

Faculdade de Engenharia da Universidade do Porto



Run-time selection of customized accelerators

José Miguel Campos

FINAL VERSION

Dissertation carried out within the scope of
Mestrado Integrado em Engenharia Eletrotécnica e de Computadores
Major Automação

Orientador: João Canas Ferreira

06/07/2020

© José Miguel Campos, 2020

Abstract

Moore's law is coming to an end.

In the last twenty years the trend was to explore the parallelization of the task execution and therefore parallel computing.

Despite being obvious that one could only achieve better performances with another type of machine like the quantum computers, it is obvious as well that these machines, that we already have, could be always of use because they are very good when it comes to performing certain tasks or task sets.

Being so, there are other situations where the best options would be the ones that can offer the most capacity of parallelization such a multiprocessor, multithreaded environment, or with a more specific environment such as are the GPUs (Graphical Processing Unit), ASICs (Application Specific Integrated Circuit) or FPGAs (Field Programmable Gate Array).

This work targeted the development of a hybrid computing system with an FPGA with runtime partial dynamic reconfiguration capabilities.

Performance and power efficiency requirements of embedded systems are constantly becoming more demanding as a result of growing algorithm complexity and autonomy. One important approach to the solution of these conflicting requirements for FPGA-based systems is to use customized accelerators to improve execution efficiency of hot-spots. The more the accelerator is customized to the code, the better the performance improvements. However, an accelerator customized for a given hot-spots, may not be ideal for other hot-spots or may not be of use at all and, therefore, we may need to reconfigure the FPGA any time we want to process a different task.

In order to improve overall performance and yet, to switch accelerators dynamically at run-time using partial reconfiguration each time we have a different task set it is needed to study the task set or try to predict the cycle of execution.

The objective of this work is to:

- Implement a hardware prototype that supports N customized accelerators and dynamically chooses the most appropriate one taking into consideration the time and energy

required to switch the accelerators. The choice may also depend on the sequence of hot-spots to be accelerated.

- Study and analyses of the partial reconfiguration characteristics and tools, and the study of the static and dynamic execution time components and model.
- The development of specific accelerators to a task or task set.
- The development of a scheduler for an operative system that would reduce the losses in the performance brought through the overall effects of the partial dynamic reconfiguration of the FPGA, being able to improve the performance of the system and reducing the power consumption at runtime.

Results obtained about different algorithms are presented. Results obtained about a simple scheduler are as well presented and finally we try to conclude what are the efforts we have to make in order to further develop this proposed scheduler in order to predict the task set and minimize the time needed to reconfigure the system.

We present conclusions if and when an FPGA is a good option to accelerate a task within a heterogeneous computing system.

The prototype was evaluated using existing benchmark programs.

Resumo

A lei de Moore está a chegar ao seu fim.

Nos últimos vinte anos, a tendência tem sido explorar a paralelização da execução das tarefas e, portanto, a computação paralela.

Apesar de óbvio que só se poderá obter melhores desempenhos com outro tipo de máquina, como os computadores quânticos, também é óbvio que estas máquinas, já existentes, podem ser sempre úteis, porque são muito boas quando se trata de executar determinadas tarefas ou conjuntos de tarefas.

Sendo assim, existem outras situações em que as melhores opções seriam aquelas que podem oferecer a maior capacidade de paralelização, como um sistema multiprocessador, programação multithreaded ou com um ambiente heterogêneo, como com as GPUs (Graphical Processing Unit), com ASICs (Application Specific Integrated Circuit) ou com FPGAs (Field Programmable Gate Arrays).

Este trabalho teve como objetivo o desenvolvimento de um sistema de computação híbrido com uma FPGA com recursos de reconfiguração parcial dinâmica em tempo de real.

Os requisitos de desempenho e eficiência de energia dos sistemas embarcados estão se tornando cada vez mais apertados como resultado da crescente complexidade dos algoritmos. Uma abordagem importante para o cumprimento desses requisitos apertados para sistemas com FPGAs é usar aceleradores personalizados para melhorar os tempos de execução das tarefas e melhorar a performance. Quanto mais o acelerador for personalizado para o código, maiores serão as melhorias de desempenho. No entanto, um acelerador personalizado para uma determinada tarefa pode não ser ideal para outras tarefas ou pode não ser de todo útil e, portanto, talvez seja necessário reconfigurar o FPGA a qualquer momento que desejar processar uma tarefa diferente ou ter os vários aceleradores instanciados na configuração inicial da FPGA.

Para alternar dinamicamente os aceleradores em tempo real usando a reconfiguração parcial cada vez que temos um conjunto de tarefas diferente e, no entanto, melhorar o desempenho geral e reduzir a perda de performance trazida pela necessidade de reconfigurar o sistema, é necessário estudar o conjunto de tarefas ou tentar prever o ciclo de execução.

O objetivo deste trabalho é:

- Implementar um protótipo de hardware que suporte N aceleradores personalizados e escolha dinamicamente o mais apropriado, levando em consideração o tempo e a energia necessários para alternar os aceleradores. A escolha também pode depender da sequência de tarefas a serem aceleradas.

- Estudo e análise das características e ferramentas de reconfiguração parcial.
- Estudo dos componentes e modelo estáticos e dinâmicos do tempo de execução.
- O desenvolvimento de aceleradores específicos para uma tarefa ou conjunto de tarefas.
- O desenvolvimento de um escalonador para um sistema operativo que reduza as perdas no desempenho causadas pelos efeitos gerais da reconfiguração dinâmica parcial do FPGA, podendo melhorar o desempenho do sistema e reduzir o consumo de energia em tempo de real.

Resultados obtidos sobre diferentes algoritmos são apresentados.

Os resultados obtidos sobre um escalonador simples também são apresentados e, finalmente, tentamos concluir quais são os esforços que temos para desenvolver o escalonador proposto, a fim de prever o conjunto de tarefas e minimizar o tempo necessário para reconfigurar o sistema.

Apresentamos conclusões se e quando um FPGA é uma boa opção para acelerar uma tarefa em um sistema de computação heterogêneo.

O protótipo foi avaliado usando os programas de benchmarking existentes.

Acknowledgments

I want to dedicate his work to all the great minds and inventors in history, like Nicola Tesla, that were ahead of their time, true heroes for mankind.

I want to thank to all my professors and colleagues that were always kind and available to help, specially to my supervisor professor Canas Ferreira for his guidance and insight, and to Federico and Pedro for their friendship and interest.

I want to thank to all my family and friends, specially to my parents and grandparents, specially to my grandfather Albino that is with us no more, for all their support and for believing in me.

Finally, I want to thank to my girlfriend Telma, for giving me inspiration and for making me wanting to be everyday a better man.

**“The parallel approach to computing does require
that some original thinking be done about numerical
analysis and the data management in order to secure
efficient use. In an environment which has
represented the absence of the need to think as the
highest virtue this is a decided disadvantage”
Daniel Slotnick, 1967**

Table of Contents

Abstract.....	iii
Resumo	iii
Acknowledgments	viii
Table of Contents.....	x
List of figures.....	xiii
List of tables	xv
Acronyms and Symbols.....	xvi
Chapter 1	1
Introduction	1
1.1 - Heterogeneous Computing Systems	2
1.2 - FPGA as a Special Type of Processing Unit.....	4
1.2 - High Level Synthesis.....	4
1.3 - Flynn's Taxonomy	4
1.4 - Motivation and Problem Statement	5
1.5 - Objectives	5
1.6 - Approach.....	5
1.7 - Structure of the Document.....	5
Chapter 2.....	7
Review of Related Work	7
2.1 - Encyclopaedia of Parallel Computing.....	7
2.2 - Measuring the Performance of Schedulability Tests	7
2.3 - Finding Speedup in Parallel Processors	7
2.4 - Self-adaptive loop for CPSS.....	7
2.5 - CoRQ.....	7
2.6 - Dynamic partial reconfiguration in FPGAs.....	7
2.7 - Quantifying the Benefits of DPR for Embedded Vision Applications	7
2.8 - Conclusions	7
Chapter 3	17
Overview of the System	17

3.1 - Zync7000.....	17
3.2 - Developed Prototype	20
3.3 - PDR of FPGAs	21
3.4 - PDR Control	24
3.5 - Data Transfer	24
3.6 - Main Memory	27
3.7 - Measurement Infrastructure	29
3.8 - Accelerator Architecture.....	7
3.9 - Execution Model	31
 Chapter 4.....	35
Project Development, Algorithms and Static Analyses	35
4.1 - PDR	35
4.1 - Algorithm Analyses	42
 Chapter 5.....	46
The Proposed Scheduler.....	46
5.1 - Simple Version	46
5.2 - Sophisticated Version	48
 Chapter 6.....	51
Conclusion and Future Work.....	51
6.1 - Conclusion	51
6.2 - Future Work	53
 References	54

List of figures

Figure 1.1 - Generic Heterogeneous Computing System	2
Figure 2.1 - MAX2 board	11
Figure 2.2 - Proposed Manager	12
Figure 2.3 - Proposed Prototype.....	13
Figure 2.4 - SoC Overview	14
Figure 2.5 - System Architecture.....	15
Figure 3.1 - Zynq®-7000 SoC block diagram	18
Figure 3.2 - Zynq®-7000 PL	19
Figure 3.3 - High Level Block Diagram of the System	20
Figure 3.4 - High Level Block Diagram of the System with ICAP Configuration	22
Figure 3.5 - Top Level AXI HWICAP Core	23
Figure 3.6 - System Diagram	25
Figure 3.7 - System View	27
Figure 3.8 - Measuring Time	30
Figure 3.9 - High Level Block Diagram for the Template IP	31
Figure 3.10 - Execution Model for HW accelerated Task.....	33
Figure 4.1 - Typical Configuration Mode Timings	38
Figure 4.2 - Configuration Phases at Power On	38
Figure 4.3 - Reconfiguration Timings	39
Figure 4.4 - Configuration Phases with PR	39
Figure 4.5 - PR Timings	40
Figure 4.6 - PDR Timings.....	40

Figure 4.7 - Measured Time of PDR	41
Figure 4.8 - Matrix Multiplication Software Execution, ARM, Timing	41
Figure 4.9 - Matrix Multiplication IP	41
Figure 4.10 - Bubble Sort, Software Execution, ARM, Timing.....	41
Figure 4.10 - Bubble Sort, Software Execution, FPGA, Timing	41
Figure 5.1 - High Level Block Diagram of the Scheduler, Simple Version	45
Figure 5.2 - High Level Block Diagram of the Scheduler, Sophisticated Version	48

List of tables

Table 3.1 – Colour Code for the Execution Model.....32

Acronyms e Symbols

Acronyms

CAD	Computer Aided Design
CISC	Complex Instruction Set Computer
DEEC	Departamento de Engenharia Eletrotécnica e de Computadores
DFT	Discrete Fourier Transform
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DRB	Dynamic Reconfiguration Branch
DRI	Dynamic Reconfiguration Interface
DRM	Dynamic Reconfiguration Memory
EDF	Earliest Deadline First
FEUP	Faculdade de Engenharia da Universidade do Porto
FPGA	Field Programmable Gate Array
OCM	On Chip Memory
PDR	Partial Dynamic Reconfiguration
RISK	Reduced Instruction Set Computer
RM	Reconfigurable Module
RP	Reconfigurable Partition

Chapter 1

Introduction

In 1965, Gordon Moore predicted that the number of transistors placed in a single chip would double every two years, being the reduced cost one of the big advantages of integrated electronics [1].

The State of the Art of computer architecture was for 70 years based on the Von Neumann model, Complex Instruction Set Computers, CISC, and Reduced Instruction Set Computers, RISC, based on the Harvard architecture, became the most used and employed processors even when talking about super computers. The execution model was, for many years, in a sequential thread fashion, where the sequences of instructions are executed one at a time.

One of the ways to improve performance, as the number of transistors were increasing for the same area, was to increase the frequency of the processor as well. This no longer feasible due to physical constraints [2], above all, because of power consumption and power density, from which we can observe that the frequency and the supply voltage are directly proportional. It would be necessary to change and improve the refrigeration systems in order to further improve the performance augmenting the processor frequency, which is not viable. As a general rule higher frequency processor designs require exponentially more power than lower frequency designs. The standard relationship between processor execution time, T , and required power, P , is: $T^3P = k$, where k is a constant. As an example, if a high frequency design were designed to double the frequency of a particular reference design one would expect it to consume 8 times the power. While clever design may avoid the worst consequences of this, it is obvious that parallel designs will have a uniform power density that scales with area. [3]

Additionally, the time to read and write in the memory had become not negligible and it was for various years a serious bottleneck that resulted in the fabrication of General Purpose Processors, GPPs, with bigger caches, a trend that still is continuing at the present moment.

For the last twenty years the trend was to parallelize the execution of different tasks or the task itself if and as well as it would be possible in order to improve the performance of a machine, in some cases, to meet the specified time constraints of the project. That led to the creation of computing systems with a large number of multicore GPPs with big caches and to the advent of multithreaded computing.

Parallel processors have been an integral part of computer architecture and design for more than 50 years. Recent re-emphasis on various forms of multiprocessors (multi core and multithreaded) has arisen from the inability to continue to scale die frequency due to power limitations. With the continuing advance in circuit density replicating processors in an MIMD, multiple instruction multiple data, configuration is an obvious alternative. [3]

Besides that, nowadays, embedded systems can run a considerable number of applications at the same time, which takes us directly to see the improvement of performance by having a system that could actually run all those applications in different, but still equal in itself, resources, as are the multiprocessor computers. Despite having to share some of the resources as the main memory, multiprocessor computers can have a large number of GPPs, that can process different applications. Even though this improved the performance, there are still some cases when the GPPs, with the most used and established processor architectures, RISK and CISC, are not the best option or a good option at all, the alternative of processor node plus array oriented accelerator has some significant advantages especially in compute intensive static applications. [3] These computing systems, developed to solve special cases, when a GPP is not a good solution, are called heterogeneous systems.

1.1 - Heterogeneous Computing Systems

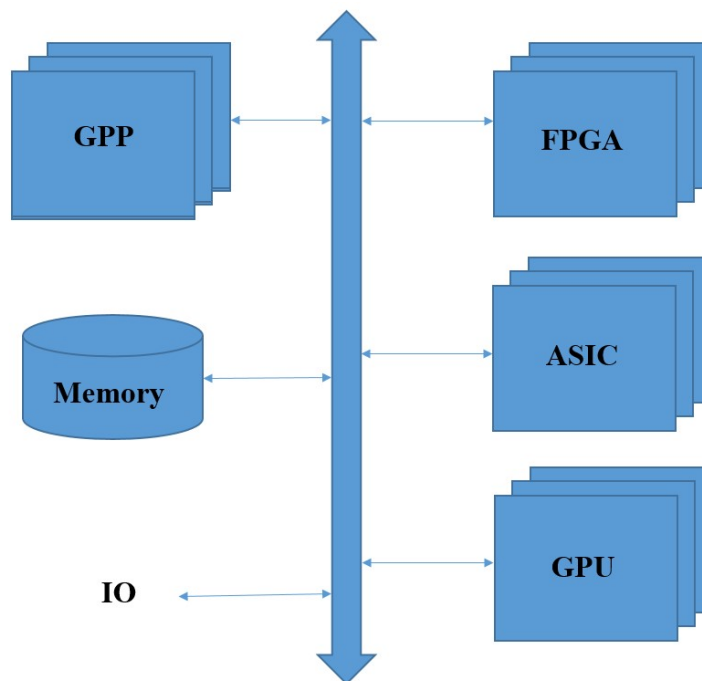


Figure 1.1 - Generic Heterogeneous Computing System

A Heterogeneous Computing System is in its essence a parallel system in which we have different types of processing units connected, that are specified and developed to process specific parts of a task set.

