UNIVERSITÀ DI PISA
SCUOLA DI INGEGNERIA
SCUOLA SUPERIORE SANT'ANNA

Master of Science in Embedded Computing Systems

Software support for dynamic partial reconfigurable FPGAs on heterogeneous platforms

Supervisors:
Prof. Giorgio Buttazzo
Dott. Mauro Marinoni

Author:
Marco Pagani

Academic Year 2015/2016

Contents

Abstract

This thesis addresses the design and implementation of a software support for real-time systems developed on heterogeneous platforms that include a processor and an FPGA with dynamic partial reconfiguration capabilities. The software support enables tasks to request the execution of accelerated functions on the FPGA in parallel with other tasks running on the processor. Accelerated functions are dynamically allocated on the FPGA depending of the availability of the area and the online requests issued by the processor, so extending the concept of multitasking to the FPGA resource domain. The performance of the allocation mechanism has been evaluated in terms of speed-up and response times. The achieved results show that the system is able to guarantee bounded delays and acceptable overhead that can be taken into account for a future schedulability analysis of real-time applications.

Chapter 1

Introduction

For the last 50 years Moore's law has been one of the leading principles guiding the semiconductor industry. Based on the empirical observation that the number of transistors in a dense integrated circuit doubles approximately every two years, the law has successfully predicted the evolution of computer processors that has dominated the last decades of the previous century.

One of the enabling factors behind the Moore's laws is the geometric scaling of complementary metal oxide semiconductor (CMOS) transistors, the basic switching elements inside integrated circuits. As transistor gets physically smaller their power density stays constant, therefore the power consumption scales downward with their area. This effect, referred to as Dennard scaling, allowed manufacturers to rise clock frequencies every generation without significantly increasing the power consumption. For decades frequency scaling has been the main method to achieve performance gains.

However, around the half of the first decade of the 2000s, geometric scaling had stopped to guarantee the same power reduction benefits. As transistors came closer in size to the atomic scale leakage current and other non-ideality factors became more important, setting the baseline for power consumption. This led to the end of frequency scaling as the main technique to achieve performance gains. To overcome this issue the industry have reacted with a paradigm shift, moving from single core processors towards multicore systems, increasing the level of parallelism instead of rising the clock frequency.

However the still growing number of transistors that can be integrated on the same die, together with the end of Dennard scaling, resulted in a trend of increasing power density. This led to the consequences that, in order to meet

power and thermal constraints, not all the transistors available in a device can be fully utilized. The portion of transistors that must be deactivated or under-clocked to meet the energy requirements is referred as *dark silicon*.

To mitigate such issues and achieve a better utilization of the transistor budget the industry has begun to move towards heterogeneous systems that may include different types of processing elements like general purpose processors, graphics processing units (GPUs) and dedicated accelerators. In such systems the computational workload can be distributed among the different processing units achieving a higher level of energy efficiency and reducing the power consumption.

However modern CPUs and GPUs are suited for performing a wide range of computational activities. To support such general purpose capabilities their internal structure is rather general, and tied to a fixed granularity of parallelism, compared to a custom microarchitecure, tailored for a specific purpose. While this flexibility is beneficial in terms of software programmability it inevitably leads to some degree of inefficiency. On the other hand the high flexibility of field-programmable gate arrays (FPGAs), together with their high performance in data-flow oriented processing, have made them increasingly attractive platforms for deploying custom hardware accelerators, optimized at microarchitecture level, combining high-performance with high energy efficiency. Moreover, the support for dynamic partial reconfiguration featured by modern FPGAs further increase their flexibility, providing many advantages over conventional static designs.

To exploit the advantages of both worlds, FPGAs manufacturers have developed heterogeneous SoCs platforms that include software programmable processors and GPUs, integrated with hardware programmable FPGA fabric. On such platforms the processors can be offloaded by distributing workloads to custom developed hardware accelerators deployed on the FPGA fabric.

1.1 Objectives

The objective of this thesis is the design and development of a software support for real-time systems on heterogeneous platforms that includes a processor and a dynamically reconfigurable FPGA. A real-time software system is composed by a set of computational activities, or *tasks*, that can be classified as periodic or aperiodic. Periodic tasks consist of an infinite

sequence of activities, called *jobs*, that are regularly activated at a constant rate. In the proposed approach each job of a periodic task can request the reconfiguration of a portion of the FPGA to accelerate parts of its execution, extending the concept of multitasking to the FPGA resource domain. The system has been built to guarantee *by design* predictable execution times and bounded delays.

The performance of the implemented system has been analyzed to evaluate the feasibly of the proposed approach. In particular the hardware acceleration speedup factors and reconfiguration overheads have been measured to estimate the suitability of the proposed approach to real-time systems.

1.2 Contributions

The first part of this thesis work has been dedicated to the study of the Xilinx's Zynq SoC architecture in order to define a generic system structure suitable for supporting different types of hardware accelerators, implementing different types of operations. Special attention has been dedicated to the the communication and control mechanisms between the processor and the accelerator modules, and to the integration of multiple accelerators in the system from a hardware and software perspective.

In the second part of this work a few simple standard algorithms have been implemented both in software and hardware. The different implementations have been tested and evaluated in terms of achieved speedups. Consequently the hardware accelerators have been used to evaluate the partial reconfiguration capabilities of the platform, swapping accelerators modules at runtime under software control.

The final part of this work has been the dedicated to the development of a support library for the FreeRTOS operating system. The library abstracts the hardware acceleration and reconfiguration mechanisms providing a simple application programming interface that enables software tasks to request the execution of accelerated operations on the FPGA.

1.3 Thesis outline

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the FPGA internal structure and the integration of such

device in heterogeneous platforms. Then dynamic partial reconfiguration is introduced and the related aspects are discussed. Next, the concepts of reconfigurable computing and reconfigurable operating system are discussed. Then, a taxonomy of the existing solutions to manage reconfigurable hardware is presented.

Chapter 3 presents an overview of the specific heterogeneous system-on-a-chip platform used in this thesis, the Xilinx's Zynq. The internal structure of the Zynq is presented, and the aspects that are relevant to this work, like the internal communication infrastructure with the related protocols, and support for dynamic partial reconfiguration, are discussed in greater detail. Then, possible architectures for hardware accelerator modules are presented and classified accordingly to the method used for sharing data with the processor.

Chapter 4 presents the architecture of the reconfigurable system developed in this thesis. The hardware and software components are presented at a system level, and the integration is discussed. Consequently a model for the execution of hardware-accelerated tasks is presented.

Chapter 5 describes the system implementation details. The interface of the hardware accelerators is described, and the set of accelerator modules developed is presented. Subsequently the software components of the system are presented in greater details, and their internal structure is discussed.

Chapter 6 concludes the thesis reporting the experimental results of the tests carried out to evaluate the platform.

Chapter 2

Background

Contents

2.1 Overview of Field-Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) are integrated circuits designed to be configured after manufacturing to implement a custom hardware functionality. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs) which are custom manufactured to implement a specific functionality.

2.1.1 Internal architecture

In ASICs chips logic functions are implemented by wiring together physical logic gates, while, in the FPGAs, logic functions are realized trough configurable elements called look-up tables (LUT).

Look-up tables

A *n-inputs* LUT is a logic circuit that can be configured to implement any combinational logic function with n inputs. The internal architecture of a *n-inputs* LUT in shown in Figure 2.1. In a simplified implementation, 2^n configuration cells store the truth table of the logic function to be implemented, while a $2^n : 1$ multiplexer route the state bit of one of the cells to the output according to the inputs values.

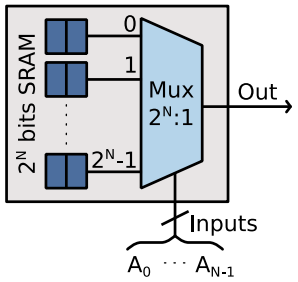


Figure 2.1: Look-up table simplified architecture.

2.1.2 Logic blocks

One or more LUTs, grouped together with carry chains and registers, constitutes the basic logic cells of the FPGAs architectures. A small set of such logic cells, called *Slices* in the Xilinx terminology or *Adaptive Logic*

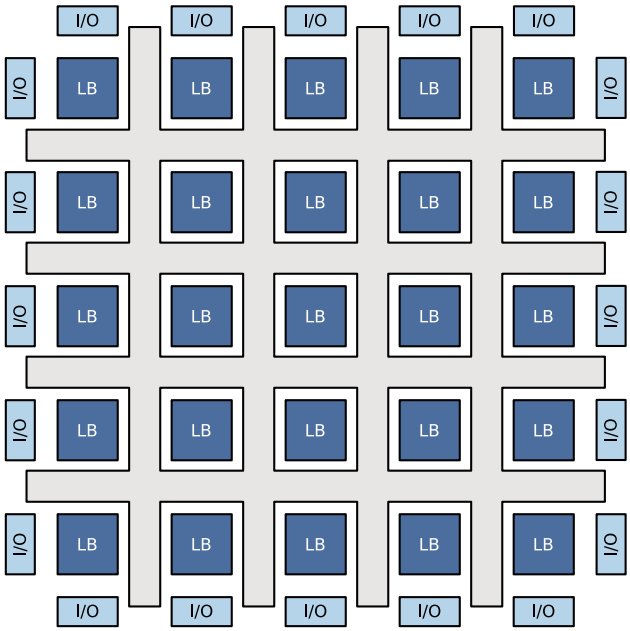


Figure 2.2: Schematization of simplified FPGA architecture: Logic blocks (LBs) are highlighted in blue, I/O blocks in light blue, programmable interconnect is in gray.

Modules by Altera are grouped together to build the basic logic block of the FPGAs structure.

In the Xilinx FPGAs architecture such logic blocks are called *CLB (Configurable Logic Blocks)* [1] while in the Altera implemetations they are referred to as *LAB (Logic Array Block)* [2]. Those blocks constitute the main logic resources for implementing sequential and combinatorial logic functions [1].

2.1.3 Interconnetion system

To build a programmable computing fabric, logic blocks are distributed in a regular 2D grid structure across the FPGA chip and connected trough a programmable routing infrastructure that allows communications between cells. At the edges of the chip, special configurable blocks, namely *IOB (I/O blocks)*, implement the input and output functions. Figure 2.2 shows an overview of a simplified FPGA architecture.

In the Xilinx 7 Series FPGA implementation, shown in Figure 2.3, each CLB is costituited by 2 *slices* and it is paired with a switch matrix to access the general routing infrastructure [1].

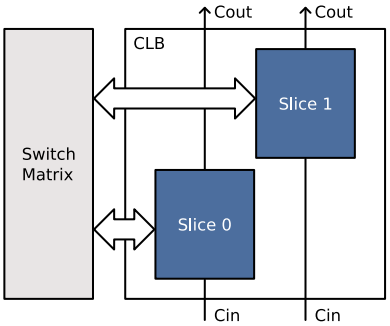


Figure 2.3: Structure of a *Configurable Logic Block (CLB)* of Xilinx 7 Series FPGAs. The CLB is constituted by 2 logic cells *(slices)* and can access the general routing resources trough a switch matrix.

2.1.4 Specialized blocks

The homogeneous fabric composed of logic blocks, I/O blocks and the interconnect is sufficient to implement logic circuits. However, in real FPGAs implementation, to increase the performance and optimize the area consumption, rows of logic blocks are interleaved with some rows of special purpose blocks, such as: *Random Access Memory blocks* (BRAMs) cells and *Adders/Multipliers* (DSPs) cells [3]. The availability of such specialized tiles allows increasing the available memory and boosts arithmetic operations that are required to implement high-speed signal processing functionality. A more realistic, non-homogeneous, FPGA architecture is shown in Figure 2.4.

2.1.5 System on a chip

To provide some degree of general purpose computing functionality, the FPGA fabric can be programmed to implement one or more fully functional microprocessors. A microprocessor that is entirely implemented with the FPGA logical resources is usually called *soft microprocessor* or *soft-core*.

The possibility of deploying an arbitrary number of processors inside the FPGA fabric provides a high level of flexibility. However, when compared to processors implementations that are optimized at silicon level, namely *hard-cores*, soft microprocessors provide a relatively low processing performance [3].

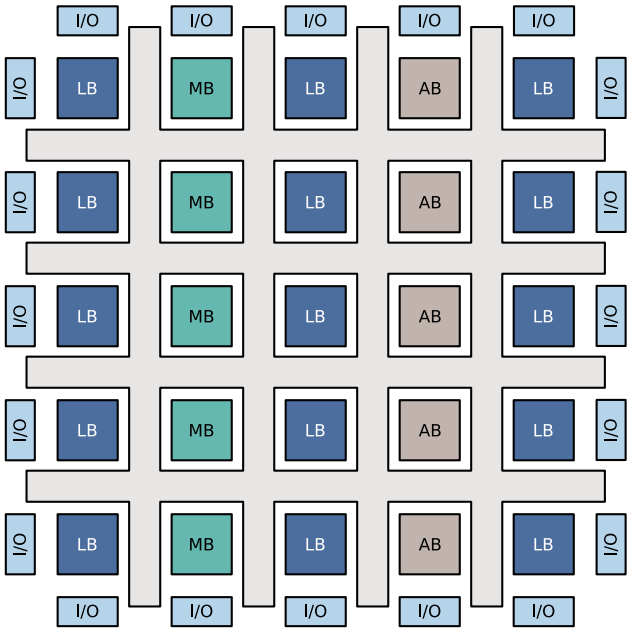


Figure 2.4: Schematization of more realistic FPGA architecture. Rows of specialized blocks: memory blocks (MBs) and arithmetic blocks (ABs) are interleaved with rows of standard logic blocks (LBs).

Heterogeneous platforms

To overcome this limitation and provide an efficient platform for heterogeneous computing, the leading manufactures of FPGAs have started to produces hybrid *systems-on-a-chip* (SoCs) where one or more CPU hard cores are tightly coupled with an FPGA device, as shown in Figure 2.5. The tight coupling between the two parts of such hybrid SoC provide a high-bandwidth and low-latency connection between the processors and the FPGA's programmable fabric.

Such SoCs platforms provide an attractive platform for deploying embedded high-performance computing solutions. The logic fabric can host custom accelerators that can offload the processors form the most computational intensive tasks.

2.2 Design flow

A typical FPGA design flow can be summarized in three main steps: *design*, *synthesis*, and *implementation*. During each step of the flow tests and

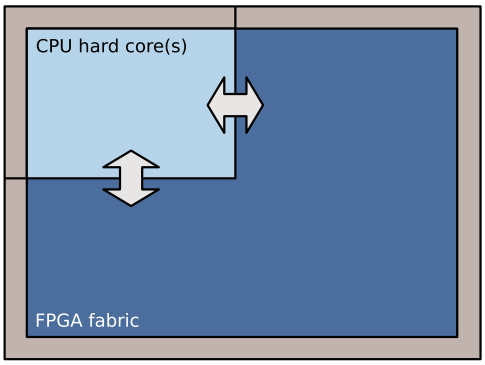


Figure 2.5: Overview of a hybrid FPGA SoC. Hard CPU core(s) and programmable FPGA fabric are integrated on the same physical structure (Die). A high-bandwidth, low-latency connection infrastructure allows data exchange between the two sides. Both sides have access to dedicated I/O resources.

verification, sub-steps must be performed to ensure the correctness of the design flow.

2.2.1 Design phase

During the design phase, a set of logic modules in the form of HDL (hardware description language) sources and/or intellectual property (IP) packages are assembled together to form a complete design. Once assembled, the design must be validated.

2.2.2 Synthesis phase

In the synthesis phase, the set of sources specified in the design phase are "compiled" into a set gate-level netlists. A netlist is a description of the connectivity between logical elements. In addition to the gate-level netlists, the synthesis tool may also provide a representation of the netlists in terms of logic elements optimized to the specific target architecture.

2.2.3 Implementation phase

The implementation phase maps the output products of the synthesis phase to the physical resources of the FPGA. The main sub-phase of the implementation is usually called "place and route". In such sub-phase the netlists elements are mapped to physical logic resources located in specific positions of the FPGA and interconnected trough the routing resources.

2.2.4 Configuration bistreams

Once the design has been implemented to a specific device, the resulting implementation must be converted to a binary representation that can be used to program the FPGA. Such a binary representation is called a *bitstream*.

When the configuration bitstream has been generated it can be transferred to the FPGA trough a programming interface (usually JTAG) to deploy the implemented design. Since the configuration is stored in volatile SRAM cells, the FPGA does not retain the configuration when power is removed. For such a reason, to allow in the field deployment, the configuration bitstreams can be stored in an non volatile (usually Flash) memory paired with the FPGA. In this way, the FPGA can perform a self-configuration every time the device is power cycled, during the boot process.

2.3 Dynamic partial reconfiguration

Dynamic partial reconfiguration is the ability to dynamically (re)configure a subset of logic blocks included in the FPGA while the remaining blocks continue to operate without interruption [4] [5].

2.3.1 Structure of a reconfigurable design

A set of FPGA blocks that can be reconfigured dynamically is generically referred as *reconfigurable partition*. A reconfigurable partition can host one *reconfigurable module* from a set of reconfigurable modules associated with the reconfigurable portion. Each module in the set can implement a different functionality. The remainder of the FPGA, that is not subject to partial reconfiguration, is usually referred as *static region*. Figure 2.6 shows an example of an FPGA design that uses partial reconfiguration. A reconfigurable module implementation is represented by a binary file called *partial bitstream*.

