Swinger: Processor Relocation on Dynamically Reconfigurable FPGAs

Henrique Miguel Santos da Silva Mendes

Thesis to obtain the Master of Science Degree in Electrical and Computer Engineering

Supervisors: Doutor Ricardo Jorge Fernandes Chaves
Doutor Nuno Filipe Valentim Roma

Examination Committee
Chairperson: Doutor Nuno Cavaco Gomes Horta
Supervisor: Doutor Ricardo Jorge Fernandes Chaves
Members of the Committee: Doutor Horácio Cláudio de Campos Neto

October 2014
Acknowledgments

First and foremost, I want to thank my supervisors Ricardo Chaves, Nuno Roma and Pedro Tomás. Not only for the amazing direction and supervision over the past 2 years, giving me their knowledge and experience in the field, but also, for giving me the opportunity to continue my studies at a point of my life I though of giving up. Thank you.

A special thanks to my colleague Nuno Neves, for working closely with me on this big project, making it possible. I also want to thank my amazing colleagues Sérgio and Hirak, for their continuous support and help in the most difficult parts of this project.

A very special thanks to Pedro Gançó, Pedro Diogo and Pedro Venâncio. You were the first friends I made in university, and we went through all the highs and lows, and made it to end together. I love you guys.

To Confraria: You are the most amazing group of creative, smart, and lovely people I ever got the pleasure to meet. I hope you know how much you helped me throughout this journey, and how grateful I am to have you. You are all stars.

Thank you to Inês do Ó and Lígia, for being the most awesome study company, and for being amazing friends. You always believed in me, and you inspire me to always give it my best.

To my international friends Piotr, Oskar, Magda, Asia, Maja, Sasha, Scott and Ross: The amazing moments I spent with you are important not just for my academic journey, but who I am as a person. The few times I spent with you are some the moments I cherish the most in my life, and hope to spend many more.

To some of my older friends, JT, Joana Neto, Joana Morte and Luís: Hope one day you will forgive me for the distance that I created this past years, while going on this amazing but difficult journey.

Finally, I want to thank my amazing family, for always supporting me in all my dreams, and for loving me for who I am, unconditionally. Thank you to my father Luís for teaching me about hard work. Thank you to my mother Laura for being the most solid rock in my life, and for being my best friend. Thank you to my brother Pedro for all the help and for inspiring me to follow this path. To my brother Afonso: you can do anything you put your mind, heart, and hard work into.
Abstract

Given the increased demand for high performance and energy-aware computational platforms, an adaptive heterogeneous computing platform composed of up to 105 cores is herein proposed. The platform is based on an aggregate of multiple processing clusters, each containing multiple processing cores, whose architectures are adapted, in execution-time, to the instantaneous energy and performance constraints of the software application under execution. This adaptation is ensured by a sophisticated hypervisor engine, implemented as a software layer in the host computer, which keeps a permanent record of a broad set of performance counters, gathered from the execution of each core in the FPGA, in order to dynamically determine the optimal heterogeneous mix of processor architectures that satisfy the considered constraints. Partial dynamical reconfiguration mechanisms ensure a run-time adaptation of the cores that integrate each cluster. When compared with static instantiations of the considered many-core processor architectures, the obtained experimental results show that significant gains can be obtained with the proposed adaptive computing platform, with performance speedups up to 9.5x, while offering reductions in terms of the consumed energy as high as 10x. In order to obtain efficiency in the reconfiguration process, a dedicated module capable of performing relocation is herein proposed, not only achieving the minimum reconfiguration time-overhead permitted by the target platform’s configuration port, but also allowing memory savings by relocating the target modules in the selected FPGA region, allowing to save up to 80% in configuration memory.

Keywords

Dynamic Reconfiguration, Heterogeneous Architecture, Energy-Aware Computation, Runtime Adaptability, FPGA, Relocation
Resumo

Respondendo a uma grande procura para alta eficiência em plataformas computacionais, em termos de execução e consumo energético, é proposto um sistema heterogêneo adaptativo composto por 105 elementos de processamento. O sistema baseia-se em múltiplos agregados de múltiplos elementos de processamento, possuindo uma capacidade de adaptação dinâmica ao nível da arquitetura, em tempo real, em relação à energia instatânea e requisitos de execução da aplicação software a ser executada. Esta adaptação é comandada por um elemento supervisor por software no computador anfitrião, que, recolhendo informação acerca de execução de todos os elementos de processamento, é capaz de determinar a configuração heterogênea ótima para satisfazer os requisitos definidos. Enviando comandos para o mecanismo de reconfiguração, instanciado como parte estática no sistema, um processo de reconfiguração parcial dinâmica garante uma adaptação em tempo real por parte dos elementos de processamento que compõem cada agregado reconfigurável. Quando comparado com uma implementação estática do sistema considerado, os resultados experimentais obtidos revelam que significantes ganhos podem ser obtidos, com ganhos de eficiência de tempo de execução até 9.5 vezes, oferecendo reduções em termos de consumo de energia de até 10 vezes. De forma a obter a maior eficiência no processo de reconfiguração, um módulo dedicado que torna possível a realocação, é proposto. Realocando os módulos instanciados na região da FPGA utilizada, é possível obter uma redução de memória de configuração de 80%.

Palavras Chave

Reconfiguração Dinâmica, Arquitectura Heterogénea, Computação energéticamente eficiente, Adaptação Dinâmica, FPGA, Realocação
# Contents

List of Acronyms  
1 Introduction  
  1.1 Motivation ................................................. 3  
  1.2 Objectives .................................................. 4  
  1.3 Main Contributions ......................................... 5  
  1.4 Dissertation outline ....................................... 5  
2 Related Technology ........................................... 7  
  2.1 FPGAs ...................................................... 8  
    2.1.1 FPGA Design Flow .................................. 9  
  2.2 Reconfiguration on FPGAs ................................ 9  
    2.2.1 Virtex device family ................................ 9  
      2.2.1.A Configuration Ports ............................ 10  
      2.2.1.B Bus Macros .................................... 11  
      2.2.1.C Multiboot .................................... 12  
    2.2.2 Altera Devices .................................... 13  
  2.3 Host-FPGA Interface ..................................... 13  
  2.4 MB-LITE in Multicore Systems ............................ 14  
  2.5 Summary .................................................. 15  
3 State of the art .............................................. 17  
  3.1 Reconfiguration ............................................ 18  
    3.1.1 Dynamic Partial Reconfiguration .................. 18  
    3.1.2 Relocation ........................................ 20  
  3.2 Multicore Heterogeneous Systems ....................... 23  
  3.3 Adaptive Processing system ............................. 23  
  3.4 Summary .................................................. 25
## Contents

4 Proposed Solution 27  
4.1 Dynamic many-core heterogeneous architecture 28  
4.1.1 Hypervisor 30  
4.1.2 Reconfiguration Engine 31  
4.1.3 Processing clusters 33  
4.2 Summary 34  

5 Architecture Design and Implementation 37  
5.1 Processing cores 38  
5.2 Communication and interfacing networks 40  
5.3 Reconfiguration 42  
5.4 SWINGER 45  
5.4.1 Read-Bitstream State Machine 47  
5.4.2 AXI Interface and Burst transfer 48  
5.4.3 Write-ICAP State Machine 49  
5.4.4 Relocation Parser 50  
5.4.4.A Floor-planning 51  
5.5 Hypervisor replacement policies 53  
5.6 Summary 54  

6 Evaluation 57  
6.1 Evaluation Benchmark 58  
6.2 Hardware resources 59  
6.3 Reconfiguration overhead 60  
6.4 Relocation Results 62  
6.5 Performance evaluation 63  
6.5.1 Runtime architecture adaptation and model definition 63  
6.5.2 Adaptive model-based policies 65  
6.5.3 Speedup and energy reduction 66  
6.6 Summary 68  

7 Conclusions and Future Work 71  
7.1 Conclusions 72  
7.2 Future Work 73  

A Annex A 79  
A.1 3-Tile Reconfigurable Module 80  
A.2 2-Tile Reconfigurable Module 80  
A.3 1-Tile Reconfigurable Module 81
## List of Figures

2.1 Reconfiguration with using external and Internal Configuration Ports .............. 11  
2.2 Partition Pins: Bridge between static and reconfigurable design ...................... 12  
2.3 Altera’s Reconfigurable Architecture .................................................. 13  
2.4 High-Level AXI Bridge for PCI Express Architecture. ............................... 15  
3.1 Connection of the reconfigurable regions. ............................................. 20  
3.2 REPLICA’s Relocation Module. ............................................................. 22  
4.1 Proposed reconfigurable multi-core heterogeneous architecture. ...................... 29  
4.2 Internal block diagram of a processing cluster. ...................................... 29  
5.1 Multicore framework components. ......................................................... 39  
5.2 Reconfiguration procedure: the Hypervisor is responsible for dispatching the workload and for issuing reconfiguration commands to the on-chip Reconfiguration Engine. .... 44  
5.3 SWINGER - Reconfiguration dedicated module with relocation parser ............... 46  
5.4 State Machine responsible for communication with Microblaze, reading configuration data from external memory ................................................................. 48  
5.5 Floor-Planning limitation in target device. ............................................. 52  
5.6 Considered Hypervisor replacement policies. ........................................... 55  
6.1 Real-time adaptation of the processing architecture, performing reconfiguration through software, having a Microblaze load into Xilinx AXI ICAP controller IP. ................ 61  
6.2 Real-time adaptation of the processing architecture using implemented dedicated module SWINGER. ............................................................ 62  
6.3 Real-time adaptation of the processing architecture, without any a priori knowledge of the computing kernels (Kernel order: 1, 2, 3, 4). .................................. 64  
6.4 Real-time adaptation of the processing architecture, without any a priori knowledge of the computing kernels (Kernel order: 1, 3, 2, 4). .................................. 64  
6.5 System real-time adaptation, according to the minimum execution-time optimization policy. ................................................................. 65
List of Figures

6.6 System real-time adaptation, according to the established power-ceiling constraint policy. 67
6.7 System real-time adaptation, according to the minimum assured performance policy. 68

A.1 3-Tile Reconfigurable Module floor-planning 81
A.2 2-Tile Reconfigurable Module floor-planning 82
A.3 1-Tile Reconfigurable Module floor-planning 83
List of Acronyms

AXI  Advanced Extensible Interface

CLB  Configurable Logic Block

CPU  Central Processing Unit

CRC  Cyclic Redundancy Check

CLB  Configurable Logic Blocks

DPR  Dynamic Partial Reconfiguration

DSP  Digital Signal Processor

ESB  Embedded System Block

FIFO  First-In First-Out

FPGA  Field-Programmable Gate Array

FPU  Floating Point Unit

FP  Floating Point

FSM  Finite State Machine

GPP  General-Purpose Processor

GPU  Graphics Processing Units

IOB  input-output Blocks

ICAP  Internal Configuration Access Port

ISA  Instruction Set Architecture

RM  Reconfigurable Module

ISA  Instruction Set Architecture

JTAG  Joint Test Action Group
List of Figures

LAB  Logic Array Block
LUT  Look-Up Table
PE   Processing Element
PRR  Partially Reconfigurable Region
RISC Reduced Instruction Set Processor
SPL  Specialized Programmable Logic
UART Universal Asynchronous Receiver/Transmitter
QoS  Quality of Service
List of Tables

5.1 Frame Address Register Description ............................................. 50

6.1 Experimental evaluation of each cluster type, in terms of hardware resources, maximum operating frequency and power consumption. ............................................. 60

6.2 Experimental evaluation of the static fraction of the reconfigurable platform, in terms of hardware resources, maximum operating frequency and power consumption. . 60

6.3 Experimental evaluation of each cluster floor-planning configuration, in terms of impact of relocation on configuration memory. ............................................. 63

6.4 Execution time and energy results. .................................................... 67
1 Introduction

Contents

1.1 Motivation .................................................. 3
1.2 Objectives .................................................. 4
1.3 Main Contributions ........................................ 5
1.4 Dissertation outline ....................................... 5
1. Introduction

The concept of reconfigurable computing appeared in the 1960s, when Gerald Estrin proposed a computer made of a standard processor and an array of reconfigurable hardware in which a main processor controlled the behaviour of the reconfigurable hardware. The reconfigurable system would be designed to execute a specific task and once the task was done the hardware would adjust to do some other task [6].

Since then, computational systems have suffered great developments in all aspects: from constant performance improvements in an ever-growing variety of applications, to the size of the hardware systems, as famously described by Moore’s Law. With signs that Moore’s Law linearity is slowly changing, as computational performance is increasingly difficult to improve, and the consumed power is a growing concern, alternatives to tackle these issues have surged.

Field-Programmable Gate Arrays (FPGAs) have followed the same improvement path as the traditional computational systems, and have now the technology necessary in order to be programmed with more complex digital systems, work at higher frequencies, and have an array of different interfaces and different built-in technologies. These improvements have changed the role of FPGAs from platforms to prototype and test digital systems, to viable platforms to run and improve tasks.

Since first introduced by Gerald Estrin’s, the concept has been applied to commercially available FPGA devices, and they now have the technology built-in. Different authors have applied it in different ways and have proven its great potential to improve performance, flexibility and power consumption of computational systems. The development of this technology on FPGAs, like the possibility of partial reconfiguration at run-time, the possibility of having different inter-changeable configuration layers on a device (Multiboot), keep increasing the number of possible approaches to reconfigurable systems and all its different uses.

While the dynamic reconfiguration of a specific region of an FPGA, with the goal of having a configuration that is better suited for a specific task, can undoubtedly improve its execution, the timing overhead introduced in order to do so is detrimental to that objective. Ideally, the time necessary to change a part of the design should not influence the system’s performance, however since the timing is directly tied to the resources to be changed, and is limited by the configuration port that perform that change, this issue is of special relevance and is one of the major downfalls of the technology.

Several authors have proposed different ways to reduce this detrimental time factor, as well as ways to reduce other loopholes on the technology. Another relevant issue introduced by the partial reconfiguration approach to FPGAs, is the potentially high number of configuration information that needs to be created. This can lead to the necessity of large storage, that is increased with the number of reconfigurable modules in the project and their complexity. In order to reduce this impact, the concept of dynamic relocation was introduced. In its premise, a single configuration file can be used to configure several similar modules that are interchangeable, by manipulating
1.1 Motivation

The configuration file itself.

A small reconfiguration time and the Relocation approach can be of special relevance for a multi-core system, for its similarity between cores, and for the necessity of having all the cores working in a parallel way, which is the starting thread of the present dissertation.

In this dissertation a thorough discussion and analysis of all the mentioned aspects is presented. By taking in consideration the different potential uses of the reconfiguration technology, and by also taking in consideration its different downfalls, the presented work is intended to be a relevant addition to the state of the art, being an improvement in all the important aspects to computational systems, like performance and power, and minimizing any timing overheads and excessive memory storage, using the different existent approaches to create a solid reconfigurable system.

1.1 Motivation

Different processor architectures can be categorised by their degree of flexibility. From the general purpose processor which are designed to be able to execute a large variety of applications, to the dedicated processor, created for specific tasks. Reconfigurable computing offers a large potential advantage in computing power over conventional micro-processor based systems, given its high adaptability.

Uniprocessor performance improvements have stalled while silicon capacity continues to grow. Consequently, the raw computational density gap widens and the role for spatially oriented programmable computations also grows. This is where the use of reconfigurable FPGAs becomes relevant. A constant focus on improving the architecture of computer system in terms of functionality, performance, power, and cost led to the development of different architectures, each trying to achieve an optimised balance of these key aspects, depending on the targeted goals. However it is apparent that these aspects are interdependent, and it is easy to improve one while sacrificing another. When a processor is customised for a specific task it loses flexibility in terms of being able to execute other types of tasks as efficiently. Given the fact that a more flexible architecture will lose in terms of performance, it is possible to say that these two factors (flexibility and performance) have an inverse relationship. Using reconfigurable computing it is possible to obtain a processor with the ability to adapt itself in order to achieve better performances on specific tasks while having a great level of flexibility.

Multi-core systems have proven to be relevant as computational forces for highly parallel application. From the description of the advantages of treating processors as reconfigurable modules described above, added to the advantages introduced by a multi-core system, and the performance results can be significant. If the system is reconfigured to a configuration dedicated to a specific task, the hardware instantiated is more fully utilized, as opposed to having unused
1. Introduction

resources instantiated, which leads to power saving.

Applying dynamic reconfiguration to a multi-core system can therefore be a great advantage, however, since it premises a hopefully large number of modules, it can result in a large amount of data that needs to be stored and processed in order to process the reconfiguration. However, since multi-core system have similar modules, relocation can be applied to mitigate the described limitation.

For the above reasons, having and adaptive multi-core solution is a viable path for the goal of improving performance and power, using dynamic relocation, towards creating a highly flexible processing system that can be applied to perform a wide range of tasks.

