NANYANG TECHNOLOGICAL UNIVERSITY

# NANYANG TECHNOLOGICAL UNIVERSITY

## EMBEDDING FPGA OVERLAYS INTO THE XILINX ZYNQ FOR RUNTIME MANAGEMENT USING RTOS

by

SYED ALI ASGHAR
(G1402228G)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2015

# Contents

# List of Figures

# List of Tables

# Abbreviations

**ACP** Accelerator Coherency Port

**AXI** Advanced eXtensible Interface

**DMA** Direct Memory Access

**DPR** Dynamic Partial Reconfiguration

**FPGA** Field Programmable gate array

**GP** General purpose

**HLS** High Level Synthesis

**HP** High Performance

**HW** Hardware

**IF** Intermediate Fabric

**MMU** Memory Management Unit

**OCM** On-Chip Memory

**PE** Processing Element

**PL** Programmable Logic

**PS** Processing system

**RTL** Register Transfer Level

**RTOS** Real Time Operating System

**Abstract**

Coarse-grained FPGA overlay architectures have been shown to be effective when paired with general purpose processors, offering software-like programmability, fast compilation, application portability and improved design productivity. Run time management of overlay architectures can be done using a bare metal software application, a general purpose operating system or even using a Real Time Operating System (RTOS). The factor that dictates the use of runtime manager is the amount of control desired over the platform response. A platform that must respond rapidly to many different deadlines and different priorities and takes care of shared resources will require a more complicated and sophisticated runtime manager. An RTOS such as uC/OS-III seems to be a better run time manager in scenarios where overlay is used as a shared resource for executing compute kernels. This report first presents experiments to evaluate the performance of a commercial RTOS (uC/OS-III) on the Xilinx Zynq by quantifying performance metrics such as context switching time, preemption time etc. We then describe the process of embedding an FPGA overlay architecture into the Xilinx Zynq platform. We also demonstrate the runtime management of software and hardware tasks using uC/OS-III. Finally, we present use-case scenarios and preferred scheduling mechanisms by considering the overlay as a shared resource among tasks requiring hardware acceleration.

# Acknowledgment

Firstly, I would like to extend my sincere gratitude to the following people who never ceased to help and support me in every way possible for the completion of this project.

Associate Professor Dr. Douglas Leslie Maskell, the project supervisor, for the unwavering guidance, insightful comments and encouragement. If not for his support, I would have not been able to complete this project.

The completion of this project could also not have been possible without the participation and assistance of Mr. Abhishek Kumar Jain. I really appreciate all the efforts he has put into this project.

Finally, I would like to thank Mr. Jeremian Chua of CHIPES for the technical services and facilities he made available in the Lab which contributed positively toward my project.

# Chapter 1

# Introduction

## 1.1 Motivation

Embedded reconfigurable platforms, such as the Xilinx Zynq, couple one or more general purpose processors with Field Programmable gate array (FPGA). Poor design productivity for these platforms has been a key limiting factor, restricting their effective use to experts in hardware design [2]. Coarse-grained FPGA overlay architectures [3, 4, 5, 6, 7, 8, 9, 10] have been shown to be effective when paired with general purpose processors, offering software-like programmability, fast compilation, application portability and improved design productivity. Run time management of overlay architectures can be done using a bare metal software application, a general purpose operating system or even using an RTOS. The factor that dictates the use of runtime manager is the amount of control desired over the platform response. A platform that must respond rapidly to many different deadlines and different priorities and takes care of shared resources will require a more complicated and sophisticated runtime manager. An RTOS such as uC/OS-III seems to be a better run time manager in scenarios where overlay is used as a shared resource for executing compute kernels. In our work, we aim to conduct the analysis of a real time operating system (uC/OS-III) as a run time manager of software and hardware tasks on the Xilinx Zynq Platform and present use-case scenarios and preferred scheduling mechanisms by considering the overlay as a shared resource among tasks requiring hardware acceleration.

## 1.2 Contribution

We first present experiments to evaluate the performance of a commercial RTOS (uC/OS-III) on the Xilinx Zynq by quantifying performance metrics such as context switching time, preemption time etc. We then present an approach for embedding FPGA overlay architecture into the Xilinx Zynq platform. We also demonstrate the runtime management of software and hardware tasks using uC/OS-III. Finally we present use-case scenarios and preferred scheduling mechanisms by considering the overlay as a shared resource among tasks requiring hardware acceleration.

## 1.3 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information on the Xilinx Zynq platform, overlay architectures and run time management systems. In chapter 3, we present experiments to evaluate the performance of a commercial RTOS (uC/OS-III) on the Xilinx Zynq by quantifying performance metrics. In chapter 4, we present our approach for embedding FPGA overlay architecture into Zynq. Chapter 5, demonstrates the runtime management of software and hardware tasks on the platform using uC/OS-III by considering the overlay as a shared resource for the tasks requiring hardware acceleration.

# Chapter 2

# Background and Literature Survey

## 2.1 Zynq as a hybrid computing platform

Both major FPGA vendors have recently introduced hybrid platforms consisting of high performance processors coupled with programmable logic. These architectures partition the hardware into a Processing system (PS), containing one or more processors along with peripherals, bus and memory interfaces, and other infrastructure, and the Programmable Logic (PL) where custom hardware can be implemented. In this report, we focus on the Xilinx Zynq-7000. The Zynq-7000 contains a dual-core ARM Cortex A9 processor equipped with a double-precision floating point unit, commonly used peripherals, a dedicated hard Direct Memory Access (DMA) controller (PS-DMA), L1 and L2 cache, On-Chip Memory (OCM) and external memory interfaces. It also contains several Advanced eXtensible Interface (AXI) based interfaces to the programmable logic (PL). The AXI interfaces to the fabric include:

- AXI_ACP – One 64-bit AXI Accelerator Coherency Port (ACP) slave interface for coherent access to CPU memory
- AXI_HP – four 64-bit/32-bit configurable, buffered AXI High Performance (HP) slave interfaces with direct access to DDR memory and OCM
- AXI_GP – two 32-bit master and two 32-bit AXI General purpose (GP) slave interfaces

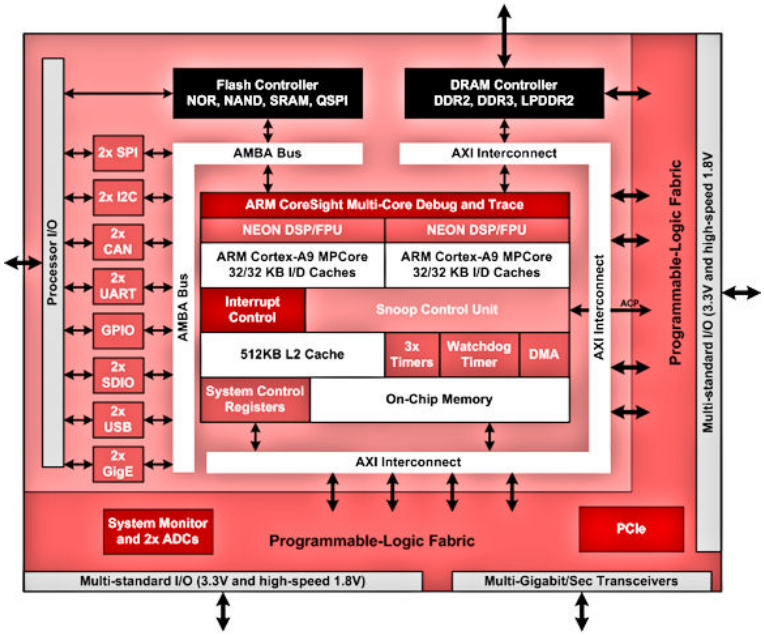



Figure 2.1: Block Diagram of the Hybrid Platform.

For more than a decade, researchers have shown that FPGAs can accelerate a wide variety of software, in some cases by several orders of magnitude compared to state-of-the-art general purpose processors [11, 12]. FPGA accelerators are normally designed at a low level of abstraction (typically Register Transfer Level (RTL)) in order to obtain an efficient implementation, and this can consume more time and make reuse difficult when compared with a similar software design. As such, design productivity remains a major challenge, restricting the effective use of FPGA accelerators to niche disciplines involving highly skilled hardware engineers. High Level Synthesis (HLS) has been proposed as a way of addressing the limited design productivity and manpower capabilities associated with hardware design [13, 14]. However, achieving desired performance often still requires detailed low-level design engineering effort that is hard for non-experts. Even as HLS tools improve in efficiency, prohibitive compilation times (specifically the place and route times in the backend flow) still limit productivity and mainstream adoption [2]. Hence, there is a growing need to make FPGAs more accessible to application developers who are accustomed to software API abstractions and fast development cycles [15].

Coarse grained configurable overlay architectures have been proposed as a method to overcome some of these issues [3, 4, 5, 6, 7, 10]. Overlays can be used for reducing the prohibitive compilation time required to map an application to the conventional fine-grained FPGA fabric. Overlays have also been shown to be effective when paired with general purpose processors [16, 4] as this allows the hardware fabric to be viewed as a software-managed hardware task, enabling more shared use. We describe FPGA Overlay architectures in the next section.