2.3.2 Benefits of partial reconfiguration

The possibility to dynamically reconfigure portions of the FPGA allows to time-multiplex hardware modules dynamically [4]. In this way, the system functionalities can be changed on-demand, at run-time, providing a higher

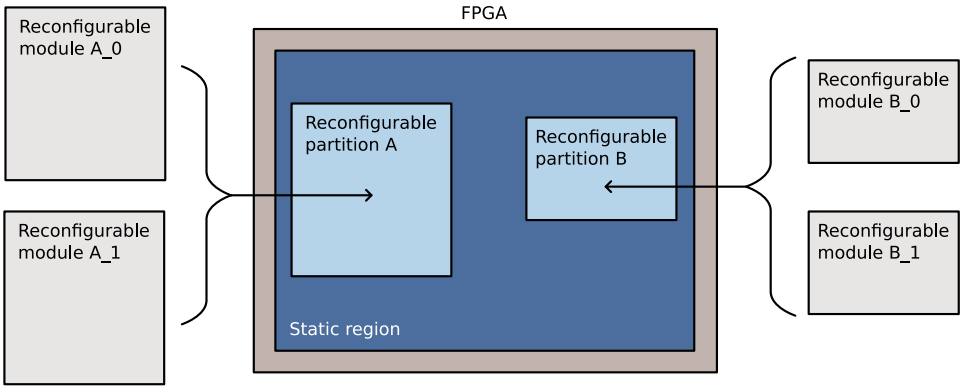


Figure 2.6: Example of an FPGA design that uses partial reconfiguration. Modules A_0 and A_1 share the reconfigurable partition A, while modules B_0 and B_1 are hosted in the reconfigurable partition B.

degree of flexibility in the choices of algorithms and protocols, implemented in harware, available to an application [5].

The total resources required by the set of all modules may exceed the resources available on the FPGA since non all the modules are deployed at the same time as show in Figure 2.6 and 2.7. This flexibility allows optimizing resource consumption, therefore area and power requirements.

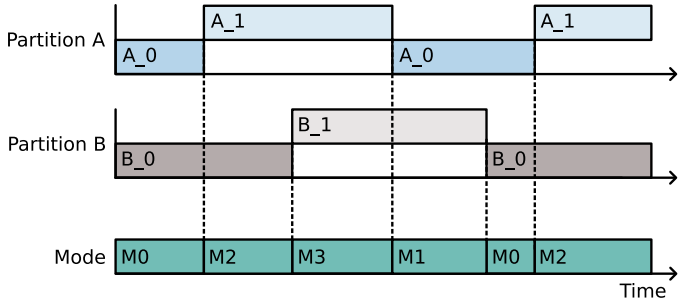


Figure 2.7: Partial reconfiguration allows to time-multiplex hardware modules dynamically. Each configuration is defined as a system mode. Reconfigurations overheads are not shown in the figure.

2.3.3 Autonomous reconfiguration

In addition to the external configuration port, modern FPGAs include an internal reconfiguration port that allows the FPGA to reconfigure portions of its own fabric. The internal configuration port controller can be driven by an internal soft-core microprocessor implemented in the static part. The

processor fetches partial bistreams from an external memory and triggers partial reconfiguration processes trough the internal reconfiguration port.

In hybrid FPGA SoC platforms the internal configuration port logic can be implemented on silicon, as a peripheral of the processor hard cores. In this way the reconfiguration process can be managed by the hard cores without wasting logic blocks.

2.3.4 Common applications of partial reconfigurations

Typical scenarios where an application may benefit from partial reconfiguration range from network applications, where multiple modules can be used to implement different protocols, to cryptographic applications where a set of cryptographic modules can be changed on-demand to extend the system functionalities [5].

Other classical examples of applications that can take advantage from partial reconfiguration are those related to *software defined radio* (SDR), where different types of baseband processing can be performed by a set of different modules, and image/video processing applications, where each module can implement a different video filter.

2.4 Reconfigurable computing

A reconfigurable computer is a device that include a reconfigurable logic device like a reconfigurable FPGA [4]. A reconfigurable computer differs from a standard computer is the sense that is able to dynamically make significant changes in the hardware *datapaths*. A datapath is defined as a set of *functional units* that perform data processing operations. In other words a reconfigurable computer features the ability to adapt its own hardware structure to the data that needs to be processed.

2.4.1 Taxonomy

From a more general perspective reconfigurable systems can be considered, following Tredennick's classification, repored in Table 2.1, as an evolution of the "classical" Von Neumann programmable computer model. In reconfigurable systems the sources that are used to configure the hardware resources are usually called *configware* [6]. The Tredennick's classification should be

Computer type	Algorithms	Resources
Hard wired	*fixed*	*fixed*
Von Neumann	*configurable*	*fixed*
Reconfigurable	*configurable*	*configurable*

Table 2.1: Tredennick's classification

intended as a general taxonomy that classifies different abstract systems models. Real hardware devices are often heterogeneous systems that may implement more than one paradigm.

2.4.2 Reconfigurable devices classification

Reconfigurable computing architectures can be classified based on the granularity [4] offered by the reconfigurable device: *fine-grained* devices allow to reconfigure elements that operate at bit-level, while *coarse-grained* devices feature the ability to reconfigure large logic blocks, like ALUs.

Modern FPGAs feature a heterogeneous fabric where rows of special resources like DSPs are interleaved with logic blocks, therefore they can be classified as *mixed-grain* devices. CPUs are not considered reconfigurable devices since the instruction stream can cause only relatively small changes in a set of fixed datapaths [4].

2.5 Software support for reconfigurable devices

The development process for an application that wants to exploit hardware acceleration dynamically, on a reconfigurable FPGA platform, is complicated and often inefficient. The lack of standard interfaces and device abstractions tie the application structure, and its development, to a specific platform and the related tools.

These issue can be overcome with the introduction of a software support for the reconfigurable device. The software support hides the complexity of the reconfigurable device, providing the developers a set of standard API (application programming interface) to ease the development process.

The problem of software support and system modeling for reconfigurable platforms has been investigated from different perspectives: on one side reconfigurable platforms have been modeled and analyzed from a theoretical

perspective to guarantee real-time predictability. On the other side research efforts have been dedicated to the development of *reconfigurable operating system* (ROS) on real hardware platforms. A ROS is an operating system augmented with a support for reconfigurable hardware.

In both categories the proposed approaches are quite heterogeneous, due to the devices evolution and the lack of dominant solutions. A possible taxonomy to classify the various approaches can be based on the following features:

- *Reconfiguration approach.* In the *mode-level* approach the reconfiguration event can be triggered by a change in the task-set, resulting from a change in the application mode. In the *job-level* approach each instance (job) of a task can trigger a device reconfiguration. In practice optimization techniques used in real implementations may blur this distinction but the differentiation remains valid to classify the proposed solutions.

- *Allocation method.* In the *slotted* approach the reconfigurable region of the FPGA is partitioned into a set of slots. The slots are interconnected by a bus or a network on chip communication infrastructure, that usually resides in the static part of the system. Typically each slot can accommodate a single reconfigurable module. More advanced solutions allow placing reconfigurable modules in more than one slot [7] [8]. On the other hand, in the *slotless* approach, communication channels are allocated dynamically or emulated trough the reconfiguration port. Therefore reconfigurable modules can be allocated with less constraints.

2.5.1 Theoretical works

Theoretical works are focused on the modeling and analysis of reconfigurable platforms to provide real-time guarantee. The system is usually modeled as a set of hardware activities executed on a reconfigurable device. Some approaches also comprehends software activities running on a processor. The following list summarizes relevant contributions in the real-time analysis of reconfigurable systems:

- Danne and Platzner [9] proposed two algorithms (one EDF-based and one server-based) for preemptive scheduling of hardware activities on a

reconfigurable device. The model adopted the allocation is quite simple and does not consider allocation constraints.

- Pellizzoni and Caccamo [10] proposed an optimization method to dynamically distribute a set of computational activities between a processor and a set of reconfigurable slots. Each activity is available either as software and hardware implementation.

- Recently, Saha et. al. [11] presented a new scheduling algorithm for preemptable hardware activities. The approach exploits the higher speed and the improved capabilities of modern reconfiguration interfaces to dynamically change the allocation every time a task terminates.

Unfortunately most of the theoretical works have a limited applicability to real hardware platforms, due to the non realistic assumption on reconfigurable modules allocation constraints, reconfiguration overheads and communication mechanisms.

2.5.2 Reconfigurable operating systems

Reconfigurable operating systems aim at creating a uniform system environment for hardware and software activities. In most of the proposed solutions, hardware actives are wrapped in software containers and integrated in the operating system software environment. Recent works in the field of reconfigurable operating systems are ReconOS [12] and R3TOS [13].

- *ReconOS* extends the classic multi-threading programming model to hardware activities executed on a reconfigurable device. "Hardware threads" interact with software threads trough a custom developed POSIX-style API, using the same operating system mechanisms, like semaphore, condition variables and message queues. Hardware threads are allocated on the reconfigurable device using a slotted approach.

- *R3TOS* wraps reconfigurable hardware with a Free-RTOS based microkernel to create a uniform hardware-software environment. R3TOS differs form other solutions for the novel approach to allocation and communication problems. Reconfigurable hardware resources are used either for computation or to establish dynamic communication channels. The lack of static communication infrastructures removes some placing constraints allowing a slot-less allocation approach.

2.5.3 Contribution of this work

The reconfigurable operating systems presented above are complete solution but they are focused on improving the average system performance rather then guaranteeing worst-case response times. Rather, this thesis addresses the development of a system prototype built to guarantee *by design* predictable execution times and bounded delays, enabling for the development, as a future work, of a response time analysis.

Chapter 3

Platform description

Contents

3.1 Zynq System On a Chip

This section presents an overview on the internal architecture of the heterogeneous platform underling the system developed in this thesis, the Xilinx Zynq system on a chip (SoC). The Zynq is a hybrid FPGA SoC device that includes a software programmable high-performance ARM processor coupled with a hardware programmable FPGA fabric.

3.1.1 Zynq internal architecture

The internal structure of the Zynq SoC can be divided in two main functional blocks referred, in the Xilinx's terminology, as: *Programmable system* (PS) and *Programmable logic* (PL) [14]. The processing system block includes two ARM cores, while the programmable logic block contains the FPGA programmable fabric. A simplified representation of the Zynq internal architecture is illustrated in Figure 3.1.

3.1.2 Programmable System

Internally, the processing system side of the device is composed by the following functional blocks [14]:

- Application processor unit (APU);
- Memory interfaces;
- I/O peripherals (IOP);
- Interconnect.

Application processor unit

The main component of the APU is a dual-core ARM Cortex-A9 processor. The processor implement the ARM v7-A instruction set architecture and can execute ARM and Thumb instructions. Internally, each A9 core is associated with a NEON coprocessor unit, a memory management unit (MMU) and a L1 cache memory divided in two 32 KB sections for data and instructions. The NEON coprocessor extends the instruction set with SIMD (single instructions, multiple data) instructions targeted for 3D graphics, image, audio and video processing.

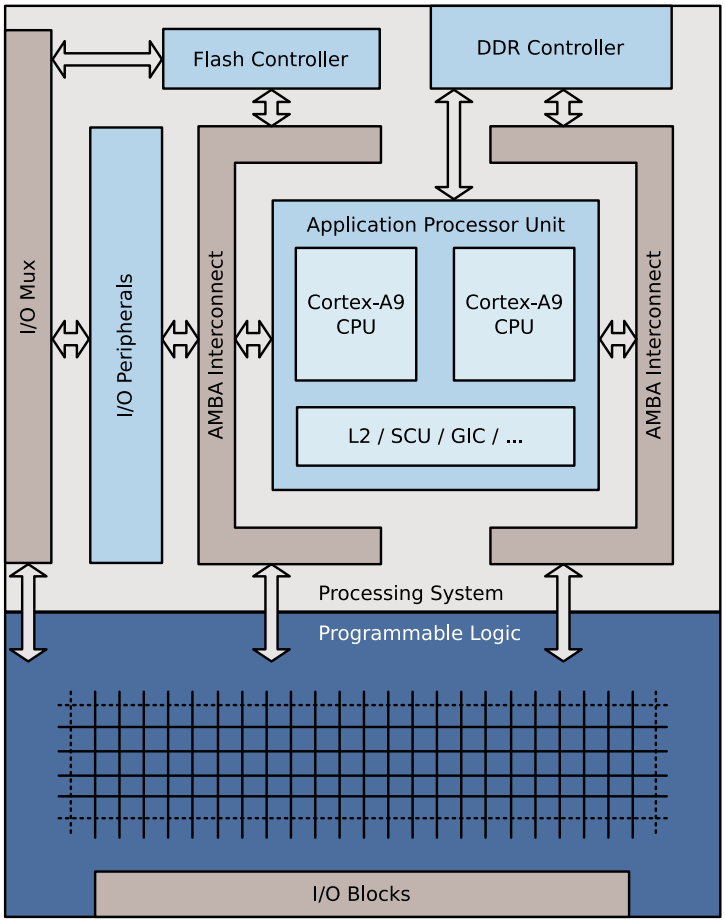


Figure 3.1: Simplified representation of the Zynq SoC internal architecture. Internal components of the Processing system and the Programmable Logic fabric are connected trough ARM AMBA interconnect.

The two ARM cores are connected with a 512 KB L2 8-way set-associative unified cache memory trough a *snoop control unit* (SCU). The SCU ensures the coherency between data caches. The cores are also connected to a 256 KB on-chip SRAM memory (OCM) module that provides a low-latency memory close to the processors.

Memory interfaces

The memory interface unit includes controllers for dynamic and static memories. The dynamic memory (DRAM) controller is a multi-protocol controller that supports different double-data rate types (DDR2, DDR3, DDR3L) memories. The static memory controller supports NAND flash, Quad-SPI flash and parallel NOR flash interfaces.

The DRAM memory controller is multi-ported to allow uniform memory access from the PS and the PL sides. The controller features four AXI slave ports [14]:

- The ARM processor cores can access the memory, trough the L2 cache, from a dedicated 64-bit port;
- Memory access from programmable logic is ensured by two dedicated 64-bit ports;
- All other masters shares a 64-bit dedicated port trough the central interconnect.

Input/Output peripherals

The I/O subsystem includes a wide range of industry-standard interfaces that allow to control, and to communicate with, external devices. The peripherals can access the physical pins of the processing system trough a multiplexer. The pins mapping is controlled by a configuration register of the MIO (multiplexed input/output) control module.

Although the I/O subsystem is part of the processing system the peripherals connections can be routed to the programmable logic to access the I/O resources included in the programmable logic side. This feature is called EMIO (extendable multiplexed input/output).

The following list summarize the peripherals included in the I/O subsystem:

- Up to 54 GPIO (general purpose input/output) signals routable to the physical pins trough MIO. 192 GPIO signals shared between the PS and PL trough EMIO;
- Two Gigabit Ethernet controllers;
- Two USB 2.0 high speed, dual-role, controllers that can operate as host or device;
- Two SD/SDIO (Secure Digital) cards controller;
- Two SPI (Serial Peripheral Interface) master/slave controllers;
- Two I2C (Inter-Integrated Circuit) controllers;
- Two CAN (Controller Area Network) Controllers;
- Two UART (Universal Asynchronous Receiver/Transmitter) controllers.

Interconnect

The functional blocks included in the programmable systems are connected to each other, and to the programmble logic, trough ARM AMBA AXI (Advanced eXtensible Interface) interconnect. The AMBA interconnect supports multiple simultaneous master-slave transactions [14]. The interfacing between the programmable system and programmable logic sides will be discussed later.

3.1.3 Programmable Logic

The programmable logic side of the Zynq SoC is based on the Xilinx 7 Series (Artix-7 or Kintex-7) FPGA fabric. The 7 Series FPGA fabric is an heterogeneous structure where the main logic resources, configurable logic blocks (CLBs), are interleaved with specialized blocks. The main types of logic blocks included insides the programmable logic are:

- Configurable logic blocks (CLBs) are the main resources for implementing logic or distributed memory. Each CLB is composed by two slices. Each slice contains four 6-input look-up tables (LUTs), eight flip-flops and other logic. Each 6-input can be configured as two 5-input LUTs [1][3].

- Block RAMs (BRAMs) are specialized blocks used for implementing dense memory storage. RAM blocks allow to implement random access storages effectively without wasting generic logic resources. Each RAM block is dual-ported and can store up to 36 Kb of data. A single RAM block can also be configured as two independent 18 Kb RAM memories. More RAM blocks can be combined together to form a large memory [3].

- Digital signal processing (DSP48E1) slices are specializied unit optimized to perform arithmetic operations. The units are targeted to signal processing and computational intensive applications in general. The DSP includes a 25x18 two's complement multiplier/accumulator and a 48-bit adder/accumulator and can be programmed to perform different computations [14]. It can be also configured in SIMD mode where it is capable to perform 2 or 4 operations on shorter operands [3].

- Input/Output blocks (IOBs) are the interfaces between the programmable logic resources and the physical pins. Each programmable IOB handles 1 bit as input or output and is compatible with voltage levels ranging from 1.2 V to 3.3 V [14].

3.2 Interconnection between processing system and programmable logic

The main interface for data exchange between the PS and PL sides of the device is implemented trough a set of AXI interfaces. Before introducing the structure of the interconnection the following sections provides a brief introduction to the AXI protocols.

3.2.1 AMBA AXI standard

Advanced Microcontroller Bus Architecture (AMBA) is a family of open standard, on-chip interconnect specification for the connection and integration of functional blocks in SoC platforms [15]. AMBA provides a standard set of standard interfaces that facilitates the integration and re-use of intellectual proprieties reducing development timescales and costs.

Advanced eXtensible Interface (AXI) standard is an interconnection standard, part of the AMBA 3.0 specifications, targeted for hi-performance

systems. The last version of the standard AXI4 is part of the AMBA 4.0 specifications. The AXI standard provides two classes of interconnections: *stream* and *memory mapped*.

AXI stream interconnections

AXI stream interconnection provide a point-to-point high-speed unidirectional link between a master and a slave interfaces. The endpoint that implements the master interface is the data producer while the endpoint that includes the slave interface is the data consumer. In the simplest form an AXI stream link comprises a set of data signals, usually 32-bit wide, and two handshake signals: *valid* and *ready*. The master controls the valid signal and assert it when new data are available. The slave applies a back pressure trough the ready signal to notify when it is ready to consume a new data. A data transfer takes place when both signals, valid and ready, are asserted true.

AXI memory mapped interconnections

Memory mapped AXI extends the protocol with the concept of memory addresses. An interconnection is memory mapped in the sense that the addresses specified in the transactions are mapped in the global memory space of the system. A memory mapped AXI (AXI for simplicity) link provides a bidirectional connection between a master and a slave interfaces. The data can be transfer as single beat or bursts. Read and write transactions are initiated by the master to read or write data to the slave. An AXI link comprises five independent channels: two channels are involved in read requests, the remaining three are involved in write requests:

- *Read channels:*
  - Read address channel: used by the master to send the address to the slave for a read request;
  - Read data channel: used by the slave to send data to the master in response to a read request;
- *Write channels:*

- Write address channel: used by the master to send the address to the slave for a write request;
- Write data channel: used by the master to send data to the slave in a write transaction;
- Write response channel: used by the slave to inform the master if a write transaction was successful.