The main objective of this dissertation is to apply dynamic re-allocation using reconfiguration on an FPGA to a multi-core system, which is capable of dynamically adapt itself in terms of the architecture depending on the type of the operations which are being executed.

1.2 Objectives

In this dissertation, an adaptable and scalable architecture is proposed that not only supports more than one hundred heterogeneous cores, but can also adapt its characteristics according to application requirements, by taking advantage of the partial dynamic reconfiguration capabilities of modern FPGA devices. One of the main features and contribution of the proposed approach is the ability to monitor the performance of the Processing Elements (PEs) as they execute an application's different kernels, in order to autonomously and immediately determine the most suitable architecture according to the current execution scenario. Given this, the proposed solution is able to change the architecture of each of the PEs in real-time, in order to achieve the best possible performance/energy efficiency.

In order to achieve this, a module dedicated to perform reconfiguration is proposed, with the capability of communicating with the entity responsible for monitoring the system. The proposed module is meant to perform the reconfiguration with the minimum time overhead, dictated by the configuration limit of the chosen configuration port. All the different reconfiguration capabilities are explored, with the goal of having a system that efficiently manages the reconfiguration of all the different modules composing that same system.

The proposed module is also capable of performing the relocation of the reconfigurable modules, by applying configuration bitstream manipulation techniques that allow for the reconfiguration to be performed in different regions of the device done with no additional overhead using the same configuration file, hence saving in memory and increasing the flexibility of the system.
1.3 Main Contributions

The present evaluation of the adaptive multi-core system validates that it is possible to apply the most advanced reconfiguration technologies in modern FPGA to provide a hardware system with increased flexibility, while increasing performance and energy efficiency. As an example, the proposed system using multiple multi-core reconfigurable clusters, with 3 distinct configurations, allows to achieve performance speedups up to 9.5x, and energy consumption reduction up to 10x.

In order to provide the proposed system with a performance efficient low-complexity reconfiguration engine, a dedicated module is implemented, capable of loading configuration data from DDR memory into the chosen configuration port, at a 3.2Gbit/s rate, by using the available burst transfer option available in AXI buses, and an asynchronous First-In First-Out (FIFO).

Included in the implemented reconfiguration module, is the built-in option of performing relocation through filtering the configuration data as it is loaded to the configuration port. The implemented relocation parser provides the system with the option of using configuration data of a specific reconfigurable cluster to reconfigure another, without increasing reconfiguration time overhead. Thus, saving in configuration memory necessary to reconfigurable systems, potentially saving as much as 80% of necessary configuration data.

The SWINGER system and, in particular, its dynamic adaptation mechanisms, are discussed in the following international journal:


1.4 Dissertation outline

The present dissertation is composed of 5 main chapters: the chapter 2 describes all the related technology relevant to the frame work, with a deeper analysis of the modern FPGA technologies, both in terms of the general functions and specifically in relation to reconfiguration.

The chapter 3 presents the study of the state of the Art in which this project is inserted in. This chapter starts with an overview of the FPGA technology, following by the presentation of the existing research on reconfiguration and research done on reconfigurable systems. The chapter is concluded with the analysis of the most relevant multi-core systems.

The fourth and fifth chapters involve the description of the proposed system and all its different layers. There is an overview of the overall frame-work that was implemented. The Multi-core system used is described, as well as the communication between the Host and the FPGA. More extensively described is the reconfiguration engine, being that it is the main focus of this thesis,
1. Introduction

presenting the details of implementation focused on complying with the defined objectives.

The sixth chapter presents the evaluation of the results obtained from the system, and dis-
cusses those results, to draw conclusions on performance, reconfiguration cost, power saving,
and bitstream storage reduction due to relocation.

The final chapter provides concluding remarks, as well as proposed future work.
2

Related Technology

Contents

2.1 FPGAs ................................................................. 8
2.2 Reconfiguration on FPGAs ................................. 9
2.3 Host-FPGA Interface ........................................... 13
2.4 MB-LITE in Multicore Systems ......................... 14
2.5 Summary ............................................................ 15
2. Related Technology

When considering the different existent solutions for acceleration platforms, most point to FPGA given the increasing configurable resources, for the increasing working frequencies, the improvements and advances of its interfaces, and improvement on the existent development tools. For this reason, a description of the current market-available FPGA technologies is presented to better illustrate how the chosen platform operates, and their main advantages and disadvantages.

One of the main reasons FPGA devices are the best candidates for the system herein considered, is the fact that the most recent releases include built-in dynamic reconfiguration technologies. The possibilities available to perform reconfiguration have progressively evolved in terms of efficiency, and different options are now available.

Since the platform is meant to be an accelerator to a main processing system, it is important to characterize the current FPGA technologies and different methods of communication between a host Central Processing Unit (CPU) and an FPGA, with the goal of finding the best interface in alignment with the requirements for the proposed solution.

Lastly, since a multi-core system is proposed, and a customizable core is required in order to have different configurations possible, the most suitable options available for this kind of solutions is also presented on this chapter.

2.1 FPGAs

An FPGA is a programmable semiconductor device that is based around a matrix of Configurable Logic Blocks (CLB) connected through programmable interconnects. The CLB is the basic logic unit in an FPGA. Every CLB in a Xilinx FPGA consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc), and flip-flops. Modern FPGA devices follow the same traditional structure, varying in the existing features, technologies, and in number of CLBs. Through the connection between the different configurable resources, virtually any digital system can be created and tested, limited by the existing resource routing constraints and working frequency. The device is programmed using a configuration file (bitstream) which contains the translation between the design and the configuration of the specific device. This fact provides FPGAs with a great flexibility when it comes to the number of systems that can be programmed. However, this is achieved with a great complexity of programming the device.

As mentioned before, current FPGAs have evolved far beyond the basic capabilities of their predecessors, and incorporate dedicated blocks (ASIC type) of commonly used functionality such as RAM, clock management, and Digital Signal Processors (DSPs) units, as well as high-performance interfaces that increase the potential applications of the platform.

Several FPGA technologies and manufacturers exist in the market, however Xilinx and Altera are the clear leaders in the high-performance market. The technologies provided from this two manufacturers share the same basic concept for their functioning, but have different modes of
configuration, communication interfaces and associated tools, design flows and approaches to design projects. In order to better understand the process, it is important to describe the available technology.

2.1.1 FPGA Design Flow

To understand the functionality of an FPGA design, is the description the process between design conception and device configuration.

The tools used for an FPGA design project usually translate an hardware description language file (VHDL/Verilog) into an actual circuit design, and have a specific design flow from implementation to debugging. From an intended functionality, which leads to an hdl file, the design is synthesized to create a netlist, which is a textual description of a circuit diagram or schematic. Next, simulation is used to verify that the design specified in the netlist functions correctly. However, this simulation only provides the conceptual behaviour of the logic circuit, and does not take into consideration functional maximum frequency, possible clock skews due to FPGA specific routing. Following a behavioural verification of the netlist, the components are translated and mapped particular components of the target FPGA, the design is place and routed. Opposed to the behaviour simulation, the post-place and route simulation is the closest verification to the behaviour of the design configured in the FPGA.

Finally, a configuration bit file (bitstream) is created, which holds specific configuration commands and configuration data for the target FPGA, and loaded to a configuration port to program the device and start execution.

2.2 Reconfiguration on FPGAs

The novel FPGA devices have not only been improved in terms of performance and available resources, but also in the associated reconfiguration technology.

In order to accurately evaluate the technology, it is necessary to understand the elements that are reconfigurable, to the available configuration ports that can be used and their functionality, and the different available relevant options.

The next subsections lists the different options and characteristics present in each technology.

2.2.1 Virtex device family

The Virtex technology is the flagship technology from Xilinx, being a midrange coarse grained approach, meaning that it is composed by larger, more powerful logic blocks. The basic cells of these FPGAs are the CLBs, which include circuitry allowing to efficiently perform arithmetic operations, contains Look-up Tables and input-output Blockss (IOBs). The smallest component that can be targeted for reconfiguration is denominated Slice, and it is composed by Look-Up
2. Related Technology

Tables (LUTs) and CLBs. Therefore, when an element is reconfigured, the topology of this element is transformed into the new desired configuration.

In terms of reconfigurability, with each new release, different options are introduced, while some are discontinued. One of the main differences across the Virtex devices is the area of the configurable fabric since with each upgrade there is a significant increase of the maximum number of available resources in the device, enabling more complex systems to be configured. Furthermore, newer releases not only improve the overall potential performance of a system (higher maximum frequencies), but also introduce different configuration options that increase the maximum frequency possible for the reconfiguration process, mainly by improving the performance of the configuration ports available for the target device.

2.2.1. A Configuration Ports

Recent Virtex devices have 3 main configuration ports available: JTAG, SelectMap and ICAP (Internal Configuration Access Port). Configuration performance, timing constraints and the physical location are the main factors that differentiate these options.

Associated with a configuration port is a data bus, with a specific width data width, where configuration data is loaded, until the entire configuration file is uploaded. The configuration time value depends on the clock frequency, data width of the configuration port, and the size of the configuration file. The performance of the chosen configuration port can be maximized by using the maximum working frequency and ensuring that with each clock period a new configuration word is loaded.

JTAG is an external port to the FPGA, usually the chosen port for the initial configuration of the complete design, also allowing to load partial bitstream files and performing partial dynamic reconfiguration. However, out of the 3 configuration ports available it is the one with the least efficient maximum performance, having a width of 16 bits, and, in the most recent Virtex releases, a maximum achievable configuration frequency of 40MHz.

On the other hand, both SelectMap and ICAP in most recent Virtex devices have a configuration width of up to 32 bits, and can work at frequencies up to 100 MHz, making them more suitable for designs that depend on the lowest reconfiguration time possible. The main difference between ICAP and SelectMap, is the fact that they are internal to the FPGA and external, respectively.

As depicted in Fig. 2.1 it is possible to see that an internal configuration port (ICAP) is typically used when performing self-reconfiguration, when the reconfiguration is performed and triggered by internal logic. In that case, and internal micro-processor, for example, reads the configuration bitstream stored in a memory and loads it to the ICAP.

In the case of an external configuration port, an external processor is responsible for fetching the data from the configuration memory to the configuration port.

Taking in consideration these differences, for a project in which the reconfiguration engine is
2.2 Reconfiguration on FPGAs

Figure 2.1: Reconfiguration with using external and Internal Configuration Ports

meant to be part of the design that can is configured on the FPGA and is meant to be performed at run-time, the best option is ICAP. Since one of the main concerns when performing dynamic reconfiguration is the reconfiguration time, by having the configuration port more accessible to the internal design, loading a stored configuration to the target configuration port will meet less added clock cycles, thus reducing the reconfiguration time.

2.2.1.B Bus Macros

Bus Macros are originally used in partial Reconfiguration to route signals in and out of reconfigurable modules, in order to protect the reconfigurable modules and meet the necessary requirements previously discussed. The bus macros guarantees the connection point between the static and the reconfigurable modules so that the same static routing work for all reconfigurable module variants. The bus macros in the earlier releases of reconfigurable devices were provided by Xilinx and had to be instantiated for each module port.

A very important optimization in regards to partial reconfiguration, was the elimination of the need for explicit Bus Macros. This Transition was made with the release of ISE 12 of partition pins. The partition pins are automatically managed and have a much simpler structure than the tri-state buffer structure of the Bus Macros, while still meeting the requirements for Dynamic Partial Reconfiguration (DPR). The interface between a reconfigurable module and the static exterior through partition pins is represented in Fig. 2.2. It is possible to observe that the partition pins 1
2. Related Technology

![Figure 2.2: Partition Pins: Bridge between static and reconfigurable design](image)

and 2 represented create a clear registered interface between the input and output signals A and D, maintaining the signals B and C in the reserved reconfigurable region RM A1.

2.2.1. C Multiboot

The Virtex 5, 6, and 7 FPGA architectures supports MultiBoot [28][32][31], allowing the FPGA to selectively load its configuration from an attached configuration file containing two or more bitstreams. The configuration file is stored in a memory present in the FPGA device making the reconfiguration process much faster. In this mode, the FPGA application triggers a MultiBoot operation, causing the FPGA to reconfigure from a different bitstream. Once a MultiBoot operation is triggered, the FPGA restarts its configuration process as usual, and the FPGA clears its configuration memory and reconfigures from the stored full-configuration with a new bitstream file.

This function of the FPGA can be looked at in an abstract way as different layers of configuration, in which each bitstream stored in the memory represents a configuration layer. After the FPGA configures itself from the initial bitstream, the FPGA can trigger a MultiBoot event and swap the configuration content. The number of bitstreams supported by the Virtex 5, 6, and 7 FPGA MultiBoot feature depends upon the density of the target FPGA, to a maximum of four configuration images.

The MultiBoot feature can be triggered by different methods. A simple and flexible way to trigger the MultiBoot feature is to create a small state machine (internal to the FPGA) to send the IPROG command sequence to the FPGA configuration control via the ICAP primitive [28][32][31]. The IPROG command is sent through ICAP to trigger a MultiBoot configuration. After a successful configuration, the MultiBoot controller determines the start address of the next bitstream, sets a specific internal status register, and then issues a programming command via the ICAP interface[28][32][31]. The MultiBoot module includes both a state machine and the ICAP primitive. The user application signals a MultiBoot event to the module via a trigger command, while the state machine handles the required configuration data ordering, passing the configuration data to the FPGA configuration engine via the ICAP primitive.

After a trigger is received from the user application, the state machine sends the another
specific command sequence to the ICAP primitive, and the configuration is dynamically switched.

2.2.2 Altera Devices

Altera FPGA Architectures are coarse grained and combine LUT-based, product-term-based and memory into one device. Signal interconnections are provided by the Fast Track interconnect. The device consist of an array of MegaLAB structures in which each MegaLAB structure consists of a group of Logic Array Blocks (LABs), one Embedded System Block (ESB), and a MegaLAB interconnect, as depicted in Fig. 2.3. Each LAB contains a four-input LUT (look-up Table) that can implement any logic function of four inputs.

![Figure 2.3: Altera’s Reconfigurable Architecture](image)

Reconfiguration options are built-in to certain devices, like the APEX family. While the full reconfiguration process is performed similarly to the set-up configuration of the device, the partial reconfiguration is done by designing logic in RAMs, creating a truth table where functions with 7 inputs and 16 outputs can be created, that can be associated with other functions through the interconnections. After this logic is created, the system can re-write the content of the truth table at any moment, changing the configuration of a part of the design. This partial reconfiguration process can be done at run-time.

2.3 Host-FPGA Interface

One of the main drawbacks of multi-core systems and multi-device platforms is the communication bandwidth limitation, making the communication between the host and the FPGA relevant
2. Related Technology

to the performance of the system, since having a connection with low-bandwidth and low overall performance can put in jeopardy the advantages of using an FPGA as an accelerator for a given application. FPGAs have different interfaces available to communicate with the exterior, in this specific case, a main host CPU, which usually differ in protocol and throughput. JTAG can be looked at as one of those interfaces, since it can receive data from the host and affect directly the FPGA in its configuration. For this specific accelerator system, however, the type of data that is meant to be transmitted is meant to be bi-directional. The bidirectional element is specially relevant to the type of system in study, since the host must be able to send information to the accelerator system, while also must be able to receive information back from the accelerator.

Among these interfaces, an Universal Asynchronous Receiver/Transmitter (UART) and Ethernet interface are usually included in the development board including the FPGA, and can be used to establish a communication and transmit/receive data between Host and FPGA. In order to establish this communication, a specific protocol compliant controller is included in the system configured in the FPGA, and is limited by its latency and overhead, as well has the bandwidth of the interface.

As a solution for the required high-speed, high-bandwidth communication between FPGA and host, recent FPGA releases include integrated Blocks for PCI Express [27]. These integrated blocks provide the Host-FPGA system with a reliable, high-bandwidth and high performance solution for communications, such as multimedia server and mobile platforms. Similarly to the description of the UART communication, a specific controller (PCIe Bridge) must be instantiated to establish the interface between the PCIe transceivers and the FPGAs internal interfaces, and, therefore, the rest of the system.

In order to more easily establish the described connection using PCIe on Xilinx FPGAs, Xilinx provides an IP PCI Express Bridge, with communication buses compliant to the AXI4 connection protocol. In Fig. 2.4, the design for the PCIe AXI bridge is illustrated, in which it is possible to see the different modules that complete the controller, responsible for translating the information received from and transmitted to the instantiated PCIe Integrated block, into a data transfer to an Advanced Extensible Interface (AXI) bus. Represented in the figure, there are 2 separate bridges that connect to the AXI bus, which represent the Master and Slave AXI ports. With the Slave and Master Ports, peripherals in the system that are slaved by the Master port of the PCI Express AXI bridge, are consequently addressable by the Host in a similar way as an internal MicroBlaze processor would.

