Runtime Support for Heterogeneous Multi-core Systems

Mojtaba Sabeghi

Mojtaba Sabeghi

Runtime Support for Heterogeneous Multi-core Systems

Runtime Support for Heterogeneous Multi-core Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op maandag 4 april 2011 om 12:30 uur

door

Mojtaba Sabeghi

Master in Computer Engineering
Ferdowsi University of Mashhad
geboren te Mashhad, Iran

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir H.J. Sips

Copromotor:
Dr. K.L.M. Bertels

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft, NL
Prof.dr.ir H.J. Sips, promotor	Technische Universiteit Delft, NL
Dr. K.L.M. Bertels, copromotor	Technische Universiteit Delft, NL
Prof. Dr. B.H.H. Juurlink	Technische Universität Berlin, DE
Prof. Dr. M. Platzner	Universität Paderborn, DE
Prof. Dr.-Ing. M. Berekovic	Friedrich-Schiller-Universität Jena , DE
Prof.dr.ir. P.F.A. Van Mieghem	Technische Universiteit Delft, NL
Dr. T. Stefanov	Universiteit Leiden, NL
Prof.dr.ir. P.M. Sarro, reservelid	Technische Universiteit Delft, NL

Mojtaba Sabeghi

Runtime Support for Heterogeneous Multi-core Systems

Delft: TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica - III

PhD Thesis Technische Universiteit Delft.

Met samenvatting in het Nederlands.

ISBN 978-90-72298-15-7

Subject headings: heterogeneous multi-core systems, runtime support, virtualization, reconfigurable systems, scheduling, profiling.

Cover designed by Javad Ghasemi, Babol. All rights reserved. gigantic.ant@gmail.com

Printed in The Netherlands

To my parents

To my wife

Runtime Support for Heterogeneous Multi-core Systems

Mojtaba Sabeghi

Abstract

Multi-core processing platforms are one of the major steps forward in offering high-performance computing platforms. The idea is to increase the performance by employing more processing elements to perform a job. However, this creates a challenge for both hardware developers who build such systems and software designers who program those platforms.

On the hardware side, we can mention the problems on the interconnects management, memory hierarchies complexities and cache coherency problem. While on the software side, problems mainly arise in resource management, resource sharing and synchronization. One more fundamental problem on the software side is the inability to program such platforms with the conventional programming models. This is mainly because programming such platforms requires in-depth knowledge of hardware design.

In this dissertation, we address the software side problems by proposing a comprehensive runtime system which is responsible to manage the system resources and resolve all the conflicting issues when accessing computing resources. Furthermore, the runtime system offers the application developers with APIs and system primitives that abstract away the platform dependent details, and provides a consistent programming model. These primitives decouple the process of software development from hardware design and results in the software to be independent of the underlying hardware platform.

The proposed runtime system consists of a scheduler, a profiler, a transformer, a JIT compiler and a kernel library. A detailed description of each component is presented and the performance of the whole system as well as the imposed overhead of the component is discussed.

Acknowledgements

Over the past years, I have benefited from the help, the inspiration and the encouragement offered by many people. This thesis in its present form would not have been possible without their generous support and I would like to take the chance here to express my appreciation.

First and foremost, I would like to thank my co-promotor, Dr. Koen Bertels for offering me the opportunity to pursue my PhD studies in Delft. He gave me the freedom to work on my own research interest and his support was always there for technical as well as personal problems. I would also like to thank Prof. Stamatis Vassiliadis for trusting my skills and accepting me in his research group. Unfortunately, I did not have the chance to use his help and support during my PhD. I sincerely thank my promotor Prof. Henk Sips for his help at the final stage of my PhD. I would like to extend my gratitude to my PhD examination committee for reading my thesis and giving me invaluable comments. A special thanks goes to Dr. Todor Stefanov and Prof. Ben Juurlink for their detailed comments on my thesis.

To respect those who first introduced me to the world of computer science research, I would like to express my gratitude to my former supervisor at Ferdowsi University of Mashhad, Prof. Mahmoud Naghibzadeh. I would also like to thank Dr. Hossein Deldari, Dr. Mohsen Kahani and Dr. Mohammad Hossein Yaghmaee from whom I learned the basics of the computer science.

The most valuable experiences during my PhD studies are associated with the people I have met in the university. In this spirit, I am very grateful to all the members of CE lab and especially to the members of the Delft Workbench

team. A very special thanks goes to Mr. Hamid Mushtaq with whom I had the chance to collaborate as his MSc advisor. I extend my sincere thanks to Dr. Cor Meenderinck and Mr. Roel Meeuws for helping me to translate the propositions and the abstract of my thesis into Dutch. I would like to especially thank Dr. Tariq Abdullah for reading my thesis and giving me detailed comments to improve my thesis quality. I am also thankful to Mr. Arash Ostadzadeh for proofreading my thesis. Special thanks go to Dr. Behnaz Pourebrahimi, Dr. Mahmood Ahmadi, Mr. Arash Ostadzadeh, Mr. Alireza Asadi and Dr. Mahmood Fazlali for being good friends and supportive whenever I needed them. I would also like to thank Zubair, Vlad, Razvan, Kamana, Yana, Jae, Ozana, Luyi and Thomas. My thanks also go to Lidwina, Monique, Bert, Eric and Eef for the support during these years. I would also like to thank the relatively large Persian community in Delft and other parts of the Netherlands. Meeting so many Iranians made me feel at home and gave me an opportunity to have a great social life in the Netherlands.

I would like to express my deepest thanks to my parents for their unwavering support, encouragement and trust in my abilities. I am really grateful to my sister, Narjes, who was always my main source of inspiration and motivation. I sincerely thank my brothers and sisters for their kindness and support.

Last but not least, I am extremely grateful to my wife for her love, support and understanding over the years. Without her encouragement and support, this thesis would not have been possible.

Mojtaba Sabeghi Delft, The Netherlands, April 2011

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms and Symbols

API	Application Programming Interface
CCG	Configuration Call Graph
DSO	Dynamic Shared Object
DSP	Digital Signal Processor
EDF	Earliest Deadline First
ExTI	Expected Time Improvement
FIS	Fuzzy Inference Systems
FGEDF	Fuzzy Global EDF
FGMLF	Fuzzy Global MLF
FPEDF	Fuzzy Partitioned EDF
FPMLF	Fuzzy Partitioned MLF
FPGA	Field-Programmable Gate Array
GOT	Global Object Table
GPP	General-Purpose Processor
HW	Hardware
ISA	Instruction Set Architecture
JIT	Just in Time
LDiF	Longest Distance in the Future
LFiF	Least Frequency in the Future
LFiP	Least Frequency in the Past
MAL	MOLEN Abstraction Layer
MLF	Minimum Laxity First
OS	Operating System
PIC	Position Independent Code
RM	Rate Monotonic
RP	Reconfigurable Processor
SW	Software
VM	Virtual Machine
XREG	Exchange Register

1 Introduction

In this dissertation, we propose a comprehensive runtime system that is integrated in heterogeneous multi-core systems to resolve all the conflicting issues between the applications running on them. Furthermore, this runtime system provides application developers with APIs and system primitives that abstract away platform-dependent details to offer a consistent programming model. It decouples the process of software development from hardware design and results in the software to be independent of the underlying hardware platform.

The runtime system consists of a scheduler, a profiler, a transformer, a JIT compiler and a kernel library. The detailed description of each component is presented and the performance of the whole system as well as the imposed overhead by each component is discussed.

1.1 Problem Overview

Employing multiple processing elements on a single chip is now becoming a trend in a wide range of computing platforms. These processing units can be homogeneous or heterogeneous. It has been proved [1, 2] that this approach has several benefits such as lowering the power consumption and improving the performance.

It also seems to be a viable solution to keep the Moore's law alive for a couple of more years. The reason lies in the fact that increasing transistor densities with high clock rates demands more power and as a consequent generates more heat, which limits how fast the processor clock can go. As a result, manufacturers try to increase the performance of computing systems by increasing number of execution units on a single chip while keeping the clock fixed or even reducing the clock frequency.

However, hosting multiple cores on a single chip instead of one core creates some complexities at the hardware and the software design parts. On the hardware side, we can mention problems with the interconnects, memory hierarchies and cache coherency. While on the software side, problems mainly arise in resource management, resource sharing and synchronization. These problems are even worse when incorporating multiple heterogeneous cores. In fact, these problems are forming a new fundamental issue which can be named as *programmability wall*. This is due to the fact that programming these platforms with all the mentioned problems is really a difficult task and requires in-depth knowledge of the underlying software and hardware.

There are two main issues on the software side. First, how to divide a program into several parts so that each part can be executed on a separate core. Second, how to allocate these parts to different cores.

Even after solving these two issues, other problems may arise when multiple applications are to be executed on the same platform. In such a scenario, many competitions may exist between the applications. Therefore, the limited system resources have to be shared in a fair and transparent manner between the applications. It should be noted that most of these conflicts are only known at runtime, which means that a design time resource manager can not deal with these problems.

In our work, we target general purpose heterogeneous multi-core systems. This is in line with the technology movements towards employing heterogeneous processing elements in general purpose machines. In such systems, serving multiple applications which are running concurrently on the same machine is

an obvious requirement. To address this requirement, a runtime environment is needed which is responsible to fully operate the system and address all the conflicting issues between competing applications. Furthermore, the runtime system has to offer the programmers with APIs and system primitives which abstract away the platform dependent details and provides a consistent programming model.

The proposed runtime system consists of a scheduler, a profiler, a transformer, a JIT compiler and a kernel library. The scheduler has to decide when and where a specific task has to be executed. It has to deal with conflicting objective such as performance, power consumption or cost. In our work, the scheduler, does not change the order of the tasks execution within an application. It only decides about the mapping. The runtime profiler analyses the code and stores statistics about computational intensity and number of referrals as well as the memory bandwidth being used of different parts of the code, the purpose of which is to allow the runtime system to identify compute intensive parts, which if implemented in hardware minimize the execution time of the running applications. The transformer has to modify the binary by augmenting the calls to the software version of the task with calls to the hardware implementation in a specific core. Furthermore, the transformer has to safely resolve and transfer the parameters required by the hardware implementation to a part of memory which can be easily accessed by that core. After the results have been calculated, the appropriate return values have to be safely sent back to the calling thread. The JIT compiler is actually a binary to binary compiler and is responsible to generate the binary for the target core from the available binary. Of course as the JIT process might be costly and time consuming, we can use the help of a library in which we keep the binaries of the tasks for different cores in order to avoid JIT compilation.

It should be mentioned, that our system serves multiple applications. Each application is composed of several tasks. Each task can be mapped to a certain core, or it can be executed on the general purpose processor. A certain task might be used in more than one application. To simplify the design and to be in line with the MOLEN programming paradigm, we assume that a task is in the

form of a function in the program code. Therefore, each function in the code is a task unless it is a very small function.

1.2 Dissertation Contribution

The main task of the runtime system is to decide where, when and how an application or its constituent parts (i.e. tasks) should be executed. Therefore, the runtime system has to first analyse the code and extract information about the program execution. The program execution information includes among other things the computation intensity and frequency of each part of the code. Then, the runtime system has to decide which part of the code has to be executed on which core to achieve the required performance.

As the platform may host heterogeneous processing elements, each core might have a different binary standard and, therefore, the runtime system should be able to submit the workload for each core based on the core's binary standard. Considering the runtime decision making on the core allocation, this implementation binding is not known until runtime. Therefore, programmers need to express operations without knowing the actual implementation. In other words, the runtime system has to abstract away a task call from its actual implementation.

Based on the aforementioned criteria, the main contributions of this thesis can be summarized as follows:

1. We introduce a comprehensive runtime system together with a detailed discussion of its components and performance evaluation. We provide a detailed overview of the system layers and show how each layer interacts with the others. The most important layer is the virtualization layer which consists of a scheduler, a profiler, a JIT compiler, a transformer and a kernel library.

2. We define and implement a new task abstraction mechanism which extends the MOLEN programming paradigm. Using this mechanism, the

task implementation is separated from the task call. We extend the model in such a way that it is suitable for multitasking scenarios.

3. We present a detailed discussion of the scheduling requirements for heterogeneous multi-core systems and present some scheduling policies together with their performance evaluation. We introduce a number of new scheduling algorithms. Our scheduling decision making is based on the *distance in the future* and *frequency in the future* as well as *expected speedup*. We also introduce the configuration call graph as a viable source of information for the scheduler.

4. We employ fuzzy logic in the decision making process of the scheduler for the systems with real-time constraints. We model the inputs of the scheduler such as laxity and deadline as fuzzy variables and use a Sugeno inference engine [3] to derive the scheduling priorities.

5. We present a novel runtime profiler whose task is to analyse the running code and produce statistics about code execution such as the computation intensity and frequency of the execution. This profiler has to run concurrently with other applications on the end user's machine. Therefore, it needs to have a low overhead. Our runtime profiler has an overhead of less than 1.5%.

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 gives an overview on the background information and the related work. This chapter starts by giving a few examples of the target architectures. Next, we give a short summary of the MOLEN and the MOLEN programming paradigm as well as the MOLEN design tool chain. Then, we briefly describe some of the similar approaches toward runtime support for heterogeneous multi-core systems, which is followed by a section that motivates the need for virtualization. Afterwards, we present our comparison methodology in the thesis. At the end, we point out the open

issues and discuss our approach towards solving those issues.

Chapter 3 presents our runtime system. In this chapter, we explain how the whole system is structured into layers. The system is divided into four layers; the application layer, virtualization layer, operating system layer, and hardware platform layer. The hardware platform layer is further divided into the MOLEN abstraction layer and the physical hardware layer. Furthermore, we explain the forming components of the runtime system. These components are the scheduler, the profiler, the transformer, the kernel Library, and the JIT compiler which are briefly discussed. Then, we explain the interaction mechanisms between different components. We also include two case study scenarios and show how the runtime system should react in those scenarios.

Chapter 4 explains how the MOLEN programming paradigm is extended to support multitasking and multi-application scenarios. Accomplish this, we use the same idea of MOLEN set and execute instructions to abstract the concept of a task. In this way, we decouple the task call from the task implementation. We also propose a binding mechanism to bind a task implementation to each task call. This is done by introducing the high level SET and EXECUTE APIs. At the end, we show the overhead of the proposed APIs using some experiments.

Chapter 5 presents our scheduler as a part of the runtime system. In this chapter, we show how we use a combination of design time and runtime scheduling in order to optimize the system performance. In the design time, we use the compiler to perform static task scheduling assuming single thread of execution. Then at runtime, the runtime scheduler performs the actual task scheduling having the scheduling decisions from the compiler as a hint. The runtime scheduler can also use the information provided by the runtime profiler. It can also use the information transferred from design time in the form of a Configuration Call Graph (CCG). Then, we present a number of scheduling policies as case studies and provide performance evaluations. We base our scheduling decisions on three different parameters. We show that *Expected Time Improvement* and *Longest Distance in the Future* are very good heuristics for the scheduling.