A Heterogeneous System is possibly comprised by a multicore multiprocessor system with GPPs, connected to various others GPUs, and possibly FPGAs and ASICs, as shown in figure 1.1. Each of these parts may be the best option to process a task, in terms of performance.

When considering different algorithms or different parts of an application running in a machine, we may observe that, by using different types of processing resources we will get different performance results, and so, we should study and optimize the distribution of the applications in order to achieve the best possible performance we can have with a specific computer that has different types of processing resources.

In figure 1.1 we can observe a generic heterogeneous system that could be implemented in a SoC, or connecting multiple different processing units as well.

A system on a chip, SoC, is an integrated circuit that has within itself most components of a computer including a central processing unit, CPU, memory, input/output ports and possibly other circuits as a FPGA. It may contain digital, analogue, mixed-signal, and often radio frequency signal processing functions. All those previously stated parts are within the same IC, integrated circuit.

1.2 - FPGA as a Special Type of Processing Unit

In the case of FPGAs or ASICs, the computation would be performed by a circuit rather than by executing instruction in a general purpose, generic processor. FPGAs were initially used in the testing phase when developing new hardware, but nowadays they are used in more diverse application areas, i.e., discrete logic, signal processing, high-performance embedded computing, and recently as accelerators for high-performance computing. With the adoption of high-speed IO standards, as, i.e., AXI interface when considering the internal interfaces in a SoC, or other high-speed bus, FPGAs became more attractive when considering and designing new heterogeneous systems. With a FPGA we have the possibility to develop accelerators tailored to process a specific task, or make a generic processor type circuit that is similar to a general purpose processor. FPGAs are extremely versatile, power-efficient, and they offer high computational performance.

The Advanced eXtensible Interface (AXI), part of the ARM Advanced Microcontroller Bus Architecture specification, is a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface, mainly designed for on-chip communication.

1.3 - High Level Synthesis (HLS)

The Xilinx Vivado HLS tool transforms a C specification, in C, C++, or SystemC, into a register transfer level (RTL) implementation. Hardware designers can work at a higher level of abstraction while creating high-performance hardware.

There are many reasons to explain why algorithmic-based approaches are getting popular, but above all, it is due to accelerated design time and because this makes the FPGA an attractive device for all programmers and software developers. Using HLS almost any software developer can create a good accelerator much faster than with HDL, hardware description languages, based design, which is well known only by hardware developers.

Many implementations are possible, that means each project holds one set of C code and can contain multiple solutions with different constraints and optimization directives.

HLS is comprised of three different parts. Scheduling determines which operations occur during each clock cycle. Binding determines which hardware resource implements each scheduled operation. To implement the optimal solution, high-level synthesis uses information about the target device. Control logic extraction, extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

1.4 - Flynn's Taxonomy

Flynn's taxonomy is a categorization of forms of parallel computer architectures. From the viewpoint of the assembly language programmer, parallel computers are classified by the concurrency in processing sequences (or streams), data, and instructions. This results in four classes SISD (single instruction, single data), SIMD (single instruction, multiple data), MISD (multiple instruction, single data), and MIMD (multiple instruction, multiple data). [4]

1.5 - Motivation and Problem Statement

This master thesis tries to answer different questions, starting with:

Is a FPGA with Partial Dynamic Reconfiguration a possible option to use within a Heterogeneous Computing System?

After this first question is answered and if we were certain that a FGPA is a good option, we can proceed to understand if:

What are the consequences of using only one FPGA and create the need to reconfigure the system at runtime, while maximizing power efficiency?

At last, we will try to get to know the problems created when we need to reconfigure the system, reducing the overall performance, and answer if:

Would it be possible to reduce the loss of performance and the costs brought by the dynamic ongoing of PDR?

1.6 - Objectives

This master thesis has two main objectives.

The first objective is to determine if a dynamically reconfigurable FPGA would be a suitable option to be a part of a heterogeneous system and what are the advantages to choose it.

The second objective of this work is to investigate what are the effects of the need to reconfigure the system and to develop a scheduler that could be able to maximize the performance of the system in different situations and would be able to reduce the loss of performance that occurs when one has to reconfigure the device at runtime, anticipating and predicting what configuration would be needed next and that reconfigures the device before it needs to accelerate a specific task.

1.7 - Approach

At first, several accelerators will be developed and studied. Then, it will be also studied the conditions created by choices during project development by obtaining data from an actual prototype implementation.

After that, a simple version of the scheduler will be presented. This version does not take in account nor tries to reduce the effects of the partial reconfiguration of the system.

Finally, based on artificial intelligence, a more sophisticated version of the scheduler, that tries to predict and anticipate the correct configuration before it is needed, would be presented.

1.8 - Structure of the Document

In chapter 2, Review of Related Work, a review of several other works and books already made about this subject in the last few years is made.

In chapter 3, Overview of the System, the proposed prototype is presented.

Chapter 4, Project Development, Algorithms and Static Analyses, presents results and conclusions that come with the options made during project development and the necessity to accelerate a particular task or task set. Results from data obtained from measurements are presented.

Chapter 5, The Proposed Scheduler, has two parts. In a first moment a simple version of the scheduler is presented and the results obtained, and then in a second part, a more elaborated and sophisticated version, that tries at runtime, to improve the computing

performance reducing the time losses brought by partial dynamic reconfiguration of the FPGA.

Chapter 6, Conclusion and Future Work, presents the conclusions and results of this master thesis.

Chapter 2

Review of Related Work

In this chapter we review some of the books and studies related to this work.

2.1 - Encyclopaedia of Parallel Computing

The Encyclopaedia of Parallel Computing [4] is an important book where we could find many subjects of importance to develop our work, namely:

2.1.1 - Affinity Scheduling

Affinity scheduling is the scheduling of tasks on the processing resources where they will be executed more efficiently. It is often related to different speeds associated with processing the task in that specific resources of the computing system.

The goal of the scheduling policy is typically to optimize time or throughput.

Another form of affinity scheduling is based on the state of the memory system hierarchy. More specifically, it may be more efficient in parallel computing environments to schedule a computing task on a particular computing node than on any other if relevant data, code, or state information already resides in the caches or local memories associated with the node.

In parallel computing environments that employ affinity scheduling, the system often allocates computing tasks on the computing nodes where they will be executed most efficiently, unless that resource is loaded. If computing tasks are always executed on the computing nodes for which they have affinity, then the system may suffer from load sharing problems as tasks are waiting at overloaded nodes while other nodes are under loaded. On the one hand, if processor affinities are not followed, then the system may incur significant penalties as each computing task must establish its working set in close proximity to a computing node before it can proceed.

Scheduling decisions cannot be based solely on task-node affinity, else other scheduling criteria, such as fairness, may be sacrificed. Hence, there is a fundamental scheduling trade-off between keeping the workload shared among nodes and scheduling tasks where they execute most efficiently. An adaptive scheduling policy is needed that determines, as a function of system load, the appropriate balance between the extremes of strictly balancing the workload among all computing nodes and abiding by task-node affinities blindly.

2.1.2 - Algorithm Engineering

The development of algorithms is one of the core areas of computer science, with the goal to prove worst case performance. A purely theoretical approach delays the transfer of algorithmic results into applications. Therefore, in algorithm engineering, implementation and experimentation are viewed as equally important as design and analysis of algorithms.

The CPU time is a good way to characterize the time used by a sequential process (without I/O), even in the presence of some operating system interferences. In contrast, in parallel programs we have to measure the actual elapsed time (wall-clock time) in order to capture all aspects of the parallel program, in particular, communication and synchronization overheads. Thus timing is usually done as follows: All processors perform a barrier synchronization immediately before the piece of program to be timed; one processor x notes down its local time and all processors execute the program to be measured. After another barrier synchronization, processor x measures the elapsed time. As long as the running time is large compared to the time for a barrier synchronization, this is an easy way to measure wall-clock time. To get reliable results, averaging over many repeated runs is advisable.

In parallel computing, running time depends on the number of processors used and it is sometimes difficult to see whether a particular execution time is good or bad considering the amount of resources used. Therefore, derived measures are often used that express this more directly. The speedup is the ratio of the running time of the fastest known sequential implementation to that of the parallel running time. The speedup directly expresses the impact of parallelization. The relative speedup is easier to measure because it compares with the parallel algorithm running on a single processor.

2.1.3 - Benchmarks

Computer benchmarks are computer programs that form standard tests of the performance of a computer and the software through which it is used.

“A benchmark is testing a software interface to a computer, and not a particular type of computer architecture”.

“The basic goal of performance modelling is to measure, predict, and understand the performance of a computer program or set of programs on a computer system”.

2.1.4 - Flynn's Taxonomy

"Developed in 1966 and slightly expanded in 1972, this is a methodology to classify general forms of parallel operation available within a processor. It was proposed as an approach to clarify the types of parallelism either supported in the hardware by a processing system or available in an application".

The classification is based on the view of either the machine or the application by the machine language programmer. It implicitly assumes that the instruction set accurately represents a machine's micro-architecture.

2.1.5 - Job Scheduling

A parallel job scheduler allocates nodes for parallel jobs and coordinates the order in which jobs are run. With enough resources available, a system can execute multiple parallel jobs simultaneously, while other jobs are enqueued and wait for nodes to become available. The job scheduler manages the queues of waiting jobs and oversees node allocation. The goals of a scheduler are to optimize throughput of a system (number of jobs completed per time unit), provide response time guarantees (finish a job by a deadline), and keep utilization of compute resources high.

2.1.6 - LINPACK Benchmark

The LINPACK benchmark is a computer benchmark that reports the performance for solving a system of linear equations with a general dense matrix.

The allowed parallelism modes include automatic parallelization done by the compiler as well as manual parallelization that uses hardware-assisted shared memory or explicit message passing on a distributed memory machine.

2.1.7 - Livermore Loops

Livermore Loops are a set of 24 Fortran DO-loops (The Livermore Fortran Kernels, LFK) extracted from operational codes used at the Lawrence Livermore National Laboratory.

They have been used since the early 1970s to assess the arithmetic performance of computers and their compilers. They are a mixture of vectorizable and non-vectorizable loops and test rather fully the computational capabilities of the hardware as well as the skill of the software in compiling and vectorization of efficient code.

The main value of the benchmark is the range of performance that it demonstrates, and in this respect it complements the limited range of loops tested in the LINPACK benchmark.

2.1.8 - Scheduling Algorithms

Scheduling algorithms aim at defining when operations of a program are to be executed. Such an ordering, called a schedule, has to make sure that the dependences between the operations are met. Scheduling for parallelism consists in looking for a schedule that allows a program to be efficiently executed on a parallel architecture. This efficiency may be evaluated in term of total execution time, utilization of the processors, power consumption, or any combination of this kind of criteria. Scheduling is often combined with mapping, which consists in assigning an operation to a resource of an architecture.

2.1.9 - Shared-Memory Multiprocessor

A shared-memory multiprocessor is a computer system composed of multiple independent processors that execute different instruction streams. Using Flynn's classification, an SMP is a multiple-instruction multiple-data (MIMD) architecture. The processors share a common memory address space and communicate with each other via memory. A typical shared memory multiprocessor includes some number of processors with local caches, all interconnected with each other and with common memory via an interconnection (e.g., a bus). Shared-memory multiprocessors can either be symmetric or asymmetric. Symmetric systems imply that all processors that compose the system are identical. Conversely, asymmetric systems have different types of processors sharing memory. Most multicore chips are single-chip symmetric shared-memory multiprocessors.

2.1.10 - SoC

A System on Chip (SoC) refers to a single-integrated circuit (chip) composed of all the components of an electronic system. SoC technology did not change the functionality of the systems, it is heterogeneous, it is used to describe chips integrating on a single silicon die what was before spread on several circuits.

A SoC may contain digital components (processor, memory, hardware device drivers, bus, etc.), analogue and radio components.

The SoC market has been driven by embedded computing systems: mobile phones and handheld devices.

2.1.11 - VLSI Computation

VLSI Computation (computation within the Very-Large-Scale-Integrated technology) concerns the analysis of the computations realized by large integrated networks, whereby the traditional distinction between networks and computations disappears (each network is "dedicated" to the execution of a particular algorithm). Specific algorithms realized by such circuits are evaluated in terms of their efficient use of the integrated technology.

2.2 - Measuring the Performance of Schedulability Tests

In the work of Bini et al [5], it is discussed and compared three different metrics that can be used for evaluating the performance of schedulability tests.

The compared metrics are the breakdown utilization, the utilization upper bound and the optimality degree.

Synthetic task generation is then investigated to make conclusions on how the random generation procedure can bias the simulation results of some specific scheduling algorithm. It is presented efficient method for generating task sets with uniform distribution with the UUniFast Algorithm.

The main result achieved from this study is that current metrics intrinsically evaluate the behaviour of RM in pessimistic scenarios, which are more critical for fixed priority assignments than for dynamic systems. The use of unbiased metrics, such as the Optimality Degree, shows that the penalty paid in terms of schedulability by adopting fixed priority scheduling is less than commonly believed.

This is an important study that gives us the possibility to understand the metrics when considering to evaluate scheduling algorithms and how to generate enough tasks sets randomly to evaluate a computing system.

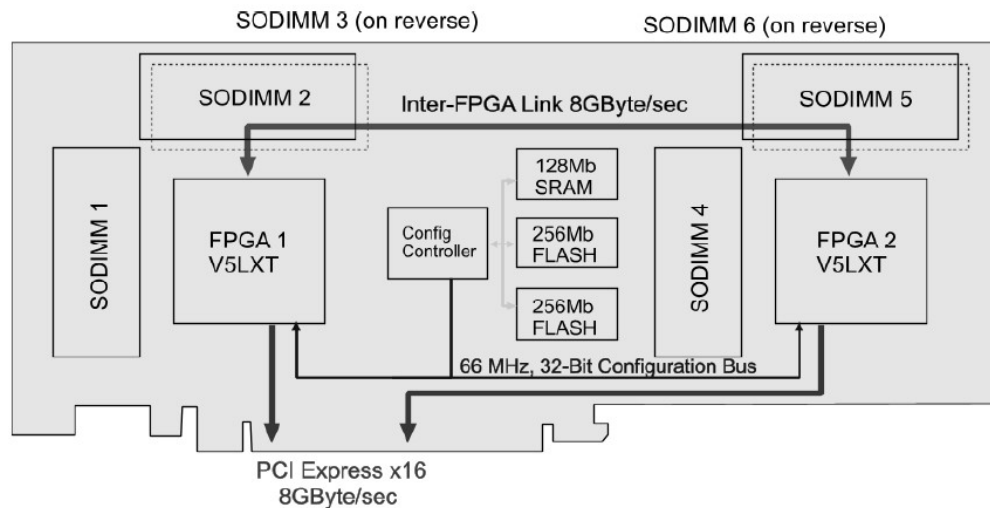


Figure 2.1 - MAX2 board [3]

2.3 - Finding Speedup in Parallel Processors

In the work of Flynn et al [3], it was proposed an acceleration methodology based on FPGA arrays. The methodology uses high performance FPGA hardware supported by a comprehensive application analysis.