2.4 MB-LITE in Multicore Systems

The requirements defined for the present work include not only hardware related technology, such as the reconfigurable platform and the high-performance Host-FPGA communication, but
also include processing elements who are configurable and can execute the necessary application in an efficient way. In order to do so, the cores must be able to function at a high enough frequency, high enough performance, and include the necessary Instruction Set Architecture (ISA) to execute the defined architecture in an efficient way. Meeting these conditions, MB-LITE, is a highly configurable solution for multi-core processing solution. This processing core is based on the Xilinx MicroBlaze 32-bit RISC processor, which has full gcc compiler support and operates at high frequency for Xilinx FPGAs. This processing core, however, is very limited in terms of its flexibility in configuration. MB-LITE on the other hand, while it complies with the same ISA and shares the same virtues as Microblaze on a Xilinx FPGA, allows for more flexibility of configuration, while facilitating the connection to other modules and peripherals.

As it relates to a reconfigurable multi-core project, altering a chosen core with added modules or peripherals, would allow to create the different reconfigurable modules that can be interchanged depending on application necessities, making MB-LITE a contender for this type of systems. By creating a configuration of the core specifically geared towards a part of the application, it is possible to maximize the performance and energy gains of this adaptation.

### 2.5 Summary

When developing an adaptive hardware accelerator, an FPGA with built-in reconfiguration technology is a prime candidate for the target platform, due to the advances made in these de-
2. Related Technology

...
3

State of the art

Contents

3.1 Reconfiguration .............................................. 18
3.2 Multicore Heterogeneous Systems .......................... 23
3.3 Adaptive Processing system ............................... 23
3.4 Summary .................................................. 25
3. State of the art

In order to have an efficient system, the existing technology should be fully explored. To do this, the state of the art must be considered. In this chapter, an overview of the existing state of the art of the different needed components is presented. This analysis starts by looking into the work published on reconfigurable systems. Also, since a specially relevant component of the present work is the relocation approach, in this section, different solutions using it are discussed, in order to prove that it is a viable solution to enhance the reconfigurable system. As stated in the objectives, the reconfiguration concept is meant to be applied to a multi-core system in order to enhance it in terms of performance and energy consumption. It is then important to analyse different Multi-core based solutions and how they can relate to the requirements for the present work. A number of solutions are studied in this chapter.

Finally, since ultimately the way that the system is meant to be enhanced is by using reconfiguration to give it the ability to dynamically change its architecture, the present current dynamically adaptive solutions, and in which ways they are used in computational systems are presented. By studying advantages and disadvantages of the existing state of the art, an improved system capable of exploring the existing technology to its maximum potential can be derived. The final section describes these different adaptive solutions.

3.1 Reconfiguration

For this section, it is important to describe and differentiate between the different forms of reconfiguration processes. It is important to distinguish between full and partial reconfiguration, as well as static and dynamic reconfiguration. Since different authors compare the different approaches, and also propose different improvements to the technology, this information is first analysed.

Following this, systems using relocation are considered. This analysis serves to not only make the concept more clear, but to also specify the conditions in which it can be applied successfully, and also to present results and conclusions based of different applications.

3.1.1 Dynamic Partial Reconfiguration

Most of the recent publications on reconfiguration, focus on modern FPGA technologies. So, the related state of the art is developed counting on modern FPGA structures that support dynamic partial reconfiguration.

[14] studied dynamic partial reconfiguration, comparing the different design flows. The authors describe the distinction between 2 kinds of reconfiguration: static and dynamic. The main distinction lays is the fact that dynamic reconfiguration is performed is done at run-time, while the remaining system is left operating. The authors also describe DPR as a way to allow designs to adapt to changing algorithms, increase performance and reduce power, which are main objectives
3.1 Reconfiguration

of the present framework, as described.

The authors conclude that there is a direct relation between the size of a partial configuration, and the number of resources that exist in that configuration, which applies to most recent reconfigurable FPGAs and tools. It is then proven that a partial bitstream is smaller in size, reducing storage needs, and leading to a smaller configuration time when compared to full reconfiguration.

Also mentioned by [14], Xilinx suggests two basic styles of DPR on a single FPGA: the Difference-based partial reconfiguration and the module-based partial reconfiguration. Difference-based partial reconfiguration can be used when a small known change is made to the design. It is especially useful in case of changing LUTs equations or dedicated memory blocks content. The partial bitstream contains only information about differences between the current design structure (that resides in the FPGA) and the new content of an FPGA, by executing a XOR operation between 2 same-sized bitstreams correspondent to the hardware to be reconfigured. Module-based partial reconfiguration uses modular design concepts to reconfigure large blocks of logic. The distinct portions of the design to be reconfigured are known as Reconfigurable Modules (RMs), and exclusively enclose hardware resources meant for the RM, and exclude logic meant for the static part of the design and other RMs, while guaranteeing a safe interface between the RM and the exterior design.

In a direct application of the concepts mentioned, [17] studied an application of Partial Reconfiguration in a Virtex-5 device for image processing, also creating an FPGA based reconfigurable hardware accelerator system geared towards image processing algorithms.

Algorithms that require executing relatively simple operations on a large amount of data usually can be efficiently accelerated with modern FPGAs. So, the authors developed an accelerator composed of pipelines that are built with several reconfigurable execution units, that can either represent an element of a FIR filter or a Median filter, as illustrated in Fig. 3.1. This is possible due to the fact that the modules that compose the 2 filters both have the same kind of inputs and outputs in the system, and changing the internal configuration of the Partially Reconfigurable Regions (PRRs) only changes the response for the same input. This way, the accelerator can be quickly reconfigured depending on the algorithm, with the goal of having a configuration more dedicated to the computational needs of the given algorithm.

One of the main challenges of DPR is assuring that the design has the necessary conditions to be partially reconfigured and creating valid partial bitstreams that cannot compromise the function of the static part, and, worst case scenario, damage the device.

With the goal of facing those challenges, PARBIT [8] was developed with the ability to transform FPGA configuration bitstreams into partial bitstreams. In order to generate the partial bitstream file, PARBIT reads the configuration frames from the original bitstream and copies to the partial bitstream only the configuration bits related to the area defined by the user. This system, similar to the one described before, allows for a greater flexibility and a reduced memory for bitstream
3. State of the art

Figure 3.1: Connection of the reconfigurable regions.

storage and is valuable for the dynamic relocation aspect of SWINGER.


The initial approach for self-configuration is to statically floor-plan the entire FPGA at design time, breaking the overall FPGA into pre-defined partial reconfigurable regions and also suggest that each module that is to be executed on the FPGA should be placed and routed for each region of the chip, so that multiple versions of each module exist.

This work reaches the conclusion that partial self-reconfiguration is a powerful and increasingly practical capability of Xilinx FPGAs. However, as is explained, the full potential of this level of reconfigurability has yet to be harnessed due to the relative youth of the technology and supporting design tools.

3.1.2 Relocation

Using a configuration bitstream of a specific reconfigurable module in order to reconfigure a similar module in a different position in the FPGA area, is defined, in this context, as relocation. Relocation on a reconfigurable system is a concept specially difficult to implement, not because of the file manipulation necessary, but for the necessary conditions the system must meet from an FPGA floor-planning standpoint, and can potentially be a limitation to the overall project.

In the framework developed by [18], two different approaches to partial dynamic reconfiguration are compared. In the first method, applicable to Virtex-II and Virtex-II Pro devices, modules must occupy the full height of the device and the topology and connectivity are limited to 1D, which they term "direct dynamic reconfiguration". The second method demonstrates how 2D modular systems can be made tractable through the use of an innovative bitstream merging process and
reserved routing, enabling modules to be assigned arbitrary rectangular regions of the FPGA and relocated at run-time. Moreover, it is possible to achieve much greater flexibility in the connectivity of the system. However, the costs of these advancements are increased complexity and reconfiguration time.

This last approach faces the problem of having signals, that are not meant to be reconfigured, be located in an area that is reconfigurable. However, this can be bypassed by making sure that the configuration bits persist in the new configuration, so that the static can pass unperturbed.

The second major innovation in merge reconfiguration is in the way the partial bitstream is loaded. The current configuration is read back from the device and modified with information from the partial bitstream before being written back, minimizing the amount of memory required to store the bitstreams, the advantage that was introduced with the concept of relocation.

The main conditions that 2 modules must meet in order for relocation to be possible are: they must include the exact same number of reconfigurable resources of the FPGA, disposed in the same structure; and must have the interface points (configurable blocks and routing) between the reconfigurable module and its exterior (static design) in the same relative location. These conditions appear from the fact that the bitstream (particularly in Xilinx devices) first address a position on the FPGA fabric, and the data that is loaded after the address composes the active resources and the routing relative address. Therefore, if the modules are identical in size, topology, and interface the exterior in the exact same points, by changing the addressing, the same data on a bitstream correspondent to one module can be used to change another.

In order to ensure that these conditions are met, the solution proposed in [9] uses Xilinx tool PlanAhead, and FPGA Editor, to, respectively, floor-plan and position the modules correctly and ensure the exact position of the reconfigurable-static interfacing. Initially, the PlanAhead routes both the modules arbitrarily, with no specific constraints associated to them other than area constraints. After the initial routing, PlanAhead's PR2ucf tool is used to extract the position of the interfacing elements (partition pins) of the module 1, FPGA editor is used to obtain the routing constraints for that module, and, by changing the relative position of those constraints and applying to module 2, it is enough to meet the necessary constraints for relocation.

Other authors have exploited, in a more direct way, the concept of relocation on reconfigurable FPGAs [5][13]. All of them ultimately manipulate partial bitstreams, either dynamically as the bitstream is loaded to the configuration port (via a parser) or by processing the entire file before proceeding with the reconfiguration.

The solution presented in the previous section PARBIT [8] does take advantage of the way bitstreams address the FPGA and perform manipulation to create relocatable modules. However, this process is done via software introducing large timing overheads.

As mentioned previously, one of the main concerns when using dynamic reconfiguration on FPGAs is the timing overhead that can impact negatively the performance of the system. By
applying relocation to the system, it is crucial that that overhead is not further increased, which can potentially happen if the partial bitstream that is loaded to the chosen configuration port has to be changed beforehand. The parser proposed in REPLICA [10] is able to download a manipulated configuration bitstream with minimal additional overhead, while enjoying the advantages of having modules that are relocatable.

After ensuring that the modules meet the necessary conditions discussed previously, REPLICA analyses the bitstream structure of Virtex devices, concluding that between relocatable modules, only a set of addressing instructions need to be changed, along with Cyclic Redundancy Check (CRC) verification instructions. So, REPLICA is able to perform relocation dynamically by parsing each word that is being loaded to the configuration port, detecting a major column address word (MJA) and a CRC register command word, which have a unique specific designation in Virtex devices. If the word is detected, REPLICA changes it with the new addressing word, calculated by a dedicated module inside the filter. Another dedicated module is used to dynamically update the CRC word. These modules are present so that if a word is to be changed, the change is done by multiplexing the different outputs of the system, in order to not introduce an overhead to the process. This module is depicted in Fig. 3.2.

From this results a module capable of performing a dynamic relocation without increasing the reconfiguration time overhead, while reducing memory requirements for configuration bitstreams.
3.2 Multicore Heterogeneous Systems

The increasing demand for computational processing power and performance observed along the last decade has driven the development of heterogeneous systems composed of one or more processing devices. Such systems typically include a host General-Purpose Processor (GPP) and one or more accelerating devices, such as a Graphics Processing Units (GPU) or an FPGA, each integrating multiple PEs. While these systems allow for a significant application acceleration, it is of fundamental importance to improve the processing performance while minimizing the energy consumption. As a consequence, new and highly efficient processing systems must be developed.

Similar to the approaches taken to improve processing performance, energy consumption is also a relevant issue. Thus, to tackle the development of energy efficient platforms, many techniques have been proposed. This includes turning-off parts of the processor [22], dynamic selective de-vectorization [12] or using dynamic voltage and frequency scaling [19], [15] to decrease energy consumption whenever the computational requirements decrease. Recently, researchers have also turned to multi-core heterogeneous systems composed of high performance big cores and low-power small cores (e.g., the ARM big.LITTLE) to decrease the power consumption of the whole system, while providing similar processing performances [33]. These systems typically exploit a common ISA among all the cores, in order to facilitate task migration from the big to the small cores (and vice-versa), which allows for a fast and efficient switching between high performance and low-power scenarios, depending on the application requirements and constraints. Nevertheless, to adequately explore the existing resources on heterogeneous multi-core systems according to a given execution profile, an adequate scheduling needs to be performed.

To efficiently manage multiple running tasks on such systems, several scheduling algorithms have been developed. [2] identify key metrics that characterize the application under execution, including the core type that best suits its resource needs. [3] proposes the combination of static analysis and runtime scheduling to achieve an energy efficient scheduling on Intel’s QuickIA heterogeneous platform [2]. [23] proposes the usage of a Performance Impact Estimation as a mechanism to predict which workload-to-core mapping is likely to provide the best performance. Nonetheless, further processing power and energy efficiency gains can still be exploited by adapting the computing system to the characteristics of the running multi-threaded applications.

The next section presents an analysis on systems that exploit this gains by using reconfiguration to provide adaptability to the system.

3.3 Adaptive Processing system

An adaptive reconfigurable system is one that is dynamically aware of how the system is executing different tasks, aware of the configuration that is most suitable, and use the ability of recon-
3. State of the art

figure parts of the design into that configuration, and in that way adapt itself to the current needs of the application. Among the objectives of this dissertation, exploring the reconfiguration technologies to improve performance and energy efficiency are the main focus, and dynamic adaptive systems have the potential to answer the defined requirements. An analysis of implementations of the concept is herein presented in this section.

In the framework of ReKonf [16] a reconfigurable adaptive multi-core Architecture is introduced. ReKonf is a reconfigurable multi-core architecture that adapts to the characteristics of the running applications by setting its reconfigurable components at runtime, to morph into a suitable configuration. ReKonf contains a Configuration Controller that keeps track of core utilization, live cache utilization and cache sharing between threads, and takes reconfiguration decisions. The tiled architecture of ReKonf makes it highly scalable in realizing many-core systems with hundreds of cores. This work proves that such a system can be created and significantly change a multi-core system with more flexibility and increased performance. ReKonf was simulated using Super Escalar (SESC), a cycle accurate simulator to evaluate multi-core architectures. SWINGER applies the same concept but actually implementing it in an FPGA, using all its reconfiguration capabilities in order to obtain a similar result.

In the work presented by [21] an heterogeneous reconfiguration architecture in combination with a Quality of Service (QoS) driven operating system is proposed, in which the granularity of reconfiguration is chosen in accordance with the computation model of the task to be performed. The main focus of this model is to explore the different granularity levels of reconfiguration to create an adaptive design to the application. The model proposed was intended to use on mobile systems in order to meet the requirements of future low-power hand-held systems.

The author also describes how reconfigurable systems are being used in wireless devices. There are a few reconfigurable architectures that have been proposed for wireless devices, since most reconfigurable architectures were targeted at simple glue logic or at dedicated high-performance computing. Moreover, conventional reconfigurable processors are bit-level reconfigurable and are far from energy efficient.

The framework presented in [25] proposes a system called ReMAP, a reconfigurable architecture meant to accelerate and parallelize applications within an heterogeneous CMP. ReMAP pairs a specially designed Specialized Programmable Logic (SPL) fabric with multiple cores of a CMP. In addition to accelerating computation in traditional reconfigurable fabrics, ReMAP can be configured to facilitate multiple forms of fine-grained communication. In contrast to previous fine-grain communication approaches, ReMAP enables custom computation to be integrated with communication. Combining these multiple modes improves performance by 45% over an area equivalent system with larger cores and dedicated hardware communication. The authors also show that ReMAP provides better energy efficiency than can be achieved by using the area consumed by the SPL for either additional or more powerful cores, providing a 44% reduction in power con-
3.4 Summary

The defined objectives and requirements relate to many different reconfiguration technologies, techniques and approaches in order to achieve a multi-core processing system that dynamically evolves into a design that is faster for the type of tasks that are being executed.

A study of the research existent that relates to the present dissertation work is presented. This includes studies on partial reconfiguration on FPGAs, different applications and techniques, and, more specifically, modular dynamic partial reconfiguration. As it relates to the objectives of this dissertation, this section described the existent state of the art performing relocation, in order to create a flexible system that requires significantly less data-storage, and conclude that it is possible to adapt the solutions to a reconfigurable system without introduce additional timing overheads.

Also, in relation to the objectives, different authors have explored the advantages in terms of performance and energy consumption of multi-core heterogeneous systems, and how processing tasks are efficiently managed and scheduled in such systems.