Chapter 6 focuses on the conditions in which we have real-time constraints. We use fuzzy logic to model the real-time constrains and to improve the scheduling decisions. Using deadline as a fuzzy parameter in real-time scheduling is more promising than laxity.

Chapter 7 describes the design and implementation of our runtime profiler, which can be used as a part of the runtime environment. The profiler is a combination of a sampling profiler and an instrumentation profiler. We discuss different parts of the profiler namely the *extractor*, *injector*, *sampler*, *profiler frame* and *daemon*. Then, we show the overhead of our profiler from different aspects. This is done by showing the overhead on instrumentation, sampling and the overall overhead. Besides, we compare the accuracy of our profiler with a popular design time profiler , *gprof*. All the presented results show that our profiler has very low overhead (less than 1.5%) and is as accurate as design time profilers.

Chapter 8 summarizes and concludes this dissertation. It gives a summary of this thesis, its contributions and findings. It then describes the remaining open issues and future research directions.

2

Background and Related Work

The increasing demand for high-performance computing platforms was always the main driving force for new advancements during the past years of development in computing technologies. This is valid in all the areas from high-end customized special-purpose computing in networking, telecommunications, and avionics to low-power embedded computing in desktop computing, portable computing, and video games.

Multi-core processing platforms are one of the major steps forward in offering high-performance computing platforms. The idea is to increase the performance by employing more processing elements to perform a job. However, this creates a challenge for both hardware developers who build such systems and software designers who program those platforms.

The basic idea of having a multi-core system is to overcome the major hardware design challenges such as the memory wall problem or the increase in power consumption. Secondly, to increase the performance by using more processing elements to execute a task, which is widely believed to be more efficient. These processing elements can be either homogeneous or heterogeneous.

In this work, we focus on heterogeneous multi-core platforms in which one or more general-purpose processors are the main processing elements and some special purpose processors work as coprocessors. Systems employing

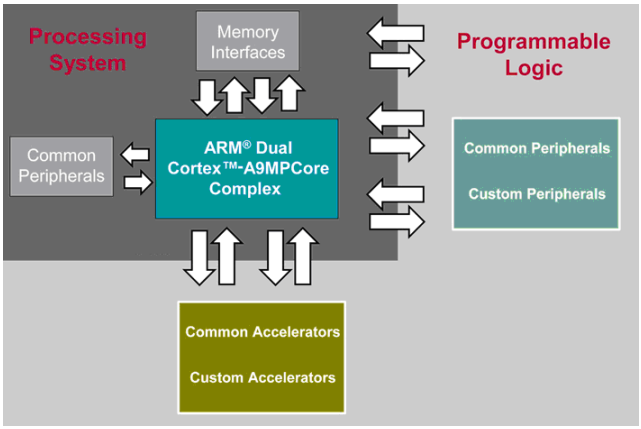

Figure 2.1: Extensible Processing Platforms

reconfigurable processors are our main concern. To clarify the scope and extent of our work, we give a brief introduction of some examples of the industrial target architectures in the following section.

2.1 Target Architectures

Our target architecture is *heterogeneous multi-core platforms*. Such platforms consist of one or more general-purpose processors and some coprocessors. There are several industrial examples of such systems. In the following sections, we give a short introduction on the Xilinx Extensible Processing Platform, Convey HC-1, Freescale QorIQ P2 Series, STMicroelectronics Platform 2012, hArtes, and Novo-G platforms.

2.1.1 The Xilinx Extensible Processing Platform

The Xilinx Extensible Processing Platform [4] offers a processor-centric design and development approach for achieving the processing power required to execute tasks involving high-performance processing and complex digital signal processing needed to meet their application-specific requirements, including

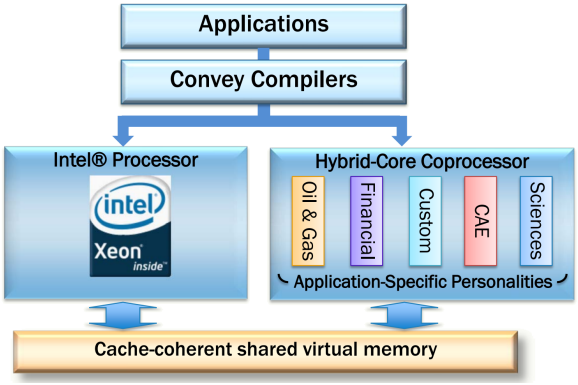

Figure 2.2: Convey HC-1

lower cost and power. It is an implementation of ARM's high-performance dual-core Cortex-A9MPCore processors and Xilinx advanced 28nm programmable logic. Figure 2.1 shows the block diagram of the Extensible processing platforms.

2.1.2 Convey HC-1

The Convey HC-1 [5] is a heterogeneous computing system based on an industry-standard Intel processor and a proprietary coprocessor that share virtual memory and an instruction stream, creating a hybrid-core computing system. The coprocessor architecture supports user-defined, dynamically loadable instruction sets.

It combines the x86 family processors with hardware-based, application-specific instructions to accelerate HPC applications. The result is a tightly integrated system that gives many times the performance of a commodity server on key applications in oil and gas, financial analytics, bioinformatics, and other markets. Figure 2.2 depicts an overview of such a platform.

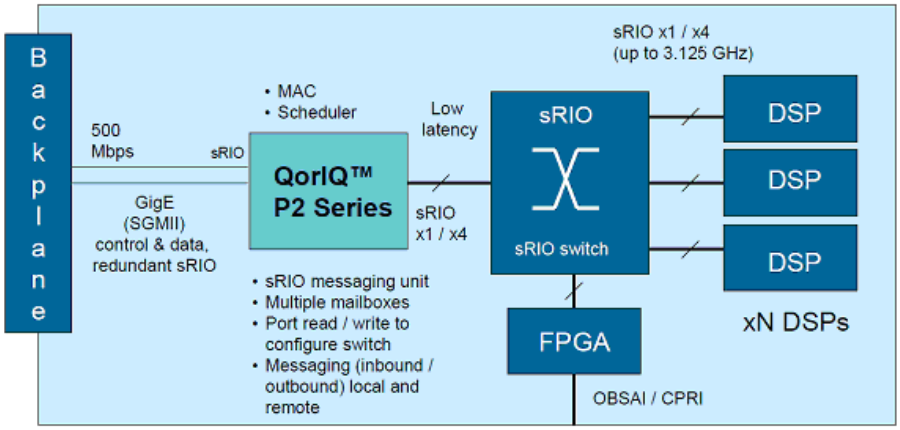

Figure 2.3: Freescale QorIQ P2 Series

2.1.3 Freescale QorIQ P2 Series

The QorIQ P2 communications platform series [6] includes the P2020 and P2010 communications processors. It delivers high single-threaded performance per watt for a wide variety of applications in networking, telecom, military and industrial markets. The P2 series delivers dual- and single-core frequencies up to 1.2 GHz on a 45nm technology low-power platform. Figure 2.3 shows the block diagram of the P2 Series.

2.1.4 STMicroelectronics Platform 2012

The P2012 provides flexibility through massive programmable and scalable architectures by enabling connection of a large number of decoupled STxP70 processors. The Platform 2012 computing fabric is composed of a variable number of “tiles” that can be easily replicated to provide scalability. Each tile includes a computing cluster with its memory hierarchy and a communication engine. The computing fabric operation is coordinated by a fabric controller and is connected to the SoC host subsystem through a dedicated bridge, having DMA capabilities. The P2012 computing fabric is connected to a host processor such as the ARM Cortex A9 via a system bridge.

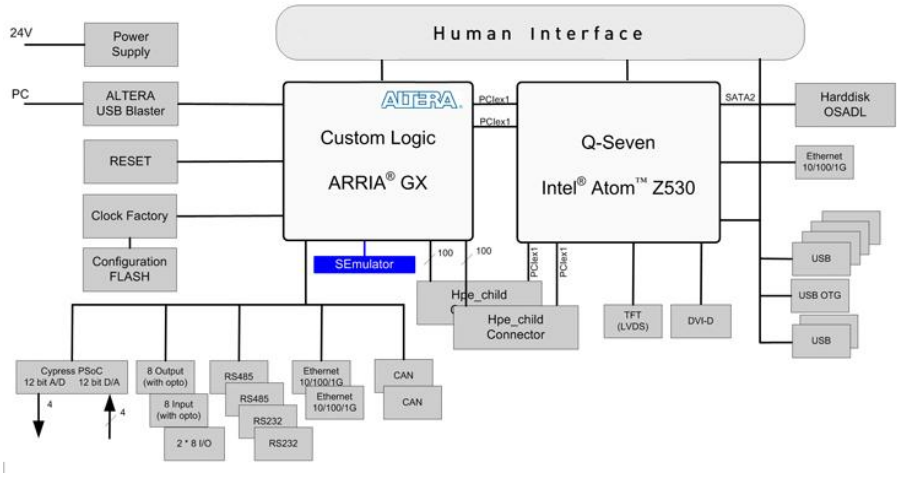

Figure 2.4: Industrial Reference Platform

2.1.5 Industrial Reference Platform

The heart of the Industrial Reference Platform, as shown in Figure 2.4, is an Intel Atom processor and an ALTERA ARRIA FPGA, connected together PCI Express to support customer specific interface and logic requirements. The Industrial Reference Platform is a highly scalable and reconfigurable platform targeted at low power, high-performance industrial automation applications.

The flexibility of the Industrial Reference Platform also makes this system a potential platform for military, automotive, and consumer applications requiring the scalable processor performance of the Intel Atom and the reprogrammability of Altera Arria series FPGAs.

2.1.6 hArtes Platform

The hArtes Platform [7], presented in Figure 2.5, provides a number of heterogeneous computing sub-systems, such as DSPs, general-purpose processors and reconfigurable elements. The system is composed of a certain number of independent blocks. Each block includes a general-purpose RISC proces-

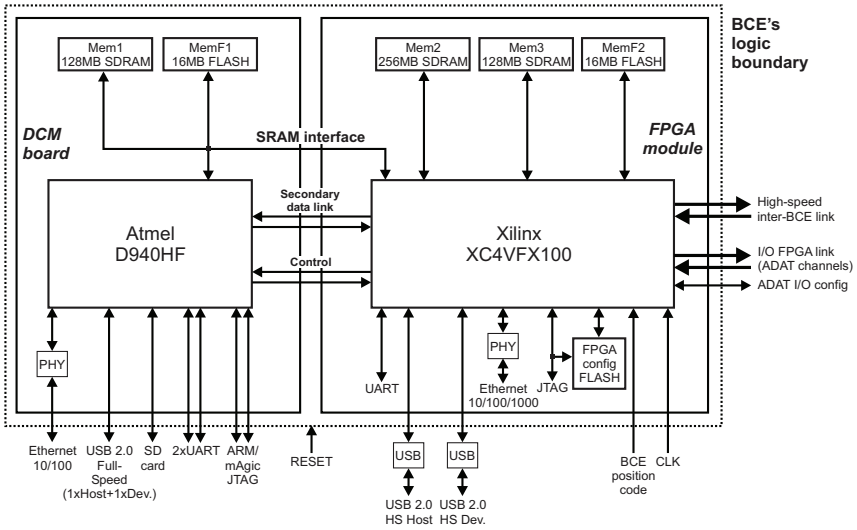

Figure 2.5: The hArtes Platform

sor, a DSP processor, and an application-specific reconfigurable block. The application-specific reconfigurable block also incorporates a RISC processor. Each block is called Basic Configurable Element (BCE).

All the BCEs share the same structure and are able to run any selected thread of a large application.

2.1.7 Novo-G

With the promise of the reconfigurable computing in offering capability of one PetaOPS at 10K Watts, it is feasible to consider large scale reconfigurable supercomputing. Novo-G [8] is such a platform, which is used in Bioinformatics and other scientific and engineering domains. Novo-G currently features 192 Stratix-III E260 FPGAs in 48 quad-FPGA boards (plus one spare) with 1TB of memory. Most of the memory is directly attached to the FPGAs (4.25GB per FPGA) and is supported by 20 Gb/s InfiniBand, quad-core Xeons, GigE, PCIe, storage, etc.

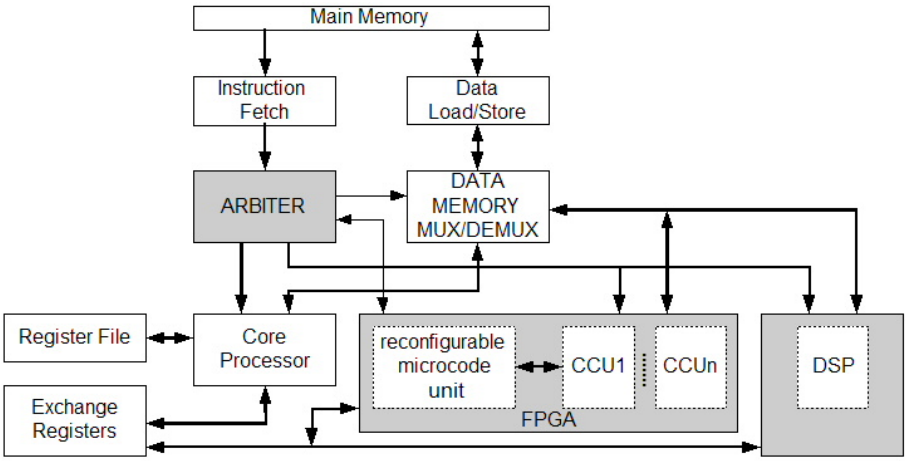

Figure 2.6: MOLEN Hardware Organization

2.2 MOLEN Hardware Organization

In the last section, we briefly introduced some of the hardware platforms that can be considered as target architecture for our work. One of the main issues in using those systems is the model of coupling and interaction among all the different processing elements as well as the memory and IO modules.

The MOLEN hardware organization is such a coupling and interaction model. MOLEN is established based on the tightly coupled co-processor architectural paradigm. Within the MOLEN concept, a general-purpose processor is assumed as the main processor, which controls all the coprocessors. Traditionally, MOLEN is designed for reconfigurable computers, however, without the loss of generality, all the MOLEN concepts can be employed effectively and efficiently in the domain of the heterogeneous multi-core systems.

In the context of reconfigurable systems, the general-purpose core processor controls the execution and reconfiguration of the reconfigurable coprocessors and tunes the latter to various application specific algorithms. Figure 2.6 represents the MOLEN machine organization for the reconfigurable systems.

In order to program MOLEN based platforms, the MOLEN programing

paradigm was introduced. We briefly describe the MOLEN programming paradigm in the next section.

2.3 MOLEN Programming Paradigm

The MOLEN programming paradigm presents a programming model for reconfigurable computing that allows modularity, general function like code execution and parallelism in a sequential consistency computational model. Furthermore, it defines a minimal ISA extension to support the programming paradigm. Such an extension allows the mapping of an arbitrary function on the reconfigurable hardware with no additional instruction requirements.