Rather than basing the HPC (high performance computing) computational paradigm on MIMD multiprocessors it was proposed an alternative based on a heterogeneous node consisting of a host processor plus and high density computational array.

A prototype consisting of a board with two Virtex FPGAs and 24GB DDR2 of buffer storage was evaluated as seen in figure 2.1.

Geophysical Modelling was implemented in a MAX2 and compared with an Intel Xeon implementation provided more than two orders of magnitude speedup.

This work, made by reference authors in 2008 is directly related to this study and gives important insights, i.e., when talking about the cylindrical model.

2.4 -Self-adaptive loop for CPSs: is the Dynamic Partial Reconfiguration profitable?

In the work of D'Andrea et al [6], that focuses in the area of edge computing, a run-time manager for the partial reconfiguration is introduced. It is adopted a metric to evaluate the impact of reconfiguration time. Its validation through its usage on a basic application implemented on FPGA is as well made.

“When exploiting FPGAs with DPR, a crucial problem is to understand whether is profitable or not to dynamically reconfigure it. In fact, the process requires that a configuration file is transferred from a storage memory to a reconfiguration memory, requiring some time, depending on both configuration file size and available bandwidth; this impact, if not well considered, can nullify the advantage obtained using a DPR” [7].

In figure 2.1 we can observe the high level block diagram of the run-time reconfiguration manager.

The monitor was made to work within various different computing systems that can be related to edge computing and can measure different variables depending on the applications and the metrics used. It was made to be the less intrusive as possible in order to do not cause interference on the application execution time.

Analysis “is a phase where a component, called analyser, interprets the raw information coming from the monitoring system, in order to obtain indications about performance” [7].

Plan is where the decision is made.

The execution phase is in reference to the actual partial reconfiguration of the system. Partial Bitstreams are transferred and the reconfiguration is made. “The configuration files are prepared at design-time, due to the complexity of the FPGA synthesis operation”.

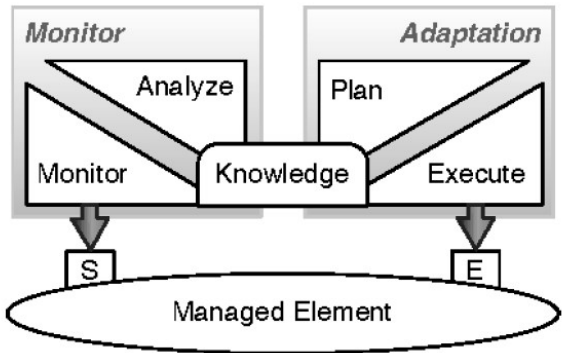


Figure 2.2 - Proposed Manager [6]

The prototype presented in figure 2.2, made to face timing performance losses in edge computing presents the manager that evaluates whether dynamic partial reconfiguration could be of use or not follows the adaptation phenomena called self-adaptive loop model [7].

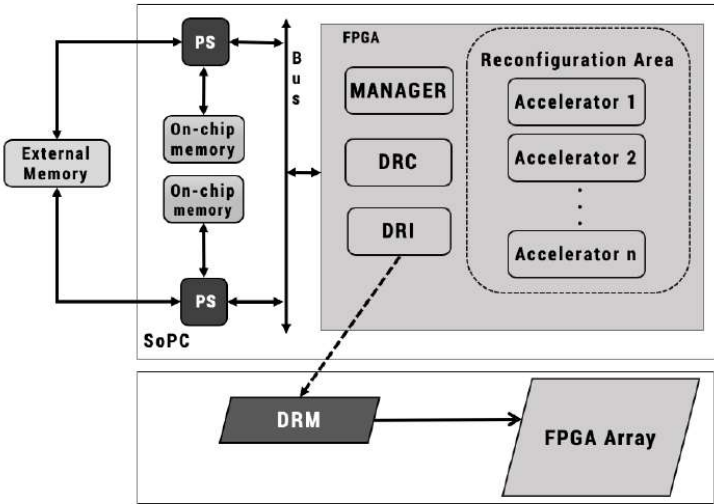


Figure 2.3 - Proposed Prototype [6]

The authors opted to store partial bitstreams in the external memory and transfer it to the on chip memory reserved to an ARM core in the processing system. Both o the cores of the ARM have its own OCM. The reconfiguration is done then by the DRC (dynamic

reconfiguration controller), using the DRI (dynamic reconfiguration interface) to store the bitstream in the DRM (dynamic reconfiguration memory). The reconfiguration of a DRB (dynamic reconfigurable branch) is considered complete when the BS is stored in the DRM.

“Dedicated experimental applications have been designed and developed to show the run-time manager profitability, i.e., the capability to adapt the system to possible application changes: it has to guarantee that timing performance are hold, in a scenario where asynchronous disturbances, i.e., that happens independently from the application, are present” [6].

“The proposed model is quite general and it has been proposed since, in the state of art, there are a number of local solutions to the DPR profitability evaluation, and only part of them consider the impact of DPR time” [6]. First validation activities have been done considering simple applications in order to infer about the profitability and reusability of the proposed manager. Future work will consider unmanned aerial vehicles able to guarantee certain timing performance on image computation using DPR.

It is an interesting work that is directly related to our study and despite being DPR an area that will be subject of developments, gives us important insights in order to proceed with our evaluation.

2.5 -CoRQ: Enabling Runtime Reconfiguration Under WCET Guarantees for Real-Time Systems

In the work of Damschen et al [8], it was presented concepts that enable runtime reconfiguration under Worst Configuration Execution Time (WCET). It was detailed “the challenges of runtime reconfiguration in real-time systems and show that conflicts while accessing a shared main memory during reconfiguration can lead to a slowdown of more than $21\times$ in reconfiguration bandwidth”. It was presented a new reconfiguration controller.

It is discussed how one approach of improving WCET guarantees of a kernel using runtime reconfiguration which is the stalling approach, a task that reconfigures an accelerator using stalling, stalls its execution for the whole reconfiguration delay, once the reconfiguration is completed, the task proceeds execution in software and executes the reconfigured hardware accelerators. It is made the assumption that the reconfiguration delay can be determined statically and can be added to the total time of execution.

An approach that enables the CPU to perform useful operations in parallel to reconfiguration is prefetching. A considerable amount of reconfiguration delay has already passed at the point in time when the accelerators are actually needed. It provides considerable performance improvements but, for real-time systems, however, prefetching poses great challenges.

The prototype, seen in figure 2.4, was evaluated and guaranteed reconfiguration delays for the stalling and prefetching approaches for a uniprocessor system.

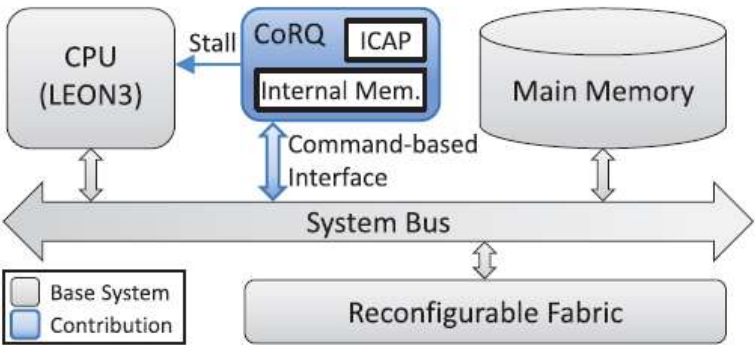


Figure 2.4 - SoC Overview [8]

2.6 - Dynamic partial reconfiguration in FPGAs

In the work of Lie et al [9], focuses on the advantages of the dynamic partial reconfiguration design flow.

They developed a prototype in order to describe the advantages of early access partial reconfiguration when in comparison with difference based partial reconfiguration or module based partial reconfiguration, seen in figure 2.5.

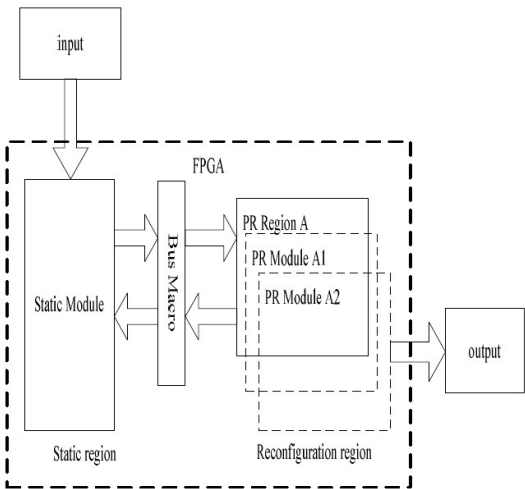


Figure 2.5 - Design Architecture [9]

The direct benefit is less space needed for storing the necessary configurations for operation. As reconfiguration times are highly dependent on the size and organization of the PRRs, an additional benefit is that the reconfiguration time is shorter.

2.7 - Quantifying the Benefits of Dynamic Partial Reconfiguration for Embedded Vision Applications

In the work of Nguyen et al [10], quantify the benefits of dynamic FPGA mapping (with DPR) over traditional static FPGA mapping for two vision applications, using a smaller FPGA but still meeting the functional and performance requirements.

2.8 - Conclusions

In this chapter we presented some of the related book and works.

In 2.1 we presented some interesting entries for this project.

From 2.2 to 2.7 we presented some of the works made in the last few years closely related to this project.

Chapter 3

Overview of The System

For the purposes of this master thesis a ZedBoard was chosen because it is a good option, when considering the questions, we want to answer. It suits well the purposes of our job.

3.1 Chosen Board: ZedBoard, Zynq®-7000 SoC

Zynq 7000 SoC is a device that integrates the software programmability of an ARM based processor, a dual-core ARM Cortex-A9, with the hardware programmability of a Field Programmable Gate Array (FPGA) logic fabric, based on Xilinx 7-series FPGA architecture, in one single chip. This is completed by industry standard AXI interfaces, which provide high bandwidth, low latency connections between the two parts. It was made to be the best price to performance-per-watt, fully scalable SoC platform.

Simplifying the system to a single chip include reductions in physical size and overall cost and other benefits while being capable in terms of resources and performance. Zynq is a very flexible system on a chip (SOC).

The development of a complete embedded system is a significant design task, and there are particular advantages to undertake the design on a platform such as an FPGA or Zynq device, which make the process more straightforward. Xilinx has a wide variety of standard IP, peripheral components and interconnections, libraries and standards with performance characteristics very well-known and integrated into the software development tools and drivers, meaning that there is no need to redesign the project when considering various different Xilinx platforms. Reusing components in the form of pre-tested and verified IP, development can be accelerated and costs can be lowered.

It was the first device of its kind in the market, never seen before. In many areas as space exploration, military, telecommunications and, for example, cryptography, this device can be a good option as it could meet the tight timing, power or flexibility constraints, keeping the time to market, TTM, low and at a low price.

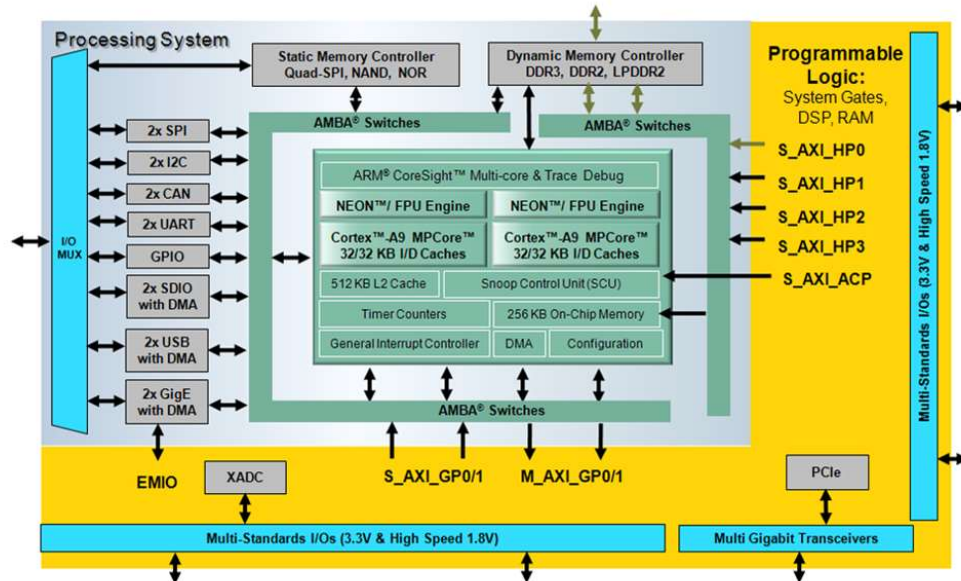


Figure 3.1 - Zynq®-7000 SoC block diagram

Mainly composed of two different parts, the Programmable System, PS, and the Programmable Logic, PL, it can be easily configured to have accelerators as peripherals instantiated in the logic fabric, and an Operating System, OS, running in the ARM that can decide which and when the resources are allocated to a specific application, while the AXI interface, part of Advanced Microcontroller Bus Architecture, AMBA, and Direct Memory Access engine, DMA, allow to transfer data and information very quickly to meet timing constraints.

As we can see in figure 3.1, the Zynq processing system is not only the ARM processor, but, as well, a set of associated processing resources forming an Application Processing Unit (APU), and further peripheral interfaces, cache memory, memory interfaces, interconnect, and clock generation circuitry.

In figure 3.2, the PL of the smaller Zynq devices corresponds to the fabric of Artix-7 FPGAs, while the larger ones are equivalent to Kintex-7. It is predominantly composed of general purpose FPGA logic fabric, which is composed of slices, Configurable Logic Blocks (CLBs), there are also Input/Output Blocks (IOBs) for interfacing. The model used in this project has five different clock regions.

In addition to the general fabric, there are two special purpose components: Block RAMs for dense memory requirements; and DSP48E1 slices for high-speed arithmetic. It is often an advantage to use distributed RAM for small memories due to efficiency, performance and a more flexible placement, but with the Block RAM, which can normally be clocked at the highest clock frequency supported, 250 MHz, we can store a large amount of data in a small physical space of the device. The LUTs in the logic fabric can be used to implement arithmetic operators of any arbitrary length, but are most suitable for arithmetic operators with short word lengths (arithmetic circuits for long word lengths can have a large footprint in slice logic, with placement and routing factors resulting in sub-optimal clock frequencies).