Lastly, the study of adaptive systems that use dynamic reconfiguration in order to improve performance and energy saving, suggest that by efficiently managing the reconfiguration, taking into consideration reconfiguration times and the way it is performed in the specific system, suggest that implementing such systems can be greatly advantageous.

The analysis present in this chapter took all these different concepts and the way they impact positively computational systems in the different aspects. Hence, it is possible to suggest that by creating a multi-core heterogeneous system, and by using an efficient task scheduling and management, and at the same time providing it with the ability of dynamically adapt the hardware platform, an energy-aware and performance enhancing system can be created, and the study presented indicates that the proposed objectives for this dissertation are feasible.

The next chapter presents the proposed solution following the objectives and requirements for this thesis, taking into consideration the current related state of the art and the related available technologies.
3. State of the art
Proposed Solution

Contents

4.1 Dynamic many-core heterogeneous architecture .................................. 28
4.2 Summary ......................................................................................... 34
4. Proposed Solution

It is possible to use the reconfiguration technology in many different approaches in order to improve different processing systems. By analysing the technology and the different ways it is used in the previous chapters, it is now possible to derive a solution that complies with all the objectives for the present work. In order to prove the advantages of using reconfiguration to create an adaptive hardware system, this presents the proposed solution.

A possible solution includes using an accelerator of multiple parallel processing elements, in order to execute certain applications commanded by the host. Having an hypervisor layer on the host capable of sending and receiving information from the processing system in the accelerator, and capable of triggering an internal configuration engine, provides the system with a dynamic adaptability.

A description of the different elements of the proposed solution is presented, from the lower level of abstraction of the single processing core, to the Hypervisor that oversees the process from the highest point of abstraction, to the reconfiguration engine that makes the adaptability possible, and the different ways and processes that tie all these elements of the design together.

4.1 Dynamic many-core heterogeneous architecture

The proposed reconfigurable platform targets the acceleration of the parallel sections of one or more applications running on a host computer. A parallel program could be partially or fully migrated to the PEs instantiated on the FPGA device, depending on performance, power and energy requirements. Furthermore, typical applications can be divided into phases, where each phase has different requirements, such as memory bandwidth, instruction-level parallelism, data-level parallelism, and even functional unit requirements\[24\]. Under these circumstances, it is very important to allow the accelerating platform to adapt to the application(s) requirements in real-time. To tackle this issue, the proposed solution is meant to have the PEs exchange performance counters with the Host, giving it information to base a decision on whether or not a reconfiguration of a number of PEs is to take place, in order to increase performance or reduce energy consumption.

Since the proposed platform targets the acceleration of highly parallel applications, a scenario in which sending data to be processed in a dedicated platform can be advantageous, a high number of PEs need to be included. Naturally, depending on their complexity, and depending on the number of resources available on the used device, a different number of PEs may co-exist on the reconfigurable fabric at different time intervals. For example, while the reconfigurable fabric may have space to hold over 100 smaller PEs, a much smaller set of bigger PEs may be supported. Under such constraints, multiple small PEs need to be swapped out when reconfiguring the system to include a bigger PE. However, when smaller PEs are better suited to perform the required computation (e.g., because the application current phase requires no floating point operations or does not allow exploring data and/or instruction-level parallelism), trading a single big PE with a
small PEs may leave reconfigurable logic unused. As a result, if one considers a fine-grained reconfiguration granularity, a likely fragmentation of the reconfigurable logic will occur. To avoid such a situation, it is herein considered that the reconfigurable fabric is equally distributed into fixed-sized clusters, where reconfiguration always occurs at the level of a full cluster.

The architecture of the proposed platform is presented in Figure 4.1.

Figure 4.1: Proposed reconfigurable multi-core heterogeneous architecture.

Figure 4.2: Internal block diagram of a processing cluster.

This comprehends a scalable multi-core accelerator, implemented on an FPGA device, tightly coupled with a host computer through a high speed interconnection bus. In the considered case, the PCIe bus was the chosen communication interface, mainly because of its high-bandwidth, and high performance. These factors are specially relevant since data must be sent and received from the Hypervisor frequently, for the desired behaviour of the platform. The accelerator is composed of a dense and heterogeneous many-core processing structure, organized in several computing clusters each composed of multiple processing cores. The type of clusters loaded is controlled by a Hypervisor module, implemented in the host computer, which is responsible for constantly monitoring the accelerator execution and for issuing appropriate reconfiguration commands that adapt the internal architecture of each individual cluster to the instantaneous characteristics of
the application being executed. To keep a permanent evaluation of the performance that is being offered by each instantiated cluster, the Hypervisor keeps a record of a comprehensive set of performance metrics relative to each core.

A high-end GPP is used as host computer and is responsible for maintaining a Hypervisor module that monitors the accelerator's execution, providing metrics for the reconfiguration process, while assigning computationally intensive tasks to each of the available processing cores. The communication between the host and the accelerator is performed with the help of a device driver for the high performance interface of the FPGA device, in this case, PCI Express.

The following subsections detail the main components that incorporate the proposed reconfiguration framework, namely: (i) the Hypervisor module, which schedules the execution of kernels on the available processing cores and manages the framework reconfiguration by issuing commands to the Reconfiguration Engine; (ii) the Reconfiguration Engine performing the actual partial reconfiguration; (iii) the processing clusters, where the application kernels are executed; and (iv) the shared-memory and inherent synchronization mechanisms.

4.1.1 Hypervisor

The adaptive nature of the proposed heterogeneous structure is ensured by an Hypervisor module, which is implemented in the host computer (see Fig. 4.1). This software module provides a bridge between the reconfigurable hardware resources that are offered by the accelerator architecture and the application executing in the host. Accordingly, the Hypervisor is responsible for three important tasks: (i) assigning application kernels to the several allocated cores in the instantiated clusters; (ii) keeping a permanent record of the processing performance that is being offered by each allocated core in the accelerator; and (iii) issuing of appropriate reconfiguration commands to the Reconfiguration Engine, located in the FPGA accelerator. In order to provide full compatibility with existing application, the Hypervisor provides a complete abstraction of the underlying selection, adaptation, and reconfiguration process.

To adequately exploit the processing capabilities of the proposed reconfiguration framework, the Hypervisor software module determines the processing topology that is most adequate to each application kernel that is being accelerated. For such purpose, it receives and maintains a permanent record of performance counters measured at each core in the instantiated processing clusters. With such information, it first measures the actual operational and computational intensity of each kernel, and further estimates the total execution time, power and energy consumption on the currently available processing cores. After obtaining these values, it re-estimates the same metrics by considering the reconfiguration of a processing cluster.

To decide on the best possible action, namely whether to execute the processing kernel on a currently available core or to re-configure a cluster and only then perform the computation, a set of policies are available, namely: (i) overall minimization of the execution time; (ii) maximization of...
4.1 Dynamic many-core heterogeneous architecture

the processing performance while establishing a ceiling for the total power consumption; and iii) minimization of the energy consumption, while guaranteeing a minimum QoS, i.e., performance level.

To improve the energy and power consumption the Hypervisor may decide to entirely disable a given processing cluster, by reconfiguring it as an blank (inactive) box. This will result in powering off the corresponding region on the FPGA device. Several reasons may lead to such a decision, namely: a) when there are no further processing kernels to be executed on the accelerator (e.g., the application has insufficient parallelism), allowing to minimize the overall power consumption; b) when the power budget will be exceeded by the active processing clusters, thus requiring parts of the reconfigurable area to be disabled; or c) when the required performance level has been reached.

Finally, if the Hypervisor decision is the reconfiguration of a processing cluster, it then issues appropriate commands to the Reconfiguration Engine (at the FPGA fabric) to adapt the processing structure.

4.1.2 Reconfiguration Engine

In order to provide the use of the reconfiguration capabilities in current FPGA devices and to provide an abstraction layer between the Hypervisor and the actual FPGA reconfiguration process an reconfiguration engine is considered. The reconfiguration engine resides on a fixed region of the FPGA device, herein designated as the static region.

In order to provide a convenient abstraction layer between the Hypervisor and the actual FPGA reconfiguration process, a Reconfiguration Engine was implemented and allocated on a static region of the FPGA device, i.e. a region that will not be subject to changes. This reconfiguration engine receives the reconfiguration commands from the Hypervisor, such as the cluster ID and a particular cluster configuration, and orchestrates the entire reconfiguration process, namely the download of the partial configuration bitstream from the external memory and the upload to the reconfiguration port. Once completed, it signals the Hypervisor about the completion of the reconfiguration.

To accomplish this task, the Reconfiguration Engine comprises a micro-controller, connected to a dedicated reconfiguration module entitled SWINGER, handles the following tasks: i) acknowledging the reconfiguration commands issued by the Hypervisor; ii) providing the SWINGER module with the address of the partial bitstream to be loaded to the configuration port; iii) ensuring the complete and ordered transfer of the particular bitstream to the reconfiguration port; and finally signalling the Hypervisor that the reconfiguration process has completed.

To make sure that the remaining system continues to operate normally, while still allowing the scheduling of tasks to any other processing cluster (apart from the cluster being reconfigured), dynamic partial reconfiguration is used.
To assure the correct layout, the entire design is floor-planned in order to reserve areas for specific architecture modules (e.g., the Cluster Interconnection Network) on the FPGA configurable fabric. This assures that the reconfiguration engine can load the partial bitstream with no risk of the process affecting the on-going activity of the remaining cores, or their communication with the Hypervisor.

The Reconfiguration Engine serves as an interface between the Hypervisor and the FPGA reconfiguration port, receiving the reconfiguration commands from the Hypervisor, such as the cluster ID and configuration bitstream, and taking care of the complete reconfiguration process. Since partial reconfiguration is used, and the Hypervisor is operated by the host, the Hypervisor can continue operating normally, scheduling tasks to any processing core (apart from those belonging to the cluster being reconfigured). Once the Reconfiguration Engine finishes the reconfiguration process, it signals the Hypervisor such as to notify that it can now start scheduling tasks to the new set of cores.

This provides the host with a simple function for the reconfiguration process, allowing it to send an order and wait for it to confirm that the reconfiguration process ended successfully, and that it can communicate with the newly configured module. The proposed solution for the reconfiguration engine uses only one configuration port to perform reconfiguration, only one module can be reconfigured at one time. However, since the reconfiguration is partial, the host can still use its ability to communicate with the remaining processing clusters and the remaining design that is not under reconfiguration.

The Reconfiguration Engine resides in a static portion of the FPGA fabric and comprises a micro-controller (e.g., a MicroBlaze) that handles the following tasks: (i) acknowledging the reconfiguration commands issued by the Hypervisor; (ii) initiating the reading of the required reconfiguration bitstreams, which were previously prepared and stored in the Bitstreams Memory (e.g., implemented by means of a flash memory); (iii) applying the required reconfiguration, by guaranteeing that each individual bitstream word is sent through the ICAP bus of the FPGA device; and (iv) signalling the Hypervisor that the reconfiguration process has completed.

As stated before, the reconfiguration task is meant to be done via Hardware and not to be processed by the instantiated micro-processor. This Hardware accelerator for the reconfiguration task, communicating with the micro-controller, is meant to substitute the DMA in Figure. 4.1, and avoid using an IP controller for ICAP. This solution is meant to avoid timing overhead while loading the configuration files to the ICAP, while also saving resources that would be needed for an IP DMA and an IP ICAP controller. This accelerator is also meant to be able to perform dynamic relocation, by having the micro-processor provide all the information to the reconfiguration modules described, and having it change words from the bitstream that as it is written to the ICAP. As previously discussed, in order for the relocation process to be an asset to the system and reduce significantly the configuration memory storage, it must be assured that it does not introduce extra
4.1 Dynamic many-core heterogeneous architecture

timing overheads. Therefore, a parser approach to perform relocation is taken. The proposed parser is included in the input data from module, which filters each configuration word as it is read from memory, and filters out the relevant words that need to be changed to reconfigure the module in a different position of the FPGA. As a relevant word is detected, the output of the parser is the word altered to fit the new module to be addressed.

The entire design is floor-planned in order to reserve areas for the specific modules on the FPGA configurable fabric where the only resources allocated in the region correspond to that module, thus, ensuring that the reconfiguration engine can load the partial bitstream with no risk of the process affecting the on-going activity of the remaining cores, or the communication between them and the Hypervisor. The floor-planning takes in consideration the number of resources needed for every configuration, with a focus on minimizing the total area occupied, while still allowing for the Place and Route tool to find an adequate routing combination for all the different configurations. It also takes into consideration one of the necessary factors for the relocation aspect of the dissertation: ensure that all the modules occupy the exact same space in the FPGA, in identical relative positions, only requiring to change the addressing words in the configuration bitstream to a different relative position on the FPGA to perform dynamic relocation.

With this permanent adaptation of the reconfigurable fabric to the application under execution, it is possible to dynamically deploy a more dedicated and better suited computational structure to the computation being performed, thus obtaining a more energy efficient computational structures and with better performances. Moreover, since each computing cluster is well defined, it can also be configured has an inactive blank box. (This process is considerably faster than the other configurations, due to a smaller bitstream). By doing so, the logic corresponding to that particular region can be deactivated, resulting in a substantial reduction of the static energy consumption. Hence, in power critical systems, the amount of active logic regions can be finely controlled, in order to limit the peak power consumption. When more power is available, more logic regions and consequently more computational cores can be activated.

4.1.3 Processing clusters

As stated above, several core architectures, with different processing capabilities, can be devised and used in the proposed reconfigurable framework. The particular set of adopted core topologies can be fitted to different kinds of kernels that naturally depends on the computational complexity of the accelerated application. In particular, either programmable or fully-dedicated processing cores can be considered in each instantiated cluster, as long as adequate interface structures are provided in each core I/O interface.

With the reconfiguration process in view, the processing cores are grouped together in homogeneous clusters (see Fig. 4.2). Since each computing cluster is well circumscribed within equal sized regions of the FPGA fabric, the number of identical cores that can be instantiated in each in-
4. Proposed Solution

dividual cluster depends not only on the amount of hardware resources that are available for each cluster region, but also on specific restrictions of the adopted internal interconnection bus, as described in Section 5.2. Hence, given the different requirements in terms of hardware resources of each core topology, clusters of more complex cores contain fewer processing cores than clusters of simpler cores. Furthermore, in different prototyping FPGA technologies, the area reserved for a cluster region can take different sizes depending on the distribution and amount of resources in the FPGA. Hence, this design approach provides a higher degree of granularity for the dynamic reconfiguration procedure, alleviating the resulting reconfiguration overhead. Accordingly, instead of individually reconfiguring each core, all the cores in a given cluster are reconfigured at once.

The communication and interface mechanisms between the cores and the host computer are managed through a dedicated controller, associated to each core. This controller was specially designed to: i) start and monitor the execution of the core, ii) receive the kernel parameterization, and iii) transmit performance metrics to the Hypervisor module (at the host), for subsequent analysis.

Besides the previously described Reconfiguration Engine, the static portion of the FPGA fabric also includes a memory controller that provides a convenient shared-memory interface to the dynamically instantiated processing cores. Such shared-memory can either be implemented by using the internal memory resources within the FPGA, or even by using an external memory connected to the FPGA device.

In addition to this shared-memory space, a restricted set of dedicated memory positions, managed by an atomic access scheme, are also provided to the implemented cores, in order to support the implementation of convenient synchronization mechanisms in the parallel processing structure.

4.2 Summary

In this chapter, the proposed solution is presented towards the necessity of an application, and the formulation of the requirements for the system.

To apply reconfiguration as a mean to enhance a multi-core processing system, by having a reconfigurable hardware accelerator, the 3 main components of the considered solution were specified: Hypervisor, Reconfiguration Engine and Processing Clusters.

The main role of the Hypervisor, given that it is the entity that is using the accelerator to enhance a processing task it is meant to run, is to trigger the execution of the accelerator, and monitor the process, from the core processing to the dynamic reconfiguration. The solution presented for the Hypervisor describes a software module that receives performance counters from the proposed accelerator and makes decisions regarding the overall reconfigurable processing system.
The proposed reconfigurable processing system comprises clusters of configurable Micro-Processors, following the requirements in order for the processing system to be not only capable of carry the processing task for the Hypervisor, but also being configurable allowing for the dynamic reconfiguration targeting an optimal processing hardware configuration for the task. The Micro-Processors proposed are part of a cluster system, which compose the targets of the reconfiguration process.