## 2.2 FPGA Overlay Architectures

Overlay architectures consist of a regular arrangement of coarse grained routing and compute resources. The key attraction of overlay architectures is software-like programmability through mapping from high level descriptions, application portability across devices, design reuse, fast compilation by avoiding the complex FPGA implementation flow, and hence, improved design productivity. Accelerators can be described at a higher level of abstraction and compiling it for overlays is several orders of magnitude faster than for the fine grained FPGAs.. An overlay provides a leaner mechanism for hardware task management at runtime as there is no need to prepare distinct bitstreams in advance using vendor-specific compilation (synthesis, map, place and route) tools. Instead, the behaviour of the overlay can be modified using software defined overlay configurations.

Despite having the implementation of the overlay architecture and its performance gain, there is no guarantee that it will surely provide reduction in kernel execution time. It depends heavily on how the overlay is interfaced to the host processor, communication mechanism between overlay, host processor and the external memory, communication bandwidth and latencies etc. Researchers have shown the effective use of coarse grained overlay architectures by pairing them with host processors as a coprocessor [17, 16] or as a part of the processor's pipeline [18]. Fig. 2.2 shows the integration of DySER [18, 19] overlay into the pipeline of a processor. Integrating an overlay within a processor pipeline can provide huge performance and energy efficiency at the expense of complete redesign of processor micro-architecture. Another possible

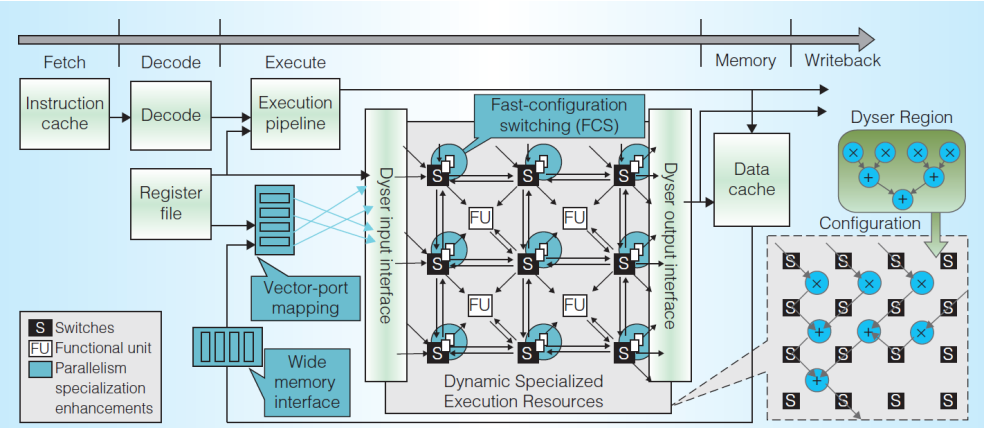



Figure 2.2: DySER Interfacing with Host Processor

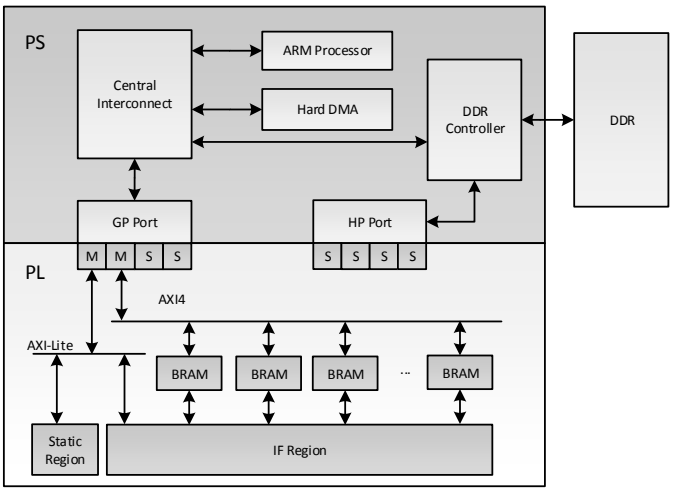



Figure 2.3: Intermediate Fabric (IF) Interfacing with Host Processor

approach is to interface the overlay (as a co-processor) with the host processor via standard communication interfaces. One example of pairing the overlay (Intermediate Fabric (IF) Overlay [5]) with a high performance ARM processor via an AXI interface in a commercial computing platform (the Xilinx Zynq[20]) is shown in Fig. 4.1.

An overlay architecture, referred to as an intermediate fabric (IF) [5], [21] was proposed to support near-instantaneous placement and routing. Standard VPR [22] algorithms were used for placement and routing of compute kernels. Unlike a physical device, whose architecture must support many applications, IFs have been specialized

for particular domains or even individual applications. Such specialization hides the complexity of fine-grained COTS devices, thus enabling fast place and route (700x speedup over vendor tools) at the cost of significant area (34% - 44%) and performance (7%) overhead when implemented on an Altera Stratix III FPGA [21]. It consists of 192 heterogeneous functional units comprising 64 multipliers, 64 subtracters, 63 adders, one square root unit, and five delay elements with a 16-bit datapath and supported the fully parallel, pipelined implementation of compute kernels.

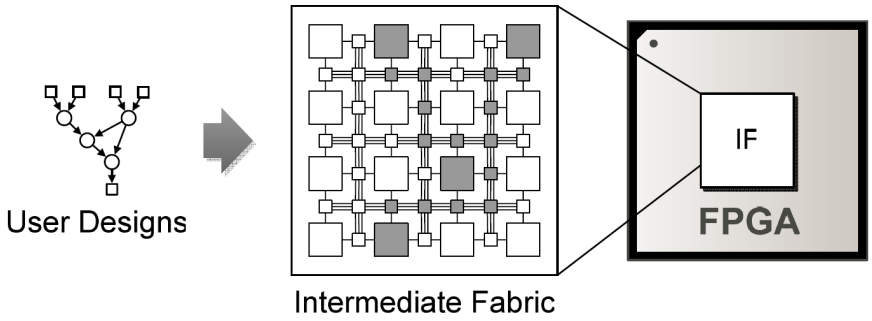



Figure 2.4: Intermediate Fabrics as Island-style Overlay.

Mesh of FU based Overlay executes a given DFG by mapping the graph nodes to the FUs and by configuring the routing logic to establish inter-FU connections that reflect the graph edges. Multiple instances of the DFGs are then executed in a pipelined fashion on the overlay to achieve high performance. It consisted of a 24×16 overlay with a nearest-neighbor-connected mesh of 214 routing cells and 170 heterogeneous functional units (FU) comprising 51 multipliers, 103 adders and 16 shift units.

DySER [7, 19] was proposed as a coarse grained overlay architecture for improving the performance of general purpose processors. It was originally designed as a heterogeneous array of 64 functional units interconnected with a circuit-switched mesh network and implemented on ASIC. The DySER architecture was then improved and prototyped, along with the OpenSPARC T1 RTL, on a Xilinx XC5VLX110T FPGA [18]. However, due to excessive LUT consumption, it was only possible to fit a 2x2 32-bit DySER, a 4x4 8-bit DySER or an 8x8 2-bit DySER on the FPGA. An adapted version of a 6x6 16-bit DySER was implemented on a Xilinx Zynq-7020 [9].

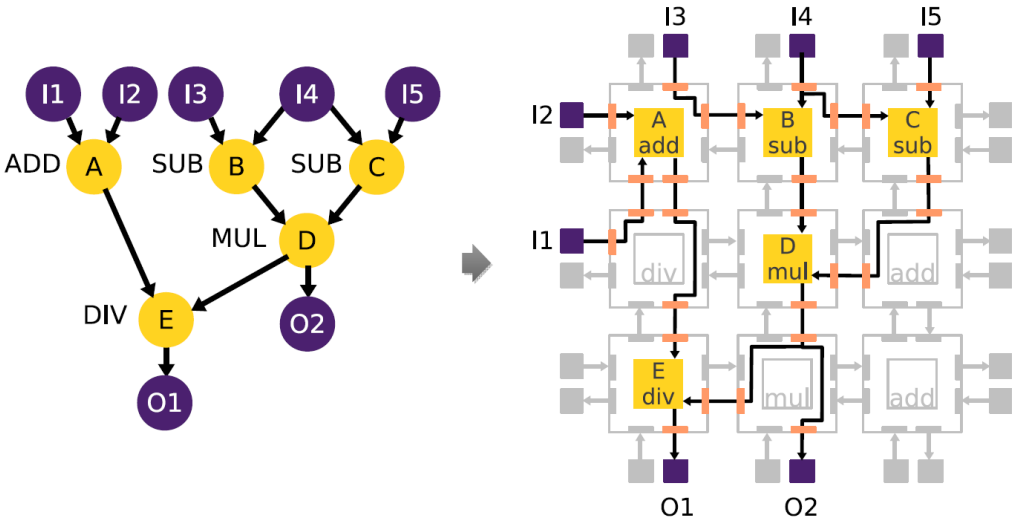



Figure 2.5: Nearest-neighbor connected Mesh of Functional units.