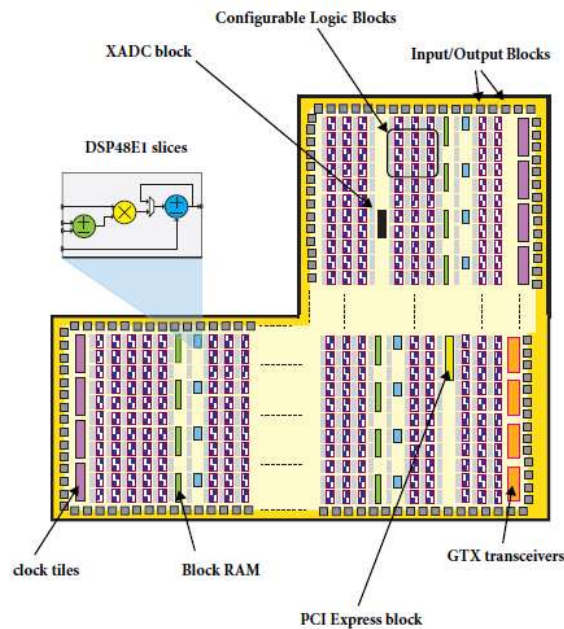


Figure 3.2 - Zynq Programmable Logic (PL)

DSP48E1s are special slices for implementing high-speed arithmetic on signals with medium to long arithmetic word lengths.

A soft processor could be implemented (i.e. MicroBlaze) in the PL, which could be an advantage when compared with other hybrid systems, for instance because it has a much more flexible configuration, the actual footprint could be changed, occupying more area for a better performance, in order to have more features or to work in a specified different clock frequency. In the other hand we could reduce the area optimizing the soft core for the

operations it will need to do. MicroBlaze resource utilization varies with configuration, starting at approximately 900 LUTs, 700 FFs, and 2 Block RAMs for the ‘minimum area’ option, and rising to about 3800 LUTs, 3200 FFs, 6 DSP48E1s and 21 Block RAMs for the ‘maximum performance’ configuration.

3.2 Developed Prototype

As seen in figure 3.3, it was decided to use one of the cores of the ARM processor to run operative system features while the other core is used to run the applications in parallel with the Reconfigurable Partitions in the PL.

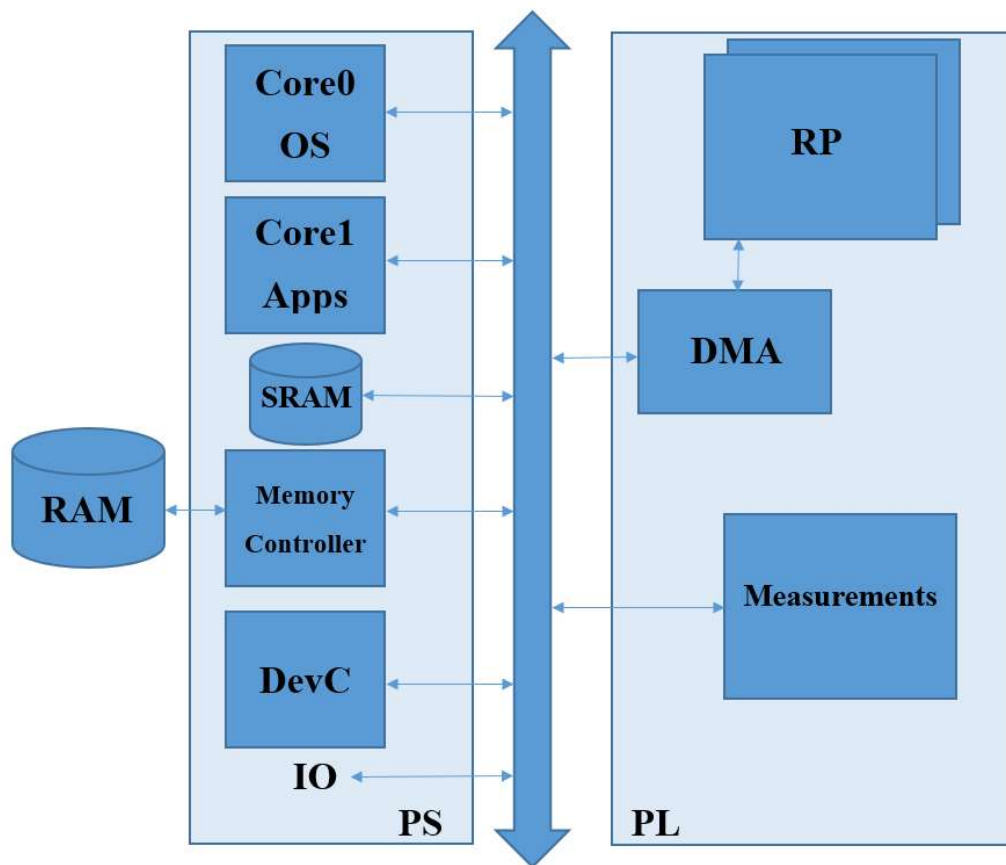


Figure 3.3 - High Level Block Diagram of the System

This is a simple system in itself, this device could have been configured in a more elaborated manner, possibly with a soft core processor that could have been tailored to control the PR features, the use of special memories in the PL, or independent memory added as peripheral to the board. Although that could have brought better performances the aim of this study is to make analyses at runtime and decide based on the obtained results,

and so, it was decided to simplify at most the possible configurations provided that one could get good results while making the desired studies.

3.3 Partial Dynamic Reconfiguration of FPGAs

Almost as old as the FPGA itself the reconfiguration of FPGAs at runtime was an important feature to automate the process of emulating and validate different circuits when designing ASICs.

As the FPGAs evolved into being devices that could be used to accelerate specific task sets within an embedded system, easing the job of the GPPs, this feature gains more importance because it maximizes the flexibility of the system, which is one of the pros brought by the FPGA while delivering high performance.

Partial dynamic reconfiguration (PDR) allows a programmer to reconfigure a portion of the FPGA dynamically, while the remaining FPGA design continues to function. One can define multiple accelerators for a particular region in the design, without impacting operation in areas outside this region. This methodology is effective in systems with multiple functions that time-share the same FPGA device resources [11]. PR enables the implementation of more complex FPGA systems and there are nowadays standard ways to implement a project with a partially reconfigurable part a runtime.

Xilinx introduced and supported the notion of a Partial Bitstream. Having a Bitstream for the static implemented design in the FPGA, and of the reconfigurable partitions as black boxes. Reconfigurable Modules, RMs, with no functionality can be delivered as part of the initial configuration, to be later replaced with a desired Reconfigurable Module. We can then create a Partial Bitstream for each configuration we need to have, with different accelerators actually implemented and functioning in the Reconfigurable Partition, RP considered to be an actual reconfiguration of the logic fabric at runtime.

The Static Bitstream is the full configuration bitstream. All PR designs start with standard configuration of the full device using a full configuration bitstream. The format and structure here is no different from a flat design solution. There is no difference in how this bitstream can be used to initially program the FPGA. However, the design itself is prepared for partial reconfiguration of the device after the full programming has been done. Bitstream compression can be very effective in this case, reducing bitstream size and initial configuration time.

All configurations use the same top-level, or static, placement and routing results. Dynamic Function eXchange, DFX, allows for the reconfiguration of modules within an active

design. DFX is a comprehensive solution that is comprised of many parts. These elements include the Xilinx silicon ability to be dynamically reconfigured and the Vivado software flow for compiling designs from RTL to bitstream [11].

The two leading companies in the market of FPGAs, Xilinx and Intel, have already, as well, their standard ways to develop a project with partial reconfigurable features and IPs to ease the process of controlling the reconfiguration.

The Xilinx AXI Hardware Internal Configuration Access Port (HWICAP) LogiCORE IP core for the AXI Interface enables an embedded microprocessor, such as the MicroBlaze processor, to read and write the FPGA configuration memory through the Internal Configuration Access Port (ICAPEn) [12]. The ARM processor can be used to control the reconfiguration as well.

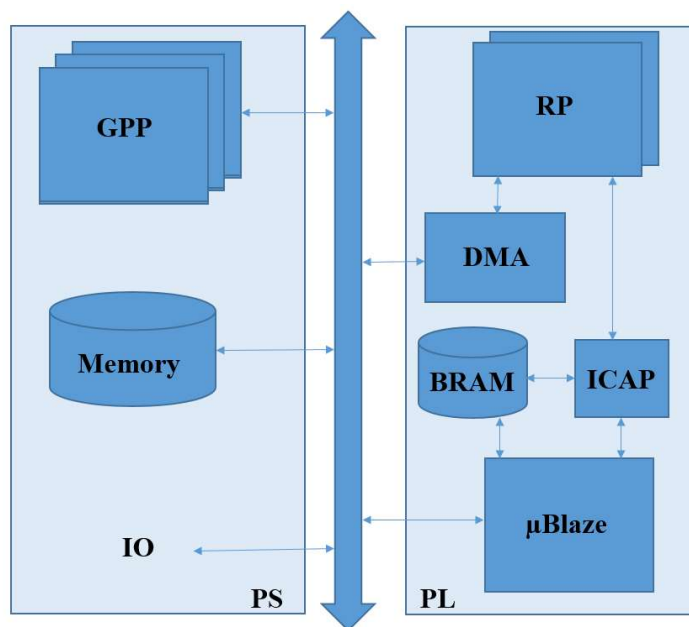


Figure 3.4 - High Level Block Diagram of the System with ICAP Configuration

As seen in figure 3.4, with the ICAP, we could have an Operative System, OS, running in one or various of the GPPs, and an application that would have to communicate with the soft core processor, in this case the μ Blaze, in order to proceed with a partial reconfiguration of the logic fabric. A soft-core processor is a hardware description language, HDL, model of a specific processor (CPU) that can be customized for a given application and synthesized for an ASIC or FPGA target. There are several soft-core processors available from commercial vendors and open-source communities. As the complexity of embedded systems continues to increase, it is expected that the usage of customizable soft-core processors will become more widespread.

In this case, of figure 3.4, not only the processor would be free to be running other applications but also, we have a special memory instantiated in the FPGA, to store the partial

bitstreams and other needed variables by the μ Blaze, for example, for controlling the partial reconfiguration. This would leave at all times the memory controller, the DRAM, and the DMA engine, free to proceed as needed with the data transfers to process the applications in the queue. The ICAP is the standard and most used way to reconfigure the FPGA at runtime for Xilinx devices.

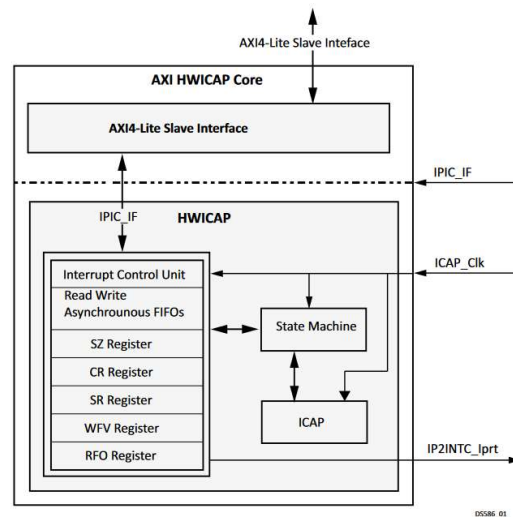


Figure 3.5 - Top Level Block Diagram for the AXI HWICAP Core, Source Xilinx

Figure 3.5 shows the internal structure of the ICAP.

This is not true in the case of Zynq where if we want to control the reconfiguration flow in the GPP, we can use the Processor Configuration Access Port, the AXI-PCAP bridge, already implemented in the PS, as part of Device Configuration, DevC. The AXI-PCAP bridge converts 32-bit AXI formatted data to the 32-bit PCAP protocol and vice versa. A transmit and receive FIFO buffer data between the AXI and the PCAP interface. A DMA engine moves data between the FIFOs and a memory device, typically the OCM, the DDR memory, or one of the peripheral memories. The 32-bit PCAP interface is clocked at 100 MHz and supports 400 MB/s download throughput for non-secure PL configuration and 100 MB/s for secure PL configuration where data is sent only every 4th clock cycle. To transfer data across the PCAP interface a DevC driver function needs to be called. The driver will take care of setting the correct PCAP mode and initiating the DMA transfer. The function call will only return after both the AXI and the PCAP transfers are complete [13].

The Xilinx Partial Reconfiguration AXI Shutdown Manager safely handles AXI4MM and AXI4-Lite interfaces on a Reconfigurable Partition when it is undergoing partial reconfiguration (PR), preventing system deadlock that can occur if AXI transactions are interrupted by PR.

One or more Partial Reconfiguration AXI Shutdown Managers can be used to make the AXI interfaces between a Reconfigurable Partition and the static logic safe during Partial Reconfiguration (PR). When active, AXI transactions sent to the Reconfigurable Module (RM), and AXI transactions emanating from the Reconfigurable Module, are terminated by the core because the Reconfigurable Module might not be able to complete them. When inactive (In Pass Through mode), transactions are passed unaltered [14].

The Xilinx Partial Reconfiguration Decoupler can be used to provide a safe and managed boundary between the static logic and a Reconfigurable Partition during Partial Reconfiguration [15]. The Decoupler can be important to avoid several undesired effects that can occur during the reconfiguration, namely, undesired values can be driven to the static logic (signals might glitch, can be driven to 1 by the interconnect or they might be driven by a reconfigurable module that has not yet been reset and is in an unknown state) and signals driven by the static logic into the Reconfigurable Partition, RP, that is undergoing reconfiguration can cause the newly loaded Reconfigurable Module, RM, to become corrupted (spurious writes to memories can occur or parts of the reconfigurable module can start to operate while other parts do not).

The Xilinx Partial Reconfiguration Bitstream Monitor can be used to identify partial bitstreams as they flow through the design. This information can be used for debugging or to help manage system applications such as blocking bitstream loads [16].

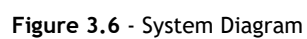
Finally, the Xilinx Partial Reconfiguration Controller core provides management functions for self-controlling partially reconfigurable designs. It is intended for enclosed systems where all of the Reconfigurable Modules are known to the controller. The optional AXI4-Lite register interface allows the core to be reconfigured at run time, so it can also be used in systems where the Reconfigurable Modules can change in the field [17].

3.4 Partial Dynamic Reconfiguration Control

As stated before, the partial reconfiguration of the PL will be done with the AXI PCAP and the DMA controller as part of DevC, it is the standard option to do so when talking of Zynq and a better option with higher throughput that would lead to better performances would be only achieved with the alteration of the ICAP possibly overclocking this IP, with the development of a new IP in order to control the partial reconfiguration or with the compression of the bitstream before the transfer for the reconfiguration is made.