The reconfiguration engine is an essential part of the system, given that the proposed solution is a direct application of dynamic reconfiguration as a proof of concept of the advantages of a dynamically adaptive hardware system. The main requirement for the Reconfiguration Engine is a low timing overhead, as well as the capability of performing dynamic relocation. The reconfiguration engine must also be able to communicate with the Hypervisor, which ultimately triggers the process. The proposed solution also involves a module which loads the configuration data from external memory into an internal configuration port, passing through a parser, meant to process the configuration file as it is being loaded, and perform the relocation.
5.1 Processing cores ........................................... 38
5.2 Communication and interfacing networks .................. 40
5.3 Reconfiguration ........................................... 42
5.4 SWINGER ................................................ 45
5.5 Hypervisor replacement policies ............................ 53
5.6 Summary ................................................... 54
5. Architecture Design and Implementation

Following the general description of the reconfigurable heterogeneous computational platform, this chapter details several key aspects in terms of its design and implementation on an FPGA device. This description initiates with the presentation of the control infrastructure that is tightly connected with each processing core. The communication infrastructure of the framework is also explained, including several aspects related with the intra-cluster communication bus, shared by all cores in each cluster, but also with the main system bus, connecting all clusters with the host computer, which was also implemented to adapt the already existent multi-core structure to the objectives of this dissertation. The host is connected with the main bus via a high performance connections, in particular, the PCIe and AXI interfaces.

A detailed description of the Reconfiguration Engine implemented is then presented, covering several aspects concerned with its internal architecture and implementation, with a special focus on how the relocation was successfully included in the engine. This chapter is concluded with a description of the considered replacement and reconfiguration policies that are implemented by the Hypervisor software layer, in order to optimize the achievable performance in terms of computational throughput and energy consumption.

5.1 Processing cores

In order to provide a wide variety in terms of the available operations and hardware complexity, three ISA were defined. As the base architecture, the 32-bit RISC MB-LITE soft-core was used for each processing core. This is a simple and portable processing structure, and is also a compliant implementation of the well known MicroBlaze ISA. Accordingly, it is possible to take advantage of the gcc support for the MicroBlaze processor. Moreover, the MB-LITE design is a low cost, highly configurable, and relatively easy to include custom modules. Its reduced hardware resources were also taken into account, providing a robust, yet low-resource, base core for scalable multi-core processing platforms. An representation of the Processing core used appears in Fig. 5.1(a).

The simplest deployed architecture (Type A) corresponds to the basic configuration of the MB-LITE core, i.e. with both the barrel shifter and multiplier deactivated, as the presence of those structures leads to significant hardware overheads and maximum operating frequency decrease . The second architecture (Type B) includes the referred structures, corresponding to the full MB-LITE architecture. For the third considered architecture (Type C), a full MB-LITE core supporting a single-precision Floating Point Unit FPU was considered.

To maximize the flexibility and versatility of the proposed reconfigurable structure, the heterogeneous accelerator architecture was designed by following a fully modular approach. In particular, each processing core was designed in order to be as independent as possible of the underlying PE architecture. In fact, the only imposed restriction to the architecture of the PEs is concerned with the provision of straightforward monitoring and communication interfaces to the
attached core controller. The core controller is responsible for the proper communication of each processing core with the entire system external to it.

Figure 4.2 depicts the main components that comprise this core infrastructure. Each core controller, depicted in Fig. 5.1(b) is implemented by a Finite State Machine (FSM), ensuring the coordinated execution of the following four sequential states in the core: i) wait for a configuration word from the host computer, while maintaining the PE in a reset (idle) state; ii) release of

Figure 5.1: Multicore framework components.
the PE from idle state (upon reception of a host message) and wait until the PE has finished reading the configuration word; iii) monitoring of the PE execution, while keeping record of a set of performance counters; and iv) transmitting the measured performance counters to the host and reverting of the PE to a reset state, upon assertion by the PE that the kernel execution has completed.

To accomplish this coordination, two separate communication interfaces are featured in each core controller and are depicted in Fig.4.2: a host communication interface and a PE communication interface. The host interface is composed of two unidirectional channels, compatible with the AXI-Stream protocol [1]. This interface is used by the core to receive configuration words from the host and to send packets to the host, containing the measured performance counters (as described above). The PE interface is dependent on the adopted PE architecture and is used by the controller to gather relevant information about the execution and performance of the PE. It is usually implemented by a simple interface, which can be either memory mapped to the PE, or connected through a custom I/O interface.

By default, the controller monitors the number of clock cycles that are required to execute a specific kernel. However, depending on the considered application, it is also possible to monitor other parameters, which are specifically customized based on the underlying architecture of the PE. Examples of additional counters currently implemented include the number of specific operations (e.g., integer divisions, FP additions or FP multiplications), counting branch mispredictions, identification of specific function calls (e.g., to detect operations that are performed through software libraries, instead of hardwired instructions). Hence, by providing this monitoring flexibility, it is possible to offer the Hypervisor with the necessary means to optimize any application in terms of the performance-energy balance, by easily tuning the reconfiguration policies to the specific characteristics of the available PEs and to the requirements of the application kernels to be accelerated.

Finally, depending on the accelerated application and on the adopted PE architecture, optional program and data local memories can also be accommodated inside each core. Such memory devices may either comprise an attached cache controller or may be characterized by a non-coherent access mechanism, comprised by a straightforward scratch-pad memory. Independently of the considered approach, such memories will represent the first level of the accelerator memory hierarchy.

5.2 Communication and interfacing networks

In order for the processing elements to form a proper multi-core system, capable of communicating with the Hypervisor, it is necessary to have dedicated interfacing networks for the task. This section describes the implementation of these important communication mechanisms.
5.2 Communication and interfacing networks

A fully compliant bus based on the AXI-Stream Protocol [1] was adopted for both the core interconnection and the cluster interconnection networks (see Fig. 4.2). In fact, despite the several available Xilinx IP Cores that implement an AXI-Stream interconnection fabric, it was decided to implement a fully compliant and custom interconnection, since not all the protocol signals are required and it is only necessary to create communication channels between the host and the cores. Moreover, this decision was taken with hardware resource overhead reduction in mind, since the complexity of the custom module is much lower than the original IP Core.

The implemented interconnection provides single-cycle communication mechanism between up to 16 peripherals which, in accordance to the protocol, corresponds to 16 AXI-Stream Master and 16 AXI-Stream Slave ports. Consequently, this interconnection features two independent unidirectional channels: a one-to-many channel and a many-to-one channel. The hypervisor can be seen as the “one” channel, while the processing elements are connected to the “many” channels. The first channel routes data signals to the corresponding core, by using a decoder driven by the 4-bit destination TDEST signal. The second channel is managed by a Round-Robin Arbiter, with a priority function based on the equations presented in [20], being the TVALID and TLAST signals used as the request and the end-of-burst acknowledge signals, respectively.

Given that each cluster can accommodate up to 15 processing cores, depending on the available area resources and core complexity, one of the interconnection ports reserved for outer-cluster communication. Hence, by daisy-chaining together a number of instantiations of the interconnection module through a specially designed bridge, it is possible to create a communication network with any number of levels.

To support the interconnection between the clusters and the host computer, a specific connection to the AXI4 bus was implemented using the Xilinx IP AXI DMA [29] core. This IP core provides a bridge between an AXI-Stream interface and an AXI4 interface, allowing the translation of a stream-based communication to a memory-mapped communication. The AXI4 bus used in the prototyping device connects to the PCIe external interface and, from there, to the host computer.

Despite the adopted simplifications in the communication protocol, it still ensures the required set of functionalities, as well as the flexibility to implement additional features. To initiate each core execution, the host sends one 32-bit word to the target core controller, containing the necessary information for the core execution. At the end of the execution, the core controller sends a packet to the host with a 32-bit word reserved for a return message, followed by a configurable amount of 32-bit words containing the measured performance counter values. The most-significant 8 bits of each packet word are reserved for the tuple \((\text{cluster\_id}, \text{core\_id})\), containing the cluster and core identifications.

Finally, to allow a flexible access of each core to the shared local memory in the cluster and to the external memory, a different interconnection module was derived from the previous one and
5. Architecture Design and Implementation

provided as a second layer of the interconnection network. Hence, while maintaining the same base structure, it is possible to obtain a single-cycle, arbitrated and shared-bus interconnection. This is achieved by exchanging the TDEST signal with an address signal, named TADDR, and by including memory and core interfaces, that see the unidirectional channels as a single bidirectional channel. Moreover, by including extra signals and providing the appropriate controllers, coherent cache-based memory hierarchies can also be implemented.

With this, it is possible to create an interconnection bus, capable of supporting and arbitrary number of levels, allowing the connection of multiple clusters to the host.

The proper communication between the Hypervisor and the considered processing elements, as well as assuring the required characteristics in order for them to be fully reconfigurable modules on the FPGA fabric, was developed for the present dissertation.

5.3 Reconfiguration

To provide the reconfiguration capabilities required to implement the proposed architecture, the Xilinx Virtex-7 FPGA was selected as target technology. This technology provides 3 different configuration ports to perform the reconfiguration, namely: Joint Test Action Group (JTAG), SelectMap, and Internal Configuration Access Port (ICAP). The main differentiating factor between them are the accessibility and the entity that is responsible for the reconfiguration process. Each configuration port also provides different data width and working frequency interfaces, resulting in different reconfiguration throughputs.

The JTAG interface is a configuration port external to the FPGA, commonly used to load the initial configuration into the device, directly interfacing it with the external flash memory, which stores the initial configuration file. This port provides a 16-bit configuration data-port, with a maximum frequency of 40 MHz. The SelectMap interface is also external, but with a higher reconfiguration throughout. Finally, the ICAP, which is essentially an internal version of the Select Map, has a 32-bit data port supporting a writing bandwidth of up to 100MHz. Being internal to the FPGA, it allows for the reconfiguration process to be controlled by an entity instantiated within the device itself. Since in the proposed framework the core reconfiguration procedure is to be controlled by the Reconfiguration Engine, inside the FPGA, with the host computer serving as the Hypervisor, the ICAP port presents itself as the most suitable interface. A lower reconfiguration time translates into a lower timing overhead, which is of special relevance since the resulting configuration files of the proposed modules are large (up to 2 MBytes).

Besides these reconfiguration means, the targeted FPGA technology also supports Multiboot reconfiguration. This reconfiguration capability offers the possibility to load different full configuration images in few cycles, allowing for different configuration layers to switch dynamically. However, this reconfiguration process only allows for full-device reconfiguration, not allowing for partial
reconfiguration. In contrast, partial dynamic reconfiguration allows the reconfiguration of specific processing groups (clusters) to be performed one by one, adjusting the system to the desired configuration.

To allow for a modular approach to the reconfiguration procedure that defines the several instantiated clusters, it is necessary to constrain each module to a specific and well defined region of the reconfigurable fabric of the FPGA. Accordingly, in order to ensure that the reconfiguration of each assigned cluster only changes a predefined region in the device, appropriate region delimitation has to be applied to each processing cluster. Hence, by evaluating the area resources required by each resulting computational cluster, it is possible to define the required reconfiguration region of each cluster, as well as the maximum amount of supported clusters in the device. Another limiting factor in mapping the clusters to the configurable logic is the location of both the PCIe module and the ICAP interfaces. These locations are conditioned, since they cannot be included in a region with an assigned reconfigurable module. These two modules are implemented by hard-cores, being allocated in specific regions of the device. Each of these regions is delimited using the floor-planner tool in order to reserve areas for each specific architecture module. This ensures that the reconfiguration engine can load the partial bitstream with no risk of the process affecting the on-going activity of the remaining cores or their communication with the Hypervisor.

In this particular implementation, the developed Reconfiguration Engine (illustrated in Fig. 5.2) is composed by a Xilinx ICAP controller (AXI HWICAP), an external memory controller connected to the on-board Linear Flash, a MicroBlaze micro-processor, an AXI Memory Controller connected to a 4KB FIFO, and an AXI PCIe Bridge, all interconnected by an AXI4 bus.

Since the reconfiguration of the clusters is triggered by the Hypervisor, in the host, a set of flags is used to communicate with the Reconfiguration Engine. These flags are implemented on a shared BRAM, accessible to both the host computer and the micro-controller in the internal reconfiguration control logic, via the PCIe Bridge. This shared memory is used to trigger the reconfiguration command, to inform the host computer of the reconfiguration conclusion, and as an indicator of which configuration is to be loaded. With this approach, the host computer does not need to actively wait for the conclusion of the reconfiguration process, being allowed to continue processing the information obtained from other clusters that might be still running. In this particular implementation, the configuration bitstreams are stored on the on-board Linear Flash. This Flash is used both for loading the initial full-configuration when the system boots (since it is a non-volatile memory) and to store the partial bitstream configurations. The option for using this Flash memory to accommodate the repository of the partial bitstreams (instead of using the DDR memory) is to avoid any impact in the computation throughput when both the Reconfiguration Engine and the processing cores would be simultaneously accessing the DDR memory. With this solution, it is possible to have a reconfiguration being performed while the computational cores access the main memory to access the data.
Upon receiving the command with the identification of the required configuration, the microcontroller of the Reconfiguration Engine issues read commands to the Linear Flash controller, in order to obtain the bitstream header. This header contains the information regarding the configuration bitstream size, thus obtaining the amount of bytes that need to be sent through the ICAP. After this initial phase, two approaches can be taken to carry out the actual reconfiguration. In the first option, the micro-controller controls each word that is transferred to the ICAP. This is performed by reading the 16-bit words from Flash memory and by packing them into 32-bit words, before sending them to the ICAP port. This option has the disadvantage of reading the Flash word by word, and of requiring the intervention of the micro-controller for each word transfer. In the second option, the data is directly transferred from the Flash memory to the ICAP port. The micro-controller only has to setup a DMA descriptor and order the DMA transfer to start. Although this last option requires the presence of the DMA engine, it allows the usage of the Flash burst mode, as well as a faster merge of the 16-bit words into 32-bit words, required by the ICAP. This results in much higher transfer rates and consequently faster reconfigurations. Once the transfer is completed, the Reconfiguration Engine controller signals the host computer, informing it that the requested reconfiguration is completed. With this approach, only one reconfiguration command can be issued at a time, since the previous reconfiguration must be concluded before a new reconfiguration command can be issued. Besides simplifying the reconfiguration procedure (avoiding the presence of reconfiguration command queues), it is worth noting that it does not significantly affect the resulting performance, since the contention to access the Flash memory would prevent greater reconfiguration throughputs.
5.4 SWINGER

One of the main concerns when considering an adaptive processing system that uses reconfiguration at run-time, is the amount of time necessary to actually carry through with that process, and how it can impact negatively the on-going tasks. The reconfiguration can be done via software, having a processor running the task of reading the configuration data and loading it into ICAP. This, however, introduces an excessive number of wasted cycles, from the execution of the software to the reading and writing of the configuration words, creating a large timing overhead for the process. For this reason, instead of having a “middle man” just to conduct a word from external memory into ICAP, having a dedicated module for the task would reduce those wasted cycles, improving the system. In this chapter, the implementation for the proposed reconfiguration module is presented.

A module dedicated to performing the reconfiguration of the processing clusters was implemented, SWINGER, and is herein described. The main reason why having a dedicated hardware for this specific task is important to implement in the proposed solution, is the fact that by having the instantiated micro-processor processing and loading the configuration data, a relevant amount of timing overhead is introduced. This is not only due to the fact that this process is being done by software, but also the fact that each configuration word is read individually, not using burst support for AXI4 buses, and is then written to the Xilinx IP HWICAP, introducing further wasted cycles. Those lost cycles when applied to a bigger configuration file, translate into timing overheads that could put in jeopardy the overall objectives for this thesis. SWINGER would also trump the second solution previously described: the use of a DMA. By having an hardware module capable of performing the specific task of fetching a number of words from memory and loading directly into ICAP, a lot of resources would be saved, and possibly power, since DMA is generally a large module in terms of its use of FPGA elements.

The different areas of the SWINGER module are represented in Fig. 5.3, which includes the ports that the module uses to communicate with the exterior, and a clear separation between the part of the design responsible for reading the specific configuration bitstream and the part responsible to writing to the ICAP. This separation is represented by two distinct State machines responsible for the control of each corresponding side of the main module.

The separation of the two sides described is done by an asynchronous FIFO, in order to minimize the overhead to the reconfiguration process and use the maximum throughput allowed by ICAP specifications. This FIFO is implemented as dual-clock in order to ensure that the side of the module responsible to load the words to the ICAP can access configuration words at no higher frequency than 100MHz, which is the maximum frequency of the ICAP, while allowing a higher frequency to write the words to the FIFO by the state machine who is getting the configuration words from memory. By having two state machines dedicated to the tasks of writing and reading
words to and from the FIFO modules, as long as there is a word in the FIFO to be read, the right state machine is constantly working. Thus, as long as the reading side of the module can ensure that at all cycles there is a word available to be written to the ICAP on the FIFO, it is possible to load a 32 bit word into ICAP every clock cycle, and achieve the maximum possible reconfiguration performance.