The larger DySER array was achieved by using a DSP block as the compute logic, thus better targeting the architecture to the FPGA.

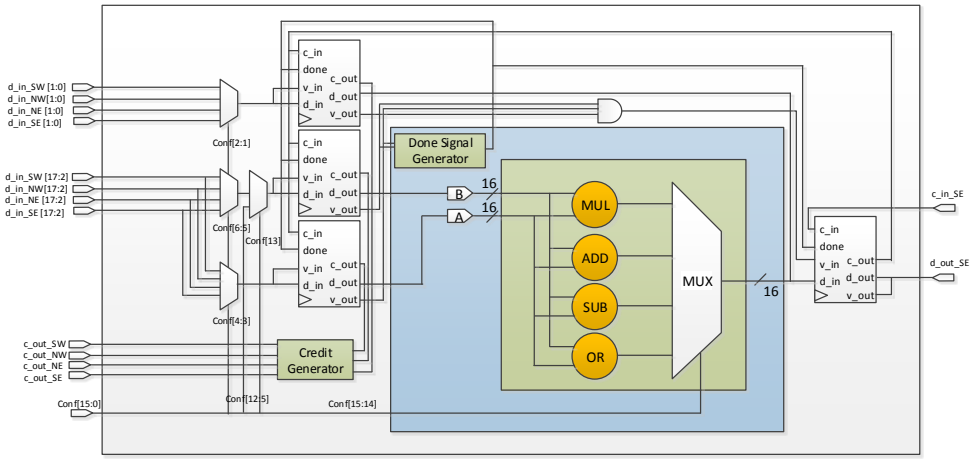
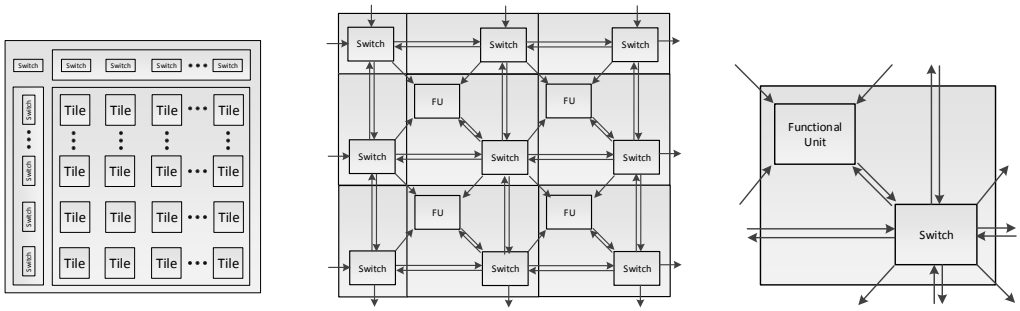



Figure 2.6: DySER functional unit.




(a) DySER block diagram. (b) Architecture of a 2×2 DySER. (c) Tile architecture.

Figure 2.7: DySER architecture as Island-style overlay.

## 2.3 Operating System for Run time management

The abstraction provided by operating system for reconfigurable computing helps in coordinating multiple Hardware (HW) tasks, managing shared reconfigurable resources among tasks and providing methods for communication and synchronization among HW and SW tasks. Scheduling is one of the most important functionality encapsulated in an OS, because which resource has to be used when by a task has to be decided by the OS. The simplest reconfiguration scheduling is to run a queue and reconfigure on demand. Scheduler must also have the feature of preemption to allocate hardware resources to a task of higher priority. various preemption schemes for reconfigurable devices were compared in [23], while mechanisms for context-saving and restoring were discussed in [24] and [25]. Communication between tasks is another important abstraction provided by the OS. Shared memory model is the common communication style used in multi-core systems but in reconfigurable computing systems several models have been proposed for effective communication such as Message passing interfaces (MPI) and Remote procedure calls (RPC).

Virtual memory is another important abstraction provided by the OS. When reconfigurable hardware is tightly coupled with a processor with Memory Management Unit (MMU) support, reconfigurable hardware can share processor's MMU. The processor can now be used by the OS to perform memory accesses and then it can feed data to reconfigurable HW for computation. This model brings good control but

reduces the ability of the processor to act as a compute unit as is kept busy in memory transactions. DMA controllers are normally used in such cases to counter this issue of handling memory transactions. Synchronization using threads is an important functionality encapsulated in the OS. Reconfigurable HW based computations are inherently concurrent where more than one hardware tasks occur in parallel with software tasks, in such scenarios synchronization between tasks is a critical issue. The simplest method is thread style synchronization on hardware tasks.

A number of researchers have focused on providing system support for reconfigurable hardware so as to provide a simple programming model to the user and effective run-time scheduling of hardware and software tasks [26, 27, 28, 29]. Several operating systems have been developed for reconfigurable hardware[27, 30, 31, 26, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42]. Several Linux extensions have also been proposed to support reconfigurable hardware [33, 36, 39, 37]. A real time operating system [38] was proposed to provide management abstraction by exploiting Dynamic Partial Reconfiguration (DPR) capability of FPGAs. Since the reconfiguration time in a dynamically reconfigurable system has an impact on overall application performance as shown in [43], in our work, we propose to use RTOS (uC/OS-III) as a run time manager for overlay architectures by demonstrating fast context switching.

# Chapter 3

# RTOS as a run time manager

## 3.1 Introduction

In order to use an RTOS as a run time manger we need to measure and compare the performance parameters of commercial RTOS. In order to quantify the performance of an RTOS, we benchmark its most crucial parameters. To evaluate the amount of overheads introduced due to services provided by the RTOS, we measure parameters such as Context switching time, Pre-emption time and message passing latency. While most of the available real time operating systems provide performance close to each other, we choose uC/OS-III because of its low memory footprint, well documented code and popularity.

## 3.2 Performance Metrics of an RTOS

A real time operating system provides many services to embedded system developers. Facilities such as mutex, scheduling, message passing, context switching and priority based pre-emption are few among many. The cost of these facilities is the overhead of these services and depends upon the way real time operating system has been implemented. Therefore, it is necessary to find out the cost incurred due to the overhead of implementation.

**Context Switching Time:** It is the time the system takes to switch the program flow control between two independent and active tasks so that the execution can be carried on from the same point at a later time. This time includes storing and restoring the context of the task. It is an important metric of any multitasking system. The time it takes for a context switch is dependent upon many factors such as host processing system and efficiency of the software implementation of Real-time operating system.

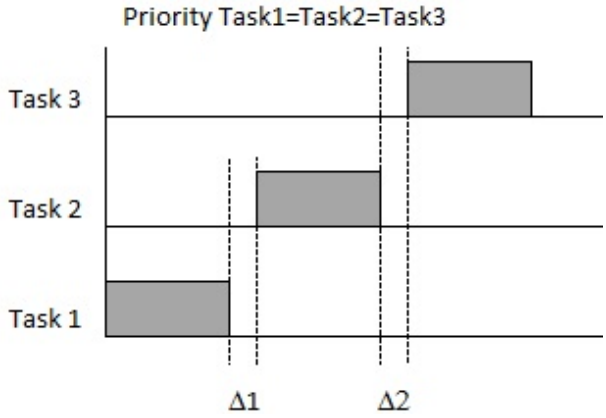



Figure 3.1: Context Switching Time.

A pictorial demonstration of this performance parameter is shown in Figure 3.1.

**Pre-emption Time:** It is the average time a higher-priority task takes to stop a lower priority task during its execution and take control of the processor. The pre-emption usually occurs when the higher priority task moves from an idle to ready state in response of an external event or when the higher priority task has been waiting for some internal event such as an expiration of a delay. Pre-emption time is another important parameter since multitasking system quiet often uses pre-emption.

Figure 3.2 illustrates pre-emption between tasks.
Pre-emption happens when a higher priority task waiting for an event becomes ready. When the event occurs and preempt the lower priority task, in our case we use message queue for that event. A message queue is a kernel object allocated by the application. It is used to pass messages from one task to another and also can be used to signal

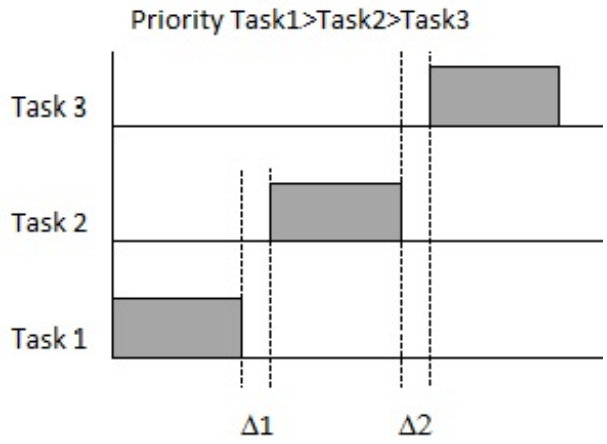



Figure 3.2: Pre-emption Time.

one task to another or from ISR to a task.