3.5 Data Transfer

“Data access plans determine where data arrays reside, typically choosing between disk, CPU main memory, FPGA DRAM memory or FPGA on-chip memory, and then develops an optimized memory hierarchy for the application.” [18]



Two DMA controllers are used. One as part of DevC, which with the PCAP, controls and executes the reconfiguration of the PL. The others instantiated in the PL as part of the static logic, controls the transfer of data from the DRAM memory to the accelerators in the PL.

The use of DMA controller to proceed with the transfer of data to the accelerators, combined with the pre fetching of data to the ARM core that runs applications is of crucial importance in order to have a good performance and to improve the throughput, because, despite the AXI interfaces already being made to fulfil this tight timing constraints, the great amount of information and the attempt of different resources trying to access the memory at the same time can lead to poor performances and a time to transfer the data 10 to 20 times greater. The throughput of the DMA controller at 100MHz is 400MB/s [18].

Unlike Programed Input/Output, PIO, that blocks the processor in each transfer, and that could drastically slowdown the overall cycle time particularly when considering big amount of data transfer being only a good option, in example, i.e., to control the flow of a task set, DMA, can significantly reduce the time to transfer the data while freeing the processor to proceed with the control of the execution of the task set. The time to transfer the data is proportional to the number of words to be transmitted in the memory mapped interface and, in the streaming interface, to the number of packets. This component can have a significant jitter if data transfer is not well controlled, making this time ten to twenty times greater than the minimum calculated value, that is because we can get to a state in which, i.e., two different applications are trying to access and control the DMA controller and each restarts the process of the transference of data not letting it finish before the other tries and gets the access to the controller. To reduce that uncertainty to an acceptable minimum, for the purposes of this study, all other communications between a parallel processing component and the DRAM memory are stalled and waiting for the DRAM memory controller to be free and had already finished the previous transfer of data, as we will see later in the chapter, when studying the execution time model.

As seen in figure 3.6, there are three interfaces connected to the smart interconnect, those are the scatter gather mode interface and the M_AXI_MM2S, memory mapped to streaming interface, from the PS, connected to the to the DRAM memory controller in order to get data from main memory to the accelerators in the PL, which must have a streaming interface. The other connection is the streaming to memory mapped interface that is needed to write data in the DRAM memory, possibly, the results of calculations of tasks accelerated in the FPGA.

Beyond those previous stated connections there are AXI Lite connections to allow the reconfiguration of the DMA controller at runtime, that could be done to transfer different amount of burst or with different burst sizes, which could be done by the operative system.

Furthermore, as seen, we can configure the controller to interrupt one of the cores of the processor in order to control the transference o data.

3.6 Main Memory

The memory used to store data and partial bitstreams will be the DRAM memory, which despite being a good option is not the best option at all as it could be bottleneck in this system because the operative system would have to share it, controlling the access to this resource.

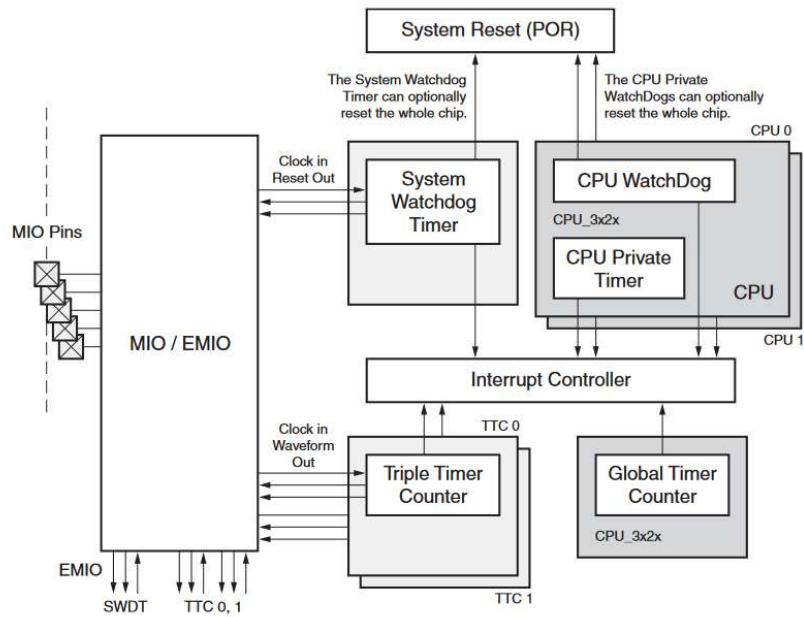


Figure 3.7 - System View [10]

That would have to be made by the DevC before partial reconfigurations of the PL to get the bitstreams, and by the Accelerators in the PL and the ARM core that runs applications, to transfer the data needed by the applications. A better solution would be to use the OCM to store the data needed by the applications and implement a special memory in the PL to store bitstreams and control the reconfiguration with the AXI ICAP as this configuration would allow to make transfers of data and bitstreams in parallel without having to stall one of the resources. It was decided to stop all data transfer operations while the DMA controller is operating whether it is for the reconfiguration of the PL whether to transfer the data, in order to minimize the variations in the time to transfer the data and bitstreams and to have more precise and expected results.

There are several possibilities when considering storing the data needed for applications and as well for booting and partial and static configurations of the PL.

The booting and OS, as well as, static and partial configurations can be loaded from, either, the flash memory or, i.e., the SD card or other added peripheral solid state memory. From that point, we can store data in different resources already present on the board.

The on-chip memory (OCM) module contains 256 KB of RAM and 128 KB of ROM (BootROM). It supports two 64-bit AXI slave interface ports, one dedicated for CPU/ACP access via the APU snoop control unit (SCU), and the other shared by all other bus masters within the processing system (PS) and programmable logic (PL). The BootROM memory is used exclusively by the boot process and is not visible to the user. OCM supports high AXI read and write throughput for RAM access by implementing the RAM as a double-wide memory (128 bits). To take advantage of the high RAM access throughput, the user application must use even AXI burst sizes and 128-bit aligned addresses [19].

The static memory controller (SMC) can be used either as a NAND flash controller or a parallel port memory controller supporting, i.e., SRAM memory, with fast access possibilities. The APB bus interface provides a memory mapped area for the software to read and write the control and status registers [19].

The DDR memory controller supports DDR2, DDR3, DDR3L, and LPDDR2 devices and consists of three major blocks: an AXI memory port interface (DDRI), a core controller with transaction scheduler (DDRC) and a controller with digital PHY (DDRP). The DDRI block interfaces with four 64-bit synchronous AXI interfaces to serve multiple AXI masters simultaneously. The DDRC performs DDR data service scheduling to maximize DDR memory efficiency. It also contains fly-by channel for low latency channel to allow access to DDR memory without going through the CAM. The PHY processes read/write requests from the controller and translates them into specific signals within the timing constraints of the target DDR memory. The DDR pins connect directly to the DDR device(s) via the PCB signal traces. The system accesses the DDR via DDRI via its four 64-bit AXI memory ports. One AXI port is dedicated to the L2-cache for the CPUs and ACP, two ports are dedicated to the AXI_HP interfaces, and the fourth port is shared by all the other masters on the AXI interconnect. The DDR interface (DDRI) arbitrates the requests from the eight ports (four reads and four writes). The arbiter selects a request and passes it to the DDR controller and transaction scheduler (DDRC) [19].

Finally, we can have special memories, as a BRAM, instantiated in the PL, and control transactions and the data flow, from and to, the DRAM memory and PL block memory. This could greatly improve performance because more parallelization could be added because different resources processing the task could access different memory spaces at the same time.

3.7 Measurement Infrastructure

Profiling is a method by which the software execution time of each routine is determined. It is used to determine critical pieces of code and optimal code placement in a design. Routines that are frequently called are best suited for placement in fast memories, such as cache memory. One can also use profiling information to determine whether a piece of code can be placed in hardware, thereby improving overall performance.

To profile a software application, one must ensure that interrupts are raised periodically to sample the program counter, PC, value. To do this, a timer must be used. The profile interrupt handler requires full access to the timer, so a separate timer that is not used by the application itself must be available in the system. Xilinx profiling libraries that provide the profile interrupt handler support the xps_timer core. These timers should be available for exclusive use by the profile libraries. The timer interrupt signal is connected directly to the processor, or it is connected to the processor through the general interrupt controller, GIC. The utility DExplorer can be used to perform fined-grained profiling to check layer-by-layer execution time and DDR memory bandwidth. This is very useful for the model's performance bottleneck analysis. Other option would be DSight, that delivers the visual format profiling statistics to let the users have a panorama view over DPU cores utilization, so that they can locate the application's bottleneck and further optimize performance [19].

As seen in figure 3.7, each Cortex-A9 processor has its own private 32-bit timer and 32-bit watchdog timer. Both processors share a global 64-bit timer. These timers are always clocked at 1/2 of the CPU frequency (CPU_3x2x). On the system level, there is a 24-bit watchdog timer and two 16-bit triple timer/counters. The system watchdog timer is clocked at 1/4 or 1/6 of the CPU frequency (CPU_1x), or can be clocked by an external signal from an MIO pin or from the PL. The two triple timers/counters are always clocked at 1/4 or 1/6 of the CPU frequency (CPU_1x), and are used to count the widths of signal pulses from an MIO pin or from the PL [19].

Running the applications in one of the ARM cores or in the FPGA, the other core is free to be running measurements related code.

We have configured the core private timer and the GIC, general interrupt controller and created a global variable in order to measure the elapsed real time when processing a task in those resources.

Next in figure 3.8 we can see an excerpt of the code where we can see the configuration of the timer and of the interrupts just before it is needed to measure the time to run an application in this case, the sorting algorithm bubble sort.

In this case we can see a software task that runs one thousand times. The elapsed real time is stored in the global variable `TimerExpired` and stored.

```

180     int Status;
181     int LastTimerExpired = 0;
182     XScuTimer_Config *ConfigPtr;
183
184     ConfigPtr = XScuTimer_LookupConfig(TimerDeviceId);
185     Status = XScuTimer_CfgInitialize(TimerInstancePtr, ConfigPtr,
186                                     ConfigPtr->BaseAddr);
187     if (Status != XST_SUCCESS) {
188         return XST_FAILURE;
189     }
190     Status = XScuTimer_SelfTest(TimerInstancePtr);
191     if (Status != XST_SUCCESS) {
192         return XST_FAILURE;
193     }
194     Status = TimerSetupIntrSystem(IntcInstancePtr,
195                                 TimerInstancePtr, TimerIntrId);
196     if (Status != XST_SUCCESS) {
197         return XST_FAILURE;
198     }
199     XScuTimer_EnableAutoReload(TimerInstancePtr);
200     XScuTimer_LoadTimer(TimerInstancePtr, XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ / 2000000);
201     XScuTimer_Start(TimerInstancePtr);
202     while (1) {
203         bubbleSort(arr, n);
204         xil_printf("time is %d\r\n", TimerExpired);
205         LastTimerExpired++;
206         TimerExpired=0;
207         if(LastTimerExpired == 1000){
208             XScuTimer_Stop(TimerInstancePtr);
209             break;
210         }
211     }
212

```

Figure 3.8 - Measuring Time, code excerpt

As soon as we have 1000 measurements the timer is stopped and the average measured time is calculated.

In this case there is an interruption that occurs each time the timer expires. The value 0xFFFF was loaded to the timer and it was configured to interrupt the ARM core, as seen in line 200, each 1/1000000 seconds, 1 microsecond.

In each interruption of the core, the variable, `TimerExpired` is incremented, what would give us when the application was run the elapsed real time, here represented in line 204 with a print which should not be done in the real case when we actually measured the time in order to not to change the actual elapsed time because of I/O interference. Instead this number was stored for later average time calculations.

3.8 Accelerator Architecture

When developing accelerators specially made to accelerate a specific task, there is a great probability that we can get to a solution that is very good in one situation but cannot process other task unless we are talking about the one the circuit was made to accelerate.

Besides this, it would be mandatory in a project with partial dynamic reconfiguration features that the module in the reconfigurable partition is seen from the outside as the same black box, whether it is ready to accelerate one task or another, for all possible

configurations that are needed, meaning that, the streaming and other control inputs and outputs must be the same, which can significantly increase the task to develop such an accelerator.

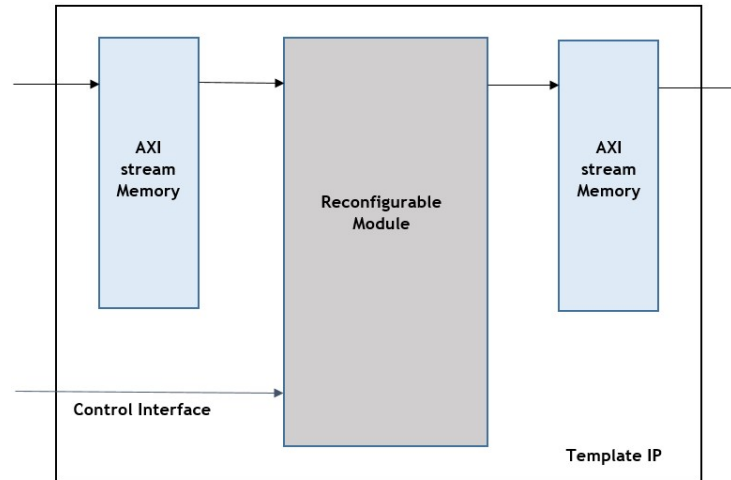


Figure 3.9 - High Level Block Diagram of the Reconfigurable IP

Figure 3.9 illustrate the high level block diagram of the template IP.

The IP must be seen from the outside as the same IP for each possible configuration needed.

The I/O are just memories with streaming interfaces and the reconfigurable module is one implementation for a specific calculations representing each task to be accelerated.

For this purpose, when using Vivado HLS, we have two possible choices. The first one would be not to care about the interfaces and then create in Vivado IP integrator one IP whose interfaces to the outside would be the IP created by Xilinx the AXI4-Stream Accelerator Adapter v2.1, that provides the AXI4-Stream interface to AXI4 infrastructure components and BRAM/FIFO interface towards Accelerator IP and complements accelerators using Vivado HLS. The second option would be to already take care of the streaming interfaces when creating the accelerator in the HLS with the libraries `#include<hls_stream.h>` and `#include<ap_axi_sdata.h>` that would allow us to create those interfaces.

3.9 Execution Model

Although we are before a new and trending area, previous studies, as this one, have shown that we have to make special considerations and options when considering a specific heterogeneous system, the set of tasks to be accelerated, and project development issues that could lead to suboptimal performances of the chosen system.

In our case, one would have to decide whether a task will be executed by the GPP or accelerated in the FPGA, with the previous knowledge that we may have to reconfigure the PL in order to do so, and, because of that, it is of extreme importance to study the timings associated with the reconfiguration. It is important, as well, to consider and verify if there is any precedence between two tasks or sub tasks of an application or if we can maximize the possibilities to parallelize those different tasks.

In table 3.1 it is explained what are the code of colours used to make illustrations of synthetic cases that can happen during the execution of different task sets.