In order to ensure that the FIFO has words to be written at all times, the module must be able to obtain the configuration words with as little latency as possible. This also provides a clear separation between the two tasks, allowing them to achieve the best performance possible to each of them, while not interfering with the other. As described before, since the configuration bitstream is too large to keep in the internal FPGA memory elements, which would have given the fastest access, they must be stored externally. However, it is possible to set up burst transfers through AXI4 bus, where by setting an address and number of words, words are loaded in each clock cycle. Thus, by using an AXI4 interface in the SWINGER module and connecting the port to the AXI4 bus, it is possible to set a fast transfer of the bitstream file with a low overhead. This AXI4 interface corresponds to an AXI Master port, since the module is dedicated to send read instructions to the AXI bus with the address of the external memory.

On the proposed solution, a micro-processor is present in the FPGA instantiated design, communicating with the Hypervisor through PCIe. In order for the micro-processor to be able to communicate with the SWINGER module, and in order to give it information on the bitstream that needs to be loaded to the ICAP (its address and size), a communication port must be chosen. Since this communication is only needed for a couple of configuration words which then trigger a much longer reconfiguration process, the efficiency in terms of number of cycles needed for
communication is less relevant in relation to the bigger process. The choice of communication port, a Fast Simplex Link (FSL) bus, was based on less complex communication and the number of necessary resources.

An option would be to make SWINGER a slave connected to the same AXI4 bus accessed by the MicroBlaze and PCIe. However this would involve an instantiated AXI Slave interface in the module and a complex state machine to comply with the AXI protocol, which would significantly increase the complexity of the SWINGER core only to establish simple occasional communications with the MicroBlaze. MicroBlaze processors can be configured with FSL ports, which is a low complexity solution to communicate with hardware accelerators. From the MicroBlaze side, FSL functions are available which make the task of sending a word to the accelerator through the FLS bus simple, which can be used to send bitstream address, size (number of words to be loaded) and a trigger message. On the SWINGER side, by including the FSL ports and by including a state machine that waits for words on the FSL bus and stores the different information on designated registers, the communication can be establish with no extra resources needed but a more complex state machine (no dedicated interface blocks). When SWINGER receives the trigger message from the MicroBlaze, the reconfiguration process starts as described, and once it completes, it sends a message to MicroBlaze (which is waiting for a word from the fsl bus), to signal the completion back to the Hypervisor.

### 5.4.1 Read-Bitstream State Machine

The most complex state machine in the proposed module, is the one responsible for fetching the configuration words from the external memory, as it not only has to deal with the signals to trigger a correct AXI4-burst transaction, but also receive commands from the Micro-Controller through the FSL bus, and act accordingly. The Fig. 5.4 shows the different states that perform these tasks.

The state machine receives information from the AXI Interface, FSL bus, and FIFO module, in order to keep an efficient control over all the different states, and these divide the state machine into 2 main groups: AXI transaction, Reconfiguration and parser control states, and FIFO states.

Initially, the state machine remains in IDLE, waiting for a word to be present in the FSL bus, by observing the signal FSL-M-Exist, indicating that in the FSL-M-Data there are control words to take in. These words are received in a sequence and stored in specially allocated registers, so that they are available for the different sub-modules. The words, in order of reception and register storage: The address in external memory, which is used for the AXI burst read interface, and incremented as the configuration words are loaded in burst groups; the size of the bitstream that is to be loaded, so that it can be decremented as the words are loaded, in order to determine when the entire file as been read from memory; relative position of the module on the interface, so that it can be used by the relocation parser to change the words to address a different location.
5. Architecture Design and Implementation

Upon receiving the last control word from the FSL bus, the bitstream transfer from external memory is triggered, and the following states control this transfer, complying with the necessary protocol for a AXI burst transfer, and is explained in further detail in the following subsection.

5.4.2 AXI Interface and Burst transfer

Since a single AXI transfer would require several control cycles for the state machine to set up the transfer, this cycles added to every word that is meant to be loaded to the FIFO module would create an undesired large latency, and would not be taking full advantage of how efficient

Figure 5.4: State Machine responsible for communication with Microblaze, reading configuration data from external memory
AXI transfers can be. This is why, the Read-Bitstream State machine sets up burst transfers, in which an address and a number of words is set up, and after all the different set-up cycles of the AXI protocol, that number of words are transferred in a serial sequence of cycles.

Hence, the Axi Transfer section of the State Machine in 5.4 is composed by the states responsible for setting up and receive the burst transfers, and load them to the FIFO. However, the bursts cannot be loaded carelessly into the FIFO module, as a burst transfer can potentially overflow a full FIFO, where configuration words could be lost, and the reconfiguration process invalidated. This is why, it is necessary for the State machine to have the information about the number of words available to be written to the FIFO, and only schedule a burst if at least one burst can fit into the module, assuring that no words are lost in the process.

The main factor for selecting the burst transfer size, is the goal of having words on the FIFO at all times, so that it is never empty, making the reconfiguration as low-impact on the system as possible. By having the maximum burst transfer size, added to the advantage of instantiated an asynchronous FIFO, with a higher writing frequency, it is possible to keep the FIFO full, and mitigate the negative effect of the AXI transfer set-up cycles.

5.4.3 Write-ICAP State Machine

The second state machine on the read side of the FIFO module, is composed by a simple mechanism that checks the bitstream word availability, and fetches the word to the ICAP. The initial state of the state machine takes as input the EMPTY signal from the FIFO, and has outputs connected to the ENABLE and WRITE signals of the ICAP module. In order to meet the main objective of loading a word in each cycle, to minimize the reconfiguration timing overhead, the output data of the FIFO is directly connected to the input data of the ICAP module. This way, if the state machine detects the EMPTY signal at 0 at the beginning of a clock cycle, it activates the WRITE and ENABLE control signals of ICAP. Similarly, when the EMPTY signal is 1, the state machine ensures that the ENABLE signal is disabled, ensuring that only valid words are written to the ICAP for a solid reconfiguration process.

Given this process, the reconfiguration time is defined by the number of clock cycles necessary to write all the information of the bitstream configuration file to the ICAP module, resulting in the following equation:

$$T_{ \text{reconf} } \approx \frac{N_{\text{writingcycles}} \cdot \text{Bitstream size (Bytes)}}{4f_{\text{clk}}}$$  \hspace{1cm} (5.1)

The reconfiguration time is dictated by the size of the bitstream and the number of writing cycles, being that the writing cycles are the only factor that can be reduced. By optimizing the reconfiguration process to the limit in which the read cycles for the DDR controller (with a 32 bit data bus) is reduced and it is possible to load a 32 word in each clock cycle (by minimizing the value of writing cycles), it is then possible obtain a reconfiguration rate of 3.2 Gbit/s. Attaining
5. Architecture Design and Implementation

this reconfiguration rate, reconfiguring a cluster (with a size of approximately 2MBytes) would introduce, approximately, 5 ms of reconfiguration time.

5.4.4 Relocation Parser

As described in the proposed solution for the relocation aspect of the present dissertation, adding the relocation option to the system allows to optimize the memory storage necessary for the different configuration files. The main role of the parser is to filter each configuration word as they come through from the AXI data bus before they are written to the FIFO module, and make changes to certain words as appropriate. By taking this parser approach, instead of having a process dedicated to processing configuration files, which would add undesirable timing overheads to the reconfiguration, a parser would be able to change the file dynamically as it is being loaded to the ICAP, not affecting the time required to do so (important requisite previously discussed).

In order for the parser to adequately filter a bitstream configuration of an RM to configure another, as implemented by [10], it must have the information regarding the different modules and their position on the fabric, as well as taking into consideration the device-specific bitstream format.

In Fig. 5.3 representing the overall architecture of SWINGER, it is possible to observe the Relocation Parser as a filter imputing the data coming from the external memory through the AXI interface, sending an output data bus with the same width, to the FIFO module. The connections that can be seen between the parser and the State-Machine, represent the information that has to be provided for a correct relocation, coming from the Micro-Processor. Values of the Column address and the line address of the module to be reconfigured are stored in registers, and available to the parser, in order to change the incoming FAR words to a new address of the intended module position.

<table>
<thead>
<tr>
<th>Address Type</th>
<th>Bit Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Type</td>
<td>[25:23]</td>
<td>Valid block types are CLB, I/O, CLK (000), block RAM content (001), and CFG_CLB (010). A normal bitstream does not include type 010.</td>
</tr>
<tr>
<td>Top/Bottom Bit</td>
<td>22</td>
<td>Select between top-half rows (0) and bottom-half rows (1).</td>
</tr>
<tr>
<td>Row Address</td>
<td>[21:17]</td>
<td>Selects the current row. The row addresses increment from center to top and then reset and increment from center to bottom.</td>
</tr>
<tr>
<td>Column Address</td>
<td>[16:7]</td>
<td>Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.</td>
</tr>
<tr>
<td>Minor Address</td>
<td>[6:0]</td>
<td>Selects a frame within a major column.</td>
</tr>
</tbody>
</table>

The way that the reconfigurable modules are addressed in a partial configuration bitstream depend on the device, since some devices address entire columns at a time, while others address in individual sections, usually referred to as tiles. For the modern Virtex-7, a FAR (Frame Address
Register) word exists addressing each tile the module occupies, for each type of resource that needs to be activated. The format of the FAR word for the target device is represented in Table 5.1, where different sections for addressing are necessary. Each FAR word, addressing one single tile, must have the information of a Minor address and a Major address, as well as the type of resources that are being configured, since all the data words that follow will activate that type of resource, starting from the position addressed.

Having the information of the Major Address and Minor Address provided by the micro-processor stored, the Relocation parser analyses each incoming word, comparing it to the word 0x3002001, which in the target device is the command used to address the FAR register, meaning that, following this command, the incoming word is the FAR word meant to be modified. Waiting for the command is an effective way for the parsing to detect the words that are meant to be changed with the new addressing values.

Following the FAR word, all the data words are forwarded by the parser without alteration, given that the design was floor-planned to allow the relocation to be performed in this fashion, as previously discussed, and as proposed by[10].

In order to address the potential issue of having a data word having the same value as the command to address the FAR register, which would cause the parser to view the following word as a FAR word and changing it, it is important for the parser to be able to distinguish between these 2 cases. In order to do so, the relocation parser also detects the command 0x30004000, which is followed by the value of data words that are present in the bit file, following the FAR word. By registering that value, and by having an Internal counter that increments for every parsed data word, it is possible to guarantee that the exact number of data words are forwarded by the parser unchanged by the parser, eliminating the potential conflict.

5.4.4.A Floor-planning

An essential component to ensure that, not only the maximum number of reconfigurable processing clusters can be instantiated in the configurable fabric of the FPGA, but also that the reconfigurable modules meet the requirements previously described for the relocation process, is the floor-planning process, which precedes the final bitstream generation.

Adding to the symmetry concerns associated with the relocation process, it is also necessary to ensure that the modules are constrained in a way that allows the routing tool to be able to find routes for all the configurations (dynamic and static), while not compromising the maximum functioning frequency of the system (which could happen if the routing tool is too restricted in terms of resources), which normally results in higher reserved resources than what is architecturally necessary for a reconfigurable module.

Also, the way that the target FPGA device is built, the way that the configuration files address it, and the FPGA configurable fabric is not symmetric, limiting the number of options for the
5. Architecture Design and Implementation

floor-planning of the present system. The present system requires a number of reconfigurable modules, which require a large number of resources, occupying more than one tile. Added to all these factors, the routing tools often do not allow reconfigurable partitions to share the same configuration tile, and a module cannot be instantiated on top of ports that are used for communication or reconfiguration (in the specific proposed solution, PCIE module and ICAP).

Fig. 5.5 depicts the floor-planning limitation for this specific solution, using a Virtex-7 FPGA, and illustrates the mentioned limitation, by showing the way that the fabric is divided, and where the PCIE and ICAP modules are located.

For the purpose of the present framework, 2 different approaches are taken to the floor-planning: i) Fit the highest number of reconfigurable processing clusters in the device; ii) find different configurations for the processing clusters (number of cores per cluster) in order to explore the relocation aspect of the framework to the fullest potential that the target device allows.

The first approach uses the number of cores that in a reconfigurable cluster as previously described, and fits the maximum number of RMs in the FPGA configurable fabric, while not jeopardizing the performance of the static system.

Figure 5.5: Floor-Planning limitation in target device.
The second approach, on the other hand, represents the floor-planning with the main focus on maximizing the number of modules that can be relocated, resulting in less configuration memory necessary, while potentially compromising the advantages of the adaptive system. Both approaches are further detailed in the Annex A.

5.5 Hypervisor replacement policies

As mentioned in Section 4.1.1, the Hypervisor software layer at the host computer implements a set of optimization policies targeting different application requirements and constraints. In the considered implementation, such policies provide three cumulative levels of optimizations: i) execution time/energy optimization; ii) power-ceiling dynamic constraints; and iii) power saving with a minimum predefined and assured performance level.

The algorithm presented in Fig. 5.6 Algorithm 1 implements a runtime performance prediction routine. In order to decide when it is advantageous to reconfigure a given cluster, before executing the required kernel, the algorithm performs two distinct steps. Initially, it searches for a configuration allowing for higher gains in terms of performance, energy, or power consumption. This decision also takes into account the reconfiguration overhead. If such configuration is found and if the required time to complete all other scheduled reconfigurations is lower than executing the kernel with the current configuration, the targeted cluster is put in a waiting list for reconfiguration at the host side. It is worth noting that this algorithm can also be used for energy and power optimizations, by changing all time-base variables to energy variables.

The algorithm presented in Fig. 5.6 Algorithm 2 adds an extra level of optimization to the previous algorithm, by introducing a dynamic power-ceiling constraint. This power constraint can change at runtime, depending on the dynamic requisites of the application. Based on the total power budget of the system at a given time, the algorithm tries to turn off clusters that are inactive until the power constraint is met. Each cluster is turned off by reconfiguring it to a blank (inactive) box, which turns off the FPGA logic in the considered cluster area. As soon as the power budget increases, an idle or turned-off cluster is searched, to be analysed with the algorithm presented in Fig.5.6 for the current kernel chunk. Under this assumption, each cluster is only reconfigured if the power overhead for the reconfiguration procedure and for the new configured architecture does not violate the power ceiling constraint.

The algorithm presented in Fig. 5.6 Algorithm 2 further adds a new level of optimization, with a minimum assured performance policy. This algorithm tries to minimize the power consumption while maintaining a given minimum performance level. Initially, the algorithm presented in Fig. 1 is executed to check for the performance requirements of the new kernel chunk. If a reconfiguration is required for an idle cluster, the reconfiguration overhead and the future performance of the new cluster architecture is checked and it only reconfigures such cluster provided that the minimum
assured performance is met. Finally, the algorithm tries to turn off idle clusters to lower the total power consumption, as long as the minimum performance is met for the current kernel.

5.6 Summary

Following the description of the proposed solution for this dissertation, which focuses on an hardware adaptive system using dynamic reconfiguration as a way of obtaining performance and power gains, the description of the carried implementation was presented.

The described system required different layers of implementation: from the simple unit PE, to the interface and communication between them and the Hypervisor, to the reconfiguration process itself. The different elements were all implemented under the clearly defined requirements, dealing with the functionalities of the available platform and tools.

The Hypervisor entity responsible for overseeing and ensuring the adaptive nature of the proposed heterogeneous structured was implemented in the host computer. Communicating through the PCIe bridge with the mani-core reconfigurable system, it is capable of receiving performance information from the PEs, and triggering a reconfiguration process towards an adapted configuration, making decisions based on different implemented replacement policies for the reconfigurable clusters in the system.

These replacement policies enforced by the Hypervisor have as an important factor, the reconfiguration time to ensure that it is advantageous to reconfigure a part of the system. In order to decrease timing overhead when performing reconfiguration of the large processing clusters, a dedicated module was implemented. The SWINGER module is connected to a instantiated Microblaze, and, together with the external memory storing the configuration words, comprise the Reconfiguration engine.

In order to push to the limit the timing limitation of the chosen configuration port (ICAP) a dedicated module was implemented. SWINGER implements via hardware the task of loading configuration words from external memory storage, and is commanded by an instantiated Microblaze module, obtaining a much better timing overhead result then executing via software. The implemented dedicated hardware module can be connected to any Microblaze processor and its simple functionality allows for a simple efficient way of performing dynamic reconfiguration using external memory and the ICAP.