This is done by the introduction of new super instructions for operating the FPGA from the software. An operation, executed by the RP, is divided into two distinct phases: set and execute. In the set phase, the RP is configured to perform the required task and in the execute phase the actual execution of the task is performed. This decoupling allows the set phase to be scheduled well ahead of the execute phase, thereby hiding the reconfiguration latency. This phasing introduces two super instructions; *set* and *execute*.

Within the heterogeneous multi-core systems domain, the same super instruction can be used to manage the process of assigning a task to a special core and to control the task's execution on that core.

To facilitate the process of program development, a design tool chain is required. In the next section we present the design tool chain which is proposed for the MOLEN based platforms.

2.4 Design Tool Chain

The design tool chain, as shown in Figure 2.7, addresses the difficulty in programming heterogeneous multi-core platforms by assuming a single application execution scenario. It provides a semi-automatic support for the hardware/soft-

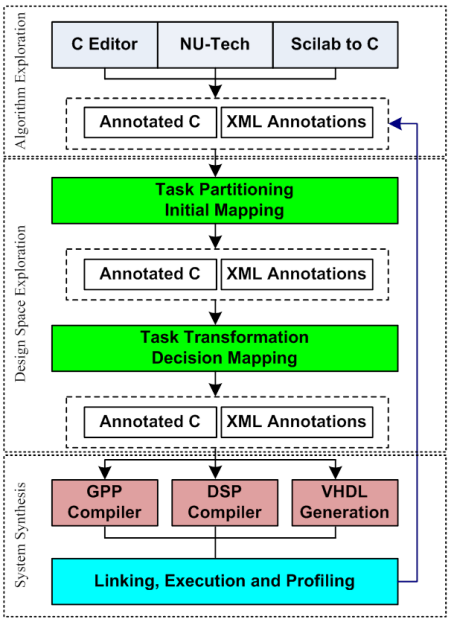

Figure 2.7: Design Tool Chain

ware co-design of such systems. The design starts with an application written in C and the final outcome is an executable with (modified) code mapped onto a multi-core platform consisting of a general-purpose processor, a DSP and an FPGA.

The tool chain embeds configuration bit streams for the reconfigurable components of the system. Thus providing a complete and operational system supported both at the software and at the hardware levels. In this way, the tool chain allows the developers to write applications for heterogeneous systems without requiring them to have intimate knowledge of the underlying hardware and its programming. The parallelism and mapping are specified in the code by using *pragmas* in the C code. These annotations can either be added manually by the developers or generated by tools. To specify parallelism, standard OpenMP pragma directives are used.

The design tool chain comprises of three toolboxes. Each of them takes a C-code as input and outputs processed files. The processed files are then fed

to the next toolbox. Figure 2.7 shows the interactions among those toolboxes. The top-level toolbox namely the Algorithm Exploration Toolbox is optional. Its purpose is just to generate C code from a Scilab code or from NU-Tech (a graphical software development tool).

Next is the Design Exploration toolbox whose task is to perform task partitioning and mapping. The task partitioning tool, known as Zebu, generates an efficient task graph [9] with the help of static profiling to identify the most efficient paths [10]. Afterwards, it performs transformation on the task graph to take into account the overhead of managing the parallel tasks. Finally, it generates a C Code annotated with pragmas for parallelization and mapping. The mapping tool hArmonic [11] is used to get an optimized mapping solution.

The last tool in the chain is the Synthesis toolbox. It contains a compiler for each processing element. The source for each processing element is compiled separately and then linked together to form a single binary. It uses a modified GCC compiler for the general-purpose processor. That compiler inserts MOLEN instructions to execute a kernel on FPGA. Moreover, Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) [12] is used to perform C to VHDL translation for the FPGA.

2.5 Runtime Systems

The main problem with the design time tool chains is that they are not aware of the runtime status of the system. For single application scenarios, that is not a serious limitation. However, when moving towards multi-application scenarios in which the load of the system varies from time to time and the load is not known at design time, there should be a runtime system, which is responsible for the mapping decisions. To perform the mapping, the runtime system can use all the information provided by the design tool chain.

Due to the inherent complexity of the task assignment on the multi-core systems and especially on heterogeneous multi-core systems, the proposed runtime

system needs to be very comprehensive. Looking at the literature, runtime systems commonly consist of many components which work together in order to operate the system and manage the resources. In this part, we give an overview on the related research on the runtime systems. It should be mentioned that all these runtime systems can be implemented - at least theoretically - on each of the target architectures presented in section 2.1.

2.5.1 BORPH

BORPH [13] handles FPGA resources as if they were CPU's by introducing the concept of hardware process which behaves just like a normal user program except it is a hardware design running on a FPGA. The BORPH kernel provides standard operating system services, such as file system access, to the hardware processes. This allows hardware processes to communicate with the rest of the system easily, and systematically [14]. BORPH is based on the Linux kernel.

BORPH introduces the concept of hardware region. A region can be physically, the entire FPGA in a multi FPGA system, or a partially reconfigurable region within a FPGA. Hardware processes are spawned in those regions.

A user starts a hardware process by executing a BORPH Object File (BOF). When a BOF file is executed, the kernel examines hardware configurations encapsulated in that file. Based on this information, the kernel chooses and configures one or more suitable hardware regions [15].

The hardware and software components of user designs may run as communicating processes within BORPH's runtime environment. The familiar language independent UNIX kernel interface facilitates easy design reuse and rapid application development. A Simulink-based design flow that integrates with BORPH is employed for developing hardware designs,

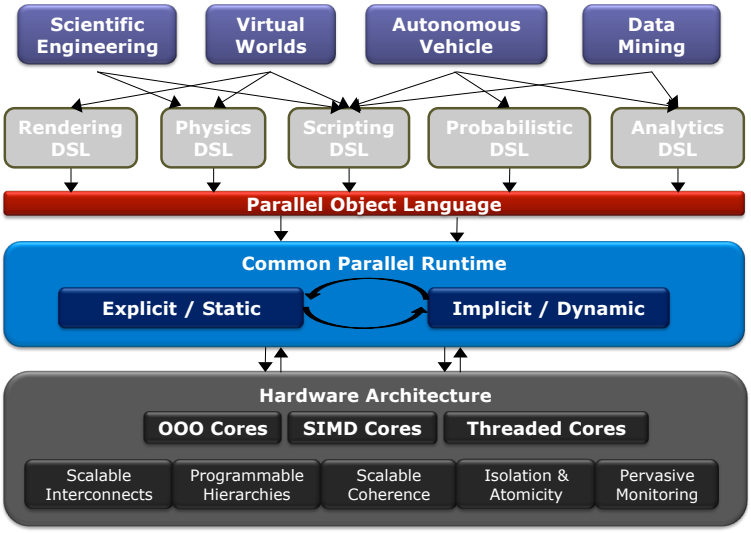

Figure 2.8: Pervasive Parallelism

2.5.2 Pervasive Parallelism

The goal of the Pervasive Parallelism (PPL) [16] is to make parallelism accessible to the average software developers so that it can be freely used in all computationally demanding applications. PPL tries to extend the Java virtual machine approach by featuring a parallel object language to be executed on a common parallel runtime system. This way it can map Java language onto the respective computing nodes. Figure 2.8 gives an overview of the PPL structure.

2.5.3 ReconOS

ReconOS [17] aims at the investigation and the development of a programming and execution model for dynamically reconfigurable hardware devices. ReconOS extends the concept of multithreaded programming to the reconfigurable logic [18]. Figure 2.9 shows an overview of the ReconOS architecture.

The described multithreaded design process allows accelerating critical system components through the reconfigurable hardware without sacrificing portability,

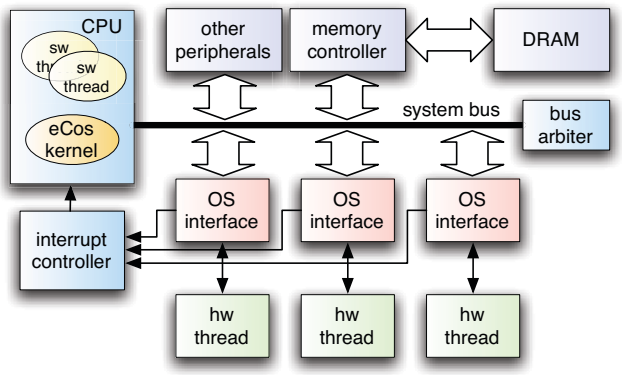

Figure 2.9: ReconOS System Architecture

flexibility, and the reusability. ReconOS enables designers to exploit both fine-grained data parallelism as well as coarse-grained thread-level parallelism [19].

In the ReconOS execution environment, it is possible to synthesize multiple hardware threads to the same location in the FPGAs reconfigurable fabric. Using partial dynamic reconfiguration, the operating system is able to load hardware threads during runtime. This allows hardware multitasking between different hardware threads [20]

The ReconOS programming model provides the main abstractions for the application design including the definition of hardware tasks and services for task synchronization, communication, and scheduling [21]. Its execution model defines the runtime system which enables multitasking in dynamically reconfigurable hardware as well as across hardware/software boundaries [22].

2.5.4 HybridOS

HybridOS [23] supports fine-grained reconfigurable accelerators integrated with general-purpose computing platforms. HybridOS is based on Linux implemented on the Xilinx XUP Boards.

The granularity of the computation on the FPGA is based on multiple data parallel kernels mapped into accelerators in HybridOS. These are accessed by

multiple threads of execution in an interleaved and space-multiplexed fashion [24].

HybridOS provides interface to the runtime system for applications using a library call approach. In addition, it interfaces the accelerator to plug into hybrid CPU/accelerator model using an accelerator framework.

2.5.5 ReconfigME

ReConfigME is an operating system for reconfigurable computing that handles the loading of the IP cores on the FPGA platform [25]. In addition to runtime resource allocation, other services provided by an operating system such as abstraction of I/O and inter-application communication provide additional benefits to the users of a reconfigurable computer [26, 27].

The ReConfigME implementation is structured into three tiers consisting of user, platform, and operating system. These tiers are connected via a standard TCP/IP network [28]. The users connect to ReConfigME through a custom-built client interface, which enables them to load applications, transfer application data and configuration information, and monitor the reconfigurable computing platform status.

The application architecture in the ReConfigME consists of a data flow graph structure, memory based I/O, EDIF application file format, and the associated software only components.

2.5.6 Warp Processing

Warp processing [29] transforms computing intensive kernels running on a general-purpose processor to hardware. The Warp profiler is implemented as hardware and is used to detect kernels at runtime. Within the Warp Processors, a Dynamic CAD tool is used to map kernels to hardware. In addition, a binary updater is used to change the binary of the program running on the general-purpose processors, so that it can uses the FPGA. The Warp Processors can

generate code for the FPGA on the fly by using a Just in Time (JIT) compiler.

The Warp processors targets single application execution. All the optimizations are within one application. However, we target multi-application execution scenarios. Our kernels are more coarse grain that Warp. The Warp is basically focusing on the optimization of the loops and small parts within a single application.

2.5.7 UltraSONIC

UltraSONIC [30] reconfigurable computing platform is designed specifically for real-time video applications. This platform incorporates a co design environment with automatic partitioning and scheduling between a host microprocessor and a number of reconfigurable processors.

It proposes the use of a task manager to combine the runtime support for hardware and software in order to improve modularity and scalability of the design. The partitioning and scheduling are done automatically. The codes for software tasks are run in software concurrently (using multi-threaded programming) with the task manager program which is based on message-passing and event-triggered protocol [31]. SONICmole [32] is a debugging environment designed for the UltraSONIC platform.

2.5.8 hthreads

hthreads [33] is a set of cooperating layers of abstraction which form a bridge from high level programming languages to low level reconfigurable devices. The first of these layers is the operating system layer, which provides runtime support to a diverse set of computations. The intermediate layer is built on top of this layer. It provides a common format for describing computations that run on the reconfigurable devices. The last layer is the high-level language support layer, which builds upon the intermediate layer to support languages, which are easy to use and familiar to most programmers. The programming model is

similar to POSIX threads [34]

To support the abstraction of the CPU/FPGA component boundary, a Hard-Ware Thread Interface (HWTI) component is created to free the designer from specifying and embedding platform specific instructions to form customized hardware/software interactions [35]. The hardware thread interface supports the generalized pthreads API semantics, and allows passing of abstract data types between hardware and software threads.

2.5.9 Other Runtime Systems

An approach to the runtime management of reconfigurable hardware tasks executing under supervision of a Linux OS is presented in [36]. The proposed system offers transparent integration of the reconfigurable resources within the software design and execution flow. In the presented system, the reconfigurable resources can be dynamically divided between the executing applications according to a policy implemented by the application.

Access to the hardware modules is encapsulated into the ghost processes in [37]. This provides a transparent interface for interactions from the kernel and other processes. The hardware and software processes use FIFOs mapped to the Linux file system as well as dual-ported memory accessible from both software and hardware processes for communication purposes. Process networks communicating via FIFO queues are a powerful model for real time digital system design, especially for data streaming applications such as multimedia devices. FIFOs also form a central part of UNIX and Linux Interprocess Communication (IPC) architectures, where they are more commonly known as pipes. It is shown that the combination of embedded Linux, reconfigurable system on chip, and FIFO communication models provide a compelling platform for efficient design and runtime implementation of the complex, high performance embedded systems [38].

2.6 Virtualization

As mentioned earlier, the computing platform has to take an active role in the resource management and resource sharing among different applications. This is done by abstracting the hardware platform details using an extra virtualization layer, which offers high level programming primitives to the application [39]. These primitives can be software or hardware implemented. The goal of such a virtualization is to virtualize all the components of a heterogeneous multi-core system including the GPP, reconfigurable processors, DSPs, etc [40].

The infrastructure presented in [41] targets the full exploitation of the underlying hardware of heterogeneous, reconfigurable parallel systems without burdening the programmer with details of the underlying hardware. Core of this framework is a lightweight runtime system performing function resolution according to the current system configuration and adhering to application requirements. This process is guided by an augmented application description, enabling declaration of implementation alternatives of individual functions and providing according meta-data such as required or provided performance data.

A virtualization layer for multi-core environments, especially FPGAs, is presented in [42] which separates applications to be run from the underlying hardware. This work is based on the Scalable Dataflow-driven Virtual Machine (SDVM). The SDVM is evolved to a virtualization layer for multi-core systems based on FPGAs. This virtualization layer allows for a transparent run time reconfiguration of the underlying hardware reducing the complexity of the systems temporal heterogeneity as seen by the application.

The goal of the IBM Lime [43] is to create a single unified programming language and environment that allows all portions of a system to move fluidly between hardware and software, dynamically and adaptively. Lime targets Java applications to be dynamically translated for co-execution on the general-purpose processors and reconfigurable logic. Another similar work is PPL [16] which tries to extend the java virtual machine approach by featuring a parallel object language to be executed on a common parallel runtime system, mapping

this language onto the respective computing nodes.

A new programming approach leveraging processor virtualization and component based software engineering paradigms is shown in [44]. This approach is intended for increasing the programmability of the multi-core platforms and for integrating heterogeneous processors efficiently.