	DMA Data or Bitstream Transfer
	OS application control
	FPGA reconfiguration
	FPGA task acceleration
	Nothing occurs, Time Delay
	Application running in the GPP

Table 3.1 - Colour Code for Execution Model

As shown in figure 3.10, after t_0 , the scheduler decides which resource to allocate to the task, in this case, the FPGA. After some time, a delay that could be experienced due to various issues, from t_1 to t_2 , the process of transference of data needed by the application starts at t_2 . This task can have significant jitter and have big variations, varying with amount of data to be transferred. Because of that it is of extreme importance to avoid conflicting situations and so it was decided to block all communications until this task is complete making a resource wait to get access to the memory until it is free again. Analyses and

measurements shown that the jitter of data transfer can lead to times ten to twenty times greater than the smallest possible value.

At the moment t_3 , the processor is controlling and starting the execution of the task in HW, which actually starts at t_4 and it is concluded at t_5 .

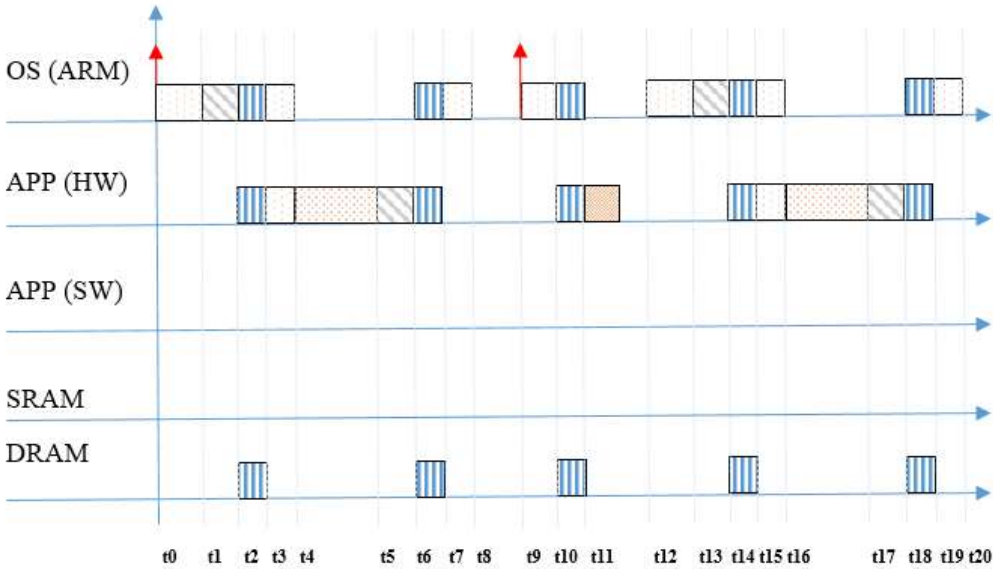


Figure 3.10 - Execution Time Model for Hardware accelerated Task (HWT)

Finally, and after the FPGA resource waits for the OS, from t_5 to t_6 , to process of retrieving the results and store them in memory is done, again stalling all other communications starting at t_6 with data transference and then from t_6 to t_7 the OS concludes the operation. The second shown HW execution of a task in the figure is similar to the first one, with exception to the first three steps and occurs when a new different task is to be accelerated in the FPGA and we need to reconfigure it before proceeding. From t_9 to t_{10} the OS already verified the we need a different configuration and started that process. First of all, we need to transfer and load the partial bitstream from the DRAM memory and then we can actually reconfigure the FPGA, which is being done from t_{11} to t_{12} possibly waiting for the OS. When it is verified that we have the correct configuration, the OS starts the execution of the task at t_{12} .

In figure 3.11 there are two different situations illustrated.

At first, it is exemplified one software execution in the ARM core. After the decision to allocate the task in the ARM, from t_0 to t_1 , there is a moment, again, that could happen for various reasons, that the system is idle, from t_1 to t_2 .

From t_2 to t_3 , all the communications are stalled due to a data transference to the SRAM, in this case reserved for ARM allocated tasks. The SRAM is used in our prototype as a reserved memory of the ARM core that runs the tasks.

Then, from t_3 to t_4 the scheduler controls the initialization of the task.

From t_4 to t_5 the ARM core is actually processing the task, and, from t_5 to t_6 , is waiting, idle for the scheduler to control the transference of results to the main memory.

The second illustrated case is the concurrent behaviour. In this case, two different tasks are executed in different resources as previously stated. The first one allocated in the ARM core, and the second whose invocation happens before the other is finished, allocated in the FPGA. This can happen simply because one of the resources is occupied, but, as well, because the calculations take us to this situation. The choice should be made not only based on the measured timing values for the elapsed real time but as well in a balanced use of the available resources.

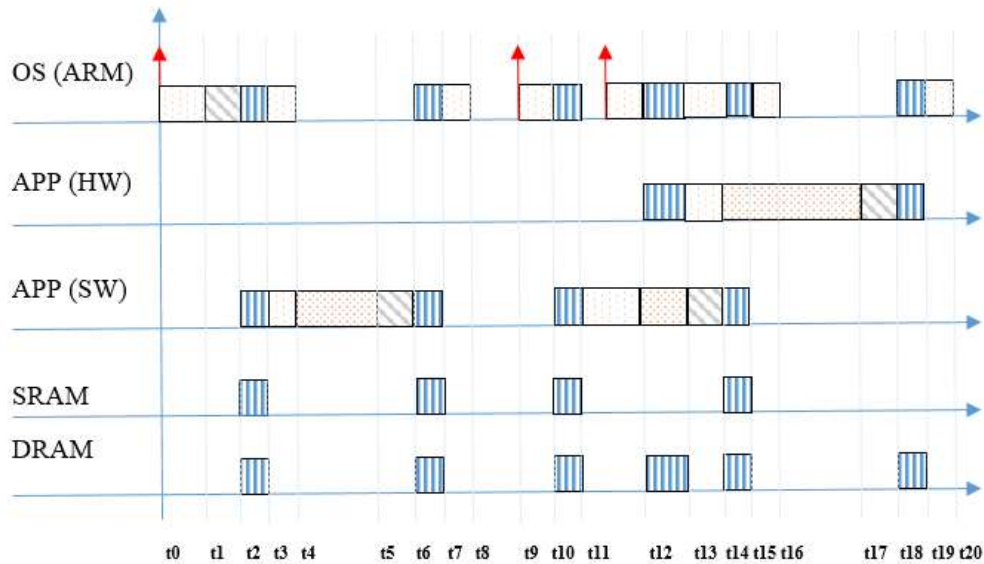


Figure 3.11 - Execution Time Model for a Software Task (HWT) and Concurrent Execution

Both the cases studied are in reference to the simple version of the scheduler demonstrated later in chapter 5. In this case we do not know beforehand the cycle of execution that can change because of different environment conditions. The simple version of the scheduler does not try to anticipate the need to partially reconfigure the system and,

as later explained, will add the average time to reconfigure the system to the average time to accelerate the task and will take those two values in account when considering to allocate the task to a specific resource. In this case, our option is to stall all other communications until the partial bitstream or the data needed is fully transferred, avoiding big variations in the time to transfer the data. This transference would be done just before it is needed.

Later studied in chapter 5 as well, are other situations not contemplated in the previous hypothetical studied cases, when the scheduler found the cycle of execution, even if only for a short period of time. Being so, the alternative would be to pre fetch the data or, particularly, the partial bitstreams in order to proceed with the reconfiguration of the system just before it is needed or already having stored the data in a memory allocated to the resource that will accelerate the task.

Chapter 4

Project Development, Algorithms and Static Analyses

Decisions at project development may be important to determine whether it's possible to have an FPGA as a peripheral with Partial Reconfiguration capabilities, to accelerate a task or not. Bottom-up/OOC synthesis (to create multiple netlist/DCP files) and management of Reconfigurable Module netlist files is the responsibility of the user.

4.1 - Partial Dynamic Reconfiguration

Standard timing constraints are supported, and additional timing budgeting capabilities are available if needed. A unique set of design rule checks, DRCs, has been established to help ensure successful design completion.

A PR design must consider the initiation of Partial Reconfiguration as well as the delivery of partial BIT files, either within the FPGA or as part of the system design.

The Vivado Design Suite includes support for the Partial Reconfiguration Controller IP. This customizable IP manages the core tasks for partial reconfiguration in any Xilinx device. The core receives triggers from hardware or software, manages handshaking and decoupling tasks, fetches partial bitstreams from memory locations, and delivers them to the ICAP.

A Reconfigurable Partition must contain a super set of all pins to be used by the varying Reconfigurable Modules implemented for the partition. If an RM uses different inputs or outputs from another RM, the resulting RM inputs or outputs might not connect inside of the RM. The tools handle this by inserting a LUT1 buffer within the RM for all unused inputs and outputs. Developing the accelerators suitable for the project with a partial reconfiguration region, particularly when this module was specially developed to accelerate a specific task

might be a difficult job, that could take 20% to 40% of the project development time and that should be done with care in order to obtain the best possible results.

One can implement an RP as a pseudo blackbox, referred to in Vivado as a greybox. Blackboxes are supported for bitstream generation. To do this, the RP must be a blackbox in the static design. The greybox has no user logic and its bitstream contains information for any static logic/routes that use resources inside the RP frames. Static routes that pass through the region, including interface nets up to the partition pin nodes, exist within this region. Programming information for these signals is included in the blackbox programming bitstream. Use of greyboxes is an effective way to reduce the size of a full configuration BIT file, and therefore reduce the initial configuration time.

When considering real time systems with very tight timing constraints, it is very important to know the worst possible scenario and to determine the time to partially reconfigure the FPGA in the worst case, even though this time is probably much smaller than the time to process the task. The time to reconfigure the FPGA and the time to transfer the data must be considered and added to the time to actually process the task. Because of that we should be careful when developing such a project.

The compression feature might also be enabled to reduce the size of BIT files. This option looks for repeated configuration frame structures to reduce the amount of configuration data that must be stored in the BIT file. The compression results in reduced configuration and reconfiguration time. When the compression option is applied to a routed PR design, all of the BIT files (full and partial) are created as compressed BIT files. This might be a good way to reduce the partial bitstream size, and therefore reduce the time to transfer bitstreams and the overall reconfiguration time. It might be important to do so in order to fulfil the project timing constraints that could be difficult to meet and make this option, of choosing an FPGA with partial reconfigurable partitions, a good option to accelerate a particular task set.

Floorplanning is required to define reconfigurable regions, per element type. One should vertically align Pblocks with frame/clock region boundaries to produce the best QoR and allows RESET_AFTER_RECONFIG to be enabled.

For user reset signals, determine if the logic inside the RM is level or edge sensitive. If the reset circuit is edge sensitive (as it may be in some IP such as FIFOs), then the RM reset should not be applied until after reconfiguration is complete.

Partial Dynamic Reconfiguration:

Despite being most of the times much lower than the overall time to process a task, the time to partially reconfigure the FPGA must not be neglected when considering an embedded system with real time constraints.

As seen in figure 4.1 there is a configuration overhead needed to configure the system in the booting phase, at power on.

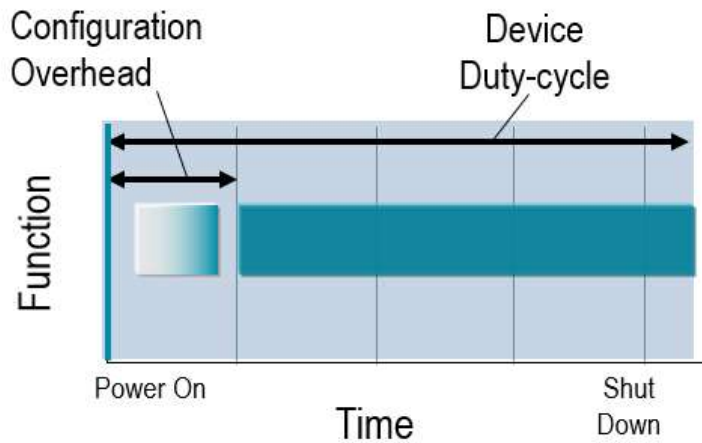


Figure 4.1 - Typical Configuration Mode Timing, Source Xilinx

The typical configuration mode, carried at power on, takes a time that could not be undervalued as it is similar to the time to process or accelerate a task.

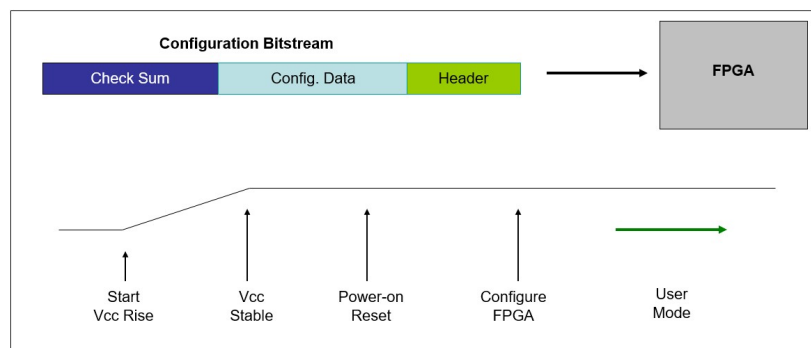


Figure 4.2 - Configuration Phases at Power On, Source Xilinx

Typically, at power on, there is a booting phase. In order to boot up an operative system and, i.e., the configurations of the FPGA, both hardware and software components of the system must be downloaded to the FPGA and program memory respectively, as we can see in figure 4.2.

During the prototyping or development phase, the hardware bitstream and software Executable and Linkable Format, ELF, file images are downloaded from the host computer to

the development board using JTAG, Joint Test Action Group, connections. We can then use Vitis to program the FPGA and debug software applications.

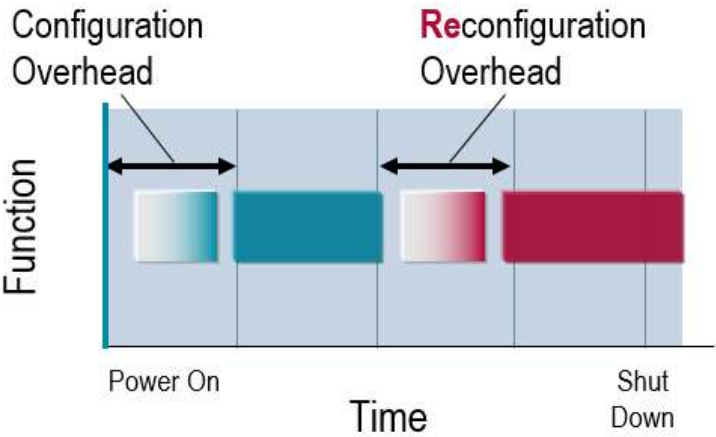


Figure 4.3 - Reconfiguration of the FPGA, Source Xilinx

At the production phase, we have a booting phase. The files stored in non-volatile memory like the OS variables or the bitstreams are loaded to, i.e., to the DRAM memory, and the FPGA is configured with the circuitry defined by those configurations.