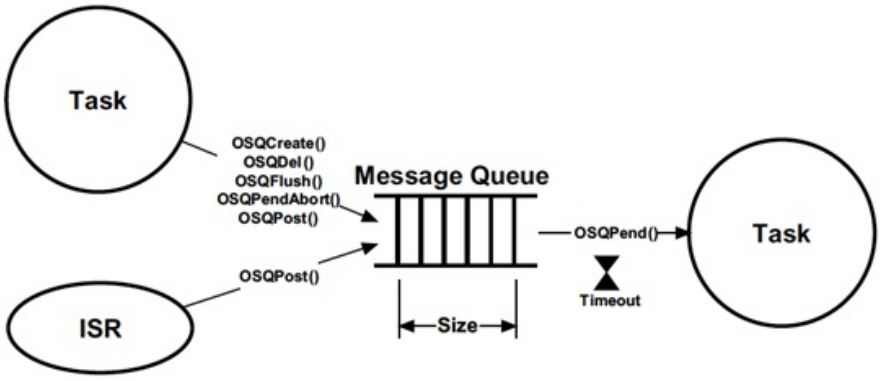



Figure 3.3: Message Queue. [1]

Figure 3.3 shows how queue can be used and shared by multiple task and Interrupt service routine.

**Message Passing latency:** This facility is provided in order to enable tasks/processes/threads to communicate and synchronize with each other. Inter-task message latency is the time from the point when a nonzero-length data message is sent from one task to another. In order to measure it properly, the sending task should stop executing immediately after sending the message and the receiving task should be suspended while waiting for it. The Figure3.4 illustrates the messgae passing between

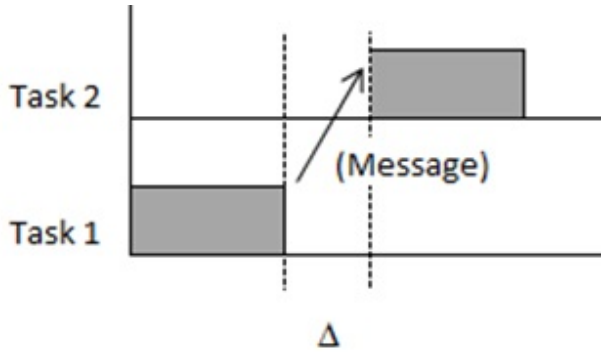



Figure 3.4: Message passing latency.

two task and the time taken for the procedure.

## 3.3 Experimental Evaluation

This section provides the details about the experiments that have been devised to perform measurements of the real time metrics on the hardware platform.

**Context Switching Time Measurement:** For the measurement of context switching time, we create two tasks (task A and task B) with equal priority. Task A starts the timer, stores the initial timer values and suspends itself. As soon as it suspends itself, it makes a context switch to task B. When Task B starts it stores the value of the timer started by task A and stops the timer. Task B now subtracts the initial value of the timer from the final value and then resumes Task A and suspends itself so that Task A can run. This is repeated 50000 times. In every iteration the context switching time is stored in variable named SUM so that at the end of the 50000 cycles,the average of the context switching time can be calculated. The maximum and minimum values are also saved for context switching.

The Figure 3.5 shows the program flow of the experiment condcuted.

**Pre-emption Time Measurement:** For pre-emption time we again create two tasks. Task A with higher priority and task B with lower priority. Task A runs and pends on an empty queue and blocks. Task B starts and enables the timer and stores

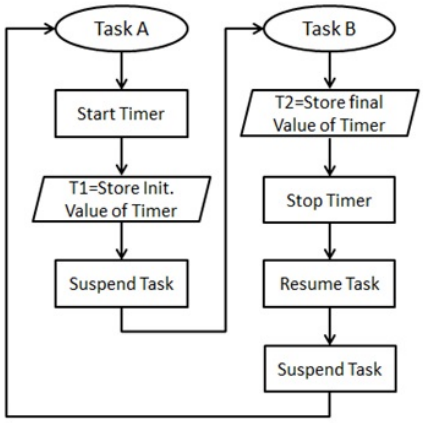



Figure 3.5: Context Switching Experiment Flow Chart.

the initial time value. Task B has now posted into the queue and hence is pre-empted by the higher priority task A which now stores the final value and subtracts the two values of timer to get the pre-emption time.

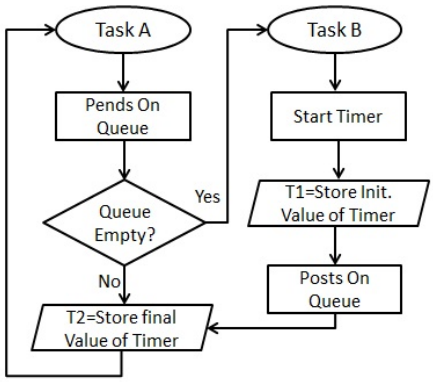



Figure 3.6: Pre-emption Time Experiment Flow Chart.

This is carried out 50000 times and maximum, minimum and average values of pre-emption time are calculated. Finally, the context switching time, which was calculated in the first experiment is subtracted to get the pre-emption time only.The Figure 3.6 shows the program flow of the experiment condcuted for pre-emption time.

**Message Passing Latency Measurement** For this experiment, again two tasks are created. Task A has a higher priority than task B. Task A starts and pends on

the empty Queue. Since the queue is empty, task A blocks instantaneously and task B starts running. Task B starts the timer and posts a message into the queue. As soon as task B posts a message in the queue, the higher priority task pending on the message becomes ready again and starts running i.e Task A. It consumes the message and the queue becomes empty again. Now task A starts pending on the message again, it blocks again and a context switch happens from task A and task B. Now task B stores the final timer value and subtracts the initial timer value from the final timer value after stopping the timer. Similar to context switching we store the maximum, minimum and calculate average after 50000 iterations. The only difference is that we subtract the context switching time from the message passing time.

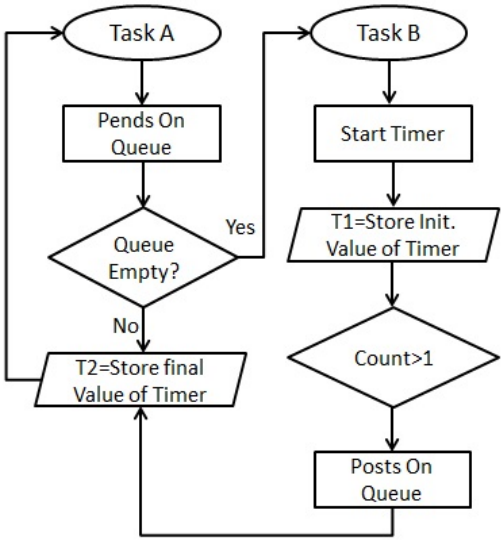



Figure 3.7: Message Passing Time Experiment Flow Chart.

Fig. 3.7 shows the program flow of the experiment condcuted for message passing time. We use Zedboard for the experiments. Experimental Setup Zed Board:

- Xilinx Zed board(ZYNQ 7020)
- Workstation/Laptop
- Xilinx Tool chain
- $u$C/OS-III Zynq-7020 port
- Compiler Optimization Flag O3

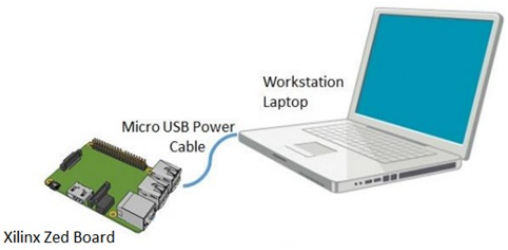



Figure 3.8: Experimental Setup Zed Board.

Table 3.1: uCOS-III: Benchmarks On Xilinx Zynq (ARM Cortex-A9)

| No. | Metric | Compiler Optimization | Average Time | Maximum Time | Minimum Time |
|---|---|---|---|---|---|
| 1. | Context Switching Time | Disabled | 1uS | 1uS | 1uS |
| | | Enabled | 0.65uS | 0.71uS | 0.66uS |
| 2. | Pre-emption Time | Disabled | 2uS | 2uS | 2uS |
| | | Enabled | 1.26uS | 1.36uS | 1.26uS |
| 3. | Messaging Latency | Disabled | 1uS | 1uS | 1uS |
| | | Enabled | 0.66uS | 0.68uS | 0.66uS |
| 4. | Task Resume Time | Disabled | 0.312uS | 0.375uS | 0.25uS |
| | | Enabled | 0.221uS | 0.273uS | 0.18 uS |
| 5. | Task Suspend Time | Disabled | 0.25uS | 0.25uS | 0.25uS |
| | | Enabled | 0.19uS | 0.21uS | 0.19 uS |
| 6. | Semaphore Post Time | Disabled | 0.183uS | 0.72uS | 0.18uS |
| | | Enabled | 0.116uS | 0.61uS | 0.11 uS |
| 7. | Semaphore Pend Time | Disabled | 0.17uS | 0.715uS | 0.166uS |
| | | Enabled | 0.11uS | 0.63uS | 0.11 uS |

Table 3.1 shows the results produced while running the experiments on the Xilinx Zynq platform. A similar set of experiments were conducted using STM32F107 platform. This platform in contrast to zed board is a low end platform. It gives us the perspective to compare the performance of a real time operating system on multiple platforms and how the underlying hardware affects the performance parameters.

