers. This was overcame by using the SD card not as peripheral to the ARM processor but using its connection pins as GPIO. Of course this resulted in having to find another way to boot the system, since the SD card that was used until this point was not an option anymore. The onboard flash memory was used as mentioned above.

- A point where no documentation at all, from Xilinx as a vendor, was on how to generate generic Filter engine drivers in order to be able to use them later in a design with Partial Reconfiguration support. As stated also before, the Xilinx HLS tool generates drivers for each IPcore, in order for the designer to be able to use it from the system software. Those drivers may have the same functionality for all our reconfigurable modules but that fact that we know it does not mean that the Software Development Kit has the same perception with us, on the contrary it “thinks” that separate drivers are needed, which by itself is against Partial Reconfiguration practice. In order to overcome this we had to do a lot of reverse engineering to several Xilinx Reference Designs.
- Finally the issue that troubled us for a long period of time, was that the interrupts from the Reconfigurable Region should be disabled before proceeding with Partial Reconfiguration and re-enabled after completion. At first we didn’t disable the interrupts from the Reconfigurable Region, which resulted in triggering spurious interrupts during reconfiguration which hang the system. The fact that the incident did not happen at every partial reconfiguration but only when we tried to reconfigure the module that wasn’t produced along with the complete configuration, lead us for a long time to believe that we were facing a hardware issue and not a software one.

After we successfully ported the RTSM on these two platforms, we were able to demonstrate that the RTSM can efficiently schedule HW and SW tasks in real life systems with partially reconfigurable FPGAs and various architecture microprocessors.

7. Future Work

Future work that could be done on this project we could identify the need for a more complex application, the grayscale application is a small one and nevertheless it has all the necessary attributes needed in order to demonstrate the correct execution of the RTSM and also the system architecture reliability, it is not big enough in order to provide us with data about the RTSM’s scalability while the number of the tasks grows.

Another extension could the addition of more Reconfigurable Regions to the system architecture. While the addition of more RR’s does not complicate the system architecture, it can help the RTSM to demonstrate its full potential accompanied with a more complex and demanding application.

During the implementation of the ZedBoard system architecture, University of Ghent did not support microreconfiguration on the Zynq SoC, but it was a work in progress. A very useful addition for the future would be the integration of microreconfiguration also on the ZedBoard platform and generally for the Zynq family SoCs.

Finally more work can be done to the towards unlocking all the RTSM features which are not supported by now namely those are

- Task check pointing in order for the RTSM to be able to do more efficient task relocation
- New functionality in order to support if else decisions from tasks
- New additions to the system architecture in order to report execution time for every issued task SW or HW in order to be able to do more efficient task scheduling for more complex architectures that use a task more than once.

8. References

1. *An execution environment for reconfigurable computing.* **Wenyin Fu, Compton, K.** s.l. : Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on, 2005. pp. 149,158.

2. *Efficient runtime support for embedded MPSoCs.* **Theodoropoulos, D., Pratikakis, P. and Pnevmatikatos, D.** Samos, Greece : IEEE, 2013. Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII) International Conference on. pp. pp.164,171.

3. *Invited paper: Acceleration of computationally-intensive kernels in the reconfigurable era.* **Papadimitriou, K., Vatsolakis, C. and Pnevmatikatos, D.** s.l. : Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012.

4. *OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES.* **DURAN, ALEJANDRO and AYGUADÉ, EDUARD and BADIA, ROSA M. and LABARTA, JESÚS and MARTINELL, LUIS and MARTORELL, XAVIER and PLANAS, JUDIT.** 2011, Parallel Processing Letters, Vol. 21, pp. 173-193.

5. *Hybrid Hardware-Software Architecture for Reconfigurable Real-Time Systems.* **Rodolfo Pellizzoni, Marco Caccamo.** St. Louis, MO : IEEE, 2008. Real-Time and Embedded Technology and Applications Symposium. pp. 273 - 284.

6. *Automatic Run-Time Manager Generation for Reconfigurable MPSoC Architecture.* **Gianluca Durelli, Christian Pilato, Andrea Cazzaniga, Donatella Sciuto, Marco D. Santambrogio.** York : s.n., July 2012. International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC).