2.7 Comparison Methodology

We presented a number of runtime systems in the Section 2.5. Each runtime system is based on some innovative thoughts and targets heterogeneous reconfigurable systems. One important open issue here is how one can compare all these systems. Traditionally in the area of the operating systems, there is always the question of which operating system has the best performance. However, there is not always a single best answer to that question.

In our case because most of the related works are research based products, this question is even more difficult to answer because most of the time there is no access to a consistent single version of those systems that can fit all the hardware platforms. One possible approach for comparison is to take the reported numbers from their publications and make the comparison based on them. However, some major problems with this approach make the comparison incorrect.

The most important problem is that in many cases the experiments are performed using synthetic test cases. Therefore, the results generated from one set of data are not always comparable with the results generated with another set of data. Furthermore, the hardware platforms being used for each system are different from the others. For example, some might use Xilinx platforms; the other used Altera, etc. In addition, the processors might be PPC, ARM, etc.

Besides, the base operating system used in different approaches is different. For example, some approaches only used a prototype operating system while some others are based on Linux (different versions) or eCos. Different operating

systems offer different scheduling algorithms, different binary standards and different C libraries. The file system being used there also has an important effect on the gained performance.

Moreover, each research might have used a different set of metrics for the experimental parts. For example, the experimental results presented for the ReconOS show the OS primitives and APIs overhead in number of bus cycles. However, as the presented primitives are unique to the ReconOS, they are not comparable for example with the primitives of BORPH.

As another example, we can mention BORPH. For the BORPH case, the experiments are mostly focused on the overhead of File I/O from hardware processes due to the special design of this system. While for example in HybridOS, the authors presented the overhead of accessing to the accelerators from the applications. They have presented four different methods for which they showed the overhead in number of cycles. Anyhow, their results show that the most effective access method depends on data transfer size and the number of times the application will use an accelerator.

Based on the above discussion and based on the trends so far in the similar researches, we will not compare the runtime system as a whole with the similar projects. However, for each component inside the runtime system we will try to compare it with the other similar works.

The metrics we will use in the coming chapters will be overhead of different components, percentage of the overhead to the total execution time, speedup, number of tasks executed on cooperating processors, percentage of the tasks executed on cooperating processors, total execution time, accuracy, response time, utilization.

We believe that these metrics can give a clear view of the system and it characteristics. Nevertheless, one of the open issues in this area is the lack of a consistent and uniform comparison methodology to compare similar works as a whole system. In future research, we will study this problem and provide a set of qualitative and quantitative metrics for comparisons purposes. Besides, we

will provide a set of standard real world application workloads that can be used as the input for the comparisons.

2.8 Open Issues

The main responsibility of the runtime system is to fill the gap between the application programmers and the hardware architectures. Application programmers are usually lacking in-depth knowledge of the hardware design. As a result, the runtime system has to provide them with the high-level primitives and services, which abstract away the hardware details. One of the main open issues in the area of the heterogeneous multi-core systems is the lack of well-defined programming model and primitives, which give the programmers a transparent view of the whole system regardless of the underlying hardware.

One major question in this regard is the coupling and interaction model of the cores in a heterogeneous environment. This has a direct effect on the programming model and system performance. It also influences the granularity of the computation, to be addressed by the system. The programming model should also provide modularity in coding.

Another issue is the binary compatibilities between the cores. The runtime system has to be able to work with different binaries for different cores. It needs to know the different binary standards, their loading requirements, code structure, etc. There is no clear approach to address binary compatibility in heterogeneous multi-core systems from the surveyed literature.

Furthermore, different vendors provide their own implementations for each computation on the special cores. This is mainly because they know the computation characteristics of the core form one side and the computation from another side and consequently, they can provide efficient implementations for the computation. The runtime system should provide easy and clear interfaces for incorporating the implementations provided by the third parties. This means the system should be adaptive in using different implementations based on the

implementation characteristics.

Another major issue is the resource management. The resource management techniques and algorithms are still open and need serious attention from the research community.

In the same line, the other problem is the lack of tools for analysing the programs binaries at runtime and produce statistics to be used by the scheduling and resource management algorithms.

The main contribution of this work is presenting an integrated runtime system, which performs the resource management activities and offers all the required programming primitives and APIs, which are needed by the programming model. We present a task abstraction mechanism through which the task implementation is separated from the task call. This way, we allow modularity, adaptively and we can easily incorporate third party implementations for each task.

We also provide a detailed discussion of the scheduling requirements of heterogeneous multi-core systems and present some scheduling policies together with their performance evaluations. To provide the scheduler with information and to ease the decision making process, we have a novel runtime profiler. The task of the profiler is to analyse the running code and produce statistics about code execution such as the computation intensity and frequency of execution.

As mentioned in section 2.7, one of the open issues in this area is the lack of a consistent and uniform comparison methodology to compare similar works as a whole system. We do not address this issue in this thesis and it is one of our future works.

2.9 Summary

In this chapter, we provided an overview on the background knowledge and the related work. We started with the target architectures. Our work focuses on the heterogeneous multi-core systems with a special attention to the reconfigurable

cores. We showed some of the available examples of such platforms.

We briefly discussed the MOLEN hardware organization and the MOLEN programming paradigm. We continued with a short overview of the current MOLEN design tool chain, which supports the automatic design, follows for polymorphic multi core systems.

By motivation why the design tool chain is not enough and there is a need for employing a runtime system, we gave a summary of the current research projects with the focus on the runtime systems. Next, we discussed the open issues in this research area and the motivations behind this work.

3

Runtime System

In the previous chapter, we reviewed the background information and related work. We showed some of the target hardware platforms as well as some similar runtime systems. Moreover, we presented the motivation of using a runtime management and virtualization.

In this chapter, we present an overview of our proposed runtime system. The runtime system is responsible to manage the system resources and address all the conflicting issues between applications. It incorporates a number of components namely the scheduler, the profiler, the transformer, the JIT compiler and the kernel library. We briefly describe the task of each component, however, a more detailed discussion on them will be given in the following chapters.

We also show how the system is structured. The whole system is composed of four layers; the application layer, the virtualization layer, the operating system layer and the hardware platform layer. Furthermore, we discuss the interaction and interfacing mechanisms between these components. The most important interface is the interface between the scheduler and the profiler. Moreover, we present two case studies which show two different scenarios of cooperation and interaction of the components.

3.1 Introduction

In a heterogeneous multi-core system, to assign the computations to different cores, a comprehensive runtime system is required. The runtime system detects the computation intensity of each task and has a general view over the system load (e.g. number of applications running) and therefore can decide about the hardware allocation based on all these information. Furthermore, the process of mapping and allocation has to be transparent for the program developers. The runtime system provides a powerful interface to exploit multiple cores available in the system [45–47].

Within the MOLEN Programming paradigm, the execution can take place either on the General-Purpose Processor (GPP) or on the cooperating processors based on the runtime dynamic conditions. In this model, the GPP is the master processor and all the applications executions start on this processor. Later on, the runtime system might decide to execute some part of each application on the other cores. This decision is very dependent on the system and the metrics that need to be optimized. It can be performance, power consumption or a smaller memory footprint.

To be able to run on the heterogeneous hardware platform, the virtualization layer needs to inspect, analyse, and do binary modification on the applications. The layer monitors the programs binary to find the compute intensive tasks. Furthermore, it determines whether these tasks can run on faster core or not and then it transforms the binary in such way that the binary can be executed on that core.

It should be mentioned, that our system serves multiple applications. Each application is composed of several tasks. Each task can be mapped to a certain core, or it can be executed on the general purpose processor. A certain task might be used in more than one application. To simplify the design and to be in line with the MOLEN programming paradigm, we assume that a task is in the form of a function in the program code. Therefore, each function in the code is a task unless it is a very small function.

The extracted tasks have to be complied for that specific core after mapping those replacements. This can be done either automatically with the help of a JIT compiler which converts the binary to the bit stream or using an available implementation from a library.

A JIT Compiler generates a considerable overhead and yet, there is no fast and reliable JIT compiler to translate the software binary to the reconfigurable hardware implementation. Therefore, having a library of the common kernels implemented for each hardware platform is a very good alternative. If a software uses kernels which are not included in the library, it can add those new kernels during the software installation process.

One of the problems with the library is the matching between the selected part in the application and the equivalent kernels in the library offering the same functionality. There is no good solution for functionality matching so far. This is proven to be a NP problem. We propose an alternative solution by assigning each kernel a unique identifier. The matching between the kernel in the application and the library can be based on this identifier.

The annotation of the source code with the identifiers may impose an extra work on the programmers. To simplify this, we can also have a software class library equivalent to the hardware kernel library. Each kernel in the hardware library has a software implementation in the class library and the programmers can use this class library during the software development.

Before going any further, we will first give a few examples of real world applications for such systems. All the following applications need huge computing power and they are very dynamic at the same time. The computation density may change according to users input. This means that during the runtime the platform has to adapt itself to the users' performance requirements.

3.2 Sample Real World Applications

In this section, we briefly introduce a few applications to motivate the need for such hardware platforms and runtime systems. Some parts of these applications are very compute intensive and also power hungry. They also have a very dynamic nature meaning that their computation intensity might change based on the user's input. Therefore, the hardware platform should be able to offer the required computation power and the runtime system should be able to manage the dynamicity and flexibility in these applications.

Interactive Multimedia Internet based Testing is similar to the TOEFL iBT exam. In such a test, several test takers (applicants) are connected to the exam server. There is a separate process on the server for each test taker. This process sends the questions containing multimedia features such as voice, video, and pictures to the corresponding test taker. Each test taker might use his own machine to connect to the server, therefore, these different machines connected with the server can have different computing power. Consequently, the server must send the question in a format, which can be easily decoded by the clients. Furthermore, the questions have to be encrypted for ensuring the security.

In the listening tests, the server should send image and voice files for each question to the client. Therefore, the server encodes image and voice files. Afterwards, the test question is sent to the client. Whenever a user answers the test, the test answer is sent back to the server. The client encrypts the test answer file and sends it to the server for ensuring a secure exam. In the speaking tests, similar to the listening tests, image and voice files inside the test question are sent to the client. Again, they are encoded and encrypted by the server before being sent to the client. In the reading tests, the server sends the simple test to the client and receives the test answer files that are encrypted by the client. For these different tests, and given that many simultaneous users will perform similar operations, the encoding and encrypting of the files can be accelerated by mapping them on the reconfigurable fabric.

Automotive Cognitive Safety System is an on-board digital system for keep-

ing the passengers safe. This system analyses, anticipates, and acts in response to the ever-changing conditions on board. It uses a number of sensors for actuating the assisted scenarios such as forward collision warning, automatic emergency braking, lane change assist, etc. The sensors generate a lot of information that needs to be processed in order to take an appropriate action. Some of the actions might have real-time constrains. The whole scenario needs a lot of multimedia processing such as image and video coding. These parts together with the real-time constrains make these systems a suitable candidate to be implemented on our target architectures and runtime system.

Mobile Devices are now being used for playing HD videos, streaming video and audio, multitasking, browsing the web, 3D gaming, etc. Mobile devices need to employ heterogeneous multi-core platforms for further increase the performance and for extending the battery life.

Multi-View Video refers to a set of N temporal synchronized video streams coming from cameras that capture the same scene from different viewpoints. Transmitting of the multi-view video requires much more bandwidth than traditional video due to the large data volume. Consequently, efficiently compressing multi-view video becomes more important than any other data. Then, the server has to synchronize multiple video into one stream and send it to the client [48].

3.3 The Proposed Runtime System

In this work, we propose a four layer architecture; the application layer, the virtualization layer, the operating system layer, and the hardware platform layer. Figure 3.1 shows an overview of the runtime system. The top most layer is the application layer. There might be several applications running at the same time. Needless to say, these applications are competing for the system resources. Moreover, they might not have a clear view of which hardware resources are available. The system has to manage the resource allocation transparently. The next level is the runtime virtualization layer. This layer is responsible to monitor the computation intensity of different applications and map the system resources

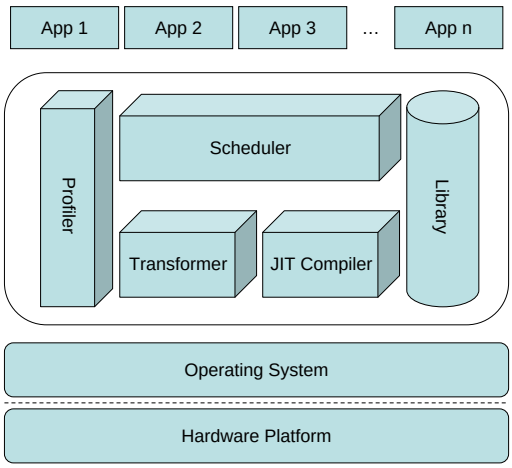

Figure 3.1: The Runtime Environment

based on the needs of the application and other runtime constrains. After the virtualization layer, there is the operating system layer on top of the hardware platform layer. We use the Linux mainstream kernel as our operating system. It is just the kernel of the Linux and we do not use any special distribution of the Linux (i.e. RTLinux [49] (real-time Linux)).

The virtualization layer is consisted of the scheduler, the profiler, the JIT compiler, the transformer, and the kernel library. A detailed description of each component is given later in this section. The runtime system is designed as a framework meaning that each component has a generic definition and can be replaced by a new component provided that the new one offers the same functionality. In order to achieve this, we have introduced well defined interfaces for each component to interact with the other components of the runtime system. For example, the interface between the scheduler and the profiler is the *profiler Frame*. If a new profiler is able to store the profiling information in the *profiler Frame*, it can easily replace our profiler. This way, the system components can be plugged in/out based on the user's preferences.

To show how the operating system and the hardware platform interact, a more detailed view of the runtime system is depicted in Figure 3.2. As it was

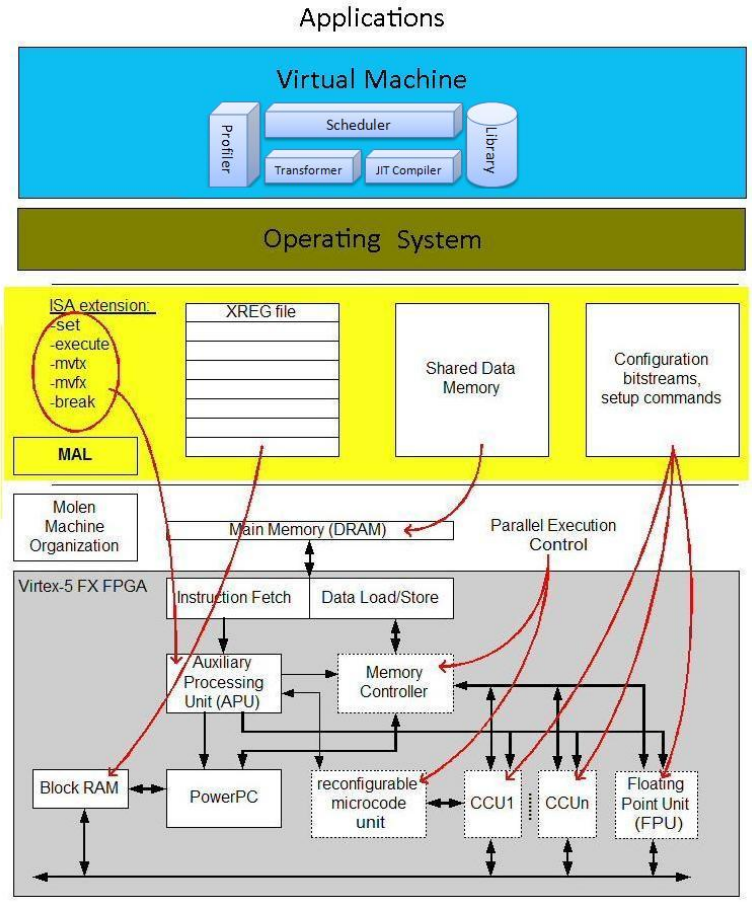

Figure 3.2: Runtime System Layers

mentioned before, our work is based on the MOLEN hardware platform. This Figure gives more detail on the hardware platform layer which is located at the bottom of Figure 3.1. As illustrated in the Figure 3.2, the hardware platform layer is consisted of the MOLEN Abstraction Layer (MAL) and the physical hardware layer.