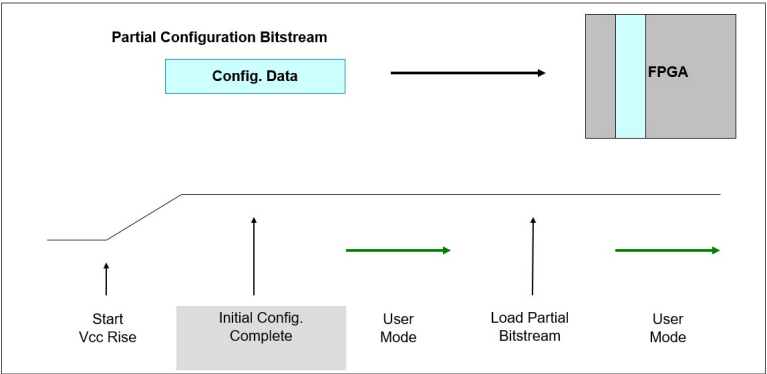


Figure 4.4 - Configuration Phases with Partial Reconfiguration, Source Xilinx

At the production phase, we can configure the FPGA with the hardware bitstream by using a configuration programmable ROM, PROM. Standard SPI or Parallel Flash memories can be used for FPGA hardware configuration. The software components of the system can be

configured by integrating them into the FPGA block RAMs. Alternately, they can be programmed into an external Flash memory or integrated into non-volatile memory.

Similar to the typical configuration mode, we can reconfigure all the FPGA at runtime. In this case, we would have a reconfiguration overhead similar to the configuration overhead, as seen in figure 4.3. The configuration memory would no longer be fixed over time as the initial bitstream with the first configuration would be loaded at power on, but later, full device bitstreams can be loaded, which would represent full devices reconfigurations.

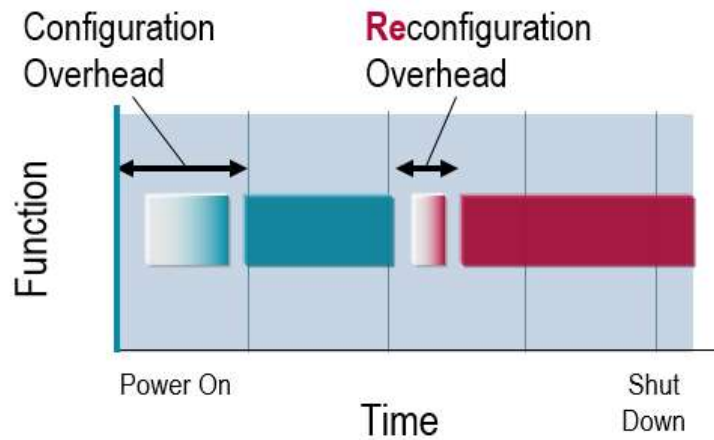


Figure 4.5 - Partial Reconfiguration of the FPGA, Source Xilinx

As we can see in figure 4.4, the partial reconfiguration of the FPGA would not need to go through all the steps previously stated to reconfigure only a part of the logic fabric, which when added to the fact that the partial bitstreams are much smaller, makes clear that a partial reconfiguration of the system would take a much smaller time to be concluded.

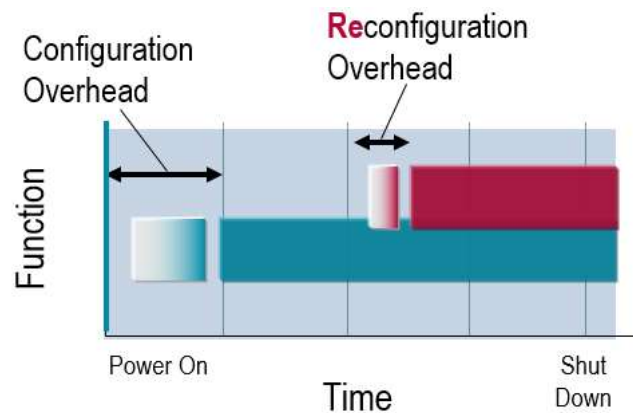


Figure 4.6 - Partial Dynamic Reconfiguration of the FPGA, Source Xilinx

As we can see in figures 4.5 and figure 4.6, with a partial reconfiguration the overhead to reconfigure is much smaller, only a subset of the configuration is altered. The difference between partial reconfiguration and partial dynamic reconfiguration is that in the first case, figure 5, all computation halts while the device is being reconfigured, and in the second, figure 6, logic layer continues operating while configuration layer is being modified and the configuration overhead is limited to the circuit that is being reconfigured.

We should place a Reconfigurable Partition within a clock region to avoid starvation. Static logic will not be placed within a RP region, therefore a Reconfigurable Module with high clock utilization can occupy an entire clock region without the clock region block range constraint. In this case 3 different clock regions of the Zync7000.

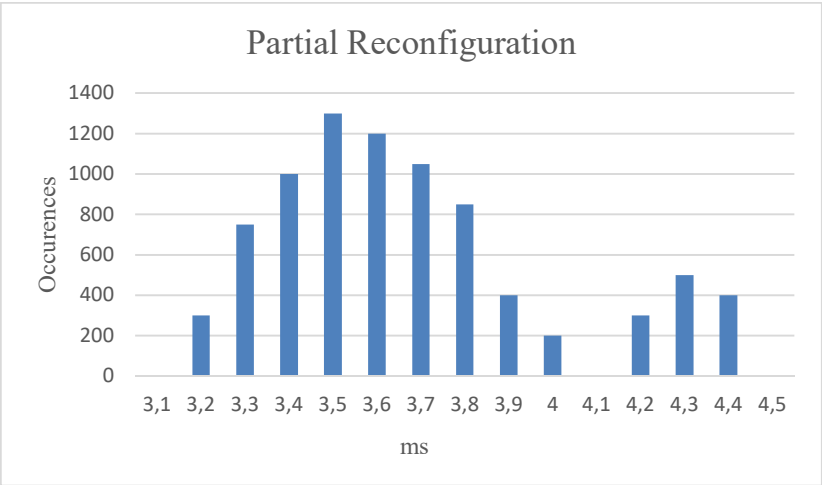


Figure 4.7 - Time to Partially Reconfigure the FPGA

As stated in the previous chapter we have made the efforts at project development to reduce the time to partially reconfigure the FPGA and reduce its variations stalling all other communications when transferring the partial bitstreams using the PCAP. We could reduce it even more if we used compression of the partial bitstreams or made alterations in the frequency of the PCAP or used a specially developed and altered ICAP with a higher throughput.

Furthermore, we have made the efforts at project development to have in our prototype a partial dynamic reconfiguration of the FPGA with static circuitry and other reconfigurable partitions working while one RP is being reconfigured.

As seen in figure 4.7, despite our efforts, the time to partially reconfigure the FPGA obtained by us had a maximum of 4,4 milliseconds, an average of 3,8 milliseconds and a

variation of 25% in 8450 measurements. This makes clear that even though this times are small, they should not be neglected and could be in many occasions a reason to process the task using the ARM processor, depending of the task set we have ahead of us.

4.2 - Algorithm Analyses

According to Flynn et al [3], “Acceleration takes place in 4 stages: (1) Analysis, (2) Transformation, (3) Partitioning and (4) Implementation.”

“The analysis stage includes understanding the algorithm, static and dynamic analysis of the program code and a software performance evaluation. The transformation stage optimizes program code, data layout and data representation for improved acceleration.” [3]

In this chapter we develop a small analyses of different algorithms in order to understand the possibilities we have to parallelize and accelerate the specific algorithm and to understand if we would have gain in terms of performance when considering to process it using a circuit implemented in a FPGA.

Besides this, we present our results for some of the implemented algorithms. Our choice was to have a checkpoint with the reconfigurable module as a black box, synthesize each of the modules out of context, and then create all the different possible configurations with the different modules.

4.2.1 Matrix Algebra

In the case of matrix algebra, we implemented a matrix multiplication.

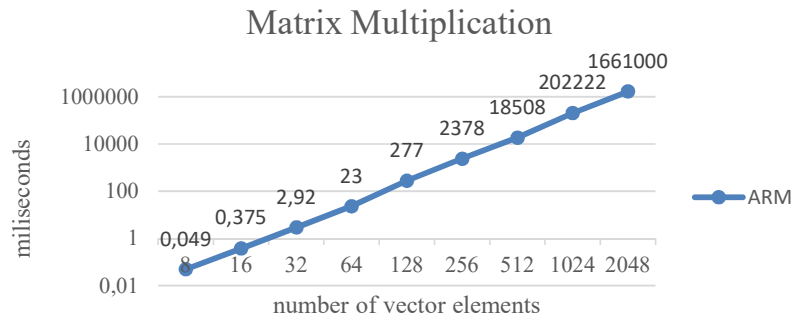


Figure 4.8: Matrix Multiplication Software Execution, ARM, Timing

As we can see in figure 4.8 matrix multiplication follows an exponential growth when considering a SW execution. Matrix multiplication exhibits high arithmetic density, has regular memory access patterns and control flow. There are already optimized hardware implementations but it was out of the scope of this work to do so.

For the implementation in HLS despite we had streaming interfaces we had to store the input values in order to proceed with the calculations. We considered square matrixes only as inputs.



Figure 4.9: Matrix Multiplication IP

In figure 4.9 we can observe the created IP block that is similar to all the other implemented algorithms.

For matrixes with dimension of 128 the HW execution time roughly match with the SW execution time. This implementation had 64 BRAM_18K, 3 DSP48E, 298 FF, 347 LUT, 0 URAM and a latency of 0.106 s. From this point forward we obtained a speedup in this case a speedup of 1.73.

For matrixes of 256X256 the values were 256 BRAM_18K, 3 DSP48E, 314 FF, 358 LUT, 0 URAM and a latency of 0.842 s and a speedup of 2.556.

For matrixes of 512X512, unfortunately the values of BRAM_18K already exceeds its maximum, we needed 1024 out of 280, 365%. For the cases of interest, with higher speedups we would have to use another board, optimize the code or obtain partial results in HW and finish the calculation in SW.

4.2.2 Sorting Algorithms

Sorting, in computer science, is the process of rearranging an unordered one-dimensional vector. Sorting is a common task in a wide range of applications. Traditional sorting algorithms were made to be processed in a GPP.

Comparator networks are devices with a fixed number of "wires", carrying values, and comparator modules that connect pairs of wires, swapping the values on the wires if they are not in a desired order. Such networks called sorting networks are designed to perform sorting on fixed numbers of values.

Sorting networks can be implemented either in hardware or in software. Donald Knuth describes how the comparators for binary integers can be implemented as simple, three-state electronic devices.

In figure 4.10 we can observe that if we use the ARM to process this task we get low results until the input vector is 4096 elements. From that point the time to sort the input vector gets higher in an exponential fashion not comparable with other execution times when compared with this and other tasks.

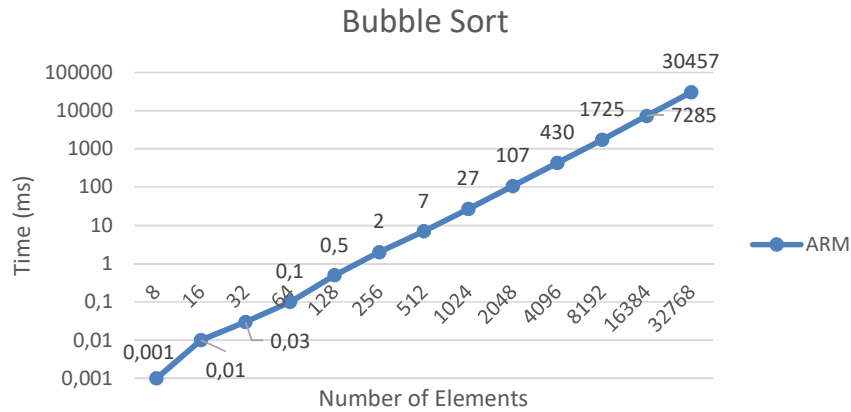


Figure 4.10: Bubble Sorting Software Execution, ARM, Timing

Our implementation obtained speedups for arrays with more than 2048 with a speedup of 1 in the worst case, 4096 with worst case scenario of 1.2 speedup, 8192 with a speedup in the worst case of 1.3, for 16374 a speedup of 1.4 and for 32768 elements a speedup of 1,4.

In example, for 16384 elements, a latency of 5s, 32 BRAM_18K, 0 DSP48E, 210 FF, 378 LUT and 0 URAM, and for arrays of 32768 elements a latency of 21.475 s with a resource utilization of 64 BRAM_18K, 0 DSP48E, 220 FF, 393 LUT and 0 URAM.

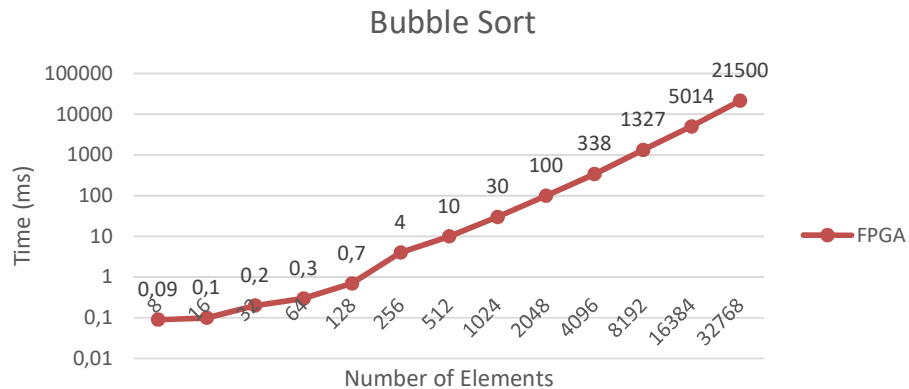


Figure 4.11: Bubble Sorting Software Execution, FPGA, Timing

In figure 4.11 we present the accelerator results for bubble sorting.

4.2.3 First Differences

Another kernel from the Livermore Loops that could be easily implemented using streaming interfaces is first differences.

When we use streaming interfaces HLS will automatically change RAM memories for FIFO, which makes ideal to implement in HW algorithms that can be accessed in a sequential order.

4.2.4 The Fast Fourier Transform, FFT

The Fast Fourier Transform, FFT, is central for many algorithms, in areas like digital signal processing, image processing or differential equations.

The FFT was one of the first algorithms to be ported to new hardware platforms and to have a hardware implementation. Xilinx and Altera have their own IP cores that can be set up for single precision and fixed point FFT computations up to 65536 elements.

Chapter 5

A Proposal for a Scheduler

In this chapter, the proposed scheduler is presented.