7. **Karel Bruneel, Karel Heyse, Tom Davidson, Amit Kulkarni and Dirk Stroobandt.** [Online] https://github.com/UGent-HES/tlut_flow.

8. **Xilinx.** Implementing Memory Structures for Video Processing in the Vivado HLS Tool. XAPP793.

9. —. Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices. XAPP1159.

10. —. Vivado Design Suite User Guide: Partial Reconfiguration. UG909.

11. **Chan.** http://elm-chan.org/fsw/ff/00index_e.html. [Online]

12. **Xilinx.** Application Note: Zynq-7000 AP SoC, "Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors". XAPP1079.

13. —. Partial Reconfiguration User Guide. UG702.

14. *A Dynamic Reconfiguration Run-Time System.* **J. Burns, A. Donlin, J. Hogg, S. Singh, M. de Wit.** s.l. : IEEE, 1997. Proc. of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.

15. *Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs.* **P. Lysaght, B. Blodget, J. Mason, J. Young, B. Bridgford.** 2006. 16th International Conference on Field Programmable Logic and Applications. pp. 1-6.

16. *Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of RealTime Tasks.* **C. Steiger, H. Walder, and M. Platzner.** s.l. : In: IEEE Transactions on computers, 2004. Vol. 53.

17. *3D Compaction: a Novel Blocking-aware Algorithm for Online Hardware Task Scheduling and Placement on 2D Partially Reconfigurable Devices.* **T. Marconi, Y. Lu, K.L.M. Bertels, G. N. Gaydadjiev.** Bangkok, Thailand : In: Proc. of the International Symposium on Applied Reconfigurable Computing (ARC), 2010. pp. 194-206.

18. *A Communication Aware Online Task Scheduling Algorithm for FPGA-based Partially Reconfigurable Systems.* **Y. Lu, T. Marconi, K.L.M. Bertels, G. N. Gaydadjiev.** s.l. : In Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines, 2010.

19. *Fast Template Placement for Reconfigurable Computing Systems.* **K. Bazargan, R. Kastner, and M. Sarrafzadeh.** s.l. : In: IEEE Design and Test of Computers, 2000. Vol. 17, pp. 68-83.

20. *Configuration Relocation and Defragmentation for Run-Time Reconfigurable Systems.* **K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck.** s.l. : In: IEEE Trans. on VLSI, 2002. Vol. 10.

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

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

23. *Operating System for Runtime Reconfigurable Multiprocessor Systems.* **D. Göhringer, M. Hübner, E. Nguepi Zeutebouo, J. Becker.** s.l. : Int. J. Reconfig. Comp, 2011.

24. *RAMPSoCVM: Runtime Support and Hardware Virtualization for a Runtime Adaptive MPSoC.* **D. Göhringer, S. Werner, M. Hübner, J. Becker.** Chania, Crete, GREECE : s.n., 2011. 21st International Conference on Field Programmable Logic and Applications. pp. 181-184.

25. *Design Framework for Partial Run-Time FPGA Reconfiguration.* **C. Conger, A. Gordon-Ross, A. D. George.** s.l. : ERSA, 2008. pp. 122-128.

26. *Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation.* **Z. Li, S. Hauck.** 2002. Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays. pp. 187-195.

27. *Architecture-Aware Reconfiguration-Centric Floorplanning for Partial Reconfiguration.* **Vipin, Fahmy.** s.l. : in Proc. ARC, 2012.

28. *FASTER: Facilitating Analysis and Synthesis Technologies for Effective Reconfiguration.* **D. Pnevmatikatos, et al.** s.l. : Elsevier Journal on Microprocessors and Microsystems (MICPRO), 2014.

29. *The DeSyRe Runtime Support for Fault-Tolerant Embedded MPSoCs.* **Pnevmatikatos, D., et al.** s.l. : IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2014 .

30. **Deliverable_4.4b.** *Initial Run-time Configuration Scheduler and Device Manager.* s.l. : FASTER project, March 2014.