MAL is actually managing the low-level hardware mapping. This means that the virtualization layer is responsible for high-level mapping decisions but it never communicates directly with the physical hardware. In fact, the virtualization layer uses the primitives provided by MAL, to do the actual communication

with the hardware.

In MAL, five instructions are required for controlling the hardware. The *set* instruction initiates the configurations of the core. The *execute* instruction controls the execution of the operations implemented on the core. The *break* instruction is utilized as a synchronization mechanism to complete the parallel execution and the *move* instructions are used to move the parameters to the exchange registers (XREG). The XREGs are used for transferring the parameters. These are accessible from both GPP and the cooperating cores. The shared memory is used to keep the data required by the computations. Furthermore, the second shared memory is used to keep the bit streams.

In the following, we describe the components of the virtualization layer.

3.3.1 Scheduler

The scheduler is responsible for deciding the mapping of the tasks to the cores. The objective is to identify certain tasks that can be accelerated on the faster cores and the remaining parts of the application will be executed on a regular general-purpose processor. We should clarify that in this thesis, a task can be a whole function or procedure but it can also be any cluster of instructions that is scattered throughout the application. These tasks can be either identified by the design tool chain at the compile time or by the runtime profiler at the runtime. In our work, the scheduler, does not change the order of the tasks execution within an application. It only decides about the mapping.

The scheduler monitors the applications binary and intercepts the kernels during the program execution. It also estimates the potential speedup that can be achieved when the kernels are executed on the reconfigurable hardware and estimates the initial cost of a hardware mapping. The scheduler decides hardware allocation after making the estimates. The hardware allocation is done according to the scheduling policy and other applications requirements. Any scheduling policy such as the Most Frequently Used, the Best Speedup and the Multi Constraint Knapsack presented in [50] can be used.

Within the MOLEN programming paradigm [51], the *set* and *execute* instructions can be effectively used as a mean for instrumentation. However, these instructions do not configure or execute anything on the hardware although they are meant to do so. The *set* just informs the scheduler of a possible future call to hardware and the *execute* instruction is a signal to the system to execute the hardware if that is possible. Both *set* and *execute* instructions include the corresponding kernel identifier. The system invokes the scheduler to decide about the actual execution after encountering *set* or an *execute* instructions.

The scheduler can also utilize more scheduling policies, which are more complex. Those policies can be based on the information from the design time, information collected at runtime or even some heuristics. A good source of information from design time is the Configuration call graph (CCG) which shows the future of the system and can help in finding a near optimal schedule. The CCG is a directed graph presenting the kernels identified by the profiler. Each node in this graph contains the kernel identifier, which uniquely identifies the kernel. The edges of the graph represent the dependencies between the configurations within the application.

3.3.2 Profiler

The profiler continually tracks the application behaviour and records statistics such as the number of references to one kernel. These statistics along with the scheduling policy, are used to determine where, when and how to execute the tasks. The profiler can also be used to find the kernels. The profiler must log the collected data in very fast and efficient data structures. Because, the overhead of retrieving information form the data structures can affect the performance of the profiler.

3.3.3 Transformer

The transformer replaces the software implementation of the kernel with a call to the hardware. It uses a binary rewriting mechanism and can again

impose a considerable overhead. This mechanism has to assure the correct input/output parameters transfer. However, we propose a mechanism within MOLEN programming paradigm that does not require binary rewriting. This will be explained in more details in Chapter 4.

3.3.4 Kernel Library

The kernel library is a pre-compiled and pre-synthesized set of kernels implemented for the underlying reconfigurable hardware. For each kernel, there might be a couple of different versions of implementations (with the same identifier) in the library. Each version differs with the others in one or more metrics for example logic size or/and power consumption. It is completely transparent from the application developers and does not impose any trouble to them.

Corresponding to each version, the library includes some metadata describing the kernel's characteristics such as the configuration latency, execution time, memory bandwidth requirement, power consumption, and the logic size. These metadata are mainly used by the scheduler based on the scheduling policy. Furthermore, it contains the physical mapping location of the kernel on the FPGA.

This library can be synthesized for each reconfigurable hardware platform resulting in the applications, independent of the underlying platform. There is a software wrapper for each operations implementation. The software wrapper is kept in the form of a Dynamic Shared Object (DSO). The application developer can also provide his own DSO along with the required metadata.

3.3.5 JIT Compiler

This is a just-in-time compiler that can be used to compile the kernels for which there is no implementation in the library. The compiler complies the binary from one architecture to the other architectures. Any just in time compiler can

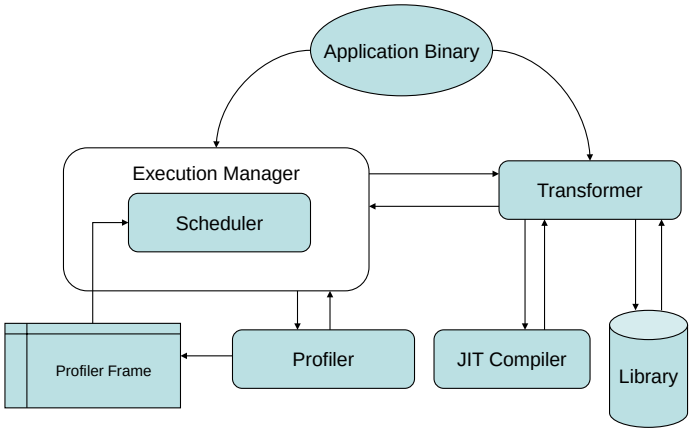

Figure 3.3: The Components Interaction

be used here nevertheless, there is no efficient one yet available.

3.4 Interfacing Components

The profiler has to communicate with the scheduler at runtime and, therefore, it needs to use data structures that allow for fast writing and reading of the profiled statistics. As the interaction between the scheduler and profiler is in real time, we propose the use of a shared memory and a double buffering mechanism to store and read the profiled data. The shared memory is shared between the profiler and the scheduler.

As shown in Figure 3.3, the profiler stores the collected information in the shared memory and the scheduler reads them. This shared memory is called *Profiler Frame*. This data is vital to the scheduler for the decision-making. As soon as the scheduler decides about mapping a task to a core, it invokes the transformer asking to transform the binary so that it can execute on that core. The transformer either performs a just in time compilation or uses an already existing implementation from the library.

To give a better overview of the sequence of activities happening during the

task scheduling, we present two case studies.

In the first case study, there are three applications and two cores besides GPP. We present this case study to show the interaction between the application, the scheduler and the kernel library. In this case study, applications *App1* and *App2* require operation *Op1* on the core *Core1* and application *App3* requires operation *Op3* on the core *Core3*. This case study clearly shows how the applications are being executed in what order their requests are served. This case study also clearly shows that the execution on the cooperating cores is non preemptive.

The sequence diagram of this case study is presented in Figure 3.4. The *OS* starts *App1* at the beginning. After sometimes, *App1* sends a request for operation *Op1* to be executed on core *Core1*. This request goes to the *scheduler* and the *scheduler* queries the *library* to check if there is an implementation for this operation in the library, which suits *core1* . The library's answer to this query is positive and, therefore, the *Core1* is configured with the *Op1* and it starts executing it. Looking at the execution of *Op1* on *Core1*, it is clear that the execution continues to the end of *Op1* and it is not preemptive.

At a later stage, the *OS* sends a suspend signal to the *App1* as its time slice is finished. Then, the *OS* starts *App2*, which needs the *Op1*. As this operation is now in busy mode on core one, the *App2*'s request is rejected. After finishing *App2*'s time slice, the *App3* starts execution. *App3* needs operation *Op2* on the *core2*. The *scheduler* therefore, queries the library, which returns a negative result. As a result, the scheduler asks JIT compiler to produce an implementation for operation *Op2* for *core2*. Then, the *Op2* is configured on *core2* and it starts execution.

The second case study shows the profiler role in the whole execution process. In this case study, there are two application and two cores. This case study shows how the profiler recognizes a kernel and interacts with the rest of the system.

Figure 3.5 presents the sequence diagram of the second case study. As it is shown in the figure, at the beginning, the applications are executing one after