Lastly, facing the possible limitation of increased configuration memory storage in a case when many different configurations are considered, a dedicated relocation parser was included in the implemented module. By ensuring that the system is correctly floor-planned into the FPGA fabric, by having equal modules instantiated in equal relative positions of the device, it is possible to reconfigure a module using the configuration bitstream of another. In order to introduce this enhancement without deteriorating the reconfiguration efficiency of the dedicated reconfiguration
Algorithm 1: Execution time optimization

Input: kernel, chunk, current_arch
Global Variables: reconf_time, inprogress_recons
Output: new_arch or 0

\[\text{cups} \leftarrow \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{current_arch})\]
\[\text{curr}_\text{time} \leftarrow \text{cups} \times \text{chunk}\]

new_arch \leftarrow 0
\[\text{time} \leftarrow \text{curr}_\text{time}\]
foreach arch do
\[\text{cups} \leftarrow \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{arch})\]
\[\text{test} \leftarrow \text{cups} \times \text{chunk}\]
\[\text{if} \ \text{test} \times 2 + \text{reconf}\_\text{time} < \text{time} \times 2 \ \text{then}\]
\[\text{new_arch} \leftarrow \text{arch}\]
end
end

if new_arch > 0 and (curr_time < time \times inprogress_recons \times \text{reconf}\_\text{time})
  new_arch \leftarrow 0
end

return new_arch

Algorithm 2: Power-ceiling algorithm

Input: kernel, chunk, idle_list, max_power, rec_list
Global Variables: reconf_time, static_power

\[\text{curr\_power} \leftarrow \text{static\_power}\]

foreach cluster do
\[\text{if} \ \text{cluster}\_\text{arch} = \text{EMPTY} \ \text{then}\]
\[\text{curr\_power} \leftarrow \text{curr\_power} + \text{get}\_\text{power}(\text{cluster}, \text{cluster}\_\text{arch})\]
end

foreach cluster in idle_list do
\[\text{if} \ \text{curr\_power} > \text{max\_power} \ \text{then}\]
\[\text{curr\_power} \leftarrow \text{curr\_power} - \text{get}\_\text{power}(\text{cluster}, \text{cluster}\_\text{arch})\]
\[\text{rec\_list} \leftarrow (\text{cluster}, \text{EMPTY})\]
end

\[\text{cluster} \leftarrow \text{find}\_\text{idle\_or\_off}\_\text{cluster}\()\]
\[\text{if} \ \text{cluster} = \text{NULL} \ \text{then}\]
\[\text{new\_arch} \leftarrow \text{time}\_\text{optimization}(\text{kernel}, \text{chunk}, \text{EMPTY})\]
end
else
\[\text{new\_arch} \leftarrow \text{time}\_\text{optimization}(\text{kernel}, \text{chunk}, \text{cluster}\_\text{arch})\]
end

if cluster \neq \text{NULL and new\_arch} > 0 then
\[\text{test} \leftarrow \text{curr\_cups} - \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{cluster}\_\text{arch})\]
\[\text{cups} \leftarrow \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{new\_arch})\]
\[\text{test} \leftarrow \text{test} + (\text{cups} \times \text{chunk} + \text{reconf}\_\text{time})/\text{chunk}\]
if test > min_cups then
\[\text{rec\_list} \leftarrow (\text{cluster}, \text{new\_arch})\]
\[\text{curr\_cups} \leftarrow \text{curr\_cups} - \text{test}\]
end

foreach each cluster in idle_list do
\[\text{cups} \leftarrow \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{cluster}\_\text{arch})\]
\[\text{if} \ \text{curr\_cups} - \text{cups} < \text{min\_cups} \ \text{then}\]
\[\text{continue}\]
end
\[\text{curr\_cups} \leftarrow \text{curr\_cups} - \text{cups}\]
\[\text{rec\_list} \leftarrow (\text{cluster}, \text{EMPTY})\]
end

Algorithm 3: Minimum assured performance algorithm

Input: kernel, chunk, idle_list, min_cups, rec_list
Global Variables: reconf_time

foreach cluster do
\[\text{if} \ \text{cluster}\_\text{arch} = \text{EMPTY} \ \text{then}\]
\[\text{curr\_cups} \leftarrow \text{curr\_cups} + \text{load}_\text{kernel}_\text{stats}(\text{cluster}\_\text{kernel}, \text{cluster}\_\text{arch})\]
end

\[\text{cluster} \leftarrow \text{find}\_\text{idle\_or\_off}\_\text{cluster}\()\]
\[\text{if} \ \text{cluster} = \text{NULL} \ \text{then}\]
\[\text{new\_arch} \leftarrow \text{time}\_\text{optimization}(\text{kernel}, \text{chunk}, \text{EMPTY})\]
end
else
\[\text{new\_arch} \leftarrow \text{time}\_\text{optimization}(\text{kernel}, \text{chunk}, \text{cluster}\_\text{arch})\]
end

if cluster \neq \text{NULL and new\_arch} > 0 then
\[\text{test} \leftarrow \text{curr\_cups} - \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{cluster}\_\text{arch})\]
\[\text{curr\_cups} \leftarrow \text{curr\_cups} - \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{new\_arch})\]
\[\text{test} \leftarrow \text{test} + \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{new\_arch})/\text{chunk}\]
if test > min_cups then
\[\text{rec\_list} \leftarrow (\text{cluster}, \text{new\_arch})\]
\[\text{curr\_cups} \leftarrow \text{curr\_cups} - \text{test}\]
end

foreach each cluster in idle_list do
\[\text{cups} \leftarrow \text{load}_\text{kernel}_\text{stats}(\text{kernel}, \text{cluster}\_\text{arch})\]
\[\text{if} \ \text{curr\_cups} - \text{cups} < \text{min\_cups} \ \text{then}\]
\[\text{continue}\]
end
\[\text{curr\_cups} \leftarrow \text{curr\_cups} - \text{cups}\]
\[\text{rec\_list} \leftarrow (\text{cluster}, \text{EMPTY})\]
end

Figure 5.6: Considered Hypervisor replacement policies.
module, the parser changes the relevant addressing words as they are loaded into the module,
rather than processing the bitstream information prior to the beginning of the transfer process,
resulting in no relevant additional overhead.
6. Evaluation

To evaluate the performance of the proposed many-core reconfiguration framework, the complete system was prototyped in a Xilinx Virtex-7 FPGA (XC7VX485T), connected through an 8x PCIe Gen 2 link to a personal computer equipped with an Intel Core i7 3770K, running at 3.5 GHz. The Synthesis and Place&Route procedures were performed using Xilinx ISE 14.5. On the Intel Core i7, cycle accurate measurements were obtained by using the PAPI library.

6.1 Evaluation Benchmark

To demonstrate the performance and energy-aware capabilities of the proposed adaptive framework, an architecture composed by 105 processing cores was prototyped and evaluated. The considered benchmark is an application composed of 4 phases, each one corresponding to a linear algebra data-parallel computation kernel with distinctive processing requirements, namely:

i) Kernel 1 performs the sum of two integer vectors (Eq. 6.1);

\[ v_i^I = a_i^I + b_i^I \mid i=1,...,N \]  

ii) Kernel 2 computes the inner product of two integer vectors (Eq. 6.2);

\[ \alpha^I = \sum_{i=1,N} a_i^I \times b_i^I \]  

iii) Kernel 3 performs the sum of two single-precision floating-point vectors (Eq. 6.3);

\[ v_i^F = a_i^F + b_i^F \mid i=1,...,N \]  

iv) Kernel 4 computes the inner product of two single-precision floating-point vectors (Eq. 6.4);

\[ \alpha^F = \sum_{i=1,N} a_i^F \times b_i^F \]  

Each input dataset \((a_i^I, b_i^I, a_i^F, b_i^F)\) is an independent and randomly generated array with 300 million integer/floating-point cells. These datasets are then partitioned and dynamically assigned to the processing clusters by the Hypervisor engine during runtime. Since the main focus of the proposed work is to evaluate the system reconfiguration capabilities (and not the memory access contention), this experimental evaluation assumes that the data is locally stored in the processing cluster. As such, all cores are implemented with an integrated scratch-pad memory, to alleviate the contention in the memory hierarchy.

Since each kernel requires operations with different levels of complexity, 3 different PE architectures were considered providing different trade-offs in terms of hardware complexity, energy consumption, and area occupancy. As the base architecture for the PEs, the 32-bit RISC MB-LITE [11] soft-core was used, due to its portable processing structure, with an implementation compliant with the well known MicroBlaze ISA [30]. Furthermore, it allows taking advantage of the GNU Compiler Collection (GCC) [7] support. Moreover, the MB-LITE design has a low hardware resource requirement and is highly configurable, making it relatively easy to design custom...
6.2 Hardware resources

modules. Its reduced hardware resources were also taken into account, providing a robust, yet low-resource, base core for scalable multi-core processing platforms.

The simplest deployed PE architecture (Type A) corresponds to the basic configuration of the MB-LITE core, i.e. with both the barrel shifter and multiplier deactivated, as the presence of these structures would impose significant hardware overheads and a lower operating frequency [11]. The second architecture (Type B) includes the referred structures, corresponding to the full MB-LITE architecture. For the third considered architecture (Type C), a full MB-LITE architecture supporting a single-precision Floating Point Unit (FPU) was considered. In this particular case, the considered 4-stage pipelined FPU is composed of three Xilinx IP Floating-Point Operator [26] cores. Since the MB-LITE core is compatible with the MicroBlaze ISA, the corresponding Floating Point (FP) instruction opcodes [30] for the addition, subtraction, multiplication, and compare were adopted.

To provide the Hypervisor with relevant profiling information about each architecture, five performance metrics were monitored during execution, corresponding to the counts of clock cycles (CLK), integer multiplications (MUL), FP operations (FP) and calls to software-emulated integer multiplications (SW_MUL) and FP operations (SW_FPU).

6.2 Hardware resources

Having defined the PE architectures, an hardware resource analysis was performed in order to define the number of processing cores to be instantiated in each cluster topology. Table 6.1 presents the required hardware resources for each type of processing core. As expected, the resource overhead increases with the complexity of each architecture. Hence, the number of processing cores in each cluster topology was dimensioned in order to ensure that the total occupied area by each cluster is approximately the same, allowing the maximum number of Type A PEs. This way, each Type A cluster contains 15 cores, each Type B cluster contains 12 cores, and each Type C cluster contains 8 cores (see Table 6.1).

By analyzing the power consumption of each cluster topology, by considering an operating frequency of 100 MHz, a rather similar value for the three topologies was observed (bottom of Table 6.1). This is explained by the fact that the three cluster topologies were properly defined in order to occupy the same amount of hardware resources, roughly including the same amount of logic.

The static part of the platform was also analyzed in terms of the required hardware resources and power consumption. Hence, all the components responsible for the reconfiguration process, such as the MicroBlaze and the dedicated reconfiguration module and relocation parser, as well as the communication part with the host, such as the AXI DMA and PCIe Bridge, were taken into account for the total power/energy consumption of the system. Table 6.2 presents the hardware
6. Evaluation

Table 6.1: Experimental evaluation of each cluster type, in terms of hardware resources, maximum operating frequency and power consumption.

<table>
<thead>
<tr>
<th></th>
<th>Available Resources per Cluster Region</th>
<th>Processing Clusters</th>
<th>Type A</th>
<th>Type B</th>
<th>Type C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>-</td>
<td>15</td>
<td>12</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td>48864</td>
<td>7655 (16%)</td>
<td>6146 (13%)</td>
<td>6569 (13%)</td>
<td></td>
</tr>
<tr>
<td>LUT's</td>
<td>24432</td>
<td>16706 (68%)</td>
<td>16614 (68%)</td>
<td>15334 (63%)</td>
<td></td>
</tr>
<tr>
<td>RAMB36E1</td>
<td>60</td>
<td>60 (100%)</td>
<td>48 (80%)</td>
<td>24 (53%)</td>
<td></td>
</tr>
<tr>
<td>RAMB18E1</td>
<td>45</td>
<td>45 (100%)</td>
<td>36 (80%)</td>
<td>24 (53%)</td>
<td></td>
</tr>
<tr>
<td>DSP48E1</td>
<td>174</td>
<td>0 (0%)</td>
<td>36 (21%)</td>
<td>56 (32%)</td>
<td></td>
</tr>
<tr>
<td>Max Freq. [MHz]</td>
<td>-</td>
<td>206.2</td>
<td>114.7</td>
<td>113.2</td>
<td></td>
</tr>
<tr>
<td>Static Power [mW]</td>
<td>-</td>
<td>210.9</td>
<td>210.7</td>
<td>209.9</td>
<td></td>
</tr>
<tr>
<td>Total Power@100 MHz [mW]</td>
<td>-</td>
<td>615.2</td>
<td>636.6</td>
<td>600.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Experimental evaluation of the static fraction of the reconfigurable platform, in terms of hardware resources, maximum operating frequency and power consumption.

<table>
<thead>
<tr>
<th></th>
<th>Registers</th>
<th>LUTs</th>
<th>RAMB36E1</th>
<th>RAMB18E1</th>
<th>DSP48E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MicroBlaze</td>
<td>2346 (&lt;1%)</td>
<td>2126 (&lt;1%)</td>
<td>22 (2%)</td>
<td>0 (0%)</td>
<td>3 (&lt;1%)</td>
</tr>
<tr>
<td>AXI4 BUS</td>
<td>21041 (3%)</td>
<td>17325 (6%)</td>
<td>4 (&lt;1%)</td>
<td>1 (&lt;1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>AXI4-Lite Bus</td>
<td>248 (&lt;1%)</td>
<td>483 (&lt;1%)</td>
<td>1 (&lt;1%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>PCIe Bridge</td>
<td>12595 (2%)</td>
<td>17954 (6%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>ICAP Controller</td>
<td>742 (&lt;1%)</td>
<td>546 (&lt;1%)</td>
<td>1 (&lt;1%)</td>
<td>1 (&lt;1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>DMA</td>
<td>2891 (&lt;1%)</td>
<td>2930 (&lt;1%)</td>
<td>1 (&lt;1%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>BRAMS</td>
<td>864 (&lt;1%)</td>
<td>1242 (&lt;1%)</td>
<td>3 (&lt;1%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Total Power Consumption [mW]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>367.9</td>
</tr>
</tbody>
</table>

According to the obtained results, it was possible to implement a 7-cluster accelerator in the considered FPGA. In the whole, this represents a number of processing cores ranging from 56 to 105, in the implemented system.

6.3 Reconfiguration overhead

In what concerns the evaluation of the real-time adaptation of the system, the expected dependency of the reconfiguration time with the size of the partial bitstream that is loaded into the ICAP was observed. Since the bitstreams for the three considered cluster topologies are approximately 2MBytes, a reconfiguration time of approximately 5ms is observed. A special case is worth noting in what concerns the bitstream corresponding to the blank-box configuration. Since this file is only 460KBytes long, a smaller reconfiguration time of approximately 2ms was observed for each reconfiguration.

It is also important to study the impact of the dedicated Hardware module created for this specific process, since it is the focus of the present dissertation. Without SWINGER, the reconfiguration process of the same modules introduce a timing overhead of approximately 450ms, by
6.3 Reconfiguration overhead

Figure 6.1: Real-time adaptation of the processing architecture, performing reconfiguration through software, having a MicroBlaze load into Xilinx AXI ICAP controller IP.

having the MicroBlaze perform the reconfiguration via software, using the Xilinx ICAP Controller IP. The reduced time has a significant impact on the considered application and can be seen in Fig. 6.1 and Fig. 6.2. On the presented graph, the 7 clusters are initialized with one of the three considered configuration. Depending on the different kernels that are being executed, it is possible to see the system changing the configuration of certain clusters, to better fit the requirements of the running kernel. The timing overhead introduced by the reconfiguration process can be seen in the grey blocks presented in the processing graph. It is possible to observe that the speedup introduced by the system is reduced significantly by the larger reconfiguration overhead, since the Hypervisor takes in consideration this factor when deciding when to trigger the reconfiguration to improve the process. Having the dedicated module results in an execution 1.3 times faster in the specific presented application example.

In order to obtain the dynamic energy that is spent in the reconfiguration procedure of each cluster, the power that is consumed by the reconfiguration engine, when it is in its idle state, was subtracted to the energy consumed by the same engine while performing a reconfiguration procedure. The obtained difference between these two measures results in an estimated reconfiguration power of about 44mW. Despite consuming significantly less power than what it is required by the actual processing clusters (210 W), this result is also considered by the Hypervisor, when making decisions regarding reconfiguration commands and energy savings.
6. Evaluation

Figure 6.2: Real-time adaptation of the processing architecture using implemented dedicated module SWINGER.

6.4 Relocation Results

To evaluate the impact of the relocation parser in the proposed adaptive multi-core system, different number of cores per cluster were considered, in the effort of analysing how floor-planning on the target device can influence the number of RMs that can be relocated.

In order to illustrate how changing the size of the RMs can improve in terms of configuration memory storage, three different scenarios were considered: i) the configurations considered in Fig. 6.1, which occupies 3 tiles on the target device’s configurable fabric ii) a configuration that can fit a RM into 2 tiles iii) a configuration that can fit a RM into 1 tile. In the Annex section of this dissertation, a more throughout explanation regarding the floor-planning for each case, and representative figures are presented.