Each channel comprises two *valid* and *ready* handshake signals. Separate channels for address and data, reads and writes, allow simultaneous bidirectional data transfer [16]. Multiple AXI master and slaves interfaces can be connected using a structure called AXI interconnect block. The interconnect routes the traffic between the master and the slave interfaces and performs the appropriate conversion if the interfaces use different configurations or versions of the standard.

The latest version of the AXI standard, AXI4, allows burst transfer of up to 256 data words. The word size can be configured up to 1024 bits. In the Xilinx's implementation word sizes from 32 to 256 bits are supported [16]. The standard also includes a simplified version of the protocol, named AXI4-Lite, tailored for those applications that require transferring only small amount of data in single beat transactions, like accessing the control registers exported by a peripheral. Such a simplified version does not support burst transactions and the data width is limited to 32 bits. Compared to regular AXI, AXI4-Lite is simpler and its implementation consumes a smaller amount of logic and routing resources [17].

3.2.2 Interconnection structure

The main interconnection between the processing system and the programmable logic comprises a set of nine AXI interfaces exported by the processing system side to the programmable logic side, as shown in Figure 3.2. Those interfaces can be used by custom peripherals and computational accelerators, deployed in the programmable logic, to access the system AXI interconnect. In this way custom modules that implement AXI interfaces can be seamlessly integrated in the system. The interfaces exported by the processing system can be classified accordingly to the mode and the interface type, as summarized in Table 3.1.

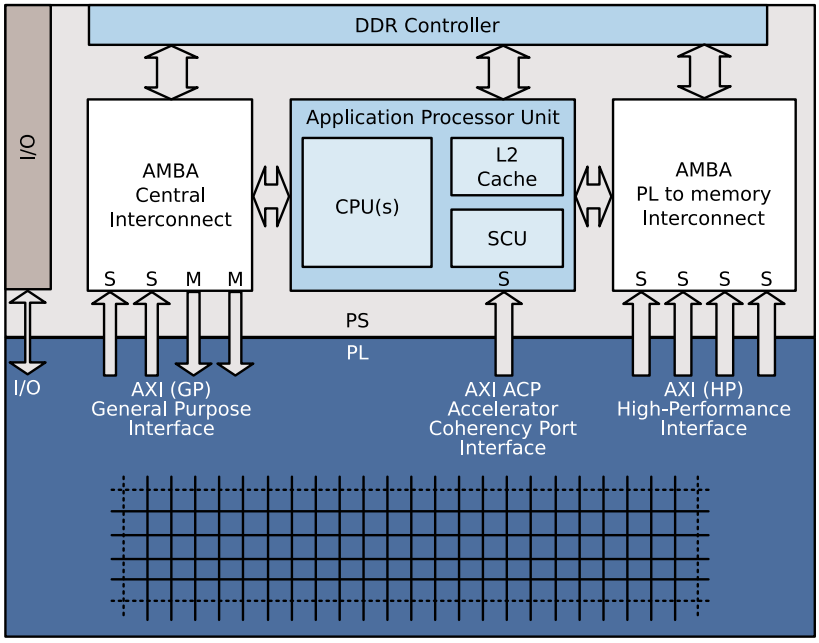


Figure 3.2: AXI interfaces exported by the processing system to the programmable logic. The direction of the arrows points from the master towards the slave, to empathize that transactions are initiated by the master.

Mode	Name	Description
Slave	`M_AXI_GP0`	General Purpose (AXI_GP)
	`M_AXI_GP1`	
Master	`S_AXI_GP0`	
	`S_AXI_GP0`	
	`S_AXI_ACP`	Accelerator Coherency Port (AXI_ACP)
	`S_AXI_HP0`	High Performance Ports (AXI_HP)
	`S_AXI_HP1`	
	`S_AXI_HP2`	
	`S_AXI_HP3`	

Table 3.1: AXI interfaces exported by the processing system side to the programmable logic side.

The slave interfaces allow modules deployed on the programmable logic, which implements master interfaces, to access the global address space where the physical DRAM memory is mapped. In this way, modules deployed on the programmable logic can share the same memory with the ARM cores included in the processing system. The different types of AXI interfaces are designed to fulfill different roles:

- *General Purpose AXI interfaces:* suited for low and medium rate data transfer and modules control. The interfaces supports 32-bit wide data transfer and are connected directly to the ports of the "master interconnect" and the "slave interconnect" that route the transaction to "central interconnect" in the processing system [14].

- *High Performance AXI interfaces:* designed for high-speed burst data transfer. Those interfaces provide a high bandwidth channel to access the system memory from the programmable logic. Each interface supports 64-and 32-bit wide data transfer and includes two FIFO buffers for read and write transactions. The interfaces and are connected to the "PL to memory interconnect" that routes the request to two dedicated ports of the DRAM memory controller [14].

- *Accelerator Coherency Port AXI interface:* designed to allow high-speed low-latency cache-coherent access to the system memory. The interface supports 64-bit wide data transfer and it is connected directly to the snoop control unit (SCU) inside the application processing unit (APU) that includes the L2 cache and the two ARM core with their private L1 caches. The cache coherent port should be used with caution since large coherent ACP transfers can cause the thrashing of the cache with severe impact on the processors performance. Also the ACP shares with the APU the same interconnect path to DRAM memory. Therefore memory accesses through the ACP, that requires access down to the DRAM memory, can potentially decrease the ARM cores performances [14].

The choice between accessing system memory trough the high performance interfaces or trough the cache-coherent interface largely depends on the granularity of the data and on the number of master modules. A complete modeling and evaluation of the possible design alternatives goes beyond the scope of this thesis. An experimental evaluation, in terms of performance

Interfaces			bandwidth (MB/s)			
Class	**Type**	**Number**	**Read**	**Write**	**Read+Write**	**Total**
Internal	AXI_GP	2M + 2S	600	600	1200	4800
	AXI_HP	4S	1200	1200	2400	9600
	AXI_ACP	1S	1200	1200	2400	2400
External	DRAM	1	4264	4264	4264	4264

Table 3.2: Theoretical Bandwidth of the AXI interfaces exported by the processing system and the DRAM external memory interface.

and energy efficiency, of the different design alternatives is presented in [18]. Table 3.2 summarizes the maximum theoretical bandwidth of the different types of AXI interfaces exported by the processing system side to the programmable logic side compared to the DRAM external memory controller. The actual throughput for the DRAM memory depends on the type of DDR memory, its specific timings and the arrangement.

3.3 Programmable logic configuration

The programmable logic side of the Zynq device is an FPGA programmable fabric made of heterogeneous logic resources and routing resources. The configuration of such resources is stored in volatile SRAM memory cells. The process of transferring the content of a bitstream file, containing a design, to the SRAM cells is commonly referred as device configuration. On the Zynq device the programmable logic fabric can be configured, and reconfigured, trough three different paths named after the employed configuration controller:

- JTAG: the programmable logic can be configured and reconfigured by the TAP (test access port) controller on the JTAG chain. This path is commonly used for development.
- PCAP (Processor Configuration Access Port): with this path the programmable logic can be configured and reconfigured, under software control, by the processing system trough its device configuration interface (DevC) subsystem. This is the most common path used for partial reconfiguration since it is entirely comprised in the processing system,

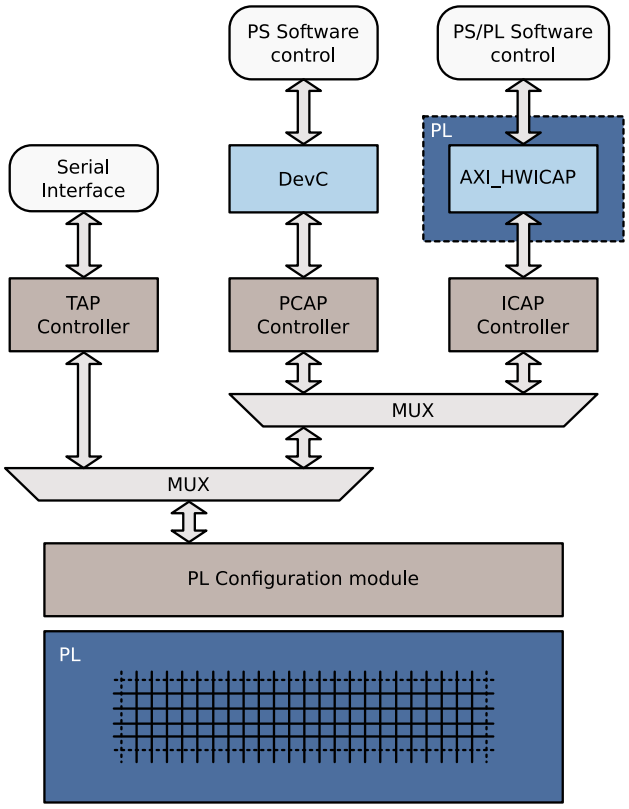


Figure 3.3: Programmable logic configuration paths.

therefore it does not require additional control modules to be deployed on the programmable logic.

- ICAP (Internal Configuration Access Port): this path allows the programmable logic to reconfigure itself autonomously trough a controller module (AXI_HWICAP) deployed in its static region. The controller is usually driven by a soft-core processor, typically a Xilinx MicroBlaze core. This path is not a common option for the Zynq.

At the end of the diffent paths the configuration module processes the bitstream and loads the data into the SRAM configuration cells of the programmale logic [14]. A complete overview of the programmable logic configuration paths is shown in Figure 3.3. The JTAG, PCAP and ICAP configuration paths are mutually exclusive among each other. Switching between the configuration paths should be performed only when all pending transactions are completed [14].

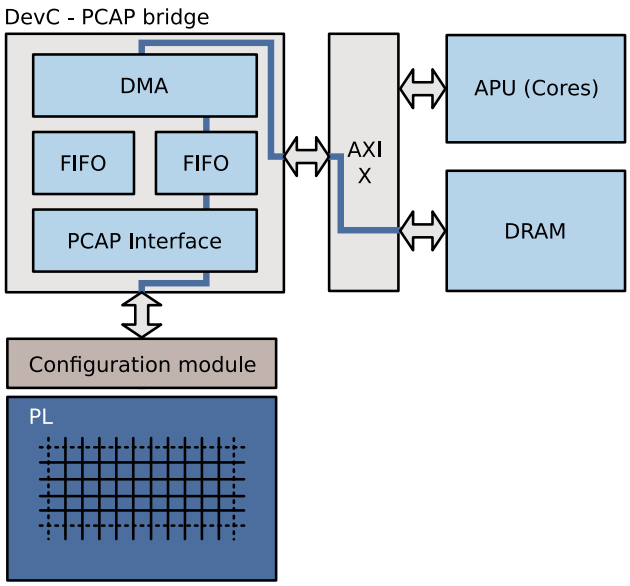


Figure 3.4: Programmable logic reconiguration path trough DevC.

3.3.1 Device configuration interface subsystem

The Device Configuration (DevC) subsystem comprises three main functional sub-blocks: AXI-PCAP bridge, Device Security Management and the XADC interface. The DevC control and status registers are mapped in the system memory space trough an AXI slave control interface connected to the main interconnect.

The AXI-PCAP bridge is the main sub-block of the DevC involved in the programmable logic configuration. It includes the PCAP interface, a DMA engine that accesses the system memory trough an AXI master interface. The DMA engine and the PCAP interface are coupled trough FIFO buffers. Through the DevC control interface the DMA engine can be programmed to transfer bitstreams from the main memory to the PL configuration memory through the PCAP interface. Once both transfers (AXI to FIFO and FIFO to PCAP) are completed, the programmable logic configuration is done and the DevC can notify the processing system by triggering the D_P_DONE_INT interrupt. An overview of a typical PCAP reconfiguration path is shown in Figure 3.4.

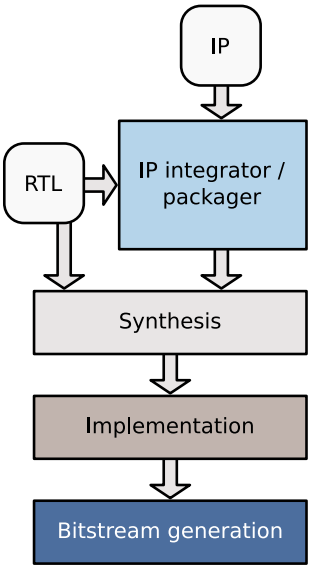


Figure 3.5: Simplified Vivado design flow. The IP integrator environment allows to integrate IP cores at system level

3.4 Design flow and tools

Vivado is the Xilinx's design suite for hardware and embedded software co-development. Vivado extends the traditional RTL (register transfer level) hardware description to device programming design flow, focusing on higher level system integration. Providing, on top of the traditional design flow, an intellectual property (IP) centric, block-based environment where IPs can be instantiated, configured and connected. Intellectual proprieties are reusable design units that can be integrated in a hardware design similarly to the way in witch software libraries can be integrated in a software design. IP cores can be designed by the user or imported from the internal Xilinx library or third party libraries. Figure 3.5 provides a schematization that highlights the main steps of a Vivado design flow.

3.4.1 System on a chip design flow

With the Vivado suite the SoC design flow can be divided in two phases: hardware development and software development. The hardware development phase involves the design an integration of custom peripherals or hardware accelerators to be deployed in the programmable logic. One of the key point in this phase is the integration between the custom logic and the processing

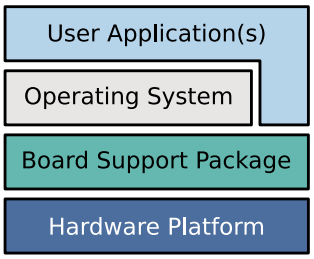


Figure 3.6: Software stack for a Vivado hardware design.

system. Inside the Vivado IP integrator the processing system (comprising the ARM hard cores) appears as a customizable IP block exporting AXI interfaces and other ports. The interconnections between this special block and the rest of the system represent the interconnections between the processing system and the programmable logic.

Once the hardware configuration has been defined, the system can be exported to the Vivado SDK (software development kit) environment to initiate the software development. For a given hardware design, the basic component exported to the SDK is the *hardware platform* representing the customized hardware design. The hardware platform contains software definitions for the registers addresses, interrupt signals, etc, and low-level control functions for peripherals and hardware accelerators.

On top of the base platform there is the *board support package* (BSP) software layer. The BSP includes a set of drivers and support functions for the internal peripherals comprised in processing system. The BSP can be customized depending on the user requirements and the upper software layers.

The operating systems runs above the BSP. During the definition of the software stack the user can choose one of the supported operating systems: Linux, FreeRTOS and others. If the operating system layer is not required, the user can choose the *bare metal* option. Figure 3.6 presents an overview of such a software stack.

3.4.2 High-level synthesis

High-level synthesis is a hardware design process that takes as input a high-level algorithmic description of a behavior and generates, as output, a hardware level description that implements such a behavior. The Vivado

suite includes the Vivado High-Level synthesis (HLS) tool. Vivado HLS transforms an high-level C/C++ behavioral description into an RTL level Verilog and/or VHDL implementation. After the high-level synthesis process, the resulting designs can be exported as IPs core package, in the IP-XACT standard format. Such IPs can be imported in Vivado and integrated into a system design, as shown in Figure 3.5. The Vivado high-level synthesis process comprises two aspects: algorithm synthesis and interface synthesis.

Algorithm Synthesis

In the algorithm synthesis process the behavior specified by the user with a C/C++ description is translated into an RTL implementation. The process is divided in two sub-phases: scheduling and binding. In the scheduling phase the operations that compose the algorithm are distributed in time among clock cycles. In the binding phase the operations are associated with physical resources on the target device like LUTs, DSP48, etc.

Trough a set of directives the user can control the algorithm synthesis process. For instance, such directives can control the actual level of parallelism of the implementation, and the preferred type of resources used to store a specific variable. Trough those directives the HLS programmer can perform an iterative exploration of design space, evaluating aspects like the trade-off between resources consumption and degree of parallelism.

Interface synthesis

In the interface synthesis process the interface of the hardware design is inferred form the arguments of the top level function of the C/C++ description. For each argument of the top function HLS synthesizes a default interface block depending on the type of the argument. The default mapping rules are specified inside the HLS user manual [19]. The user can override the default mapping by manually specifying, for each argument, the type interface block trough a set of directives.

3.4.3 Partial reconfiguration design flow

The partial reconfiguration flow is an advanced feature of the Vivado suite, currently supported only in non-project mode through *Tcl* commands in

interactive shell or batch modes. Following the Xilinx's partial reconfiguration guide [5] the design flow can be summarized in the following steps:

1. Synthesize the static part and the reconfigurable module separately.
2. For each reconfigurable region define a physical area constraint (Pblock).
3. Set the propriety `HD.RECONFIGURABLE` on each reconfigurable partition.
4. Implement a complete design comprising the static region and one reconfigurable module for each partition.
5. Save a design checkpoint for the routed design.
6. Remove the reconfigurable module from the design and save a static only checkpoint.
7. Lock the static placement and routing.
8. For each reconfigurable module: add the module to the static design, do the implementation and save the obtained configuration.
9. For each configuration: verify the configuration by running the partial reconfiguration design rules check, and generate the bitrstream in the required format.

Fortunately, this sequence of operations can be partially automatized trough a partial reconfiguration reference Tcl script provided by Xilinx. The partial reconfiguration flow used in this thesis work is based on the one used in the Xilinx's application note XAPP1231 [20]. To ease the system development process the static part of the design can be designed and synthesized through the Vivado IP integrator environment. Once the design of the static part has been completed, the following steps must be followed to generate the input files for the partial reconfiguration script:

1. Build a design that includes the static part an one hardware accelerator (reconfigurable module) for each slot (reconfigurable partition).
2. Set the `black_box` attribute on each reconfigurable module to crave out the implementation.

3. Run the synthesis and save the resulting synthesized design as a checkpoint.

4. For each reconfigurable module in the netlist view of the synthesized design, define the physical layout of the reconfigurable partition (slot) by drawing a Pblock on the FPGA area representation.