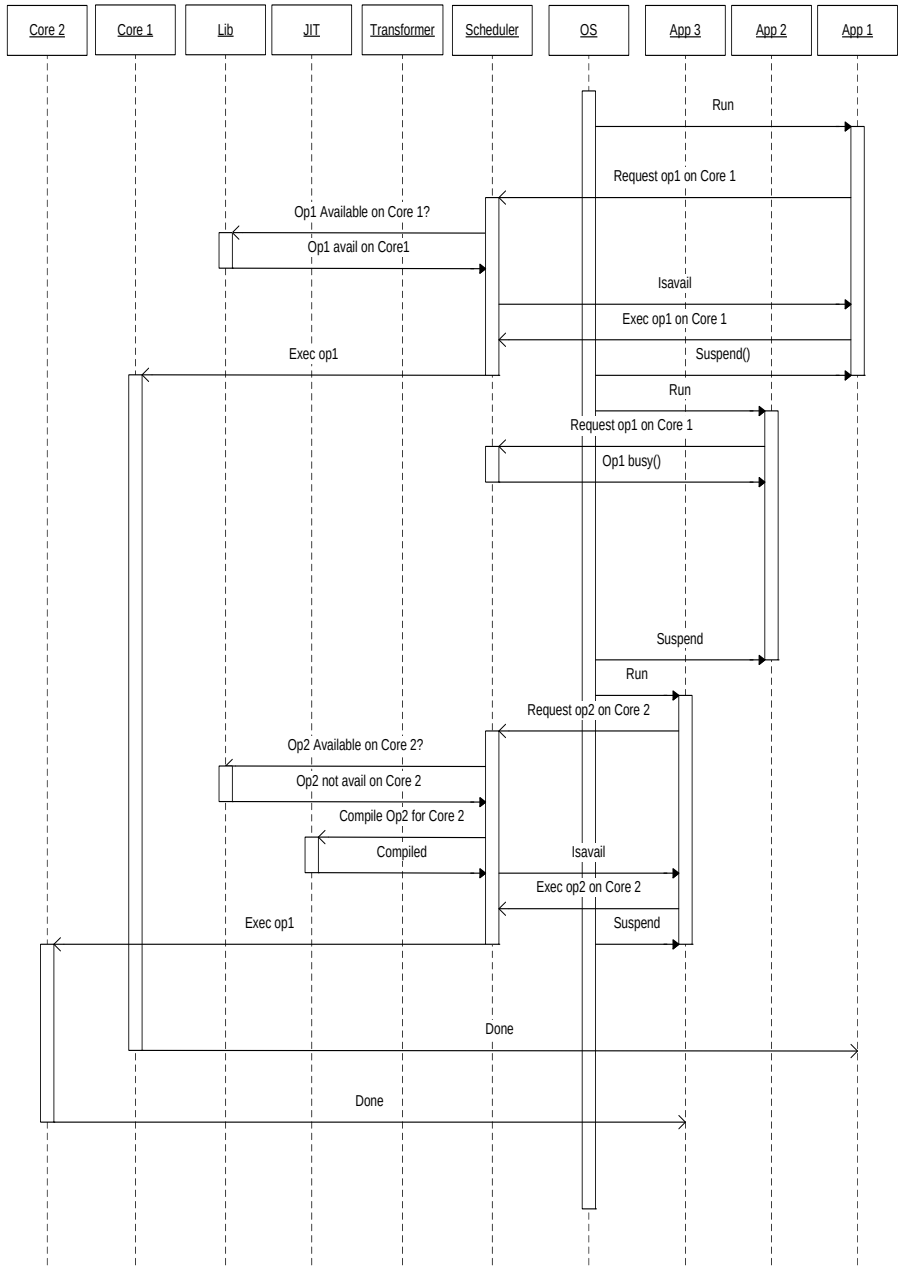

Figure 3.4: Sequence Diagram of the First Case Study

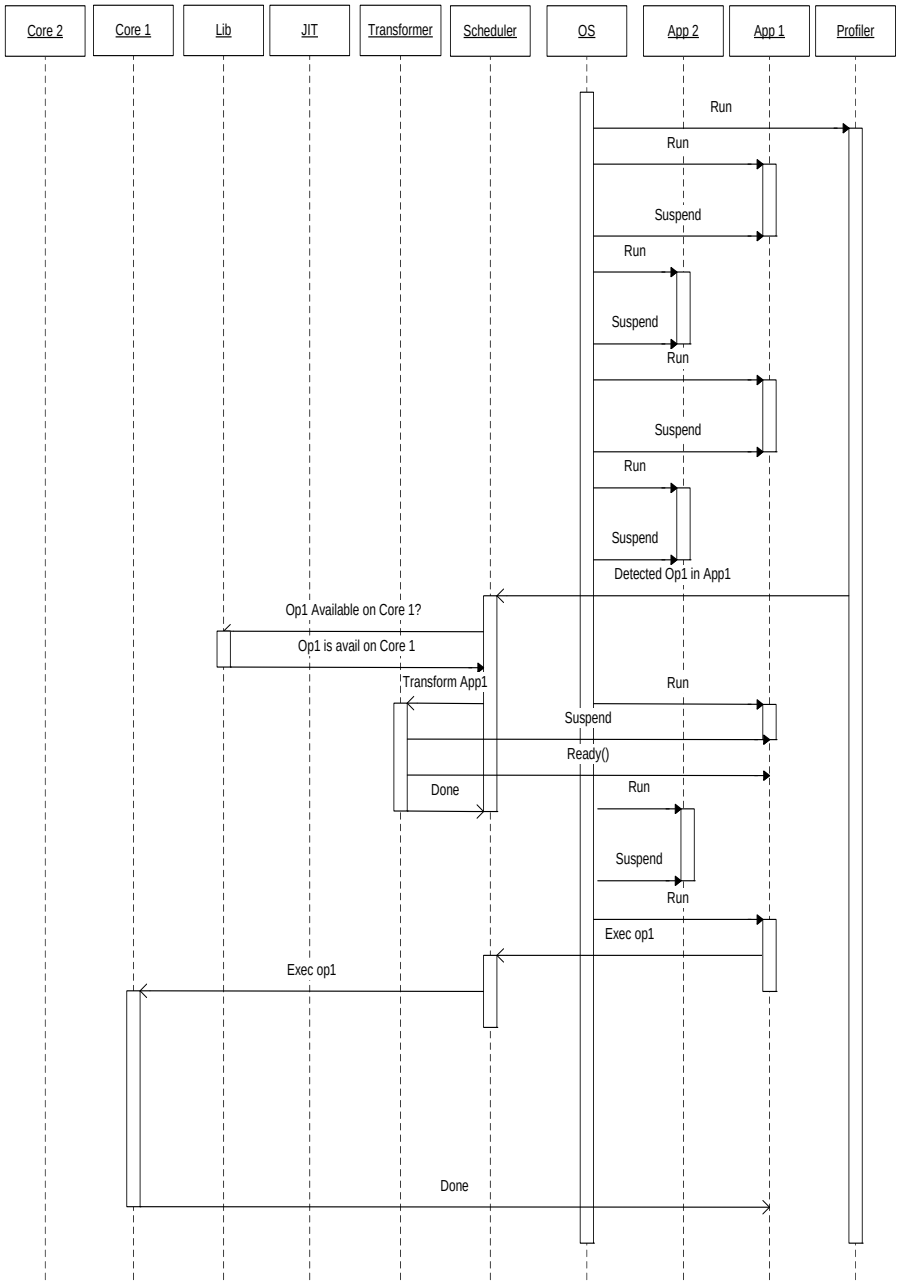

Figure 3.5: Sequence Diagram of the Second Case Study

the other for sometimes. During this period, the profiler is monitoring the behaviour of these applications. After a while, it can detect that one of a task in the application *App1* is a kernel and can be accelerated on a faster core. As a result, the *profiler* sends a signal to the *scheduler* informing it about the new kernel. The *scheduler* then checks the *library* to see if there is an implementation available for that operation which is the case in this example. After that, the *scheduler* sends a request to the *transformer* to transform the *App1* binary to be able to use this operation on *core1*. The *transformer* has to suspend the *App1* and start modifying its code. After finishing this process, *App1* can use the operation *Op1* in its later execution process.

3.5 Conclusion

In this chapter, we presented an overview over our runtime system. The runtime system is responsible to operate the system and address all the conflicting issues between the tasks. It incorporates a number of components namely the scheduler, the profiler, the transformer, the JIT compiler and the kernel library. We briefly described the task of each component, however, a more detail discussion on them will be given in the following chapters.

We also discussed the interaction and interfacing mechanisms between these components. The most important interface is the interface between the scheduler and the profiler. This interface has to be very fast and efficient to be able to address the real-time constraint of interaction between the profiler and the scheduler.

Furthermore, we presented two case studies which show two different scenarios of cooperation and interaction of the components. Other scenarios can also be envisioned.

4

Task Abstraction

In the previous chapters, we provided a summary of the MOLEN programming paradigm. The MOLEN programming paradigm is proposed to offer a general function like execution of the compute intensive parts of the programs on the reconfigurable processors. Within the MOLEN programming paradigm, the MOLEN *set* and *execute* primitives are employed to map an arbitrary function on the reconfigurable hardware. However, these instructions in their current status are only intended for single application execution scenario.

In this chapter, we extend the semantic of the MOLEN *set* and *execute* to have a more generalized approach and support multi-application and multitasking scenarios. This way, we propose the runtime SET and EXECUTES as APIs provided by the runtime system. We show how we use these APIs to abstract away a task call from its actual implementation.

4.1 Introduction

Within a heterogeneous multi-core system, different operations might be mapped to different cores. In case of a multi-application scenario, where the exact configuration of the system load is not known at design time, the task mapping has to be performed at runtime.

In our system, each application is composed of several tasks. Each task can be mapped to a certain core, or it can be executed on the general purpose processor. A certain task might be used in more than one application. To simplify the design and to be in line with the MOLEN programming paradigm, we assume that a task is in the form of a function in the program code. Therefore, each function in the code is a task unless it is a very small function. Some of these functions (tasks) are recognized by the compiler as compute intensive tasks. For calling these tasks, the compiler uses the runtime system's APIs. However, there might be other compute intensive tasks which are not recognized as such by the compiler. These tasks may be identified at runtime by the profiler and therefore, the runtime system has to change the call to these tasks in such a way that they use the runtime APIs.

For each task in the system, there might be a variety of different implementations. There are several reasons for that. First, each core might have different hardware architecture and therefore, dissimilar binary standards. This means that a task when mapped on core number one, should use a different binary than when mapped on core number two. Second, based on the desirable level of performance, each task might have different implementations even for a particular core, especially when that core is a kind of accelerator such as a FPGA. For example one implementation is optimized for better power consumption, the other is faster, etc.

It is obvious that managing all these varieties in implementations needs complicated mechanisms. It is very difficult for the program developers to concentrate on high-level algorithmic issues and at the same time deal with such complex matters. To overcome the complexity of handling all these issues, a task abstraction mechanism is needed in which the task in its conceptual view should be abstracted from its actual implementation. In this way, the programmers can only express the computation in its logical and conceptual view and then at runtime the actual binding to a certain implementation happens.

The compute intensive operations are usually implemented on the faster cores for increasing the system performance. Different vendors provide their own

implementation for each specific operation. The main challenge is to integrate these implementations - whenever possible - in new or existing applications. Such integration is only possible when application developers as well as hardware providers adopt a common programming paradigm.

The MOLEN programming paradigm [51] is a sequential consistency paradigm for programming multi-core, polymorphic machines. This paradigm allows parallel and concurrent hardware execution and is currently intended for single program execution. However, the movement towards many applications, multitasking scenarios adds new design factors to the system such as dealing with the core as shared resources. These factors prevent using the MOLEN primitives in their existing form. They should be extended in such a way that besides offering the old functionalities, they have to resolves the conflicting issues between different applications at the time of primitive usage. In this chapter, we present how the MOLEN programming paradigm primitives are extended and adapted into the proposed runtime system.

The MOLEN hardware organization is established based on the tightly coupled co-processor architectural paradigm. Within the MOLEN concept, a general-purpose core processor controls the execution and reconfiguration of reconfigurable coprocessors. A more detailed view on MOLEN is given in chapter 3. In the next section, we describe the MOLEN programing paradigm.

4.1.1 MOLEN Programming Paradigm

The MOLEN programming paradigm presents a programming model for reconfigurable computing that allows modularity, general function like code execution and parallelism in a sequential consistency computational model. Furthermore, it defines a minimal ISA extension to support the programming paradigm. Such an extension allows the mapping of an arbitrary function on the reconfigurable hardware with no additional instruction requirements.

The MOLEN programming paradigm introduces new super instructions to operate the FPGA from the software. An operation executed by the RP is

divided into two distinct phases: set and execute. In the set phase, the RP is configured to perform the required task and in the execute phase the actual execution of the task is performed. This decoupling allows the set phase to be scheduled well ahead of the execute phase, thereby hiding the reconfiguration latency. This phasing introduces two super instructions; *set* and *execute*.

The *set* instruction requires as a single parameter the beginning address of the configuration microcode. When a *set* instruction is detected, the Arbiter reads every sequential memory address until the termination condition is met and then the Arbiter configures it on the FPGA. After completion of the set phase, the hardware is ready to be used for the targeted functionality. The execution of the functionality is done using the *execute* instruction. This instruction utilizes as a single parameter the address of the execution microcode. The execution microcode performs the real operation. This consists of reading the input parameters, performing the targeted computation, and writing the results to the output registers.

As it is obvious, these two instructions are based on the assumption of a single thread of execution. There is no need for changes in the operating system as long as there is only one application dealing with the FPGA. That is because there is no competition for the resources and the application has full control over the FPGA. In case of serving several concurrent applications on the same system, *set* and *execute* cannot be used in the same way as they are used in the single application paradigm. Each application might issue its own *set* and *executes* which most probably have conflicts with the others *set* and *executes*. The operating system has to resolve all the conflicts in such a scenario.

In this chapter, we extend the semantic of MOLEN *set* and *execute* to have a more generalized approach and support multi-application, multitasking scenarios. This way, we propose the runtime SET and EXECUTES as APIs provided by the runtime system. In the next section, we review the runtime execution environment in which the MOLEN primitives are used to operate the FPGA. After that, we present the runtime primitives in Section 4.2.

4.1.2 The Runtime Environment

In chapter 3, we presented the proposed runtime environment. The runtime environment is a virtualized interface that decides how to allocate the hardware at runtime based on the dynamically changing conditions of the system. Moreover, this layer hides all platform dependent details and provides a transparent application development process.

The runtime environment components include a scheduler, a profiler, and a transformer. It might also incorporate a JIT compiler for on the fly code generation for the target cores, e.g. FPGA bit streams.

The runtime system also includes a kernel library of a set of precompiled implementation for each known operations. This means, we might have multiple implementations per operation. Each implementation has different characteristics, which are saved as metadata and can contain the configuration latency, execution time, memory bandwidth requirements, power consumption, and physical location on the reconfigurable fabric.

This library can be synthesized for each reconfigurable hardware platform resulting in the applications, independent of the underlying platform. There is a software wrapper for each operations implementation. The software wrapper is kept in the form of a Dynamic Shared Object (DSO). The application developer can also provide his own DSO along with the required metadata.

In the next section, we present the runtime primitives. The SET and EXECUTE APIs here are an extension of the MOLEN *set* and *execute* instructions respectively.

4.2 MOLEN Runtime Primitives

To keep the changes in the compiler and design tool chain [52] as limited as possible and to provide legacy compatibility, we propose the MOLEN runtime primitives as follows.

We proposed two APIs in the runtime system; the SET and EXECUTE. The functionality of these APIs is almost identical to the original MOLEN *set* and *execute*. Besides the normal MOLEN activities, these APIs also take care of the sharing of the FPGA among all the competing applications. This means, the runtime system is responsible to check the availability of the FPGA or targeted core at the time of the call. Furthermore, it can impose some sort of allocation policies such as priorities and performance issues.

Figure 4.1 shows the sequence diagram of the operation execution in the proposed runtime system. When an application encounters a call to the SET for a specific operation, it sends its request to the runtime system (VM). The VM then checks the library to look for all the appropriate implementations. If no such implementation is found, it sends a FAIL message back to the application. The FAIL message means the SET operation cannot be performed. Otherwise, based on the scheduling policy it selects one of the implementations (IM) and configures it on the FPGA. The OS driver is the low-level interface between the operating system and the physical FPGA fabric. Finally, the VM sends the address of the IM to the application.

Comparing Figure 4.1 with Figure 3.4 and 3.5 in Chapter 3, we should mention that the VM in this figure is equal to the Scheduler in those figures. The Scheduler in this figure is actually representing the scheduling policy which will be explained in the next chapter. Figure 4.1 is actually a more detailed view of the *Exec* arrows of the Figure 3.4 and 3.5 in Chapter 3.

Later on, when the application encounters the EXECUTE instruction, the system checks if the IM is still configured and ready. The execution can start right away when the IM is configured and ready. The system has to follow the SET routine again and start the execution when the IM is not configured and is not ready. If any problem occurs during this process, a FAIL message will be sent back to the application. A FAIL message received by the application means the software execution of the operation has to be started. In the following two sections, we describe the SET and EXECUTE APIs in more detail.

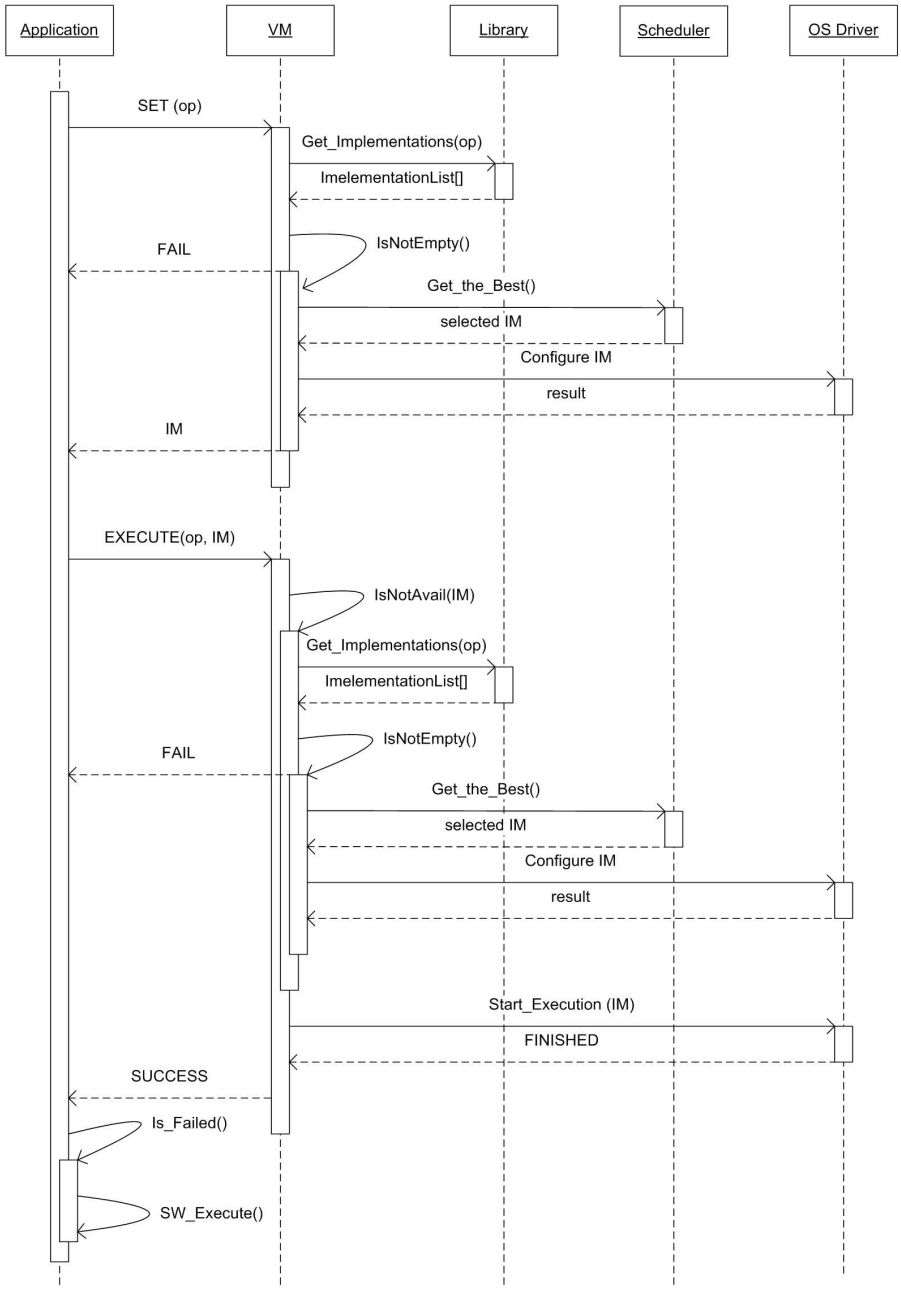

Figure 4.1: The Operation Execution Process

4.2.1 SET

The SET API receives the operation name as an input. We assume all the supported operations have a unique name. This assumption is based on the idea of having a library of a number of different implementations per operation in the runtime environment. Listing 4.1 shows the pseudo code corresponding to the SET API.