5.1 - Simple Version

“Partitioning involves the identification and evaluation of code partitioning options and data access plans. For code partitioning we choose whether to implement particular pieces of program code using the host PC or the accelerator array.” [1]

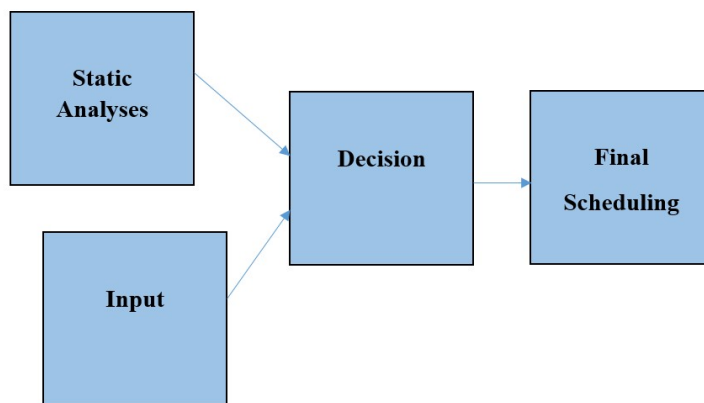


Figure 5.1 - High Level Block Diagram of the Scheduler, Simple Version

The simple version of the scheduler is presented next. In figure 5.1 we can observe the high level block diagram of the scheduler.

5.1.1 - Static Analyses

In the block Static Analyses, all the information about the tasks that will be processed is stored. The average timing values, and the worst case of the execution time, are stored in a table for each different possible input.

As we have seen, as the number of elements is getting bigger, for the proposed task set, there are more advantages to choose for a hardware acceleration compared with the execution in the GPP core in the ARM.

5.1.2 - Input

In the block Input, some parameters are inserted when the task is invoked, i.e., in the case of the sorting, the number of elements and the data itself is given to the scheduler in order to make the choice of allocating the ARM core or the FPGA to accelerate the task and proceed when possible with the control of the data transfer needed for the processing of the task. These parameters can vary from task to task as they are a specific characteristic of the task itself.

The task period remains a secret for the scheduler and will change from time to time creating a situation that could lead to different decisions in order to understand if the scheduler is making the right choice to allocate that specific resource even though that is based mostly in the average time to process the task stored in Static Analyses but as well in the current state of the system.

One of the resources can be already occupied and the scheduler may opt for a different resource if the execution timings are equivalent.

5.1.3 - Decision

The block Decision, is where the decision of allocating a resource is made.

The simple version of the scheduler would have a stalling approach when talking about the transference of the bitstreams and of the data to the memories allocated to the different resources, as studied in chapter 3. The reconfiguration would only be one if the allocated resource is the FPGA and we do not already have the right accelerator in the FPGA, in other words, if we need to have another configuration for the system. This would only be done right before that task would be accelerated, and so, in this case, we do not have a queue to store various different bitstreams or data needed by the tasks.

Here we have two different possibilities, the first one would be to consider the average time to reconfigure the system and the average time to accelerate the task, add them together in order to have the total time to process the task. The second option, a more

pessimistic approach that would be the safest one, is to consider the worst timings case to get this value, what should be done when considering systems that cannot fail to meet the timing constraints, or otherwise it could be critical.

5.1.4 - Final Scheduling

After the choice is made, and the scheduler had already decided for a software or hardware execution, and has already allocated a specific resource to do so. The block Final Scheduling only represents the actual scheduling of the task. A queue created in order to hold a number of tasks still waiting for initialization would be maintained by the scheduler. The allocation of a specific processor could be altered before the tasks start to be executed following three main objectives. The first one would be do not exceed the deadline of a task and the second represents a prediction of the time of execution of the task compared with the others in the queue and to the time of execution of the same task using different processors. These two proposals together try to minimize the total time of execution. The third objective is to balance the utilization of resources and again to minimize the total time of execution.

5.2 - Sophisticated Version

The sophisticated version tries to predict and anticipate the need to partially reconfigure the system before it is needed.

Given the assumption that, within an embedded system, the execution cycle remains the same for a given period of time, and that it could be changed only after that period, mainly because outside changes of conditions, even though we don't know what tasks would be part of the cycle we can assume that those would remain the same for a period of time and therefore it would be of use trying to discover and store information about the cycle in order to predict what task will be invoked and when will the machine will have to process a certain task.

A vector to store information about one hundred previous tasks invoked should be created. In this vector we store a structure that has information about what task was invoked, the period of the task, the number of elements of the task, the time it was needed to process it and what resource it was used to allocate the task. This is done in the block runtime analyses. At first only periodic tasks should be considered in order to smooth the analyses to be made.

We consider three different situations.

The first situation is when the system does not know what is the execution cycle. In this case the flow is similar to the simple version of the scheduler. The scheduler decides what resource to allocate when a task is invoked and reconfigures the FPGA if needed. In this case, the average time to partially reconfigure the FPGA is added to the average time to process

the task using this resource and it is considered to the total time to accelerate the task. If we get even so any improvements, the task waits in a queue until the FPGA is free.

In a second situation, we already have enough information about the task set but we still do not know the execution cycle. In this case, if one of the previous events one of the tasks take much more time than the others the system configures itself to accelerate that task, and the ARM core is used to process the others.

Finally, if we can determine the execution cycle, the system tries to anticipate the reconfiguration and has a blanking configuration in other times.

In any case, if one of the assumptions fails we go to a state of the first situation where we only reconfigure the system before it is needed.

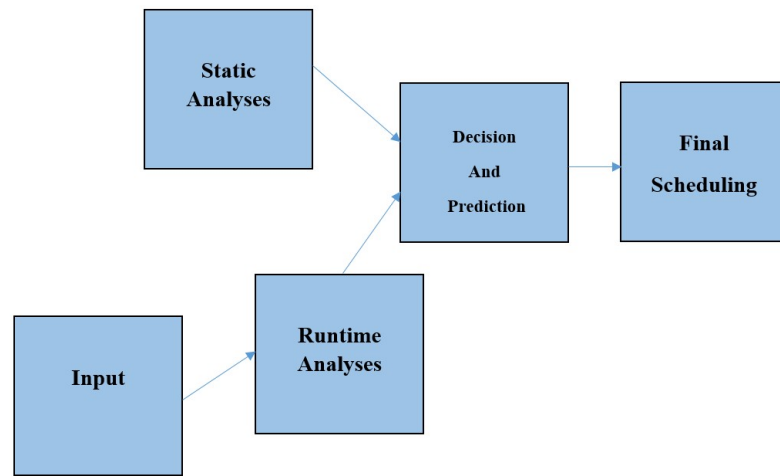


Figure 5.2 - High Level Block Diagram of the Scheduler, Sophisticated Version

In figure 5.2 we can observe the high level block diagram of the sophisticated scheduler.

5.2.1 - Static Analyses

The static analyses remain unaltered from what we have stated in 5.1.1.

5.2.2 - Input and Runtime Analyses

Similar to the case in 5.1.1, we should here use the measurement infrastructure to make calculations at runtime for the time to accelerate the tasks and to reconfigure the system. In the case of the sophisticated scheduler those calculations should be made at runtime in order to have closer values to what is really happening in the previous moments. The scheduler should not represent a big delay in the response of the system, so we would to have care in

order not to have a heavy operating system neither slow responses in comparison to the time to process the tasks.

5.2.3 - Decision and Prediction

In the block Decision and Prediction, we have implemented an algorithm to discover the cycle of execution.

As stated before we consider 3 different states for the system.

In the first case the option would be to use again the stalling approach when the transference of the bitstreams and the data.

In the second case, the scheduler starts to make risky options to maintain the same configuration even though it could not be accelerating the specified task.

In the third case, when it has discovered the execution cycle, until the cycle changes, the option would be prefetching the bitstreams and reconfigure the system just before it is needed.

The system should be prepared to this changes at runtime knowing that

- Any reconfiguration that has started should be finished until it is done;
- There will be a need to have a queue to store the bitstreams that will not be considered any time the cycle changes;

5.2.4 - Final Scheduling

Again, this remains similar from what we have stated in 5.1.4 but with the creation of more queues allocated directly to a processor in order to store partial bitstreams and data that are pre fetched when a prediction is made. This can be changed and not considered if the actual order of tasks is changed before the task starts to be executed.

Chapter 6

Conclusion and Future Work

In this chapter we present conclusions and we make reference for the work to be done in the future.

6.1 - Conclusion

We have presented prototype of a heterogeneous computing system with an ARM dual core GPP and a FPGA with the possibility of partial dynamic reconfiguration.

We came to the conclusion that it is of extreme importance to study the algorithms and if possible of the cycle of execution in order to better allocate the resources and to decide how to schedule a specific task once decided how the system would be.

Another important conclusion, about the memory hierarchy, is that we should study develop the best possible interface with different memories allocated to different processing units in order to better schedule the tasks.

This being said, our conclusion is that an FPGA with the possibility of partial dynamic reconfiguration, would be a good option depending on the tasks to be accelerated. As we shown in chapter 4, some specific algorithms are suitable to have hardware implementations and we could have very high speedups doing so, i.e., in the case of the FFT, using already developed IPs in the market, we can have a speedup of 10 o 20 times but, as we have seen, there are already other implementations with higher speedups.

Using a FPGA is much cheaper than developing an ASICs, and with one FPGA we can have a hardware implementation as much as good but with a much lower price.

Finally, with a FPGA with Partial Dynamic Reconfiguration, we can have a system that could be reconfigured as it is needed, implementing different circuits in different situations, that brings much more flexibility to the system while having lower power consumption.

Partial dynamic reconfiguration can be an important feature in areas like cryptography, space exploration or in embedded systems that could perform slightly different sequences of tasks for different conditions of the environment.

It is important, during project development, to consider and study the process of the reconfiguration of the FPGA. Xilinx has already IPs related to Partial Dynamic Reconfiguration of FPGAs, we can use, in the case of Zynq7000, the ICAP or the PCAP to control the reconfiguration of the system. If needed to have a higher throughput we would have to develop another IP to control this process.

Floorplanning is important in order to have static elements or other circuitry that continue to work while the reconfiguration is done, each of the reconfigurable partitions must be implemented in different clock zones and may occupy all the resources within that clock zone in exception to logic associated with the clock.

Another way to further reduce the time to reconfigure the system, at project development, is to use compression of the partial bitstreams. Reducing the size of the bitstreams would reduce the time to transfer the data needed for the reconfiguration which can be important when considering to minimize the undesired effects of a reconfiguration of the system, namely, loss of performance when we have to wait while the partial reconfiguration of the FPGA is done.

To improve performance in a heterogeneous computing system with partial dynamic reconfiguration we must reduce the time to reconfigure the system. This can be done:

- At project development with improvements in the logic that controls the reconfiguration;
- Reducing the size of bitstreams;
- Anticipating the need of reconfiguration of the system;

We presented a proposal for a scheduler that decides how to best allocate the different processing resources available and that controls the flow of execution for different feasible periodic task sets. This simple version of the scheduler takes in account the mean time or the worst case to partially reconfigure the reconfigurable partitions in addition to the average time or the worst case of execution of the task itself to decide which resource should be used.

The simple version of the scheduler only reconfigures the system just before it is needed to accelerate a task and will not reduce the average measured time to reconfigure the FPGA.

Furthermore, we proposed a sophisticated version of the scheduler that analyses the previous one hundred tasks and makes measurements in order to make improvements to the average values used and to predict what configuration should be used in the FPGA.

Our conclusion is that to further improve the performance, in a system with partial dynamic reconfiguration, whether we already know the task set sequence and this does not change, or we have to discover the sequence of tasks in recent periods of time.

6.2 - Future Work

Finally, there is still work to be done that we have not finished but as well out of the scope of this work but that would be interesting to further develop and implement.

6.2.1 - Accelerator development

It is a hard job to develop an accelerator suited for partial reconfiguration and specially developed to accelerate a specific task. We could further develop such a IP with much better performances giving emphasis to the interfaces that could ease the decision to allocate a resource to that task.

6.2.2 - Memory map

As we have seen, it is of great importance, in concurrent heterogeneous systems to develop the best possible memory hierarchy. Having the capacity to store in a queue the data needed by the tasks in a specific memory dedicated to each processing unit, we could maximize the possibilities to have a concurrent behaviour. It is something very difficult and that would take many attention, in order to have a good resource utilization.

6.2.3 - Simple Scheduler

Elaborate a series of sufficient tests with enough different task sets, equally distributed in order to make clear conclusions.

Implement the system in other platform with real constraints in an embedded system.

6.2.4 - Sophisticated Scheduler

We have come to conclusions and made many assumptions about what would be necessary to do in order to minimize the effects brought by the need to reconfigure the system. But it is out of the scope of this project to actually implement the proposed sophisticated scheduler. It would be of use to further make analyses and conclusions and implement a scheduler based in artificial intelligence, i.e., a neural network, that could predict the best possible configuration for the system at any time and would be able to better allocate a resource to a task.

References

- [1] G. Moore, Cramming more components onto integrated circuits, *Electronics*, Volume 38, Number 8, April 19, 1965.
- [2] J. Peterson, P. Bohrer, L. Chen, E. Elnozahy, A. Gheith, R. Jewell, M. Kistler, T. Maeurer, S. Malone, D. Murrell, N. Needel, K. Rajamani, M. Rinaldi, R. Simpson, K. Sudeep and L. Zhang, Application of full-system simulation in exploratory system design and development, *IBM J. Res. Dev.* 50(2,3) (2006).
- [3] M. Flynn et al, Finding Speedup in Parallel Processors, 2008 International Symposium on Parallel and Distributed Computing
- [4] D. Padua, *Encyclopaedia of Parallel Computing*, Volume 4, Springer.
- [5] F. Bini, G. Buttazo, Measuring the Performance of Schedulability Tests
- [6] G. D'Andrea, T. di Mascio, G. Valente, "Self-adaptive loop for CPSs: is the Dynamic Partial Reconfiguration profitable?", *MECO* 2019.
- [7] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41-50, Jan 2003.
- [8] M. Damschen, L. Bauer, J. Henkel, CoRQ: Enabling Runtime Reconfiguration Under WCET Guarantees for Real-Time Systems, *IEEE EMBEDDED SYSTEMS LETTERS*, V. 9, N. 3, 2017.
- [9] Wang Lie, Wu Feng-yan, Dynamic partial reconfiguration in FPGAs, 2009 Third International Symposium on Intelligent Information Technology Application.
- [10] M. Nguyen, R. Tamburo, S. Narasimhan, J. Hoe, Quantifying the Benefits of Dynamic Partial Reconfiguration for Embedded Vision Applications, 2019 29th International Conference on Field Programmable Logic and Applications
- [11] UG909, Vivado Design Suite User Guide, Dynamic Function eXchange.
- [12] PG134, AXI HWICAP LogiCORE IP Product Guide Vivado Design Suite.
- [13] XAPP1159, Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices.
- [14] PG305, Partial Reconfiguration AXI Shutdown Manager v1.0, LogiCORE IP Product Guide.
- [15] PG227, Partial Reconfiguration Decoupler v1.0, LogiCORE IP Product Guide.
- [16] PG304, Partial Reconfiguration Bitstream Monitor v1.0, LogiCORE IP Product Guide.
- [17] PG193, Partial Reconfiguration Controller v1.3, LogiCORE IP Product Guide.
- [18] PG021, AXI DMA v7.1, LogiCORE IP Product Guide.
- [19] UG585, Zynq-7000 SoC Technical Reference Manual.