Experimental Setup STM32:

- STM32F107(ARM CORTEX M3)

- Workstation/Laptop
- IAR Tool Chain
- $u$C/OS-III ARM CORTEX M3 port
- Compiler Optimization Flag O3


Figure 3.9: Experimental Setup STM32.

Table 3.2: uCOS-III: Benchmarks On STM32F107 (ARM Cortex-M3)

| No. | Metric | Compiler Optimization | Average Time | Maximum Time | Minimum Time |
|---|---|---|---|---|---|
| 1. | Context Switching Time | Disabled | 19.69uS | 63.72uS | 19.11uS |
| | | Enabled | 12.83uS | 45uS | 12.5uS |
| 2. | Pre-emption Time | Disabled | 30.16uS | 66.8uS | 29.47uS |
| | | Enabled | 19.02uS | 45.94uS | 18.72uS |
| 3. | Messaging Latency | Disabled | 51.13uS | 95.6uS | 49.94uS |
| | | Enabled | 33.72uS | 65.5uS | 33.16uS |
| 4. | Task Resume Time | Disabled | 4.97uS | 9.24uS | 4.78uS |
| | | Enabled | 3.51uS | 6.72uS | 3.47 uS |
| 5. | Task Suspend Time | Disabled | 1.38uS | 5.06uS | 1.37uS |
| | | Enabled | 1.07uS | 4.42uS | 1.08 uS |
| 6. | Semaphore Post Time | Disabled | 1.1uS | 4.32uS | 1.08uS |
| | | Enabled | 0.7uS | 3.7uS | 0.66 uS |
| 7. | Semaphore Pend Time | Disabled | 1.02uS | 4.29uS | 1uS |
| | | Enabled | 0.717uS | 3.83uS | 0.7 uS |

Table 3.2 shows the results produced while running the experiments on the STM32 ARM CORTEX M3 platform.

## 3.4 Summary

In order to access real time operating system as a run time manager for FPGA overlay architecture, we needed to analyze the performance of the operating system by measuring the key performance parameters.

This chapter highlights the significance of each of the experiments conducted and the performance measurements made followed by the explanation of the experiments devised and conducted. The chapter is concluded with the illustration of the experimental setup which can be utilized if there is a need of mimicking these experiments. Finally, we present the results obtained by running these experiments.

The experiments are performed on multiple hardware to give us a perspective on the impact different hardware make on the performance of the run time manager. The high end hardware platform Zynq-7020 had the worst time of 2 *us* whereas we saw 95 *us* as the worst time when running the experiments on the ARM CORTEX M3.

# Chapter 4

# Embedding Overlay into the Zynq

## 4.1 Introduction

While the Zynq platform provides high speed AXI interfaces for communication, the limited size of the reconfigurable fabric means that it is ideally suited to applications with often used and reconfigured accelerators, rather than static accelerators. Hence, integration of overlay architecture with an embedded processor(s) and the associated memory subsystem is crucial to enable efficient management and sharing of limited reconfigurable hardware resources among multiple tasks. One example [16] of pairing the overlay (Intermediate Fabric (IF) Overlay [5]) with a high performance ARM processor via an AXI interface in a commercial computing platform (the Xilinx Zynq[20]) is shown in Fig. 4.1. We use the same example here for the explanation of embedding process. Zynq platform partition the hardware into a processor system (PS), containing one or more processors along with peripherals, bus and memory interfaces, and other infrastructure, and the programmable logic (PL) where custom hardware can be implemented. The two parts are coupled together with high throughput interconnect to maximize communication bandwidth. We explain the system architecture, run time management of the system and concept of sharing the overlay in the following sections.

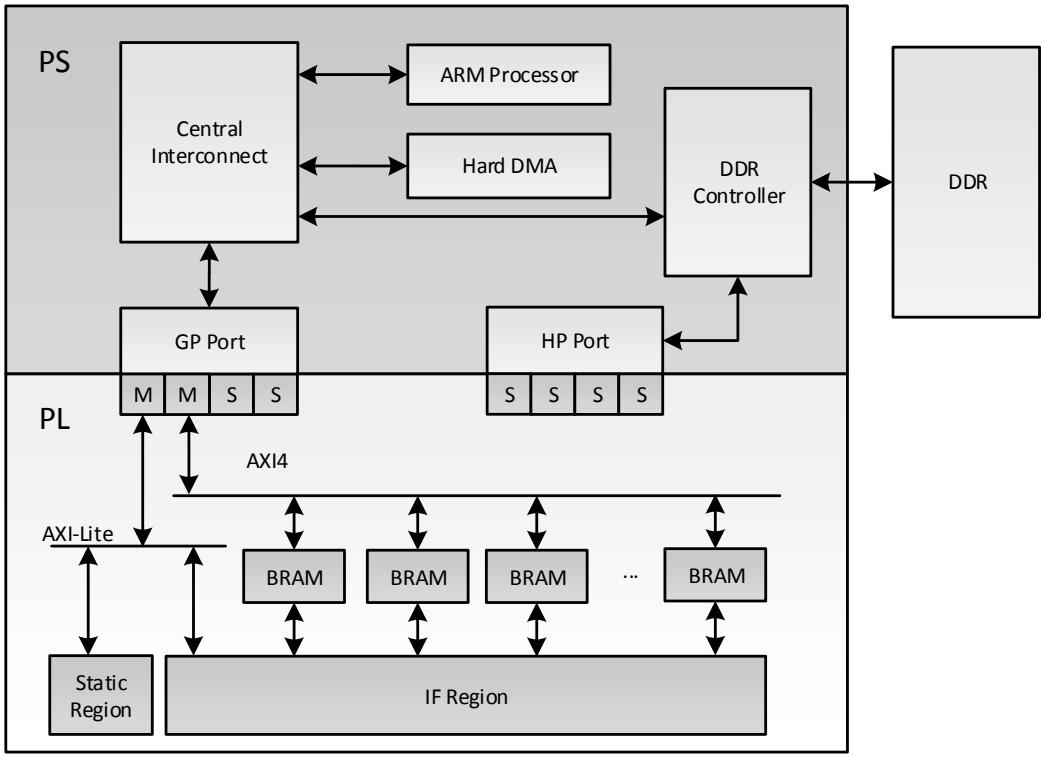



Figure 4.1: Intermediate Fabric (IF) Interfacing with Host Processor

## 4.2 System Architecture

The Overlay architecture described in this report is aimed at streaming signal processing circuits and consists of a data plane and a control plane as shown in Fig. 4.2. The data plane (shown in Fig. 4.3) contains programmable Processing Element (PE) and programmable interconnections, whose behaviour is defined by the contents of the configuration registers referred to as context frame registers. The control plane controls the movement of data between the data plane and the Block RAMs, and provides a streaming interface between them. The control plane consists of the context frame registers and a context sequencer (finite state machine) which is responsible for loading the context frame registers from the context frame buffer (CFB). The context frame buffer (CFB) can hold multiple sets of context frame registers for multi-context execution.

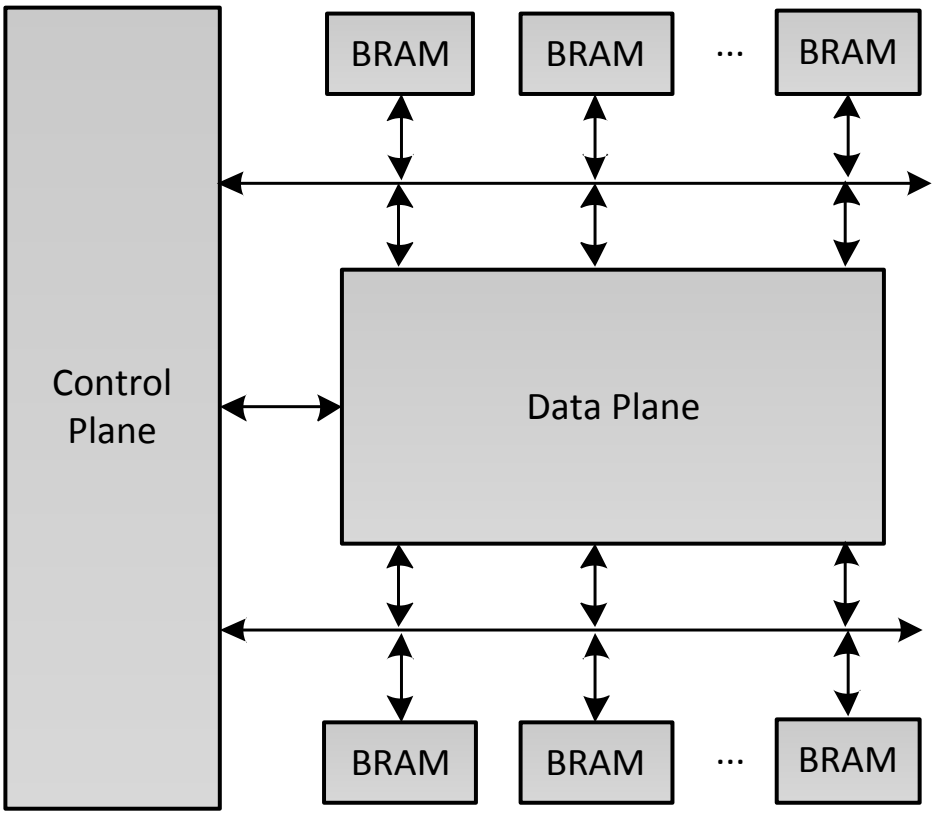



Figure 4.2: Block Diagram of the IF overlay.