In Listing 4.1, line 2 creates a list of all the existing implementation for the operation. If the physical location corresponding to any of those implementations is busy, e.g. another application is using that resource, that implementation is removed from the list in line 4-1 and the loop continues to the next element in the list. Some of the implementations might already be configured on the FPGA. This means there is no need for configuring them again. Those implementations are added to another list in line 4-2 and the best candidate (here the fastest one and can be a different one based on the scheduling policy) is return to the main program in line 5. If such an implementation does not exist, the algorithm goes further to choose one of the other implementations and starts configuring it in line 6. This selection is very dependent on the scheduling (line 6-1). The configuration process is discussed in Section 4.2.3.

Listing 4.1: The SET API

```
SET (input: Operation op): return Implementation IM

1- SET begins
2- Assume im_list the list of all the implementations
  corresponding to the op in the library;
3- Assume co_list as an empty list;
4- foreach IM in im_list
        4-1- If the corresponding physical location of IM is
          busy
                Remove IM from im_list;
                Continue;
        End if
        4-2- If IM is already configured on the FPGA
                Add IM to the co_list;
```

```
        End if
End for
5- If co_list is not empty, return the IM with the minimum
  execution time from co_list;
6- If im_list is not empty
        6-1- Choose IM from the im_list based on the scheduling
           policy; //Calls the Scheduling Algorithm e.g. IM =
          LDiF(im_list);
        6-2- If IM is NULL, Return FAIL;
        6-3- Configure IM on FPGA;
        6-4- Return IM;
End if
7- Return FAIL;
8- SET ends
```

By looking more carefully to the SET procedure, it is obvious that the hardware execution part is non-preemptive. We check the busy status of the physical locations in step 4-1 before making any replacement decision. In preemptive hardware scheduling scenario, the system has to offer the ability to save and restore hardware status among other services.

Preemptive scheduling is outside the scope of this thesis. However, more detail on the preemptive scheduling can be found in [53] and [54]. Moreover, we do not have task migration from CPU to FPGA. As soon as a task starts on the CPU, it cannot be executed on the FPGA. However, at a later stage in another run of that task, it might be executed on another core. This means that we support weak code mobility but not strong code mobility. Strong code mobility involves moving the code and the execution state from one core to another.

4.2.2 EXECUTE

The EXECUTE is also an API offered by the runtime system. It has two input arguments; the operation name and the address of the configured implementation in the SET phase. Listing 4.2 shows the pseudo code corresponding to the EXECUTE.

Listing 4.2: The EXECUTE API

```
EXECUTE (input: Operation op; input: Implementation IM)

1- EXECUTE begins
2- If IM is not NULL and IM is not busy
        Execute IM;
        Return SUCCESS;
End if
3- IM = SET (op);
4- If IM is not NULL and IM is not busy
        Execute IM;
        Return SUCCESS;
End if
5- Return FAIL;
6- EXECUTE ends
```

The operations might be shared between different applications (This task sharing is one of the motivations behind the idea of using dynamic shared object (discussed in section 4.2.3). On the other hand, there might be a gap between the occurrence of the SET and EXECUTE (e.g. because of the compiler optimizations to hide the reconfiguration delay). Therefore, the control might go from current application (aap1) to another application (app2). The second application (app2), therefor, might use the implementation which is set by the first application (aap1). That is why the busy status of the IM (in line 2 of Listing 4.2) has to be checked. If it is not busy, it can start execution.

As the EXECUTE performs a SET in any case, it is possible to directly call the EXECUTE without any prior SET or any successful prior SET. In such a case, ECECUTE is called with IM being null. In case of having a busy implementation or a null, the SET has to be performed again. This is done in line 3 of Listing 4.2. Finally, the algorithm executes the implementation in line 4 of Listing 4.2. If any problem occurs during the EXECUTE, it return a FAIL which means the operation has to be executed in software. The execution process is discussed in section 4.2.3.

4.2.3 Dynamic Binding Implementation

The actual binding of the function calls to the implementation happens at runtime. To do that we use the ELF binary format delayed symbol resolution facility and the position independent code.

For each operation implementation in the library, there is a software wrapper with two functions. One function performs the low level configuration of the operation (the original *set*) and the other performs the low level execution of the operation (the original *execute*). In the runtime SET, when the reconfiguration takes place (line 6-2 in Listing 4.1), the low level SET from this software wrapper is called. Similarly, in the runtime EXECUTE (lines 2 and 4 in Listing 4.2) the low level *execute* is called. The reason that we can use the traditional *set* and *execute* at this point is that the sharing controls has already been performed by the runtime system and it is safe to call the original *set* and *execute* instruction.

As it is mentioned in Section 4.1.2, this software wrapper is kept in the form of a Dynamic Shared Object (DSO). Given the name of a DSO by the SET (line 6-2 in Listing 4.1), the system dynamically loads the object file into the address space of the program and returns a handle to it for future operations. The name of the DSO is actually the name of the chosen implementation. We do this process by using the Linux *dlopen* function. The *dlopen* is called in LAZY mode. The LAZY mode says to perform resolutions only when they are needed. This is done internally by redirecting all the requests that are yet to be resolved through the dynamic linker. In this way, the dynamic linker knows at request time when a new reference is occurring, and resolution occurs normally. Subsequent calls do not require a repeat of the resolution. To find the address of each function in the DSO, we use Linux *dlsym* facility. The *dlsym* takes the name of the function and returns a pointer containing the resolved address of the function.

At the time of original *set* call (line 6-3 in Listing 4.1), all the required parameters needed by the FPGA have to be transferred to MOLEN XREGS. Then, it

starts configuring the FPGA. At the time of the original *execute* (lines 2 and 4 in Listing 4.2) call, the address of the second function is resolved using the *dlsym*. By this function pointer, we can invoke the required operation.

Furthermore, a support tool is proposed for simplifying the creation of DSO files to be added to the runtime library (especially for third-party modules) . The idea is simple: It shows a template of the wrapper and the program developer has to add a few lines of code to it. Besides, the program developer has to write explicitly the parameters transfers instruction in the pre-defined template (moving the parameters to XREGs). At the end, the tool compiles the code for Position Independent Code (PIC) and converts it to a DSO. The tool provides a very simple interface to gather the metadata required by the runtime scheduler such as the configuration latency, execution time, memory bandwidth requirements, power consumption, and physical location on the reconfigurable fabric and stores them in an appropriate format.

4.3 Evaluation

The overall performance improvement through acceleration and the invoked overhead are the two important parameters for the evaluation of the proposed mechanism. In this section, we first discuss the overhead involved in the proposed mechanism and then we present the experimental results. The execution time overhead imposed by loading of a DSO occurs at two places; at run and load-time.

At runtime, each reference to an externally defined symbol must be indirected through the Global Object Table (GOT). The GOT is a table that contains the absolute addresses of all the static data referenced in the program. The compiler/assemble make all data references through this table. During runtime linking this table is initialized by the runtime linker. In most cases, the only runtime overhead is the need to access imported symbols through the GOT. Each access requires only one additional instruction.

At load-time, the running program must copy the loaded code from an object file into the memory and then link it to the program. The load-time overhead is the time spent to load the object file. For a null function call in our system, the load time is about 0.75 milliseconds. For a typical wrapper function, the load time increases to about 2 milliseconds. We should mention that the increase in the input parameters' size might increase the size of the wrapper function since each parameter needs a separate instruction to be transferred to the MOLEN XREGs.

In the next section, we present the time it takes to load some of the well-known kernels using the proposed mechanism by hardware emulation.

4.3.1 Overhead in a Single Call

To show the overhead of loading DSOs, we performed some experiments. We take some well-known kernels and try to show the overhead when calling them using the proposed mechanism. We developed a very simple program which call each kernel once using the SET and EXECUTE APIs. All the kernels are implemented in the form of a DSO.

The experiment workload is obtained from an interactive multimedia Internet based testing application [55]. The workload's kernels and the results are listed in Table 4.1. The timing is based on the software or hardware execution of the tasks on the Xilinx Virtex-5 FPGA Family. The software execution time is on a hard PPC. The configuration time is the time for partial reconfiguration of each task. The numbers in the first three columns are generated using the REC-BENCH tool [55]. These numbers are used as the input to our emulator. The last two columns are obtained from employing the proposed mechanism. It should be mentioned that the hardware execution is emulated in these experiments.

The last column in Table 4.1, shows the operation total execution time when it is executed only once. This means the execution time is the sum of the software wrapper load delay plus the reconfiguration delay plus the HW execution time. The first column is the software only execution time (no FPGA) which is

Table 4.1: Workload Kernels

Kernels	SW execution time (ms)	HW Execution time (ms)	Configuration Delay (ms)	SW wrapper Delay (ms)	HW total execution time (ms)
Epic-Decoder	19.87	8.56	5.82	2.11	16.49
Epic-Encoder	11.87	5.22	2.49	1.17	8.88
Mpeg2-Decoder	77.35	2.43	3.64	1.47	7.54
Mpeg2-Encoder	10.39	1.94	4.87	1.81	8.62
G721	42.42	4.64	5.82	2.57	13.03
Jpeg-Decoder	68.39	8.63	8.72	3.41	20.76
Jpeg-Encoder	169.33	35.23	10.98	4.51	50.72
Pegwit	166.06	36.34	5.88	2.59	44.81

mentioned just as a point of reference.

As shown in Table 4.1, the software wrapper delay over the total execution time varies between 5 to 20 percent for different kernels.

However in general, when a kernel is loaded (incurring one wrapper and reconfiguration delay), it executes more than once which means the overhead decreases as the number of executions increases. To show such a reduction in execution time, we evaluate the overall execution time in the following section.

4.3.2 Overall Overhead

To show overall system performance, we used 5 different workloads from interactive multimedia Internet based test; the workload varies based on the number of tests taken and the number of kernels, which are used in each test. We have workloads for 12 applicants (821 kernels), 24 applicants (1534 kernels), 36 applicants (2586 kernels), 48 applicants (3032 kernels), and 60 applicants (4164 kernels). It should be mentioned that each test taker has its own process in the system and therefore the number of applications are equal to the number of test takers. In such a scenario, each test taker corresponding application is competing against the others to obtain the FPGA resources.

Furthermore, we have implemented a very simple scheduling algorithm. This

Table 4.2: Overall Execution Time

No application No kernels	12 821	24 1534	36 2586	48 3032	60 4164
SW only (ms)	135654.08	260508.60	381329.44	501860.74	641478.23
SW/HW (ms)	59580.79	121977.13	186415.10	256929.84	335276.90
Wrapper overhead (ms)	2983.03	5884.87	7654.71	10814.62	11463.15
Wrapper overhead percentage	5	5	4	4	3
Speedup	2.28	2.14	2.05	1.95	1.91

algorithm picks the fastest implementation and executes it. If the fastest implementation is not available, the scheduler picks the next one.

We compared the software only execution with the hardware/software execution. As shown in Table 4.2, the overall system speedup varies between 2.28 to 1.91. The wrapper overhead to the overall execution time is between 3 to 5 percent. As the number of test takers increases, the chance of executing an already configured kernel increases and as a result, the wrapper overhead reduces.

On the other hand, since the system loads increases, the overall speedup also decreases. That is because the FPGA resources are limited and fixed. Therefore, when the system load increases the HW/SW execution time gets closer to the SW only solution and as a result, the speedup reduces.

One might argue that the obtained speedup of around 2 is not as impressive as it should be. There are some reasons behind this number. First of all as we used the processor coprocessor paradigm with only one GPP, the GPP load is a bottleneck in our system. Most parts of the code which are not compute intensive are executed on the GPP and we can not exploit any parallelism in those parts. Furthermore, we use the library to load the kernels binary from which increases the overhead of the system. Operating system overhead (for example the system calls overhead) is also another source of system overhead. However, we believe that the imposed overhead is actually the price we pay to achieve a more flexible system.

4.4 Conclusion

In this chapter, we extended the MOLEN programming paradigms primitives to use them in presence of an operating system and in multi-application, multi-tasking scenarios. The MOLEN primitives in their current status are just for single application execution. We discussed the details of the SET and EXECUTE APIs and presented the dynamic binding mechanism, which is used by these APIs to bind a task call to a proper task implementation. Our experiments show that the proposed approach has a negligible overhead over the overall applications execution.

5

Scheduling

One of the major challenges in heterogeneous multi-core systems is the scheduling and allocation of the tasks into different cores. In this chapter, we present a two level scheduling mechanism. To overcome the complexity of identifying kernels at runtime, we use the compiler support. The compiler provides the runtime system with a configuration call graph which is used as a viable source of information for the scheduling algorithm. We combine the configuration call graphs from all running applications and extract our scheduling parameters for each task from this graph. We also present a number of scheduling policies in this chapter.

5.1 Introduction

Current reconfigurable multi-core systems can serve several applications as they have huge computing power and huge system resources. Therefore, the system should be able to share its computing resources among the tasks within a single application or even among different applications.

The tasks in a static single application environment can be scheduled efficiently by partitioning and design time scheduling of the tasks [56–61]. However, such predictable application schedules form only a small subset of all the applications.

Hence, the reconfigurable resources need to be managed at runtime for the multitasking environments by a runtime scheduler [62–71].

The main problem is to assign the limited system resources based on the needs of the applications at the execution time. The objective is to identify certain tasks that can be accelerated on the faster cores and the remaining parts of the application will be executed on a regular general-purpose processor. We should clarify that in this thesis, a task can be a whole function or procedure but it can also be any cluster of instructions that is scattered throughout the application.

Task scheduling can have different conflicting objectives, like improving performance, power consumption, or the memory footprint. As a result, many different scheduling algorithms can be used for task scheduling. For example during the program execution, a scheduler can estimate the potential speedup that will be achieved when running a task on a reconfigurable hardware and the initial cost of a hardware mapping. The scheduler decides how to allocate the hardware based on the scheduling policy and considering the other applications requirements

In the proposed system, task scheduling takes place in two phases. Firstly, at compile-time, the compiler performs static scheduling of the reconfiguration requests assuming a single application execution. The main goal in this stage is to hide the reconfiguration delay by configuring them well in advance the execution point. Then at runtime, the runtime system performs the actual task scheduling. In this stage, the SET and EXECUTE instructions are just a hint to the runtime system. The runtime system decides based on the runtime status of the system and it is possible to run a kernel in software even though the compiler already scheduled the configuration. In the following, we explain each phase in more details.