The results presented in table 6.3 suggest that the memory saving increases as the number of tiles that are occupied by a single RM decreases, while the number of maximum cores that can be instantiated in the FPGA is not reduced. The table presents the number of clusters that were instantiated on the FPGA area, without compromising the functioning of the static part of the system, as well as the number of cores in every configuration could fit into the area. By indicating the memory size of each of the configurations for each module, and the number of cores that can be relocated, the memory saving can be obtained.

Since the main obstacle to the relocation process is, as previously described, being able to instantiate RMs in equal relative areas of the FPGA, the results were expected. The smaller
Table 6.3: Experimental evaluation of each cluster floor-planning configuration, in terms of impact of relocation on configuration memory.

<table>
<thead>
<tr>
<th>Clusters</th>
<th>Type A cores</th>
<th>Type B cores</th>
<th>Type C cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Tiles</td>
<td>7</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>[Annex 1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Tiles</td>
<td>11</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>[Annex 2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Tile</td>
<td>23</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>[Annex 3]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bitstream size</th>
<th>Memory - no realoc</th>
<th>Realoc. Clusters</th>
<th>Memory realoc.</th>
<th>Memory Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Tiles [Annex 1]</td>
<td>2MB</td>
<td>42 MB</td>
<td>2</td>
<td>30MB</td>
</tr>
<tr>
<td>2 Tiles [Annex 2]</td>
<td>1.45MB</td>
<td>47.85MB</td>
<td>6</td>
<td>21.75MB</td>
</tr>
<tr>
<td>1 Tile [Annex 3]</td>
<td>0.75MB</td>
<td>51.75MB</td>
<td>20</td>
<td>9MB</td>
</tr>
</tbody>
</table>

cluster considered allowed for a much higher number of its copies to be instantiated across a greater number of tiles, while allowing for the necessary static part of the design and the relevant configuration and communication ports to work properly.

Ensuring that all the relocatable RMs are routed in the required way, it is possible to obtain a reduction on reconfiguration memory necessary, of up to 82%, while having still a 100+ possible number of PEs.

6.5 Performance evaluation

To further evaluate the proposed system, the following subsections present its characterization in terms of the offered adaptability. This is performed by first considering two scheduling scenarios without any previous knowledge of the application being executed, resulting in the definition of an optimized execution model. This model is then used to demonstrate the optimization policies proposed in Section 5.5. The presented results are shown in terms of the attained performance and energy savings.

6.5.1 Runtime architecture adaptation and model definition

In order to demonstrate the adaptive capabilities offered by the implemented Hypervisor, to provide the best fitted architecture for a given kernel, two different execution scenarios were considered with no a priori knowledge of the kernels is assumed. In particular, these two scenarios only differ in the order the computing kernels are executed.
6. Evaluation

Figure 6.3: Real-time adaptation of the processing architecture, without any a priori knowledge of the computing kernels (Kernel order: 1, 2, 3, 4).

Figure 6.4: Real-time adaptation of the processing architecture, without any a priori knowledge of the computing kernels (Kernel order: 1, 3, 2, 4).

In Figures 6.3 and 6.4, it is possible to see the Hypervisor allowing each cluster to execute its assigned chunk of a kernel with its currently assigned configuration. Upon completion of such kernel chunk, the Hypervisor sends a reconfiguration command to that same cluster, in order to adapt its architecture to the currently executing kernel, according to the set of received values.
6.5 Performance evaluation

Figure 6.5: System real-time adaptation, according to the minimum execution-time optimization policy.

of the performance counters. The longer execution time observed in Fig. 6.4 is due to the fact that the first chunks of kernels 3 and 4 are initially executed in clusters of Type A and Type B, and only after the first execution does the Hypervisor know that FP operations are needed. This means that the FP operations present in those kernels are initially executed with software libraries, resulting in an increased latency.

6.5.2 Adaptive model-based policies

After the first execution of the application in the previously described "untrained" mode, the obtained model of the application can be used to demonstrate the other developed optimization policies. In these scenarios, since there is a previously obtained execution model, when a kernel is to be executed, the Hypervisor can immediately trigger the reconfiguration process to adapt the assigned cluster to the best fitted architecture for that kernel.

The execution time policy described in Fig.5.6 Algorithm 1 allows the system to dynamically select the set of clusters that provide the best performance for each kernel under execution. The experimental results for this policy are presented in Fig. 6.5 and allow concluding that the system
6. Evaluation

was able to adapt to the best possible configuration, while also achieving a well balanced data chunk distribution to the several processing clusters.

The second considered optimization policy considers the maximization of the system performance, while establishing a given power-ceiling (see Fig.5.6 Algorithm 2). To ensure a more realistic test, this power-ceiling was also varied in run-time. In Fig. 6.6 it is possible to observe idle clusters being replaced by empty blank-boxes when the power-ceiling decreases, in order to meet this constraint. On the other hand, as soon as the allowed power consumption level increases, the system reactivates these turned-off clusters, in order to maximize the accelerator throughput.

The last proposed optimization policy, previously described in Fig.5.6 Algorithm 3, considers the minimization of the power consumption while assuring a minimum performance level. To show the adaptivity of the proposed system, it is further assumed that the application under execution establishes a different minimum throughput for each kernel, as shown in Fig. 6.7. As it can be observed, the system is able to adapt the clusters in real-time, not only to ensure the required performance level, but also to minimize the power consumption, by disabling inactive clusters.

6.5.3 Speedup and energy reduction

To evaluate the performance gains and energy savings resulting from the proposed adaptive system, the dynamic execution policy presented in Fig. 6.5 was compared with four different static configurations (i.e., without reconfiguration), each one composed by 7 independent clusters of PEs: i) a system with 7 Type A clusters; ii) a system with 7 Type B clusters; iii) a system with 7 Type C clusters; and iv) a heterogeneous mix composed of 2 Type A clusters, 2 Type B clusters and 3 Type C clusters.

Table 6.4 presents the obtained results in terms of execution time and energy consumption for the considered setups. Despite containing 105 cores, it can be observed that the system with only Type A clusters represents the worst case, both in terms of performance and energy. This is explained by the fact that Type A PEs must perform the multiplication operations of Kernel 2 through a combination of logic shifts and additions, and the floating-point operations of Kernels 3 and 4 through calls to software libraries. Naturally, this represents a large energy overhead, which results in a total consumption of 240 Joules. The best homogeneous static configuration is obtained by using only Type C clusters. Even though only 56 cores can be implemented in this case, it performs about 4× faster than the worst-case configuration. The best static configuration was achieved by using the considered heterogeneous configuration (2× Type A + 2× Type B + 3× Type C), which provides a trade-off between complexity and execution time/energy consumption.

Finally, it can be observed that the offered adaptive capabilities allow the dynamic system to combine all the advantages of the above described configurations. By adapting, at run-time,
6.5 Performance evaluation

Figure 6.6: System real-time adaptation, according to the established power-ceiling constraint policy.

Table 6.4: Execution time and energy results.

<table>
<thead>
<tr>
<th>Cluster Type</th>
<th>Time [s]</th>
<th>Energy Consumption [J]</th>
<th>Speedup vs. Dynamic System</th>
<th>Energy Loss vs Dynamic System</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Type-A Clusters</td>
<td>55.715</td>
<td>239.93</td>
<td>9.575</td>
<td>10.31</td>
</tr>
<tr>
<td>7 Type-B Clusters</td>
<td>29.784</td>
<td>132.72</td>
<td>5.118</td>
<td>5.70</td>
</tr>
<tr>
<td>7 Type-C Clusters</td>
<td>11.997</td>
<td>50.44</td>
<td>2.062</td>
<td>2.17</td>
</tr>
<tr>
<td>Static Heterogeneous</td>
<td>18.378</td>
<td>79.13</td>
<td>3.158</td>
<td>3.40</td>
</tr>
<tr>
<td>Dynamic System</td>
<td>5.819</td>
<td>23.28</td>
<td>2.24</td>
<td>-</td>
</tr>
</tbody>
</table>

to the requirements of the different kernels, it is assured that the system always provides the best optimized configuration for each application phase, by trading core complexity with the total number of cores. Thus, it is possible to achieve execution speedups ranging from $2.1 \times$, when compared with the best static case, to $9.5 \times$, when compared to the worst static case, while consuming from $2.2 \times$ to $10.3 \times$ less energy.
6. Evaluation

Figure 6.7: System real-time adaptation, according to the minimum assured performance policy.

6.6 Summary

In this chapter, results obtained from the proposed framework were presented, suggesting significant gains in performance, energy consumption reduction, and data memory saving.

By using the created dedicated module SWINGER, significant efficiency increase for the reconfiguration process was observed, when compared to a software-based solution. The proposed module, while being relatively low in number of resources it needs from the static part of the FPGA area, is efficient and essential for the positive gains observed in the system.

By using the relocation parser, the results of the different considered approaches to the floor-planning suggest that significant reductions in required configuration memory. These results also suggest that the memory reductions would be more significant with a larger number of configurations implemented, since the relocated modules can be obtained using all the configuration images of the main module.

Overall, by applying the different execution policies proposed, the proposed adaptive system
is able to change dynamically to the executing kernels and comply with the underlying policy. The results show that the objectives defined for this dissertation were attained.
6. Evaluation
7. Conclusions and Future Work

7.1 Conclusions

In this thesis, a new morphable heterogeneous computing platform is proposed. This platform integrates hundreds of processing cores and offering run-time adaptation capabilities to better meet the performance and energy constraints imposed by the application under execution is proposed in this paper. The adaptive nature of this platform is achieved by monitoring, in real-time, the performance of the computational cores while they execute the application kernels, and by determining the processing architectures and topologies that maximize the performance and/or energy efficiency. To perform this decision, a Hypervisor software module is also proposed, which is not only responsible for scheduling the computing kernels to the available processing cores, but also able to trigger the reconfiguration of the currently instantiated cores, by issuing appropriate commands to an on-chip Reconfiguration Engine. This latter module performs the actual adaptation, by exploiting the existing partial dynamical reconfiguration mechanisms of modern FPGA devices.

By having the dedicated module as part of the reconfiguration engine for the presented platform, the best performance achievable by the configuration port used for reconfiguration can be reached. Therefore, a performance efficient module that can perform reconfiguration in a variety of different reconfigurable hardware systems was implemented, and its impact shown relevant for the gains of the adaptive system. Moreover, facing the potential problem of an excessive configuration data storage necessary to comport such a system, specially if considered many more configurations for the RM, a relocation parser was also implemented. The study presented showed, by testing different floor-planning options and reducing the size of the considered RMs, it is possible to reduce the configuration memory requirements in up to 82%, since by having more modules that can fit into exactly symmetrical positions of the FPGA, the less number of configuration data is necessary to reconfigure those modules.

To perform the adaptation of this hundred core heterogeneous platform, different algorithms (policies) were considered, corresponding to typical optimization goals, namely: minimization of the execution time; maximization of the processing performance for a given power ceiling; and minimization of the power consumption, while guaranteeing a minimum Quality of Service (QoS) (performance level). To evaluate the proposed system and the corresponding policies, a set of computation kernels from the algebraic domain were used. The obtained experimental results allow concluding that the proposed policies provide a significant reduction of both the execution time and energy consumption when compared with static homogeneous or non-homogeneous implementations with a fixed number of cores. The proposed reconfigurable system achieves performance gains between $2\times$ and $9.5\times$, while the energy consumption was reduced between $2\times$ and $10\times$.

Accordingly, the proposed morphable heterogeneous structure has shown to be a highly viable
and efficient approach to provide an adaptable architecture, being able to morph the computing cores according to the instantaneous system restrictions, resulting not only in improved performances but, more importantly, in an energy-aware computing platform.

7.2 Future Work

Although the proposed framework comprises an efficient adaptive multi-core system, structural and conceptual improvements can be made, pushing even further the potential of such a platform.

While the different considered application suggest performance and energy consumption gains, the kernels executed by each core, while chosen to fit the different hardware configurations, do not represent more complex algorithms that might be used in a multi-core system. It is possible to study and improve the presented framework when executing different kind of applications by having different configurations and exploring different replacement policies.

Another aspect that can be further improved is the relocation implementation. The proposed system admits that all configuration data that enters the reconfiguration module, and is filtered by the relocation parser, is correct and loaded in the right order. In the event that this isn’t the case, the reconfiguration process is quite possibly invalidated. One of the approaches to solve this limitation is to perform CRC as the words are loaded into the dedicated module, by having a dedicated CRC calculating hardware section that keeps track of what the CRC word must be. By including this feature, another layer of protection in the reconfiguration process would be provided making the system more robust, stable and reliable.
7. Conclusions and Future Work
Bibliography


[31] Xilinx Inc. 7 Series FPGAs Configuration, v1.6, January 2013.


Annex A
A. Annex A

In order to correctly floor-plan the proposed reconfigurable design into the reconfigurable fabric of the target device, as to ensure that relocation can be performed by the proposed relocation parser, different aspects, such as symmetry of the configurable fabric and the tile-based addressing of the configuration, must be taken into consideration. Since, for the target device, the configuration bitstreams load configuration data for each tile occupied by the reconfigurable module, 3 different examples are illustrated and explained in this section, based on modules that occupy 3, 2 and 1 tiles on the device’s configurable area.

A.1 3-Tile Reconfigurable Module

The implementation described throughout the dissertation uses a processing cluster which occupies a larger portion of the reconfigurable fabric, as presented in table 6.3. This first implementation allows for the instantiation of a 100+ core processing system, and proved to be able to introduce performance and power consumption gains. In terms of relocation ability of the chosen RMs, this implementation proved to be too limited in terms of floor-planning to allow bigger configuration memory savings due to relocation. The considered design’s floor-planning is illustrated in Fig. A.1.

Due to the higher number of resources needed for the reconfigurable clusters, associated with the asymmetry between the different columns of tiles, it is not possible to perform relocation on clusters in different columns, and only between RMs that share the same column section. Also, due to the specification of the implemented solution, the areas which comprise the ICAP module and the PCIe are not included in a RM reserved area, causing those specific regions to not comprise with the relocation requirements. Hence, only 2 of the considered modules can be reconfigured using the relocation parser: the module starting in the tile X0Y2, and the one starting in X1Y2, both being relocated using each of the RMs above them in its column.

This configuration, due to its limitations, by ensuring that the 2 modules considered are constrained in the exact relative positions of its relocation associates, it provides a configuration memory saving of 29%, as presented in table 6.3.

A.2 2-Tile Reconfigurable Module

The second case considered, presents a reduction in the number of cores present in the RM, so that it can be instantiated in 2 tiles of the target device’s configurable area. By doing so, it is possible to bypass some of the limitations due to the asymmetry of the distribution of resources across the FPGA, and, have a larger number of modules that can be relocated, while still allowing for a 100+ processing core system. In Fig. A.2 it is possible to see that 11 symmetric reconfigurable modules were instantiated in the fabric, allowing for a higher number of same column relocation.
Due to a lower number of resources needed for the reconfigurable clusters, a higher number of reconfigurable clusters can be reconfigured using relocation. 2 of the Clusters can be reconfigured from the bitstream of the cluster number 1: the cluster number 2 and 3, since they share the same column, having the exact same distribution of resources. Similarly, it is possible to reconfigure the cluster 5 from the bitstream of cluster 4.

This configuration, by ensuring that the 6 modules considered are constrained in the exact relative positions of its relocation associates, introduces a configuration memory saving of 46%, as presented in table 6.3.

### A.3 1-Tile Reconfigurable Module

The third case considered, presents a reduction in the number of cores present in the RM, so that it can be instantiated in 1 tile of the target device’s configurable area. By doing so, it is possible to bypass most of the limitations due to the asymmetry of the distribution of resources across the FPGA, and, have a larger number of modules that can be relocated, while still allowing for a 100+
A. Annex A

Figure A.2: 2-Tile Reconfigurable Module floor-planning

processing core system. In Fig. A.3 it is possible to see that 23 symmetric reconfigurable modules were instantiated in the fabric, allowing for a higher number of same column relocation, and even across different columns.

Due to a lower number of resources needed for the reconfigurable clusters, a higher number of reconfigurable clusters can be reconfigured using relocation. 20 of the Clusters can be reconfigured from the bitstream of the cluster number 1: the cluster number 2, 3, 4, 5 and 6, since they share the same column, having the exact same distribution of resources. Similarly, it is possible to reconfigure the cluster 8, 9, 10 and 11, from the bitstream of cluster 4. Blocks 13, 14, 15, 16, 17, 19, 20, 21, 22 and 23 can all be relocated from the top module present in the same column.

This configuration, by ensuring that the 6 modules considered are constrained in the exact relative positions of its relocation associates, introduces a configuration memory saving of 82%, as presented in table 6.3.
Figure A.3: 1-Tile Reconfigurable Module floor-planning