5. For each Pblock set the proprieties `RESET_AFTER_RECONFIG` and `HD.RECONFIGURABLE` and save the resulting constraint file.

Once the static part has been synthesized and the reconfigurable partitions (slots) has been floorplanned, the resulting output files, together with the hardware accelerators source IPs, can be imported into the scripted environment. The scripted flow can be summarized as follows:

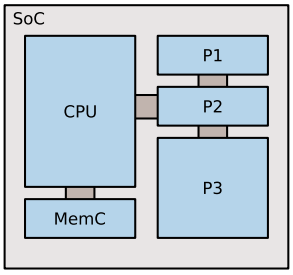
1. Import the static design synthesis checkpoint.

2. Define each hardware accelerator as a reconfigurable module. Each module will be synthesized out-of-context, without the static design part.

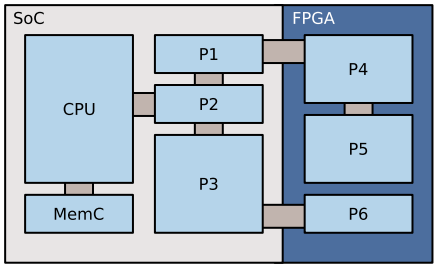
3. For each hardware accelerator define a configuration for the implementation phase. The configuration includes the static part and the specific hardware accelerator (reconfigurable module) instantiated in each reconfigurable partition (slot). The set of defined configurations will be implemented. Actually the static part will be implemented only once for the first configuration and then imported by the following configurations.

4. Once the configurations have been successfully implemented each configuration will be verified trough the partial reconfiguration design rules check. If the test is successful, full and partial bitstreams will be generated. The partial bitstreams will be further converted to the binary PCAP format.

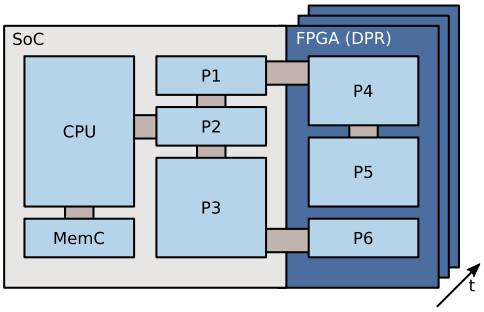
At the end of the process, for each hardware accelerator, a full bistream and a set of partial bistreams, one for each slot, will be generated.



(a) System on a chip comprising a CPU, a memory controller (MemC), and a set of peripherals (P).



(b) Heterogeneous FPGA - SoC. The system includes custom peripherals deployed on the FPGA.



(c) Heterogeneous, dynamic partial reconfiguration enabled, FPGA - SoC. Through dynamic partial reconfiguration the set of peripherals deployed on the FPGA can be changed at runtime.

Figure 3.7: Comparison of traditional SoC architecture, comprising a processor (CPU) and a set of peripherals (P) against an heterogeneous FPGA - SoC extensible architecture. Dynamic partial reconfiguration allows to change the set of modules programmed on the FPGA at runtime, extending the configurability in time domain.

3.5 Heterogeneous FPGA SoC Architecture

Traditional SoC devices comprise one or more main processors and a set of peripherals, controllers and computational accelerators, connected through an interconnect, and integrated on the same chip, as shown in Figure 3.7a. In heterogeneous FPGA SoC, like the Zynq, the programmable logic FPGA fabric can be used as a "canvas" to deploy custom peripherals and hardware accelerators that extends the capabilities of the processors and the peripherals built in the system, as shown in Figure 3.7b. Custom hardware modules can be seamlessly integrated in the system through the AXI interconnect. Such a tight coupling ensures that the FPGA fabric can be effectively used for the deployment of hi-performance hardware accelerators.

From a certain perspective the Zynq SoC can be considered an extensible SoC device. The possibility to reconfigure portions of the programmable logic fabric, while the system is running, extends this flexibility by another degree of freedom, allowing to change dynamically, in the time domain, the set of hardware accelerators modules deployed in the programmable logic.

3.5.1 Hardware accelerator classification

The way in which an accelerator module interacts with the main processor and other modules largely depends on the type of interface trough which it is connected to the rest of the system. The class of interface also has a large impact on the internal structure of the module. Following this approach hardware accelerator modules can be classified, according to the interface adopted, as follows:

- Custom (Non AXI) accelerators: this class includes all kind of accelerator modules that use custom buses, network on chip or point to point links to communicate among themselves and with the rest of the system. The main reason for developing a custom communication infrastructure is to give support for important features that are currently not supported by vendor's standard tools, using AXI interconnect, like bitstreams relocation, multi-slot allocation, floating placement, etc. The main drawbacks of those approaches are: higher complexity, limited portability and dependence upon custom third-party tools that often are device specific. Another drawback is that the performance of such

custom solutions are not guaranteed and need to be evaluated against the standard interconnect. For this reason those approaches were not considered in this thesis.

- AXI based accelerators: this class includes all accelerators modules that rely on the standard interconnect to communicate with the rest of the system. The main benefits of this approach are: support from the vendor's standard tools, platform independence and guaranteed performance, since it relies on the native interconnect used by all other modules in the system. Another benefit is the design portability, since AXI is an open industry-standard defined by ARM and adopted by both leading FPGAs manufacturers: Xilinx and Altera. The main drawbacks comes from the fact that desirable features, like bitstreams relocation or multi-slot allocation, are currently not supported by the vendor's standard tools. Third-party and research tools that support such features, do not rely on the AXI interconnect but use custom buses. Therefore, without the support of those advanced features, some compromises have to be made in the design of the reconfigurable system.

The AXI based accelerator modules can be classified as *master* and *slave* modules. Master accelerator modules are able to autonomously retrieve the data they need to process from the system memory, while slave modules should be "feed" with data by the processing system.

3.5.2 AXI based slave accelerators

The typical architecture of this kind of accelerator includes an AXI slave port, used for both control and data transfer, and an interrupt signal to notify the processor. The module can be connected to one of the processing system's general purpose master ports, as shown in Figure 3.8. In this way the module's control and data registers are mapped in the system memory space. If the module requires a larger buffer, registers banks can be implemented with BRAMs elements to increase the storage density. However, even by using BRAMs, the maximum amount of storage available is practically limited to hundreds of KB, depending on the programmable logic total area and on the share allocated to the module.

In order to perform a computation, the processor should load the input data into the slave module by performing a sequence of write operations in

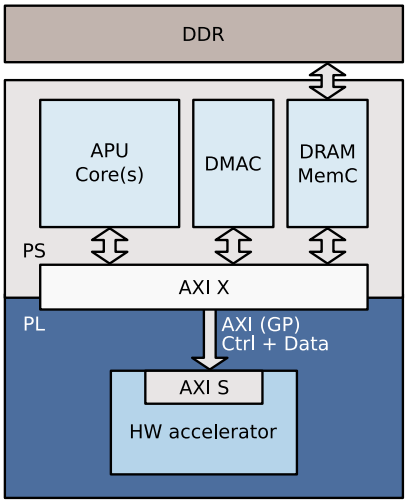


Figure 3.8: AXI Slave accelerator internal structure. The control flow and the data flow are directed by the processing system.

the module's data registers. When the data have been loaded the module can be started by writing in the module's control registers. Once the module has been started the processor must wait until the computation has been completed. After the computation has been completed, the module notifies the processor trough the interrupt signal. At this point the processor can retrieve the processed data by reading again the module's data registers.

Slave accelerators are the simplest type of AXI based accelerator. The main advantage is that the development of the interface part is simpler compared to other solutions. Also the complexity of the control software is reduced since, at the most basic level, it comprises a set of functions for reading an writing registers, as well as an interrupt service routine, if the accelerator will be used in interrupt mode. Also the number of FPGA resources required to implement the slave interface logic is small.

The main drawback of those kind of accelerator is that they are unsuitable for operations that require the processing of large amounts of data, since data transfer is done actively by the processor. If a large chunk of data is required to be transfer, a large number of processor cycles will be wasted, degrading the system performance.

A more sophisticated approach relies on the processing system's internal DMA controller (DMAC) to move data from the system memory to the slave accelerator module. However the relatively small amount of data transferable

per transaction and the increased software complexity reduces the overall versatility of this approach.

To summarize, this approach is unsuitable for stream processing operations that require to transfer and process large amount of data. Such operations represents a large share in the space of operations that FPGAs are required to accelerate. In practice, slave accelerators are limited to very specific applications and do not provide a structure that it generic enough to fit conveniently most of the operations that can be accelerated on the programmable logic.

3.5.3 AXI based master accelerators

Master accelerators features the ability to retrieve and write back data autonomously from the system memory space. From the processor perspective such a kind of accelerators requires only control operations while the data flow path passes trough another independent channel. The main advantage of this architecture is the data access decoupling between the processor and the accelerator. Such a decoupling enables the accelerator to process large chunks of data without affecting the processor.

A further possible classification of master accelerators modules can be made conceptually partitioning the master module structure in two parts: the logic required to access and move the data, and the logic that performs the data processing.

AXI stream accelerators

In this architecture the accelerator module comprises two separated parts: a memory access part and a data processing part, an shown in Figure 3.9. The memory access and data movement operations are typically performed by a companion DMA module that includes an AXI master interface required to access the memory space. Only the data processing part defines the module specific functionality since the DMA is typically implemented with a standard "library" element. The DMA and the data processing part are connected together through one or more AXI stream channels.

From the processing system perspective, in the simplest possible implementation, only the data movement part of the accelerator, the DMA engine, is visible in the memory space trough its control register. In order to perform

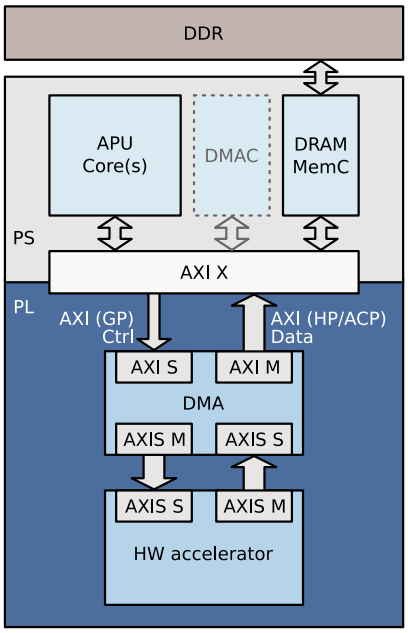


Figure 3.9: AXI Stream accelerator internal structure.

an operation, the accelerator's DMA engine must be programmed by the processing system to specify the area of the memory space where the data that need to be processed reside and the destination area where the results should be written.

The working cycle of a typical stream accelerator can be summarized as follows: first the DMA engine reads the input data from a specified location in the memory space, and transfer it to the accelerator trough an AXI input stream channel. When the accelerator receives the first data block of the input stream from the DMA it starts processing the data. Once the computation has been completed the accelerator sends back the processed data to the DMA trough an AXI stream output channel. As soon as the DMA receives the processed data from the accelerator it will copy them to a specified set of locations in the memory space.

Stream accelerators appears, to the processing system, as peripherals that process sequences of data. As the name suggests, this class of accelerators fits perfectly with stream processing operations that dominate the FPGA processing. However, when the data that need to be processed are organized in complex structures that are allocated in memory in a non-contiguous fashion, the complexity of the accelerator's control software may increase

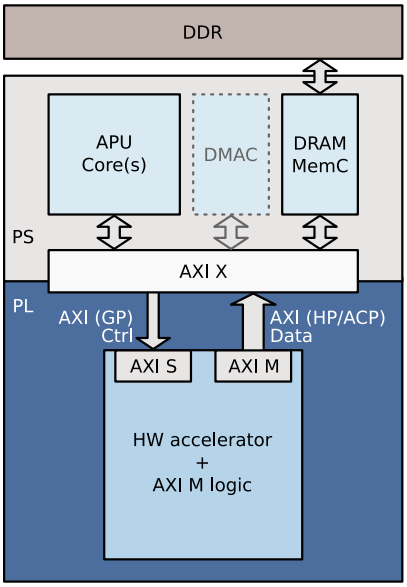


Figure 3.10: AXI Master accelerator internal structure.

significantly, requiring periodic interventions of the processing system to reprogram the DMA. More sophisticate approaches may take advantage from vectored I/O support, referred by Xilinx as scatter/gather mode. However, in this case the whole design complexity increases and the possibility of performing conditional executions inside the accelerator module may still require a dedicated software support.

To summarize, stream accelerators are optimized for stream processing operations, but their specific structure and related programming model may not fit generic co-processing units.

AXI master accelerators

With this architecture the accelerator module is a monolithic structure that comprises both: the full AXI master logic, required to access the memory space, and the computational logic required to process the data. This is the most generic type of hardware accelerator, since its behavior entirely depends on the internal structure.

In practice, if a master module must be controlled by the processing system, it should also implement an AXI slave control interface. In this case the complexity of the related control software largely depends on the module implementation and its behavior.

Actually, this is more a super-class of hardware modules, rather than a class. Also stream accelerators, if the data processing part and the DMA engine are packed together, can be seen as monolithic modules that export the DMA AXI master interface.

The main drawback of this architecture is the increased complexity of the hardware implementation, since the module must include the full master logic required to access the interconnect and move the data. Also, due to its monolithic nature, the data access part and the processing part cannot be separated and should both be placed inside the module structure in the reconfigurable partition.

Chapter 4

System architecture

Contents

4.1 System Description

A real-time software system is composed by a set of computational activities or *tasks* that can be classified as periodic or aperiodic. Periodic tasks consist of an infinite sequence of activities, called *jobs*, that are regularly activated at a constant rate. In a conventional system tasks are executed on a processor in a sequential fashion.

In the system developed in this thesis tasks can accelerate parts of their computations by requesting the execution of accelerated operations on the reconfigurable device included in the heterogeneous system. Each accelerated operation is implemented as a hardware accelerator module that can be dynamically configured on the programmable logic FPGA fabric. When a task request the execution of an accelerated operation, it will be suspended until the request accelerator has been configured on the programmable logic. Therefore, compared to other approaches to real-time reconfigurable computing, in this system each job can request the reconfiguration of a portion of the FPGA.

4.1.1 Platform Parallelism

With respect to a conventional system, where tasks execution is serialized on the processor, the proposed approach can provide relevant speedups on the tasks execution times. The factors that contribute to such speedups depends on the parallelism achievable both at operation level and system level:

- *Operation level parallelism.* The execution time of a single task can be substantially reduced, when part of its computation are performed by a hardware accelerator, due to the high level of parallelism achievable on the FPGA.
- *Platform level parallelism.* Depending on the number of slots, an equal number of hardware accelerator can run concurrently on the FPGA, enhancing the system parallelisms and reducing the computational load on the processor

4.2 System architecture model

From a model perspective the hardware structure of the heterogeneous SoC platform, underlying the system, comprises the following elements:

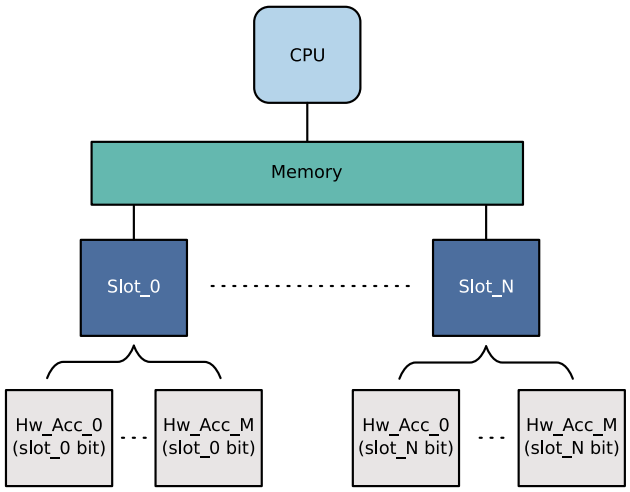


Figure 4.1: System architecture model. The system can be described as a shared memory architecture with dynamically interchangeable co-processors.

- a processor;
- a set of N reconfigurable slots;
- a set of M hardware accelerator modules;
- a shared physical memory.

Each slot can accommodate a hardware accelerator as a reconfigurable module. The processor and the hardware accelerators share and access the same memory space where the physical memory is mapped. Hardware accelerators are able to access the physical memory autonomously to retrieve data. Such shared memory is used to exchange data between the processor and the hardware accelerators. The whole system structure resembles a uniform memory access (UMA) shared memory architecture since the access time does not depend on the location of the data in memory [21]. Figure 4.1 shows a graphical representation of the system architecture.

4.3 Software structure

The software part of the system comprises the following components:

- a set of periodic computational activities (tasks);
- a real-time operating system;

- a support library to extend the operating system with functions to manage partial reconfiguration and the execution of hardware accelerated operations.

4.3.1 Software support library

From the client programmer perspective the concept of hardware acceleration is wrapped in two software entities: a set of objects named *hardware operations* and a *reconfiguration service*. The set of hardware operations represents all the accelerated operations that the user can request to accelerate the execution. The reconfiguration service is the component through which tasks can request the execution of accelerated operations.

Hardware operation

The hardware operation objects wraps the set of data required to define an accelerated operation. Each hardware operation object includes the following proprieties:

- a set of bistreams, one for each slot, containing the configuration data required to reconfigure the hardware accelerator that implements the operation.
- two functions that are used to prepare the input data before the execution on the accelerator, and to transform the output data, after the execution, before being retrieved by the task. Typically, such functions are used to ensure the cache coherency of the data.
- an interface function used by the programmer to specify the pointers to input and output data before requesting the execution of the operation. The data will be stored inside the operation structure.

The internal structure of the hardware operation object will be discussed with greater detail in chapter 5.

Reconfiguration service

The reconfiguration service abstracts the reconfiguration and a hardware acceleration mechanisms. From the client programmer perspective the reconfiguration service software interface consists of a single function that can

be used to request the execution of an accelerated operation. The requested operation is passed to the service as an argument of the function. When the function is called the reconfiguration service use the hardware operation parameter to specialize the hardware structure of the device for the requested operation, through the reconfiguration mechanism. The internal structure of the reconfiguration service will be discussed with greater detail in chapter 5.