The data plane consists of programmable PEs distributed across the fabric in a grid as shown in Fig. 4.3. A PE is connected to all of its 8 immediate neighbours using programmable crossbar (CB) switches. The operation of the PEs and CBs is set by PE and CB configuration registers, respectively. Fig. 4.4 shows the internal architecture of the DSP block based PE, which consists of a DSP block and a routing wrapper. The DSP block can be dynamically configured using the information contained in the PE configuration register. One of the key benefits of using DSP blocks in modern Xilinx devices is their dynamic programmability and wide range of possible configurations that can be set at runtime using control inputs. By building often used functions into optimised compact primitives, area, performance, and power advantages are achieved over equivalent "soft" implementations in the logic fabric.

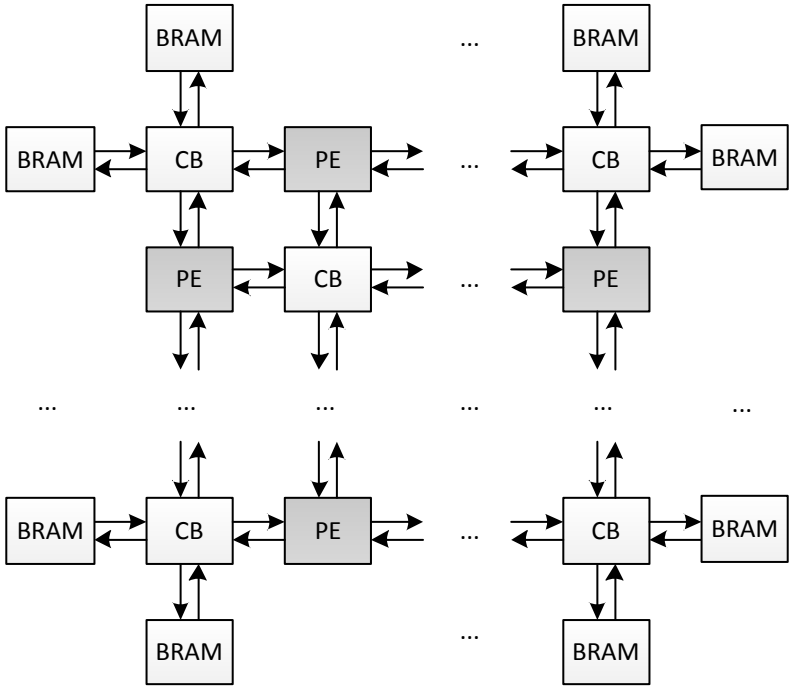



Figure 4.3: Block Diagram of the Data Plane.

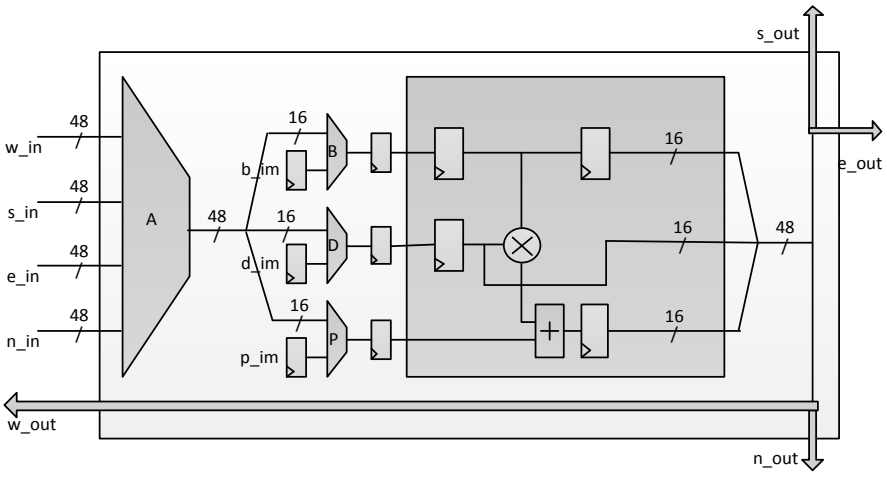



Figure 4.4: Internal architecture of processing element.

The control plane consists of a state machine based context sequencer and a context frame in the form of configuration registers. A context sequencer (CS) is needed to load context frame registers into the configurable regions and to control and monitor the execution, including context switching and data-flow. The control register is used by the PS to instruct the CS to start a HW task by checking the start bit of the control register. The status register is used to indicate the HW task status, such as the completion of a context or of the whole HW task. In the *IDLE* state, the CS waits for the control register's start bit to be asserted before moving to the *CONTEXT_START* state. In this state, it generates an interrupt *interrupt_start_context*, and then activates a context counter before moving to the *CONFIGURE* state. In this state, the CS loads the corresponding context frame from the CFB to the control plane of the Overlay to configure the context's behaviour. Once finished, the CS moves to the *EXECUTE* state and starts execution of the context. Once execution finishes, the CS moves to the *CONTEXT_FINISH* stage and generates an *interrupt_finish_context* interrupt. The CS then moves to the *RESET* state which releases the hardware fabric and sets the status register completion bit for the context. This behaviour is shown in Fig. 4.5.

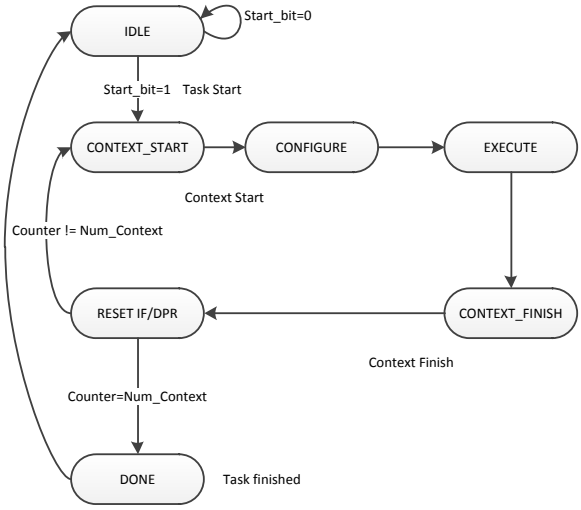



Figure 4.5: State-machine based Context Sequencer.

## 4.3 Run time Management

When paired as a coprocessor, run-time management, including overlay configuration loading, data communication, can be carried out using an operating system (Linux) [17] and also using a commercial hypervisor [44]. Firstly, user needs to identify a kernel, as described in code listing 4.1, to be implemented on top of overlay. Then data flow graph (DFG) can be extracted after compiling this code using compiler front-end. After that a place and route tool can be used to map the DFG on top of overlay. After generating configurations based on the placement and routing, kernel code can be transformed in the code containing overlay APIs as shown in code listing 4.1. The working of modified C description is pretty straight-forward as shown in code listing 4.1. First it allocates input and output BRAMs as overlay memory. Then it loads the overlay configuration for the task. After that it transfers the input data to input BRAM and triggers the overlay. Finally, the Overlay starts processing the input data in a streaming fashion and transfer the processed data to output BRAMs.

Table 4.1: Source Code Transformation

(a) Original C description

```
#include<math.h>
#define SIZE 1000

#ifdef KERNEL
int kernel(int x)
{
   int temp = 16*x;
   return (x*(x*(temp*x-20)*x+5));
}
#endif

#ifndef KERNEL
int main(void)
{
   int i;
   int in[SIZE];
   int out[SIZE];
   for (i=0; i<SIZE; i++){
      out[i] = kernel(in[i]);
   }
   return 0;
}
#endif
```

(b) Modified C description

```
#include <overlay.h>
#include<math.h>
#define SIZE 1000

void kernel(int *a, int *b, int length){
   //allocate BRAM as overlay memory
   memory_a = overlay_malloc(size_a);
   memory_b = overlay_malloc(size_b);
   //load overlay configuration for the task
   load_configuration();
   //copy inputs
   overlay_transfer_data(a, memory_a, size_a);
   //Trigger overlay to process data
   overlay_trigger_and_wait();
   //copy_outputs
   overlay_transfer_data(memory_b, b, size_b);
}
int main(void){
   int in[SIZE];
   int out[SIZE];
   kernel(in, out, SIZE);
   return 0;
}
```

## 4.4 Overlay sharing among multiple HW tasks

In the previous section, run time management of overlay was described for a single task using a set of APIs. Since the overlay can be used as a shared resource, multiple tasks can use it in a time-multiplexed manner for accelerated execution. Fig. 4.6 presents a scenario where streaming input devices, such as audio source, are attached to the platform (requiring accelerated processing on streaming data). Input buffers can accept data from the devices which then needs to go to the overlay local memory for execution (using $overlay\_transfer\_data$ API mentioned in the previous section).