4.3.2 Software activities

The computational actives that can be executed on the system consists of a set of periodic tasks. Each task in the task-set is subject to timing constraints, including its execution period, computation time, and relative deadline equal to the period. The tasks are scheduled by the real-time operating system according to a fixed priority scheduling policy. During its execution a task can request the execution of one or more accelerated operations through the reconfiguration service.

Structure of a hardware accelerated task

The structure of a task that requests the execution of a hardware accelerated operation is presented in Listing 4.1. Io order to use an accelerated operation the programmer must define an hardware operation structure in the initialization section the task, as shown at line 7. Consequently the hardware operation structure must then be initiated, as done at line 10.

The body of the tasks can be divided in three sections or chunks. In first chunk, at line 15, the task can perform any software computation, like a regular software block. Before the end of the chunk the task must prepare the input data for the accelerated operation inside a memory buffer. Next the the pointers to input and output data buffers must be set inside the hardware operation structure, through the related function, as done at line 18. At this point everything is ready and, at line 21, the reconfiguration service can be called to reserve the execution of the operation on the reconfigurable device. After the call the task will be suspended.

When the hardware operation has been completed the task can resume its execution starting the second chunk, as shown at line 23. In this final chunk the task can consume the output data produced by the hardware operation and perform any other computation. After the end of the second chunk, at

Listing 4.1: Structure of a hardware accelerated software task.

```
void accelerated_task()
{
  // Task initialization (executed only once)
  << Initialization part >>

  // Define an instance of an accelerated operation
  Hw_Op hardware_op;

  // Initialize the hardware operation
  hw_op_init_optype(hardware_op, "Operation name");

  // Task body
  while (1) {

    << Software elaborations chunk >>

    // Set operation parameters in the hardware operation structure
    hw_op_optype_set_args(hardware_op, input_data, output_data);

    // Call the reconfiguration service to execute the hardware operation
    rcfg_manager_execute_hw_op(hardware_op);

    << Software elaborations chunk >>

    // Wait for the next job
    suspend_unitl(last_wake_time, period);
  }
}
```

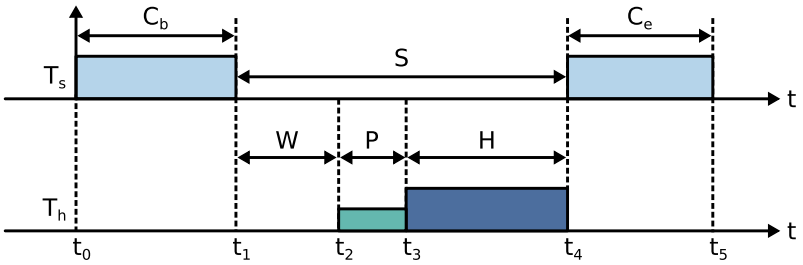


Figure 4.2: Execution model of a software task that requires the execution of an accelerated operation.

line 26, the task will be suspended until the next activation.

Structure of the reconfiguration function

A pseudocode description of the reconfiguration function is shown in Listing 4.2. When the task call the reconfiguration service function, at line 21 of Listing 4.1, for requesting the execution of a hardware operation, the reconfiguration service first check for a vacant slot, at line 4. If all the slots are occupied the calling task will be suspended until one of the slots has been released. When at least one slot is available the function search if any of the vacant slot already contains the accelerator required by the hardware operation, as shown at line 8. If none of the vacant slots contains the required accelerator one of the vacant slot the will be reconfigured with the corresponding bistream of the required accelerator, as done at line 14. The calling task will be suspended until the reconfiguration has been completed. Once the requested accelerator is available, whether it was already allocated or it has been reconfigured, it will be configured and started, at line 19. The calling task will be suspend again until the hardware accelerator has finished its execution. When the computation has been completed, at line 24, the slot will be released.

Execution model of hardware accelerated tasks

The execution of a hardware accelerated task can be modeled accordingly to the structure shown in Figure 4.2. The sequence of operations performed by the task can be summarized as follows:

- At time t_0 the task is activated;

Listing 4.2: Pseudocode the reconfiguration function call. The GOTO statement is used in this description for the sake of simplify. The real implementation does not contain any GOTO statement.

```
rcfg_manager_execute_hw_op(hardware_op);
{
  // Try to take a slot, wait if there isn't one available
  slots_semaphore.wait();

  // Search if one of the free slots already
  // contains the require hardware module
  for (slot : free_slots) {
    if (slot.id == hardware_op.id);
      GOTO execution;
  }

  // Start device reconfiguration
  transfer_bitstream(slot, hardware_op.bitstream);

  << wait until the device has been reconfigured >>

  // Start hardware accelerator
  execution: start_op(slot, hardware_op.args)

  << wait until the the accelerator has finished >>

  // Free the slot
  slots_semaphore.signal();

}
```

- Between time t_0 and time t_1 the task executes software computations and prepare the input data for the accelerated operation;
- At time t_1 the task call the reconfiguration service to request the execution of an accelerated operations;
- If all the slots are busy the task will be suspend until time t_2, when one of the slots becomes available. The time interval $W = [t_1, t_2]$ is the slot contention delay;
- Once a vacant slot is available, the reconfiguration service starts to reconfigure the slot at time t_2;
- At time t_3 the reconfiguration has been completed and the hardware accelerator can be started. The time interval $P = [t_2, t_3]$ is the time required to reconfigure the FPGA. It depends on the size of the bitstream and on the throughput of the reconfiguration port;
- At time t_4 the hardware computation has been completed and the software task can be resumed. The time interval $H = [t_3, t_4]$ is the time required by the hardware accelerator to compute the hardware operation;
- At time t_5 the task concludes its execution.

The total suspension time for a task that requires the execution of an accelerated operation is $S = W + P + H$.

4.4 Reference platform

Before discussing the deployment on the reference platform a brief overview of the different SoCs featuring the Zynq family is presented. Such overview is useful to compare the features of the specific Zynq SoC, used as development platform in this thesis, with the other SoCs members of the family. Consequently the development board used to develop and test the system is presented. Important features provided by the development board are the DDR system memory and the video output support.

Device	Z-**7010**	**Z-7015**	**Z-7020**	**Z-7030**	**Z-7045**	**Z-7100**
ARM Cores Max clock	667 MHz (-1) 766 MHz (-2) 866 MHz (-3)			667 MHz (-1) 800 MHz (-2) 1 GHz (-3)		667 MHz (-1) 800 MHz (-2)
PL type	Artix-7 FPGA			Kintex-7 FPGA		
LUTs	17.6 k	46.2 k	53.2 k	171.9 k	218.6 k	277.4 k
FlipFlops	35.2 k	92.4 k	106.4 k	157.2 k	437.2 k	544.8 k
BRAMs	60 (240 KB)	95 (380 KB)	140 (560 KB)	265 (1060 KB)	545 (2180 KB)	755 (3020 KB)
DSPs	80	160	220	400	900	2020

Table 4.1: Summary of the characteristics of the Zynq-7000 SoCs.

4.4.1 Zynq SoC family

The main difference between the SoCs of the Zynq family is the type and size of the programmable logic fabric. The programmable logic of the smaller Zynq SoCs is based on Artix-7 logic fabric while, for the larger ones, is based on Kintex-7 fabric. Other differentiation factors are the availability of PCI Express ports and hi-speed communication interfaces. The processing system is the same among all the members of the family being that the only difference is the maximum frequency of the ARM cores. Table 4.1 summarizes the features of the Zynq-7000 family SoCs.

4.4.2 ZYBO board

The ZYBO (diminutive of Zynq Board) is an entry level development board built around the Zynq Z-7010 SoC, the smallest device of the Zynq-7000 family. With respect to other popular Zynq boards, like the ZedBoard, the ZYBO does not feature high density I/0 FMC connectors. The main features of the board are:

- ZYNQ XC7Z010-1CLG400C SoC;
- 512 MB of DDR3 memory;
- Ethernet (1Gbit/100Mbit/10Mbit);
- MicroSD slot;
- VGA / HDMI video output;

- On board JTAG programming and UART to USB converter;
- Audio codec.

4.5 Test implementation

To evaluate the performances of the system, and estimate the suitability of the proposed approach to real-time computing, a test implementation of the system has been developed on a ZYBO board. The processor referred in the model is implemented by one of the ARM cores included in the APU inside the processing system of the Zynq Z-7010 SoC. The physical memory is mapped on the DDR3 memory featured on the ZYBO board. The hardware accelerators are deployed on the Zynq programmable logic.

4.5.1 Programmable logic structure

In order to support the deployment of dynamically interchangeable accelerators modules the programmable logic area is divided in two main regions: a static region and a reconfigurable region. The static region contains the static portion of communication infrastructure and other support modules while the reconfigurable region is organized as a set of reconfigurable partitions that implement the slots. Each slot can accommodate a hardware accelerator as a reconfigurable module. A schematization of the programmable logic structure, in a two slots configuration, is provided in Figure 4.3.

For a given set of M hardware accelerators, required to support the related set of M hardware operations, each of the N slots must be able to accommodate every accelerator in the set. Since bitstream relocation is not supported by the Xilinx's standard tools [4][5], each hardware accelerator is implemented as a set of N bitstreams, one for each slot. Therefore the total number of bitstreams is NxM. In this way each slot can accommodate all its specific implementations of each hardware accelerator.

Video output

The static portion of the programmable logic also include a video output path used to control a standard "VGA" (DE-15 connector) compatible monitor. The video output is used to display processed images for test and debug purposes. The video path consist of VDMA (Video DMA) unit and a custom

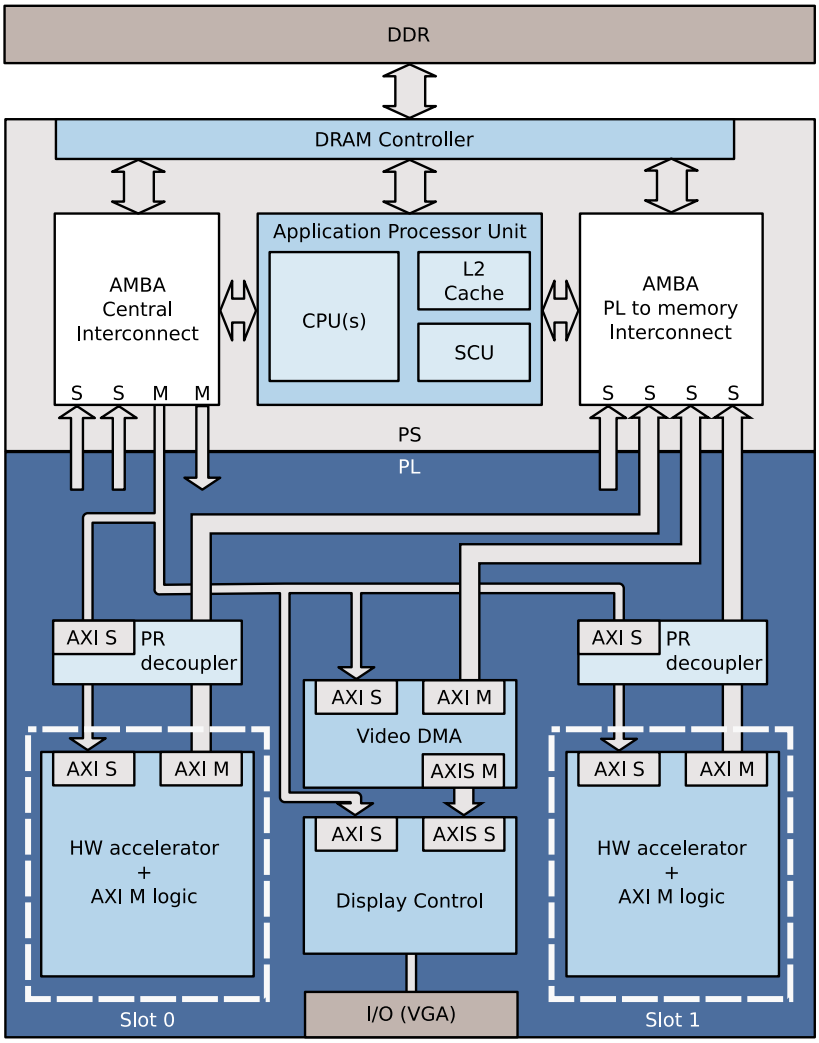


Figure 4.3: Schematization of the programmable logic architecture in a two slots configuration. AXI interconnect blocks and other support blocks are not shown.

display controller IP developed by Digilent. The VDMA engine is connected on one side to the processing system through one of the AXI HP ports, on the other side to the display controller through an AXI stream channel. The VDMA module periodically reads the video data from a memory buffer named *framebuffer* and stream them to the display control module. The display control module receives the video data and generates a set of video output signals with the appropriate timings. The digital signals generated by the display control are converted to analog values through a set of simple resistor ladder converters.

4.5.2 Decoupling logic

During the partial reconfiguration process the reconfigurable modules do not output any valid data until the reconfiguration is completed and the module is reset. Therefore, to avoid spurious transactions, the static part of the system must ignore the signals received from reconfigurable module during the partial reconfiguration process [5]. In this system such decoupling is achieved through the *Partial Reconfiguration Decoupler* IP included in the Vivado library. Each reconfigurable slot is paired with a decopler located in the static part of the system. The decoupler isolates the slot from the rest of the system during the partial reconfiguration process. The decoupler are controlled, through the AXI bus, by the software support library running on the processor inside the processing system.

4.5.3 Hardware accelerator structure

For each slot (reconfigurable partition) all the modules (hardware accelerators) that are required to fit in that slot must export an interface that match the slot's interface [5]. Therefore, to allow each hardware accelerator to be accommodated in every slot, all hardware accelerators must have the same interface.

As discussed in section 3.5.1, the interface of a hardware accelerator has an impact on the internal structure and on the programming model. To support a wide range of operations while maintaining the accelerators structure as generic as possible, without contrasting the computation a specific data processing paradigm, the standard accelerator structure has been defined, following the naming convention adopted in this thesis, as an AXI master

accelerator structure. In this way the processing paradigm and the memory access strategy depends only on the internal structure of the accelerator. With this approach accelerators that implement operations of different classes can coexist wrapped in the same generic structure.

Standard interface description

The standard interface is similar to the one by adopted by Sadri et al. [18]. The interface includes an AXI master interfaces for accessing the entire system memory, an AXI slave interface through which the accelerator can be controlled by the processor and an interrupt signal to notify the processor. In the current implementation the AXI master interfaces exported by the accelerators hosted in the slots are attached to high-performance (HP) ports exported by the processing system, while the AXI slave control interfaces are attached to the AXI master general purpose (GP) ports.

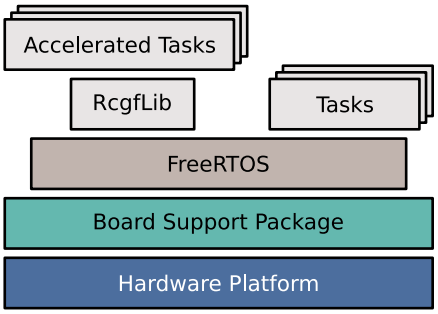


Figure 4.4: System software stack.

4.5.4 Software stack

The software stack of the test implementation comprises the following components:

- A set of tasks and hardware-accelerated tasks;
- A support library enabling hardware acceleration and partial reconfiguration;
- The freeRTOS operating system;
- The software support layer exported by Vivado, including the hardware platform and the board support package.

A graphical representation of the software stack is presented in Figure 4.4. In order to test the implementation a set of four accelerated operations has been developed: three naive implementations of image convolution filters: sobel, blur and sharp, and a simple implementation of a matrix multiplier. The underlying hardware accelerator modules has been developed using Vivado HLS. The implementation details of the hardware accelerators will be discussed in chapter 5

Chapter 5

Implementation details

Contents

5.1 Hardware accelerated operations

To evaluate the performances of FPGA based hardware acceleration on the developed platform, a set of four standard operations has been implemented both in software, as regular functions, and in hardware, as accelerator modules.

In general the computational complexity of an operation depends on the algorithm and on the implementation. Both can be optimized but optimization that performs well on software implementations does not necessary fit well hardware computation. This because, on a real platform, the performances depends both on the operations and on the memory access patterns.

In this work, to be as fair as possible, test operation has been realized using both naive algorithms an implementations. The software versions have been implemented with standard C code while the hardware counterparts have been implemented using Vivado high-level synthesis (HLS) tool.

The set of test operations developed in this work includes: three image filters: a blur filter, a sharpening filter and a sobel edge-enhancement filter, and a matrix multiplier. Before going into the details of the implementation of each operation the following section provides a description of the standard interface that each hardware accelerator implements.

5.1.1 Hardware accelerators interface

As stated in the previous section 4.5.3, in this system every accelerator module uses a standard interface structure that includes:

- An AXI master interface to access the system physical memory mapped in the memory space;
- An AXI slave port through witch the module can be controlled by the processor;
- A interrupt signal to notify the processor.

Vivado HLS synthesize the interface of the hardware module according to the type of the arguments of the top level function. The default mapping rules can be overridden by manually specifying the type of interface for each argument through a set of directives. Listing 5.1 show the HLS interface specification for the accelerator modules used in the system developed for this thesis work.

Listing 5.1: Vivado HLS interface specification.

```
1  void slot_N(volatile args_t *id, volatile data_t *mem_port, volatile args_t
       args[ARGS_SIZE], volatile scope_t *state_out)
2  {
3    /* -------- Interface specification -------- */
4
5  // AXI Lite control bus
6  #pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
7  #pragma HLS INTERFACE s_axilite port=id bundle=CTRL_BUS
8  #pragma HLS INTERFACE s_axilite port=args bundle=CTRL_BUS
9
10 // AXI Master memory port
11 #pragma HLS INTERFACE m_axi port=mem_port depth=512
12
13 // Scope output
14 #pragma HLS INTERFACE ap_none port=state_out
15
16   // Operation implementation
17   hw_mod(id, mem_port, args, state_out);
18 }
```

The top level function `slot_N()` arguments have the following meanings:

- `id` is used to specify a numeric identifier for the accelerator module. It can be accessed by the control software to know what type of hardware accelerator is allocated in a slot. Typically can be used for debug or optimization purposes.

- `args[]` is an array of unsigned/signed elements used by the control software to send control data to the hardware accelerator. Depending on the type of operation, if the amount of data that the accelerator is required to process is huge, the array can be used to pass pointers to the memory locations where the data resides. Otherwise, if the input data set is very small, the array can be used to carry the input data directly, without the need to retrieve the input data from the memory by the accelerator, saving a memory access.

The `id` and `args[]` arguments, including the function return are bundled together in a AXI4-Lite interface as specified by the the directives at lines 6-8 in Listing 5.1. With those directives HLS exports those arguments in a single AXI4-Lite interface. For the function return HLS generates an interrupt signal that will be issued when the execution has completed. When the

accelerator is connected to the processing system the function arguments will be mapped in the memory space through the AXI4-Lite interface. By default Vivado HLS generates a set C functions and addresses defines to access such memory locations.

- `mem_port` is a data pointer through which the internal logic of the hardware accelerator can access the memory space. The directive at line 11 tells Vivado HLS to generate an AXI master interface for this argument. From the HLS programmer perspective whole memory space appears, through the pointer, as a regular array that can be addressed using the index notation or pointer arithmetic. Single access to the memory array are translated to single memory transfers. If the memory array is accessed inside a pipelined loop or by using the `memcpy()` function a burst memory access is generated.

- `state_out` is an 8-bit wide output port meant to be connected to the I/O resources of the programmable logic. It has been used for testing and validation purposes during the modules development process.

The volatile keyword is used to prevent HLS from performing any kind of assumption about the pointer accesses. The keyword is recommended whenever a function pointer argument is accessed multiple times.

5.1.2 Blur and sharp Filters

Blur and sharp are image processing filters implemented using a convolution operator. Convolution is a signal processing operation defined between two functions: a source function $f(x)$ and a filter function $g(x)$. The convolution operator is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

In image processing a similar operator can be defined considering a discrete bi-dimensional spatial domain instead of the continuous time domain. In such a case a discrete convolution operator can be defined as:

$$(F * K)[x, y] = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} F[x-i, y-j]\, K[i, j]$$

Where F is the source image and K is an $N \times N$ matrix, usually called kernel. From a practical perspective it means that for each input pixel $F[x, y]$ the resulting value is computed as a weighted sum of neighboring pixels, through the weights specified by the elements of the kernel matrix.

For a given input image and a kernel the whole output image can be produced by "sliding" the kernel over the image. The output pixels are computed as the result of the convolution between the source image pixels and the kernel weights.

The weights inside the kernel matrix determine the effects of the filtering on the image. The filters developed in this work use two different 5×5 kernels.

The kernel used for the blur filter is a matrix of ones, called blur-box. This kernel has the effect of averaging the value of each pixel with the values of its neighboring pixels, acting like a low-pass filter.

$$K_B = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

For the sharpening filter the kernel used is a high-pass operator that accentuates the comparative differences in the values with the neighboring pixels.

$$K_S = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & 2 & 8 & 2 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

Implementation

In the vivado HLS implementation the source image is read line by line updating an internal window buffer. The window buffer stores the last $N = 5$ lines of the image, where N is the size of the kernel. Once a new line has been read from the main memory, and copied into the window buffer, the convolution kernel will be applied to all the lines inside the window buffer.

The resulting output pixels will be stored inside a single line output buffer. When the computation is complete the output buffer will be copied back to main memory. A pseudocode description of the implementation is provided in Listing 5.2.

In the actual implementation the blur filter differs from the sharp filter to take advantage from the fact that the kernel is matrix of ones.

Since the kernel size is fixed the computational complexity the operations depends only on the image size. For an image of size $N \times M$ the complexity of this implementation of a convolution filter is $O(NM)$.

5.1.3 Sobel filter

Sobel filter is a classical algorithm used in image processing and computer vision application for the detection of object edges. The algorithm uses two 3×3 kernels: G_x and G_y to compute the approximate derivatives over the horizontal and vertical directions.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

For each pixel the approximated derivatives are computed considering its intensity value against the values of his neighboring pixels. The resulting derivatives are the components of the approximated gradient of the intensity in that point. An approximation of the absolute magnitude of the gradient is computed as the sum of the absolute values of its components.

$$|G| = |G_x| + |G_y|$$

The intensity value of each pixel of the output image is set proportional to magnitude of the intensity gradient in that point of the image.

Implementation

The implementation of the convolution part of the sobel filter is similar to the implementations described in the previous section. Before being convoluted with the kernels source pixel are converted from the RGB 24-bit color representations to 8-bit grayscale representation. The luminance information is extracted using standard BT.601 “full swing” approximated

Listing 5.2: Image convolution filter pseudocode.

```
void image_convolution(source_image, dest_image)
{
  const var kernel;
  const var kernel_sum;
  var window_buffer;
  var output_buffer;

  var red, green, blue;
  var pixel_out;

  << pre-fill window buffer >>

  // For each line of the source image
  for (line : source_image) {

    // Update window buffer. Read a new line for the memory
    window_buffer.read_mem(source_image[line]);

    // Apply the kernel to the lines inside the window buffer
    for (x : IMAGE_WIDTH) {

      // Convolution
      for (i : KERNEL_WIDTH) {
        for (j : KERNEL_WIDTH) {

          // Multiply and accumulate each color component
          red   +=  window_buffer(x, i, j).red * kernel(i, j);
          green +=  window_buffer(x, i, j).green * kernel(i, j);
          blue  +=  window_buffer(x, i, j).blue * kernel(i, j);
        }
      }

      // Normalize componets values
      red /= kernel_sum;
      green /= kernel_sum;
      blue /= kernel_sum;

      // Range componets values
      red = range_value(red);
      green = range_value(green);
      blue = range_value(blue);

      // Pack
      pixel_out = pack_rgb(red, green, blue);

      // Store the pixel in the output buffer
      output_buffer.store(pixel_out);
    }

    // Write the line to the memory
    output_buffer.write_mem(dest_image[line], pixel_out);
  }
}
```

conversion. First a 16-bit luminance value is computed as the weighted mean of the 8-bit color components:

$$Y' = \begin{bmatrix} 66 & 129 & 25 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Then 16-bit luminance value is scaled down to 8-bit:

$$Y = (Y' + 128) \gg 8$$

After the convolution the absolute values of the derivative components are summed together to calculate the approximate magnitude of the gradient. To enhance the edges separation effect the magnitude is compared with low and high saturation thresholds. A pseudocode description of the sobel implementation is provided in Listing 5.3. As for the blur and sharp filters, for an image of size $N \times M$, the complexity of the implementation of this Sobel filter is $O(NM)$.

5.1.4 Matrix multiplier

This operation implements the matrix multiplication between two $N \times N$ integer square matrices. Given two matrices A a B each element of the product matrix $C = AB$ can be calculated as:

$$c_{ij} = \sum_{k=1}^{N} a_{ik}\, b_{kj}$$

Implementation

The hardware implementation follows straightly the naive approach, resulting in a computational complexity of $O(N^3)$. For each element of the matrix $C : c_{ij}$ the correspondent line of the matrix $A : a_i$ and column of matrix $B : b_j$ are copied into local buffers. The element c_{ij} is computed as the dot product of the elements inside the buffers. The process is repeated and the elements c_{ij} are stored inside a local buffer. When an entire row of matrix $C : c_i$ has been computed the local buffer will be copied back to memory.

Listing 5.3: Sobel filter pseudocode.

```
void image_convolution(source_image, dest_image)
{
  const var gx;
  const var gy;
  var window_buffer;
  var output_buffer;

  var pix_luma;
  var dluma_w, dluma_h;
  var pixel_out;

  << pre-fill window buffer >>

  // For each line of the source image
  for (line : source_image) {

    // Update window buffer. Read a new line for the memory
    window_buffer.read_mem(source_image[line]);

    // Apply the Gx and Gy to the lines inside the window buffer
    for (x : IMAGE_WIDTH) {

      // For each pixel in the central line of the window
      for (i : KERNEL_WIDTH) {
        for (j : KERNEL_WIDTH) {

          // Compute pixel brightness value
          pix_luma = rgb_2_luma(window_buffer(x, i, j));

          // Calculate the approximations of the
          // horizontal and vertical derivatives
          dluma_w  +=  pix_luma * gx(i, j);
          dluma_h  +=  pix_luma * gy(i, j);
        }
      }

      // Sum derivative components
      dluma = ABS(dluma_w) + ABS(dluma_h);

      // Invert
      luma_out = (MAX_LUMA - dluma);

      // Threshold
      if (luma_out > H_LUMA)
        luma_out = MAX_LUMA;
      else if (luma_out < L_LUMA)
        luma_out = 0;

      // Store the pixel in the output buffer
      output_buffer.store(luma_out);
    }

    // Write the line to the memory
    output_buffer.write_mem(dest_image[line], pixel_out);
  }
}
```

Listing 5.4: Matrix multiplier pseudocode.

```
void matrix_mult(matrix_a, matrix_b, matrix_c)
{
  // Local buffers
  var a_row[];
  var b_col[];
  var c_row[];
  var dp_temp;

  // Iterate over the rows of the A matrix
  for (i : N) {

    // Copy (burst) matrix A row (i) from memory to local buffer
    memcpy(a_row, matrix_a.row(i));

    // Iterate over the columns of the B matrix
    for (j : N) {

      // Copy (burst) matrix B column (j) from memory to local buffer
      memcpy(b_col, matrix_b.col(j));

      // Inner product between matrix A row and matrix B column
      dp_temp = 0;
      for (k : N) {
        dp_temp += a_row[k] * b_col[k];
      }

      c_row[j] = dp_temp;
    }

    // Copy back (burst) matrix C row (i) from local buffer to memory
    memcpy(matrix_c.row(i), c_row);
  }
}
```

A pseudocode description of the implementation is reported in listing 5.4. To allow burst data transfer matrix B must be stored in memory in column major order.

5.2 Support library software structure

The support library abstracts the hardware acceleration and reconfiguration mechanisms, providing simple API, as discussed is section 4.3.1. The internal structure of the library follows a modular design. The main components of the library are the reconfiguration service and the hierarchy of hardware operations.

5.2.1 Reconfiguration service

The reconfiguration service is the main component of the library. The service comprises three software modules:

- `Rcfg_Manager` module is the main component of the service, used by the tasks to request the execution of accelerated operations.
- `Dev_Cfg` module wraps the device configuration interface driver. It's used by the `Rcfg_Manager` to drive the reconfiguration of the programmable logic.
- `Slot_Drv` module is the component that performs the low level operations on the hardware accelerators. The `Rcfg_Manager` use this module to control the hardware accelerators and the slots decouplers.

A UML class diagram summarize the reconfiguration service structure in Figure 5.1. The diagram should be intended as a language-agnostic *model* of the service. The real implementation is written in ANSI-C language. Since the C language does not feature language-level support for object-oriented programming, the real implementation relies on programming conventions to emulate object support.

Rcfg_Manager module

This modules manages the execution of accelerated operations and the allocation of the hardware accelerators. When a task requests the execution of an accelerated operation, through the `execute_hw_op()` function, first it check the availability of a free slot by the `take_slot()` function. The function uses a counting semaphore to count the number of vacant slots. Initially the semaphore is set to the number of slots available in the system. If no vacant

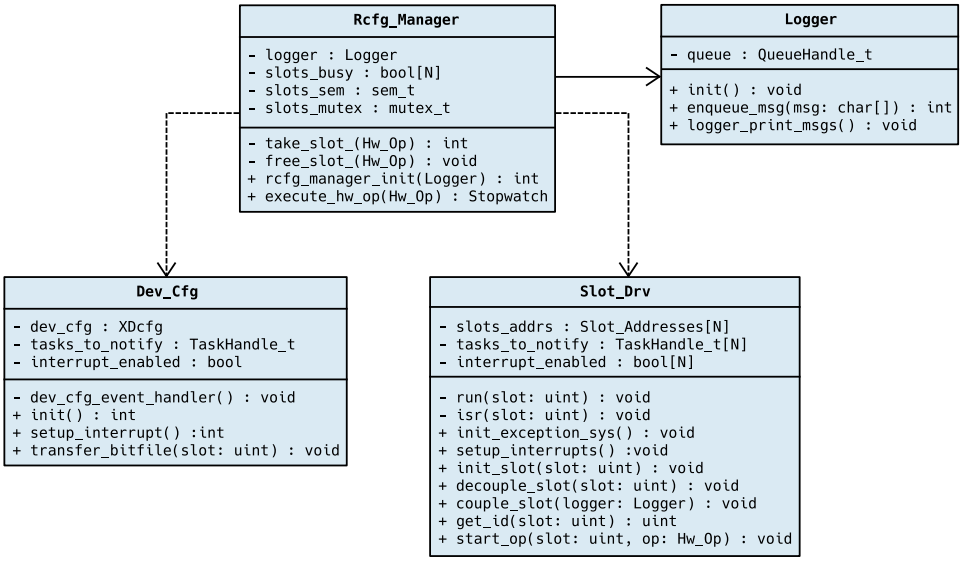


Figure 5.1: UML class diagram model of the reconfiguration service.

slots are available the task will be suspended on a queue associated with the semaphore. Once a vacant slot is available the `take_slot()` function handles the allocation of the hardware accelerator, eventually reconfiguring the device reconfiguration through the `Dev_Cfg` module. When the accelerator is ready the `Rcfg_Manager` module can control it through the `Slot_Drv` module, starting the hardware computation. Once the computation has been completed the `free_slot()` function updates the current slots allocation state.

Slot_Drv module

This module abstracts the low-level details of the hardware accelerators control, providing a simpler interface for the `Rcfg_Manager` module. Since all hardware accelerators use the same interface structure also the control and data registers are the same for all accelerators. Therefore, also the low level control functions are uniform. The mapping of the control and data registers into the memory space depends on the specific slot where the hardware accelerator resides at that moment. In other words the register's addresses are associated with the slot, not the accelerator.

For this reason the `Slot_Drv` module includes, for each slot, a specific structure that contains the related addresses. Such structures are used by

the control functions to select the accelerator, hosted in the specific slot, depending on the slot index argument.

After a hardware accelerator has been allocated in a slot, in order to execute an accelerated operation, the `Rcfg_Manager` uses the `Slot_Drv` module to perform the low-level control operations. The `start_op()` function loads the data contained in the hardware operation structure into the accelerator and starts the computation. The calling task will be suspended and later restored, when the computation has been completed, using the FreeRTOS *task notifications* mechanism as a binary semaphore.

Dev_Cfg module

This module controls the device configuration interface (DevC) used to reconfigure the slots inside the programmable logic. During the allocation process of a hardware accelerator the `Rcfg_Manager` can request a device reconfiguration through the `transfer_bitfile()` function. The underlying task will be suspended, until the reconfiguration has been completed, using the *task notifications* as a lightweight binary semaphore. Before and after device reconfiguration the `Rcfg_Manager` module controls the slot decoupling with the related functions provided by the `Slot_Drv` module.

Logger module

The `Rcfg_Manager` module shares with other components of the system a logger module used to print log messages. Internally the logger uses a FreeRTOS queue object to store the messages. The messages accumulated in the queue are flushed, trough the serial port, to the host computer during IDLE time or after the completion of other tasks.

5.2.2 Hardware operation objects

In the system accelerated operations are represented by hardware operation objects. The set of hardware operation objects defines the set of accelerated computations available to the user. A hardware operation is a data structure that contains all the necessary information to define an accelerated operation including:

- the set of bistreams containing the implementation of the underlying hardware accelerator;

- the input parameters for the hardware accelerator;
- two optional support functions to prepare the data before and after the hardware execution.

Conceptually the set of operations is organized as a class hierarchy as shown in Figure 5.2. The base class defines the interface and the derived classes implement the specific behavior. The components of the reconfiguration service relies on the interface defined by the base class. In some sense the hardware operation object "specialize" the programmable logic and the `Slot_Drv` control module for the specific hardware accelerator.

As for the reconfiguration service the operation objects are implemented in ANSI-C. Therefore the object-oriented support is only partially realized through programming conventions.

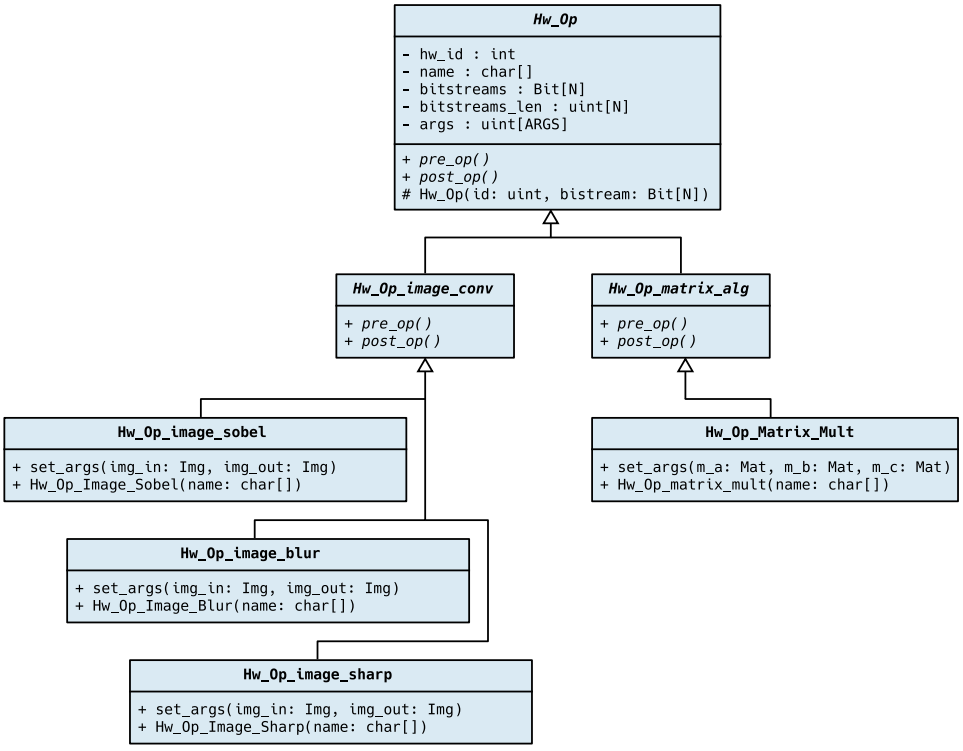


Figure 5.2: UML class diagram model of the hardware operations hierarchy.