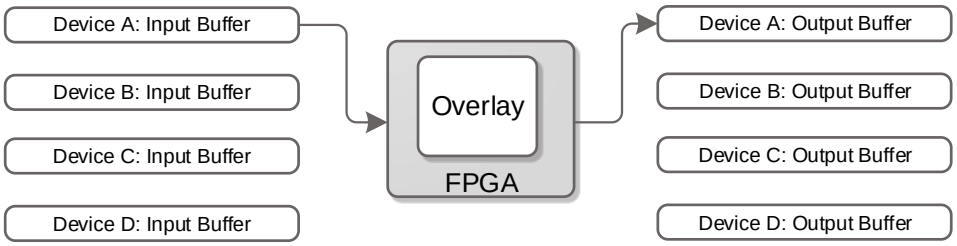



Figure 4.6: Overlay Sharing among multiple HW tasks

If only one device is active, then the kernel function (mentioned in the previous section) can be used (having same input buffer and output buffer as function arguments) within a while loop. In this way, the device will keep sending data in the input buffer and the overlay will keep processing the data available in the input buffer. However, if all the attached devices are active, the while loop should invoke multiple kernel calls in a round robin fashion having separate input buffer and output buffer as function arguments. Using bare metal software application, it is very difficult to manage the overlay as a shared resource in a scenario where multiple tasks need to access the same overlay resources.

## 4.5 Summary

In this chapter, we presented an approach of embedding an overlay withing a hybrid computing platform. We described the system architecture and run time management process. In a scenario, where multiple hardware tasks need to use the same resource, it is very difficult to manage the resource using bare metal software application. In the next chapter, we explain the concept of RTOS for run time management of the overlay architecture.

# Chapter 5

# Runtime management of Overlay using RTOS

## 5.1 Introduction

Runtime management of overlay using bare metal programming is not the most efficient method available. Bare metal programming has no service available for protection of a shared resource and since it runs through the programs in a sequential manner it is not able to do anything else other than kill time when in delay or waiting state. These problems can be taken care of by using a multi-tasking run time manager such as a real time operating system. Real time operating system provides facilities for resource management, such as a semaphore/mutex, they are also equipped with services of critical section and scheduler locking to avoid any interruption in execution of a process. Moreover; no time is wasted while any delay or wait state is encountered and instead of just killing time other processes are made to run. This way the execution is more efficient and shared resources, such as overlay, are protected. We use uC/OS-III as an RTOS for managing SW and HW tasks on the platform as shown in Fig. 5.1. The following sections shed light on the benefits gained by using an RTOS instead of bare metal programming method. Also, we discuss the methods in details which are used to protect hardware calls and functionalities.

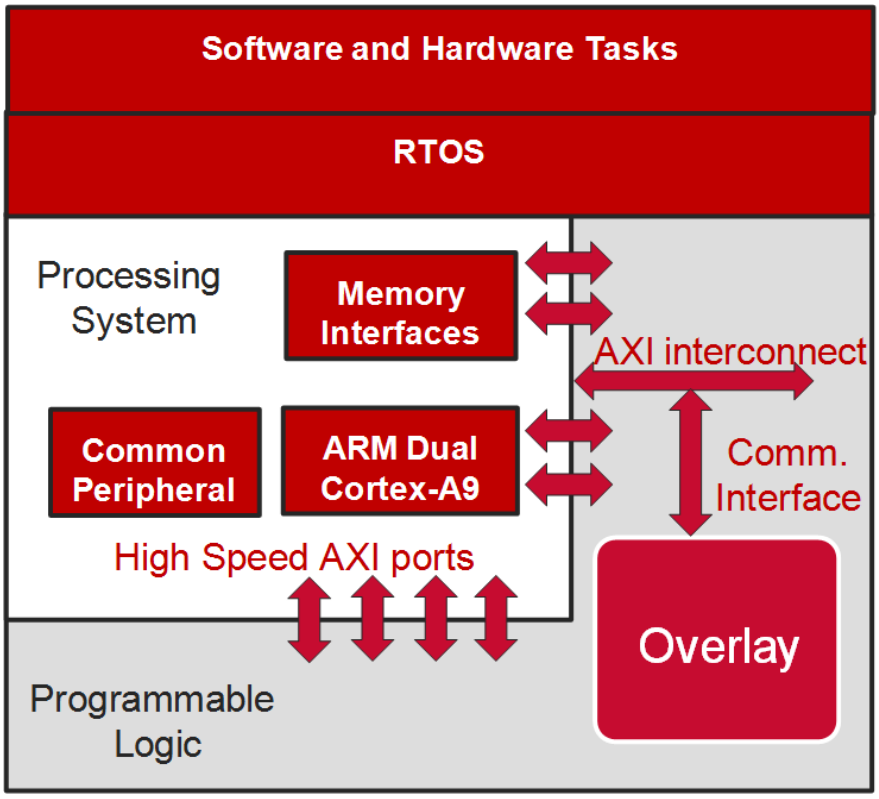



Figure 5.1: Overlay Sharing among multiple HW tasks

## 5.2 Resource Management

One of the major features of RTOS is the management of resources between multiple tasks. RTOS uses various methods to share resources between the tasks such as semaphore, mutex and critical sections. The purpose is to avoid contention for the resource and to make sure the resource is not corrupted. It is therefore important to ensure that each task can access the resource exclusively. The shared resource could be a register, a variable, a data structure, some memory location or an I/O device. In our case it is an overlay which is shared between multiple tasks. It has to be taken care that when one task is communicating with the overlay no other task should take control between the processes and corrupt the data. This can be accomplished by different methods.

The most standard methods of getting complete access to shared resources and to create critical sections are:

- Critical Section
- Locking Schedular
- Using semaphores/Mutex

**Disabling interrupts** Interrupts have higher priority than the task code present in main program, hence disabling them should be the last resort and should be done in only certain scenarios. Interrupts should only be disabled when the part of the task for which the interrupt are disabled very short, or else it would lead to high interrupt latency which could be disastrous. uC/OS-III provides facilities in the form of APIs for this purpose. The code enclosed between these function calls will be atomic and cannot be interrupted by any interrupt or other tasks. Fig. 5.2 shows the code segment which protects the shared resource by containing it in a critical section.

```
void foo1 (void)
{
    CPU_CRITICAL_ENTER();          (1)
    Access the resource;           (2)
    CPU_CRITICAL_EXIT();           (3)
}
```

Figure 5.2: Critical Section by diabling Interrupts.

**Locking the scheduler** Another method to avoid any conflict between tasks is to lock the scheduler. The scheduler is responsible for giving the control of the microprocessor from one task to another and locking the scheduler would let the task handling shared resource go undisturbed. But unlike critical sections the task could be interrupted by I/O devices. Locking the scheduler essentially means that the task which is locking the scheduler has the highest priority. This method is better than Interrupt disabling in the respect that it does increase the interrupt latency. Fig. 5.3 shows the code segment which protects the shared resource by containing it in a piece of code that locks and unlocks schedular.

```
void foo(void)
{
    OSSchedLock(&err);              (1)
    Access the resource;            (2)
    OSSchedUnlock(&err);            (3)
}
```

Figure 5.3: Resource protection by schedular locking.

**Resource protection using Mutex** Mutex/Semaphore are a mechanism which is used to govern access to shared resources in a multithreaded environment. A semaphore or mutex is a kind of an access key which is taken by the process running to protect some part of the code which is a shared resource. Unless the key is released by that process no other process can access or alter the shared resource. Once the key is released the other process will now take the key and in turn access the shared resource. Fig. 5.4 shows the code segment which protects the shared resource by containing it in a piece of code that locks and unlocks schedular.

```
while (DEF_ON)
{
    OSMutexPend((OS_MUTEX *)&MyMutex,                    (1)
                (OS_TICK   )0,
                (OS_OPT    )OS_OPT_PEND_BLOCKING,
                (CPU_TS    *)&ts,
                (OS_ERR    *)&err);
    Acquire shared resource!!                            (2)
    OSMutexPost((OS_MUTEX *)&MyMutex,                    (3)
                (OS_OPT    )OS_OPT_POST_NONE,
                (OS_ERR    *)&err);
}
```

Figure 5.4: Resource protection by mutex/semaphore.