Chapter 6

Experimental results

Contents

6.1 Experimental system setup

After the development of the test implementation on the ZYBO board some preliminary experiments has been carried out to evaluate the performances of the proposed approach. The experimental setup is based on the test implmentation, described in section 4.5, in a two slots configuration. The static region includes the static part of the communication infrastructure and the video output path. The ARM cores included in the processing system run at 650 MHz while the clock frequency for the programmable logic FPGA fabric is set to 100 MHz.

6.1.1 Programmable logic area allocation

The programmable logic area of the Zynq Z-7010 SoC, included in ZYBO the board, is divided in four clock regions. Each clock region occupies roughly a quarter of the total area and contains 25% of the total *slices*. In the experimental setup the area comprising two clock regions is allocated to the static system. The remaining area, organized in two clock regions, is divided into two equal size reconfigurable partitions allocated to the two reconfigurable slots. In reality the clock regions allocated to the slots contain twice the amount of BRAMs resources compared the clock regions occupation by the static system, while the amount of LUTs and DSP48 is almost the same. A graphical representation of the Zynq Z-7010 area, comprising the implementation of the static part, is presented in Figure 6.1. Resource consumption is represented by highlighting with colors the resources used for the implementation over the dark fabric of unused resources.

6.1.2 Hardware operations

The set of hardware operations comprises the four operations described in chapter 5: three image processing filters and a matrix multiplier. The image processing operations have been synthesized to process images of size 800×600 pixels, with 24-bit color depth. The matrix multiplier has been synthesized to multiply 512 elements integer matrices. For each hardware accelerator the high-level synthesis resources utilization estimates, provided by Vivado HLS, are reported in Table 6.1.

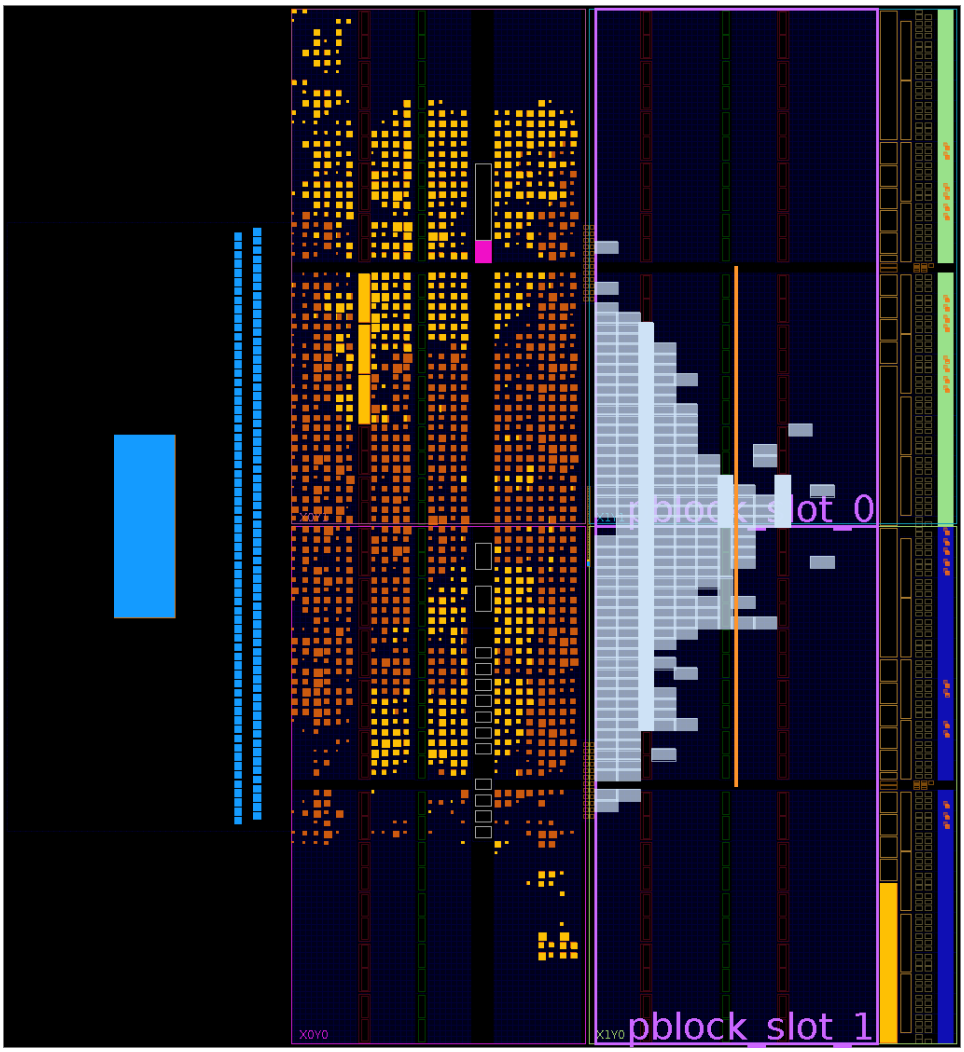


Figure 6.1: Device view of the implementation of the static part. From the left side of the image processing system is highlighted in light blue. On the right side the programmable logic area is divided into four clock regions. Two clock regions on the left are allocated to the static part of the design. The logic resources used for the implementation of the static part are highlighted in orange. The light orange blocks indicate the resources used for the implementation of the video output path. The remaining dark orange blocks are used to implement communication and support functions.
On the right side the two remaining clock regions are allocated to the reconfigurable slots, physically constrained by the two *Pblocks* highlighted in violet. No resources are placed inside those partition since they are reserved for the reconfigurable modules (hardware accelerators). The gray boxes inside the Pblocks are the ports for interfacing the reconfigurable modules with the static logic.

Hardware module	**LUT**	**FF**	**DSP48E**	**BRAM18K**
Sobel	3044 (17 %)	2711 (7 %)	29 (36 %)	10 (8 %)
Blur	4198 (23 %)	2801 (7 %)	8 (10 %)	14 (11 %)
Sharp	4198 (23 %)	3140 (8 %)	5 (6 %)	14 (11%)
Mult	2091 (11 %)	1782 (5 %)	24 (30 %)	19 (15 %)
Zynq Z-7010 Total	17600	35200	80	120

Table 6.1: High-level synthesis resource utilization estimates. The utilization percentage refers to the whole Zynq device.

The RTL code resulting from high-level synthesis of the hardware modules has been synthesized, and then implemented, in two different versions, one for each slot. Since the two slots have equal size, also the resulting bistreams have the same size. The whole partial reconfiguration flow, for the entire set of modules, produces eight different partial bistreams, two for each module, of size 338 KByte. The flow also generates a full bitstream, comprising the whole area of the Zynq Z-7010, of size 2 MByte. The full bistream includes the static part and two accelerators placed in the slots. Table 6.2 provides a comparison between the HLS resources utilization estimates and the more accurate estimate at synthesis level. Further optimization are possible in the implementation phase. Figure 6.2 provides a graphical representation of the implementation of the sharp hardware accelerator inside the slot-1.

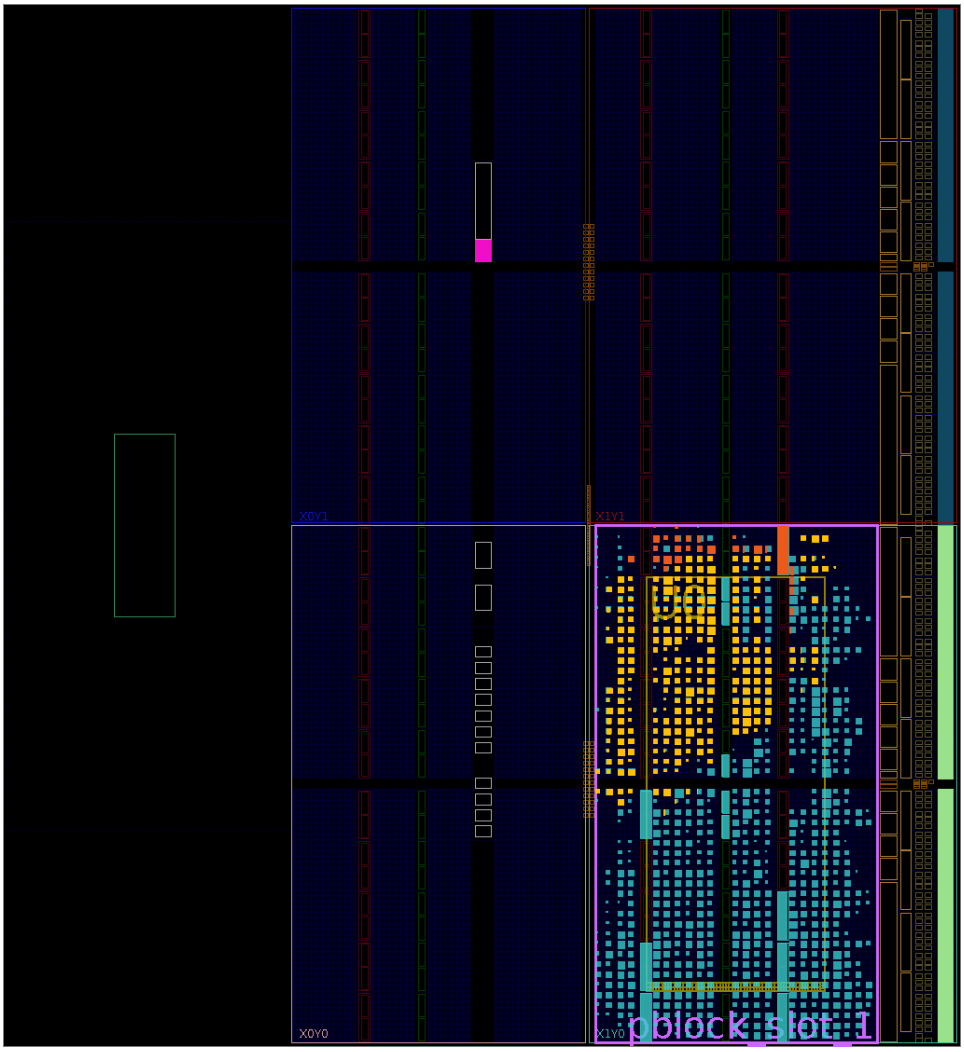


Figure 6.2: Device view of the sharp hardware accelerator implemented in slot 1. The logic resources used to implement the computing logic are highlighted in teal. The AXI master logic is realized by the orange cells. The resources highlighted in red in the upper part implements the AXI slave control logic.

Hardware module		LUT	FF	DSP48E	BRAM
Sobel	*HLS*	3044	2711	29	10 RAMB18
	Synth	2584	2188	5	5 RAMB36
Blur	*HLS*	4198	2801	8	14 RAMB18
	Synth	2698	2237	8	7 RAMB36
Sharp	*HLS*	4198	3140	5	14 RAMB18
	Synth	2870	2635	5	7 RAMB36
Mult	*HLS*	2091	1782	24	19 RAMB18
	Synth	1632	2410	18	9 BRAMB36 1 BRAMB18
Zynq Z-7010 Total		17600	35200	80	120

Table 6.2: Comparsion between resources estimates at HLS level and logic synthesis level.

6.2 Speedup evaluation experiment

The first experiment was carried out to evaluate the speedup factors achievable from hardware acceleration on FPGA. The execution times of the four accelerated operations, running on the PL, have been compared to their software counterparts, running on the ARM core, for a significant number of runs.

For each operation the tests have been performed using a single benchmark task, without any other load on the system. Each job of such task first requests to the reconfiguration manager the execution of the accelerated operation under test. Once the operation has been completed, the reconfiguration manger returns to the task a data structure containing the execution time of the hardware operation. Then the task executes the software version, measuring the execution time. Since such a task is the only activity running on the system, the measure is not subject to interference from other activities. Finally, the task computes the speedup ratio and write the test results into the log queue through the Logger module.

Once the task has been executed for the required number of jobs, it will be no longer activated. At this point the Logger module will flush the log queue to the host PC through the serial port, allowing the user to retrieve

Operation	FPGA-ET [ms]		Processor-ET [ms]		Speedup	
	pACET	pWCET	pACET	pWCET	Avg	Min
Sobel	21.523	21.526	179.067	179.074	8.319	8.318
Blur	26.387	26.391	374.780	374.803	14.203	14.201
Sharp	26.390	26.395	304.985	305.015	11.557	11.555
Mult	1698.245	1698.246	8768.543	8769.744	5.163	5.163

Table 6.3: Comparison of the profiled worst-case (pWCET) and average-case (pACET) execution times of hardware operations and software counterparts.

the test results. The complete results are reported in Table 6.3. Average speedup factors are graphically visualized in Figure 6.3.

6.2.1 Results evaluation

The results show that, despite the lower clock frequency of the FPGA fabric, the hardware implementations provide a relevant speedup over the software versions. It is worth noting that the measured speedup factors are dependent upon the specific implementation and optimization techniques. In general, it is safe to assume, for stream processing oriented operations, an average speedup factor ranging between 5 and 20, due to the higher level of parallelism achievable on an FPGA.

The low variance observed in the results depends on the fact that the operation are stream-processing oriented algorithms. For a given stream of input data, a series of operations, usually referred as kernel function, is applied to each element in the stream. The execution flow does not contain any branches and does not depends on the specific value of the input data.

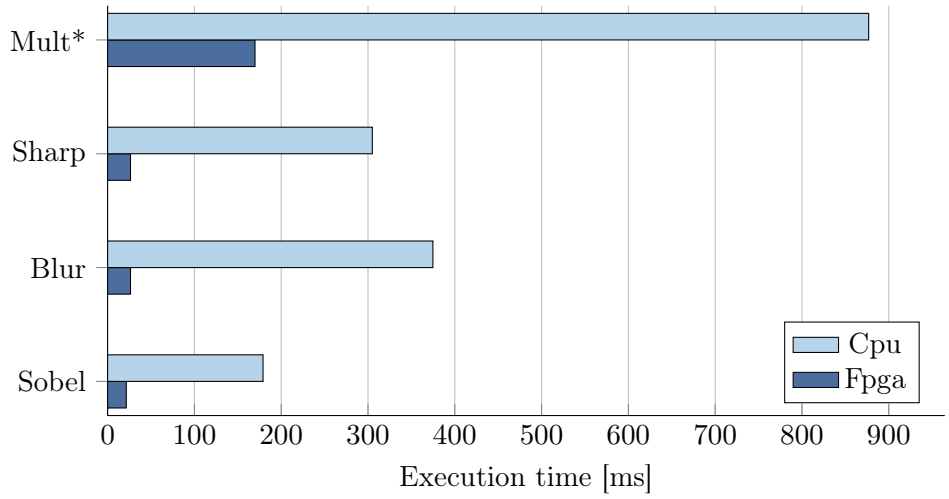


Figure 6.3: Comparison of the average execution times between hardware and software implementations. The execution times of the *Mult* operation have been scaled down by a factor of 10.

6.3 Worst-case response time experiment

The second experiment was carried out to evaluate the system behavior in a scenario where the number of tasks, requiring the execution of hardware operations, exceeds the number of slots. In particular, this experiment aims at measuring how the priorities assigned to the tasks impact on the observed worst-case response times.

The task-set used for this test comprises four tasks. Each task requests the execution of one of the four accelerated operation: *sobel*, *blur*, *sharp* and *mult*. The tasks have the following periods:

- Sobel task: 100 ms
- Blur task: 150 ms
- Sharp task: 170 ms
- Mult task: 2500 ms

The underlying operating system, FreeRTOS, uses a fixed-priority preemptive scheduling policy. In a first test all tasks have been assigned the same priority. Therefore, they are scheduled according to FIFO order, without preemption among the tasks. In a second test task's priorities are assigned

according to the Rate Monotonic order. Therefore, each task is assigned a priory proportional to its request rate. Both tests runs for 5 minutes.

Table 6.4 summarizes the tasks proprieties and reports the profiled worst-case execution times (pWCETs) and average-case execution times (pACETs) for both tests. Figure 6.4, 6.5, 6.6 and 6.7 show the distributions of the response times, normalized with respect to periods, for all the tasks.

6.3.1 Results evaluation

The results of both tests confirm that the system is able to sustain an area overload condition where the number of hardware operations exceeds the number of available slots. It is worth to noticing that, in this condition, the "virtual area", required to implement all the functionalities exceeds the physical area available on the device. Here the area is measured in terms of the number of available logic resources. In such situations the feasibility of the system can be achieved only by sharing, in the time domain, the FPGA resources through dynamic partial reconfiguration.

The comparison between Rate Monotonic and FIFO shows that Rate Monotonic order is able to guarantee lower response times for the tasks with the higher rate, preserving the feasibility of the scheduling. This is an expected behavior since Rate Monotonic is optimal, among the class of fixed priority assignment, in the sense of schedulability. The profiled worst-case response times are promising results in the direction of applying this approach to real-time systems.

Task	**Period [ms]**	**Jobs**	**HwOp pWCET [ms]**	**RM resp-time [ms]**		**FIFO resp-time [ms]**	
				Worst	**Avg**	**Worst**	**Avg**
Sobel	100	3000	21.526	47.480	27.391	93.330	36.615
Blur	150	2000	26.391	78.623	40.883	67.052	34.218
Sharp	170	1765	26.395	84.030	39.103	80.544	36.178
Mult	2500	120	1698.246	1737.880	1714.244	1737.888	1704.976

Table 6.4: Summary of the worst-case response time experiment.

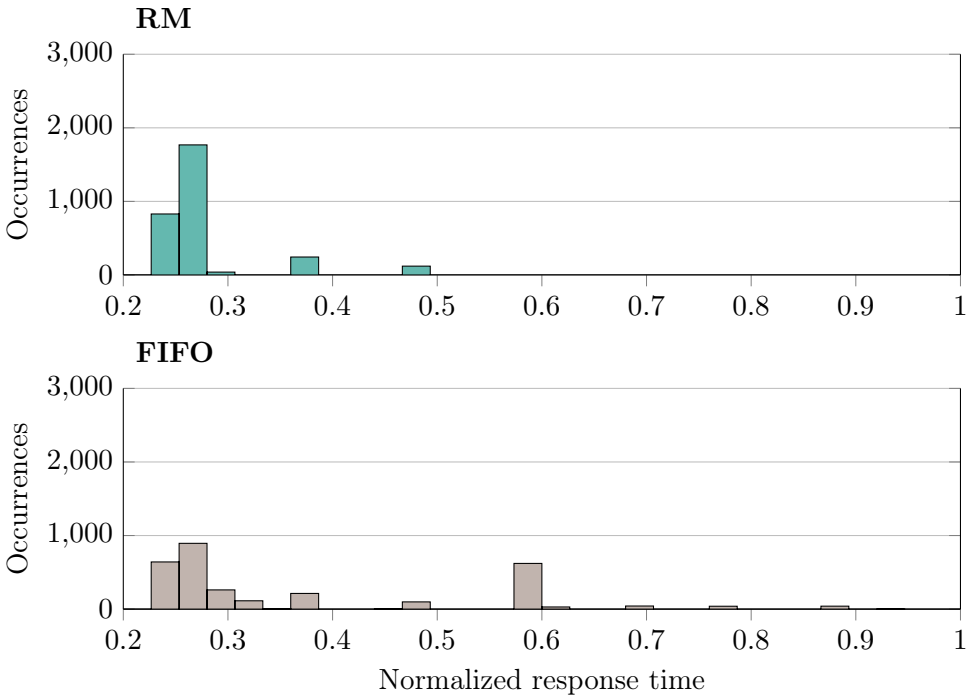


Figure 6.4: Response times distributions for the *Sobel* task.

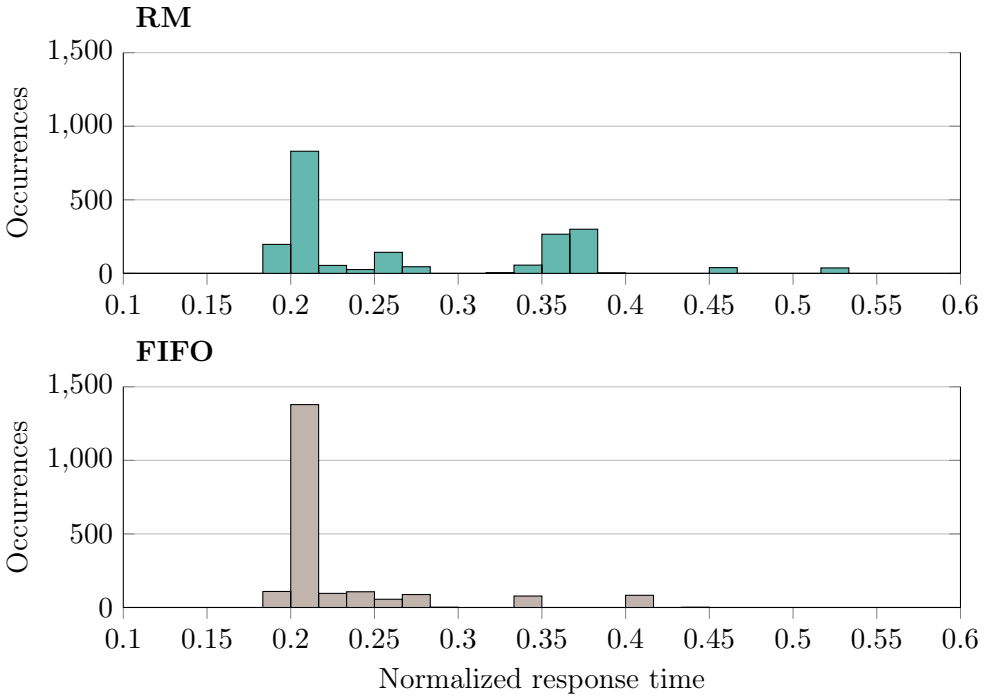


Figure 6.5: Response times distributions for the *Blur* task.

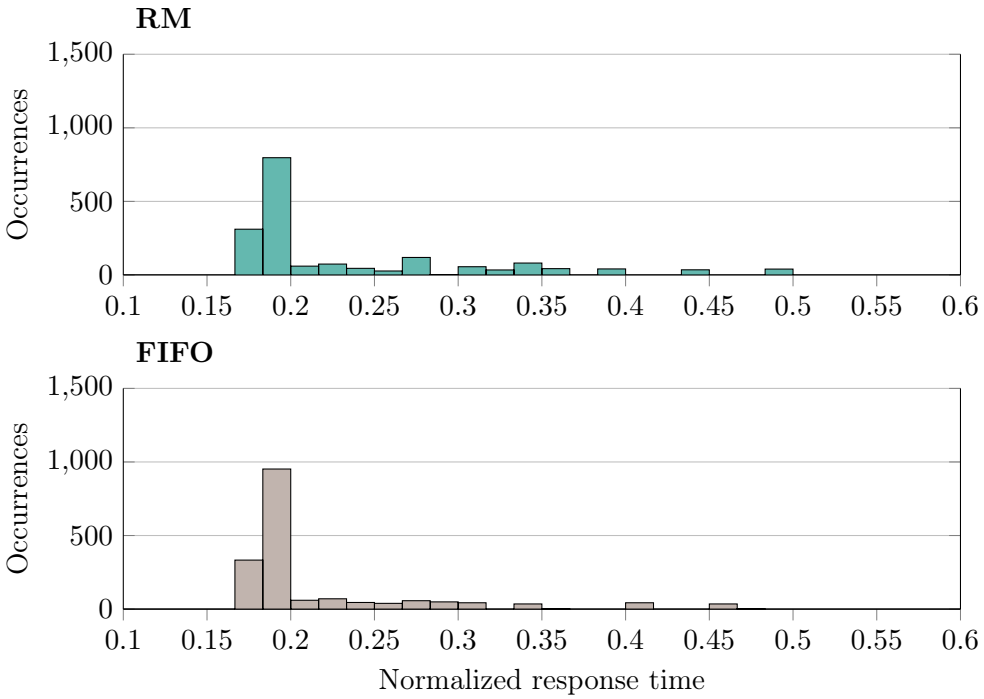


Figure 6.6: Response times distributions for the *Sharp* task.

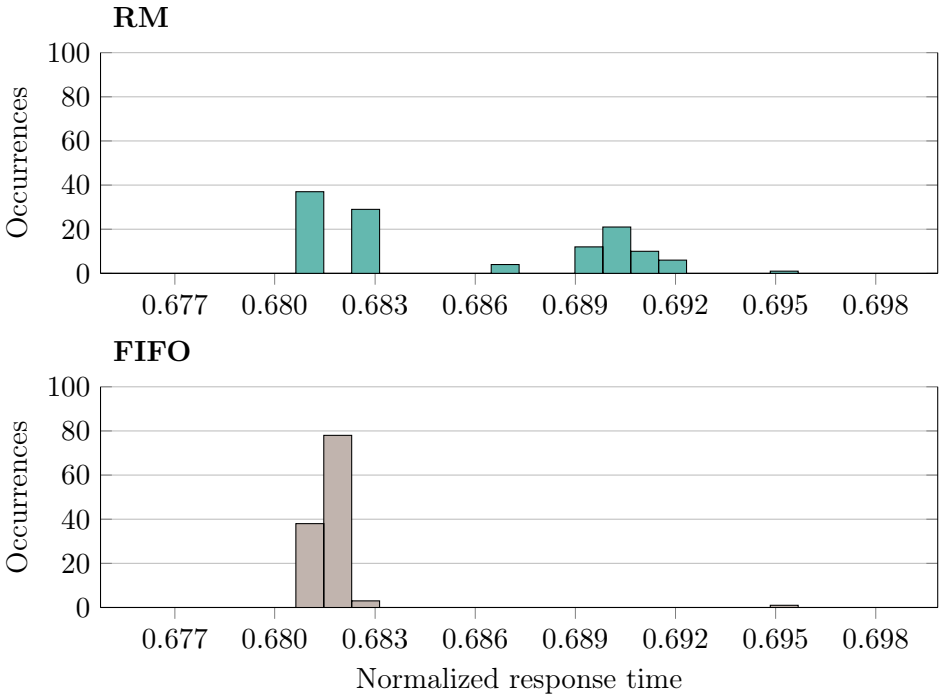


Figure 6.7: Response times distributions for the *Mult* task.

6.4 Reconfiguration times profiling

This experiment concerns the profiling of the reconfiguration times. As discussed in Section 3.3 the reconfiguration of programmable logic fabric is performed by the *Device Configuration* (DevC) subsystem contained in the processing system. Such a module transfers bitstreams from the main memory to the programmable configuration memory trough the processor configuration access port (PCAP) using an internal DMA engine. The DMA accesses the system memory through an AXI master interface connected to the internal AXI interconnect. Unlike the Application Processing Unit, and the master modules connected to the High-Performance AXI ports, the DevC subsystem does not have a direct path to the DRAM controller and must share the access to the controller with other internal modules through the central interconnect.

In general, the throughput achievable by the Device Configuration internal DMA depends on the traffic conditions of internal Interconnect, and the load on the DRAM controller. The modeling and performance evaluation of the Interconnect goes beyond the objectives of this thesis. However a first test was carried out to evaluate how a memory intensive software task could interfere with the Device Configuration throughput, affecting the reconfiguration times.

The task-set used for this test includes the four tasks described in the previous test, with the addition of a memory intensive software activity, continuously running in background. The software activity performs memory transfers between two 32 Mbyte memory buffers. The sizes of the buffers exceed the size of the Application Processing Unit L2 cache. Therefore, such a memory transfer generates a continuous stream of request to the DRAM controller, simulating a generic memory intensive software application. To summarize, in the test system the DRAM controller can receive requests from the following modules:

- The Accelerator modules in the first slot;
- The Accelerator module in the second slot;
- The video DMA engine;
- The Processor (memory transfer activity);
- The Device Configuration (through central Interconnect).

The objective of the test is to estimate how the memory transfers generated by the software activity affect the reconfiguration times. The base task-set used for this test includes the four tasks used for the prevision test with the same periods. The task's priorities are assigned according to the Rate Monotonic order. Table 6.5 shows the results of test, comparing the reconfiguration times observed in two different runs of the same duration. In the first test only the four tasks included in the base task-set are active. In the second test also the memory traffic generator is active. Figure 6.8 illustrates the distribution of reconfiguration times.

Experiment	**Reconfiguration time [ms]**		
	Min	**Avg**	**Max**
4 Task-Hw	2.7942	2.8149	2.8434
4 Task-Hw + MemTask	2.8502	2.9230	2.9638

Table 6.5: Reconfiguration times.

6.4.1 Results evaluation

The results of this test show that memory intensive software activities can affect the reconfiguration times, although the impact is very small, in the order of 0.1 ms. Therefore, the system is able to sustain memory intensive software activities without significant impacts on the reconfiguration performances. This result is important in the perspective of real-time systems, where bounded reconfiguration delays are essential to guarantee a predictable system behavior. Given the size of the partial bistreams: 338 KByte, the average observed throughput for the Device Configuration Interface are 117 MByte/s for the test with the base task-set and 113 MByte/s for the test with the base task-set and the memory intensive software activity.

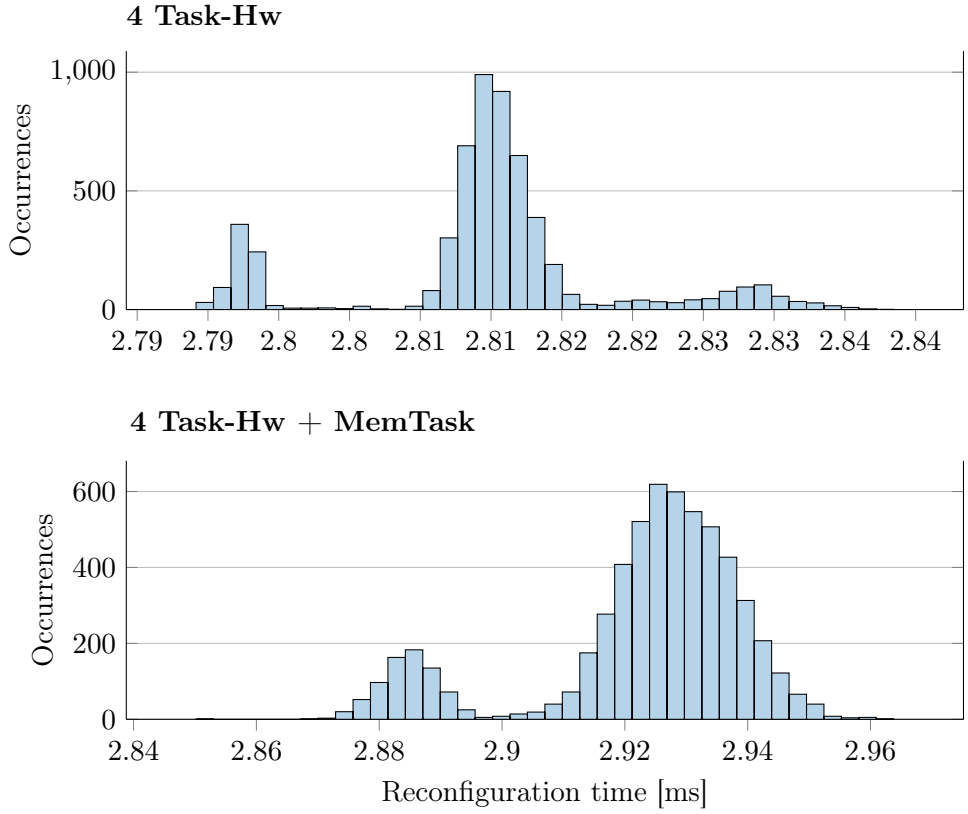


Figure 6.8: Distribution of reconfiguration times.

Conclusions

This thesis has been dedicated to the design and implementation of a software support for real-time systems developed on heterogeneous systems on a chip platform that includes a processor and a dynamically reconfigurable FPGA. The software support allows real-time applications to exploit the dynamic partial reconfiguration capabilities of the heterogeneous platform, extending the concept of multitasking to the FPGA resource domain. With respect to conventional approaches, based on system-level reconfiguration, this work proposes a job-level approach, in which each job of a hardware accelerated software activities can request the reconfiguration of the FPGA fabric.

The performance of the system has been evaluated in a case study that includes hardware accelerated image processing operations and a linear algebra matrix operation. The measured speedup factors for accelerated operations ranges between 5x and 14x, providing a relevant advantage over equivalent software versions. Moreover, the time required to reconfigure a portion of a modern FPGA can be estimated in the order of milliseconds, depending on the amount of logic resources involved in the process. For the specific device used in this case study, the Zynq Z-7010 SoC, the time required to reconfigure roughly 25% of the logic resources of the internal FPGA is less than 3 milliseconds. The reduction in the measured response times shows that the speedup factors achievable due to FPGA hardware acceleration and the system-level parallelism provided by the multi-slot architecture overcomes the overheads introduced by the partial reconfiguration.

As a future work, the mechanisms used in the proposed implementation can be incorporated in a real-time operating system, extending the interface between the system and userspace with specific system calls to manage reconfigurable hardware. Moreover, the system architecture deserves further studies and evaluation in the belief that integration between hardware accelerators and software support is the key element of the system.

Bibliography

[1] *7 Xeries FPGAs Configurable Logic Block*. UG474. Rev. 1.7. Xilinx. Nov. 2014.

[2] *Stratix V Device Handbook*. SV51002. Rev. 1.3. Altera. Nov. 2011.

[3] Louise H Crockett et al. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[4] Dirk Koch. *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Vol. 153. Springer Science & Business Media, 2012.

[5] *Vivado Design Suite User Guide: Partial Reconfiguration*. UG909. v2015.4. Xilinx. Nov. 2015.

[6] Ming Liu et al. "Run-time partial reconfiguration speed investigation and architectural design space exploration". In: *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE. 2009, pp. 498–502.

[7] Dirk Koch, Christian Beckhoff, and Jürgen Teich. "Recobus-builder—a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs". In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE. 2008, pp. 119–124.

[8] Christian Beckhoff, Dirk Koch, and Jim Torresen. "Go ahead: a partial reconfiguration framework". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, pp. 37–44.

[9] K. Danne and M. Platzner. "Periodic real-time scheduling for FPGA computers". In: *Third International Workshop on Intelligent Solutions in Embedded System, 2005*. Hamburg, Germany, May 2005, pp. 117–127.

[10] R. Pellizzoni and M. Caccamo. "Real-Time Management of Hardware and Software Tasks for FPGA-based Embedded Systems". In: *IEEE Transactions on Computers* 56.12 (Dec. 2007), pp. 1666–1680.

[11] S. Saha, A. Sarkar, and A. Chakrabarti. "Scheduling Dynamic Hard Real-Time Task Sets on Fully and Partially Reconfigurable Platforms". In: *IEEE Embedded Systems Letters* 7.1 (Mar. 2015), pp. 23–26.

[12] Enno Lübbers and Marco Platzner. "ReconOS: Multithreaded Programming for Reconfigurable Computers". In: *ACM Transactions on Embedded Computing Systems* 9.1 (Oct. 2009), 8:1–8:33.

[13] Xabier Iturbe et al. "Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS)". In: *ACM Transactions on Reconfigurable Technology and Systems* 8.1 (Feb. 2015).

[14] *Zynq-7000 AP SoC Technical Reference Manual*. UG585. v1.10. Xilinx. Feb. 2015.

[15] ARM. *AMBA Specifications*. URL: `http://www.arm.com/products/system-ip/amba-specifications.php`.

[16] *AXI Reference Guide*. UG761. v14.3. Xilinx. Nov. 2012.

[17] *Leveraging Data-Mover IPs for Data Movement in Zynq-7000 AP SoC Systems*. WP459. v1.0. Jan. 2015.

[18] Mohammadsadegh Sadri et al. "Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ". In: *Proceedings of the 10th FPGAworld Conference*. ACM. 2013, p. 5.

[19] *Vivado Design Suite User Guide - High-Level Synthesis*. UG902. v2015.4. Xilinx. Nov. 2015.

[20] *Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite for Zynq-7000 AP SoC Processor*. XAPP1231. v1.1. Xilinx. Mar. 2015.

[21] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.