## 5.3 Context Switch Overheads

It is often the case that the hardware is being utilized by multiple processes and the shared resource is accessed by more than one process/thread. When the hardware functionality is spread across multiple tasks one parameter becomes significantly important i.e. context switching time. When a function is divided in chunks and each part is completed by different task the context switching time between one task to

another takes great prominence. The greater the context switching time the greater the overhead and in turn slower response time. We ran experiments in order to quantize the context switching time. Since context switching can happen due to multiple reason we tested out the context switching for few common scenarios such as task suspension, switching due to mutex held by a task and switching due to an empty queue. The results are shown in a figure below. These experiments are devised to measure the context switching time in different scenarios. Brief description of these scenarios are described below.

**Task Suspension** The experiment is devised so that a timer measures the time of switching from one task to another by using task suspension. These achieved by calling the task suspension API. It is repeated significant number of times and then averaged to get the context switching time.

**Mutex Access** The second experiment measures the context switching time by also using a timer.The timer records the time a task asks for a mutex and doesn't get it and the switch is made to the other ready task.This time is saved multiple number of times and averaged out to give us the overhead.

**Queue Empty** This experiment measures context switching time when the switch happens due access to an empty queue. When a task tries to read from an empty queue the control is transferred from one task to another.This transfer fo control is measured by the timer and similar to the other experimients averaged out. Table 5.1 shows the context switching time measured using various experiments.

Table 5.1: Context Switching Time

| No. | Metric | Average Time |
|---|---|---|
| 1. | Task Suspension | 1.72uS |
| 2. | Mutex Access | 4.38uS |
| 3. | Empty Queue | 2.08 uS |

## 5.4 Summary

This chapter highlight the contrasting factor between bare metal programing and real time operating system for run time management of overlay architecture. The chapter points out the downside of using bare metal programming for run time management of overlay and sheds light upon the benefits attained by using real time operating system.

This chapter also explains the resource management facilities made available to the user when using real time operating system as a run time manager. The chapter is concluded with emphasis on the importance of context switching as a parameter which can gauge how well a multi-threaded system can perform.

# Chapter 6

# Conclusions and Future Work

This chapter concludes and summarizes this report. Furthermore, in this chapter we discuss future research directions in detail.

## 6.1 Conclusions

This report proposed an approach for embedding FPGA overlay architecture into the Xilinx Zynq platform and demonstrated the runtime management of overlay using uC/OS-III. This work included developing an understanding of hardware acceleration concept, overlay architectures and RTOS concepts. Experiments were designed to evaluate the performance of uC/OS-III on the Xilinx Zynq by quantifying performance metrics such as context switching time, preemption time etc. A set of use-case scenarios and preferred scheduling mechanisms were presented by considering the overlay as a shared resource among tasks requiring hardware acceleration. Before we could begin embedding overlay into Zynq, an in-depth knowledge of the current trends and previous efforts in the field of overlay architectures were studied to compare and contrast their features.

A performance analysis of uC/OS-III was presented in chapter 3. Observations and results achieved by measuring standard RTOS metrics in a reliable and unbiased manner were discussed. We observed that the parameters such as Context Switching, Pre-emption Time and Message passing Latency introduce significant overheads

(up to 100 *us*) for a low end device (STM32 Microcontroller). We observed a significant reduction in the overheads when employing high end device (Xilinx Zynq SoC). Overheads of up to 2 *us* were observed in case of Zynq. An approach for embedding FPGA overlay architecture into the Xilinx Zynq platform was presented in chapter 4. Use-case scenarios and preferred scheduling mechanisms were presented in 5. Furthermore, the approach presented in this report facilitates high level application developers to use uC/OS-III as a run time manager of overlay architectures for hardware acceleration.

Embedding multiple overlay instances Real life application demonstration

## 6.2 Future work

Some of the main future research directions are embedding multiple overlay instances and real life application demonstration. We describe these directions in detail as follows:

- **Embedding multiple overlay instances**: Due to single overlay instance in the platform, hardware tasks need to wait for the release of overlay resources. We plan to use multiple overlay instances within the system to run multiple hardware tasks in parallel.
- **Real life application demonstration**: We plan to demonstrate some real life applications on the platform where fast context switching is required for the hardware tasks.

Finally, with these initiatives we hope to demonstrate that overlay architectures can be used as high performance programmable accelerators within a hybrid computing platform managed by a real time operating system.

# Bibliography


[1] Message queue. `https://doc.micrium.com/display/osiiidoc/Message+Queues`.

[2] Greg Stitt. Are field-programmable gate arrays ready for the mainstream? *IEEE Micro*, 31(6):58–63, 2011.

[3] C. Plessl and M. Platzner. Zippy - a coarse-grained reconfigurable array with support for hardware virtualization. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, pages 213–218, 2005.

[4] Neil W. Bergmann, Sunil K. Shukla, and Jrgen Becker. QUKU: a dual-layer reconfigurable architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 12:63:1–63:26, March 2013.

[5] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–22, October 2010.

[6] Davor Capalija and Tarek S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[7] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying

functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.

[8] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient overlay architecture based on DSP blocks. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2015.

[9] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell. Adapting the DySER architecture with DSP blocks as an Overlay for the Xilinx Zynq. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2015.

[10] Cheng Liu, C.L. Yu, and H.K.-H. So. A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme FPGA frequency. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 228–228, 2013.

[11] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[12] O.T. Albaharna, P. Y K Cheung, and T.J. Clarke. On the viability of FPGA-based integrated coprocessors. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 206–215, 1996.

[13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based Processor/Accelerator systems. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 33–36, 2011.

[14] Yun Liang, Kyle Rupnow, Yinan Li, and et. al. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012(649057):1–14, January 2012.

[15] David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Queue*, 11(2):40:40–40:52, February 2013.

[16] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell. Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform. *Journal of Signal Processing Systems*, 77(1–2):61–76, Oct. 2014.

[17] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of cgra. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 9–16, 2014.

[18] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the dyser hardware accelerator into opensparc. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[19] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 503–514, 2011.

[20] Xilinx Ltd. Zynq-7000 technical reference manual. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2013.

[21] G. Stitt and J. Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters*, 3(3):81–84, September 2011.

[22] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for fpga research. In *Field-Programmable Logic and Applications*, pages 213–222, 1997.

[23] K. Jozwik, H. Tomiyama, S. Honda, and H. Takada. A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems. In

*Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2010.

[24] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 223–228, 2005.

[25] K. Rupnow, Wenyin Fu, and K. Compton. Block, drop or roll(back): Alternative preemption methods for RH multi-tasking. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 63–70, 2009.

[26] Gordon Brebner. A virtual hardware operating system for the Xilinx XC6200. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 327–336. 1996.

[27] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, November 2004.

[28] M. Vuletic, L. Righetti, L. Pozzi, and P. Ienne. Operating system support for interface virtualisation of reconfigurable coprocessors. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, pages 748–749, 2004.

[29] K. Rupnow. Operating system management of reconfigurable hardware computing systems. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 477–478, 2009.

[30] Herbert Walder and Marco Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 284–287, 2003.

[31] Aws Ismail. *Operating system abstractions of hardware accelerators on field-programmable gate arrays*. Thesis, August 2011.

[32] Xun Changqing, Wen Mei, Wu Nan, Zhang Chunyuan, and H.K.-H. So. Extending BORPH for shared memory reconfigurable computers. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 563 –566, August 2012.

[33] H.K.-H. So, A. Tkachenko, and R. Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 259–264, 2006.

[34] Benjamin Krill, A. Amira, and H. Rabah. Generic virtual filesystems for reconfigurable devices. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1815–1818, 2012.

[35] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl. ReconOSAn operating system approach for reconfigurable computing. *IEEE Micro*, 2013.

[36] Enno Lübbers and Marco Platzner. ReconOS: multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):8, October 2009.

[37] John H. Kelm and Steven S. Lumetta. HybridOS: runtime support for reconfigurable accelerators. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 212–221, 2008.

[38] X. Iturbe, K. Benkrid, A.T. Erdogan, T. Arslan, M. Azkarate, I. Martinez, and A. Perez. R3TOS: a reliable reconfigurable real-time operating system. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 99–104, 2010.

[39] D. Gohringer, S. Werner, M. Hubner, and J. Becker. RAMPSoCVM: runtime support and hardware virtualization for a runtime adaptive MPSoC. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[40] Grant B. Wigley, David A. Kearney, and David Warren. Introducing ReConfigME: an operating system for reconfigurable computing. In *Field-Programmable Logic and Applications*, pages 687–697. January 2002.

[41] A. Hofmann and K. Waldschmidt. SDVMˆR: a scalable firmware for FPGA-Based multi-core systems-on-chip. In *Symposium on VLSI (ISVLSI)*, pages 387 –392, April 2008.

[42] Andreas Hofmann, Klaus Waldschmidt, and Jan Haase. SDVMˆR - managing heterogeneity in space and time on multicore SoCs. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 142–148, June 2010.

[43] K. Vipin and S. A. Fahmy. ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq. *IEEE Embedded Systems Letters*, 6(3):41–44, September 2014.

[44] Khoa Dang Pham, Abhishek Kumar Jain, Jin Cui, Suhaib A Fahmy, and Douglas L Maskell. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2013.