We should mention that in our system, each application is composed of several tasks. Each task can be mapped to a certain core, or it can be executed on the general-purpose processor. A certain task might be used in more than one application. To simplify the design and to be in line with the MOLEN programming paradigm, we assume that a task is in the form of a function in

the program code. Therefore, each function in the code is a task unless it is a very small function. Some of these functions (tasks) are recognized by the compiler as compute intensive tasks. For calling these tasks, the compiler uses the runtime system's APIs. However, there might be other compute intensive tasks which are not recognized as such by the compiler. These tasks may be identified at runtime by the profiler and therefore, the runtime system has to change the call to these tasks in such a way that they use the runtime APIs. Besides the help of the compiler and the runtime system, the program developer can also directly use the runtime system's APIs in the program code.

5.2 Compile Time Scheduling

One of the tasks of the runtime system is to intercept the kernels. Then, it can map them to the faster cores. Intercepting the kernels at runtime is not trivial. It requires complex profiling tools to provide the required information to the scheduler. Besides using the runtime profiling information, we can also use the compiler and design time tools hints for identifying compute intensive tasks.

Within the MOLEN programming paradigm, the set and execute concept can be effectively employed as an abstraction primitive which can separate the notion of task call from the real task implementation as described in Chapter 4. The runtime scheduler then employs a binding mechanism to attach each task to a proper implementation at runtime.

The compiler performs static scheduling of the reconfiguration requests assuming single application execution. The goal of this scheduler is to minimize the impact of the reconfiguration latency over the application performance. The reconfiguration latency is a major drawback when using reconfigurable accelerators. However, by configuring the tasks well in advance, the compiler can reduce the effect of reconfiguration on the overall application performance [72]. Here, we show a motivational example presented in [73].

In Figure 5.1, op1, op2, and op3 are the kernels, which can be placed on an

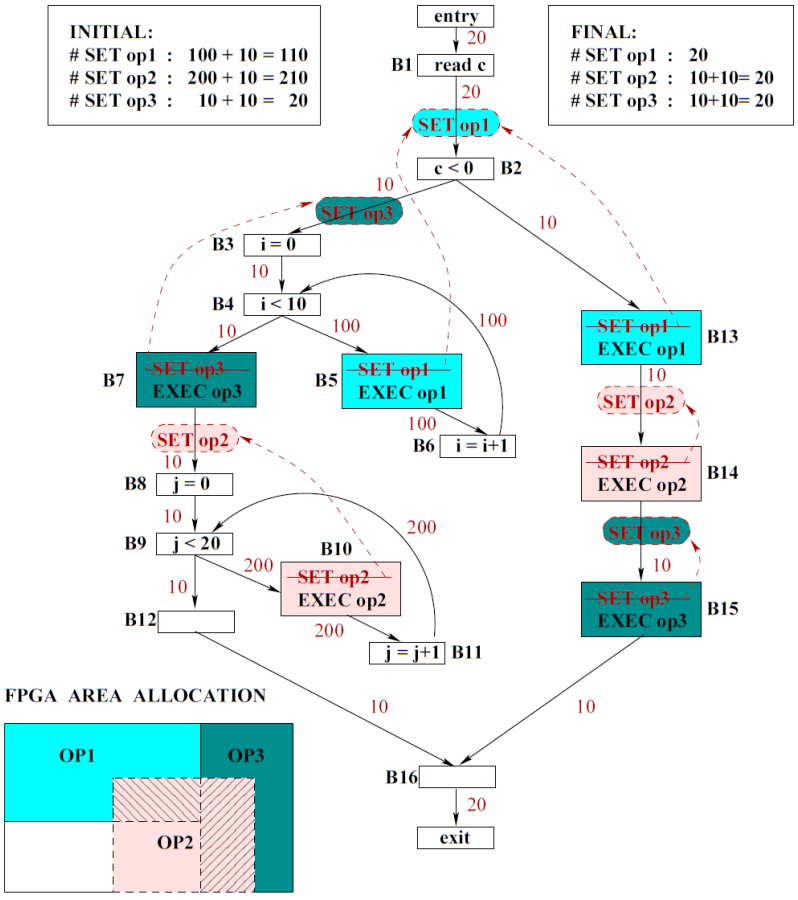

Figure 5.1: Compiler Instruction Scheduling

FPGA. op2 has physical conflicts with both op1 and op3 and therefore can not be place with any of them concurrently. However, op1 and op3 can be placed at the same time. The numbers associated with each edge of the graph represent the execution frequency of the edge. As it is obvious in the figure, operation op1 is configured 100 times in B5 and 10 times in B13. By moving SET op1 instruction on the (B1, B2)-edge, we can reduce the number of configuration calls for op1 to 20. The hardware configuration for op2 at B10 cannot be moved earlier than node B7, as it will change the hardware configuration for op3 that must be performed at B7. Moving forward the SET of the other operations such

as B15 hides the configuration latency to some extent.

Therefore, we make use of the compiler optimization in order to optimize a single application execution. As shown in the motivational example, the compiler changes program by moving the SETs to an earlier place in the code to hide the configuration latency.

However during runtime, all these optimized applications run together. This concurrent execution causes some conflicts when using the computing resources. Therefore, the runtime system might not be able to perform all the optimizations done by the compiler. In the next section, we describe how the runtime scheduler works and how it resolves the conflicts.

5.3 Runtime Scheduling

One of the basic limitations of multitasking reconfigurable computers is the size of the reconfigurable fabric. The size of the required logic for all the applications usually exceeds the size of the fabric. Therefore, the most important design factor for a runtime scheduling mechanism is the replacement policy. This replacement algorithm determines which part of the logic area should be replaced whenever some space is needed.

The scheduler should calculate which tasks are likely to be needed soon and if they are already configured, not to replace them with other tasks. This is often in combination with pre-cleaning, which guesses which configured tasks currently on the FPGA are not likely to be needed soon. However, this cannot be managed by the runtime systems because, it is impossible to compute reliably how long it takes before a task will be used again, except when all the applications running on a system are known beforehand.

One way to predict the future behaviour of the system is to look at the past behaviour of it. However, it has been proven in [74] that the past behaviour is not a good heuristic to be used for replacement in the reconfigurable computers. We really need to know the future behaviour to be able to make good decisions.

One solution is to provide information from the design time using CCG. With the help of CCG, the scheduler can look at future of the application behaviour and extract useful information about the tasks execution. For example, the scheduler can calculate parameters such as the distance to the next call and the frequency of the calls in future, which can be used as the replacement decision parameters.

In the followings, we give more details on the replacement policy and the parameters we use for the replacement decisions.

5.3.1 The Replacement Policy

Many decision parameters can be used to decide for the replacement such as the frequency of use in the past, the distance from last call, or even a random selection.

All of these parameters are heuristics to keep the best kernel configured. Having information about the future is indeed a real success factor in this decision. In this section, we introduce a few decision parameters such as the distance to the next call, the frequency of the calls in future, the frequency of calls in the past and time improvement.

At each scheduling point, we define the distance to the next call for the task T as follows. Let d_i be the distance to the next call for application i. This represents the number of task calls between the current execution point and the next call to the task T, in the breadth first traverse of the CCG. We use the minimum value of d_i as the distance to the next call for task T.

Similarly, for a specific task T, we define the frequency of the calls in future for an application as the total number of calls to the task T in all the successors of the current execution point. Afterwards, the frequency of the calls for the task T in the whole system is the sum of the frequencies of all applications.

Deciding based on the distance to the next call means we want to remove the task that is used furthest in future. Similarly, considering the frequency of calls

in future as the decision parameter, means the replacement candidate is the task that is used less frequent in the future.

The time improvement heuristic is defined by two parameters: the first is the reduction in task execution time, which is obtained by comparing the hardware execution time of a task on FPGA and the software execution time on the general-purpose processor. The second parameter is called distance to the next call which is defined above.

These parameters except the frequency in the past can be calculated from the CCG. CCG is a directed graph that is generated by the compiler and gives information about the execution behaviour of an application. In the next section, we provide a detailed explanation of the CCG.

5.3.2 Configuration Call Graph

A Configuration Call Graph is a directed graph, which provides the runtime system with information about the execution of the application. CCG is generated by the compiler. The runtime scheduler can predict the future behaviour of the system from a CCG. Each CCG node represents a task. The edges of the graph represent the dependencies between the configurations within the application. The nodes of a CCG are of three types: operation information nodes, parallel execution nodes (AND nodes), alternative execution nodes (OR nodes).

The parallel execution nodes specify that all of their successor nodes have to be executed in parallel. The alternative execution nodes specify that only one of their successors have to be executed. However, the selection of the successor depends on a condition that is specified in the program. The output edges of the alternative execution nodes are weighted with the probabilities of execution of the corresponding successors. The compiler derives those probabilities from the edge execution frequencies. The weights of the other edges are set to one. Figure 5.2 shows a sample CCG.

Each node in a CCG is weighted. The weight of a node is defined by a vector of n values; $W_{node} = \{V_1, V_2, .., V_n\}$. Each V_i can be another vector or a single

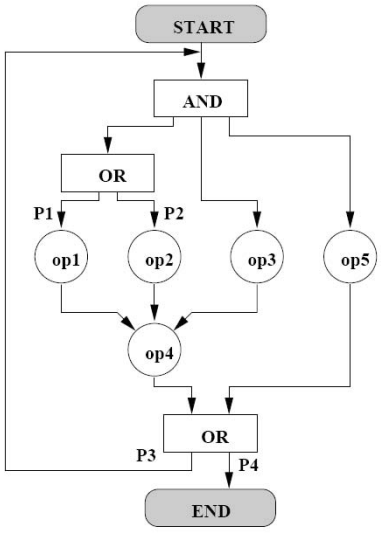

Figure 5.2: A Sample CCG

value which shows a specific characteristic of the implementations of a task. The distance to next call to the same operation and the frequency of the call of the same operation are among the parameters that the scheduler uses to perform the task mapping. The weights of the CCG can be calculated off line thereby reducing the runtime overhead.

The probabilities of the OR nodes and some of the values in the weight vector of the node might be dependent on the compile time profiling input dataset. This means that the CCG is not accurate at runtime because another input dataset might be used at runtime. However, we use the CCG as a heuristic and it does not need to be one hundred percent accurate, though a more accurate heuristic results in better performance.

The algorithm to calculate the distance to the next call for each node (task) is listed in the Listing 5.1. This is in fact a Breadth-first search algorithm. However, it is extended to keep the level for each node. The level represents the number of edges from the root of the tree to the node in the execution order. A Breadth-first algorithm traverses the graph level by level. The parameter level is used to limit the traverse depth. This is discussed later in this section.

Listing 5.1: Distance to the Next Call in a Single CCG

```
DistanceToNextCall (input: Node node, CCG ccg, Integer level) :
   return Integer distance

1- Begin
2- node.level=0;
3- Enqueue node;
4- While the queue is not empty do
        4-1- Dequeue a node n;
        4-2- Mark n;
        4-3- If (n.level > level), Return level+1;
        4-4- if (n.operationName = node.operationName, Return n
          .level;
        4-5- For all unmarked children of the node
                4-5-1- child.level=n.level+1;
                4-5-2- enqueue the child node;
        End for
End while
6- End
```

The algorithm presented in the Listing 5.2 is similar to the algorithm in Listing 5.1. This algorithm counts the frequency of the calls for each operation from each node. It should be noted that CCG is a directed graph. This algorithm also used the parameter level to limit the traverse depth as will be explained later.

For avoiding ambiguity, we only traverse the node with the highest probability in the alternative execution nodes (OR nodes) in the CCG traversal algorithms listed in Listings 5.1 and 5.2. Furthermore, we traverse the loop body only two times when encountering a loop. This is not shown in the algorithms listed in 5.1 and 5.2 in order to make them more readable. To do this, in the loop in line 4-5 in Listing 5.1, we check the type of the node. If it is an OR node, we only enqueue the child connected via an existing edge with the highest weight in line 4-5-2. In case of loops, marking mechanism should be changed and it is a little bit more complicated. In fact, a loop begins (or ends) with an OR node. When passing an OR node, we reduce the probability of the taken branch in such a way that allows only one more pass through that branch.

Listing 5.2: Frequency of Calls from Current Node in a Single CCG

```
FrequencyInFuture (input: Node node, CCG ccg, Integer level) :
  return Integer Frequency

1- Begin
2- node.level=0; count=0;
3- Enqueue node;
4- While the queue is not empty do
        4-1- Dequeue a node n;
        4-2- Mark n;
        4-3- If (n.level > level), Return count;
        4-4- if (n.operationName = node.operationName, count++;
        4-5- For all unmarked children of the node
                4-5-1- child.level=n.level+1;
                4-5-2- enqueue the child node;
        End for
End while
6- End
```

One of the main issues in calculating the distance and frequency is a concept we call it the *nearness degree*. Looking at the distance is good in general. However, we are not interested to look very far as that might have a negative influence on the scheduling decisions. For example, an operation might be called several times but that is not in the near future. If we look very far, it means that this operation should have a very high priority and it should be kept configured. However, in the near future, this particular operation is not used and it is wise to remove it and make space for the other operations.

To calculate the *nearness degree*, a maximum traverse depth is specified to restrict the level of movement in the CCG in the traversal algorithms listed in the Listings 5.1 and 5.2. This limitation also has a positive impact on the time overhead of the algorithm. We performed experiments to identify the maximum traverse depth. Figure 5.3 shows the total execution time of the applications by increasing the traverse depth for the different workloads. More details about our experimental setup can be found in Section 5.8. As it is shown in Figure 5.3, by increasing the traverse depth, the execution time is approaching to a fixed value.

At the beginning of each diagram, the execution time varies a lot. After around traverse depth of 10, the execution time is getting more flat. Therefore, we set the *nearness degree* to 10. This means that in all of our traversal algorithms we traverse at most 10 levels of CCG from the current execution point.

As we said before, the *nearness degree* can influence the time complexity of the traversal algorithms. In the worst case, when the GCC is full (i.e. all the nodes have two children), we have to traverse 1024 nodes with the *nearness degree* of 10. However most of the times, the CCG is not full and most nodes have only one child. In that case, we traverse much fewer nodes than 1024. However, this number is very dependent on the workload. This means that if we want to use a different workload in our system, we have to change the *nearness degree* accordingly. This is a design parameter and should be identified at the system design time. If the designer can not provide this number, the algorithms traverse the tree to the end and because this is an off line stage, it does not influence the applications' execution time at runtime.

It may seem that this replacement algorithm is similar to the page replacement algorithms for virtual memory managements. For example, recency and frequency of use are the most dominant decision factors for virtual memory page replacement algorithms. However, there are some major differences here; firstly, the virtual memory replacement algorithms replace at least one page by a new page when the memory is full. Nevertheless, here, evicting no kernel is also an option. Secondly, the cost of the replacement in virtual memory is fixed and has no effect on the decision, yet here every kernel has its own configuration latency. Thirdly, most of the algorithms for virtual memory are based on the temporal and spatial locality in the reference pattern of the main memory. However, in our case, the spatial and temporal locality of references is not yet proven or accepted as a general principle and needs more investigation. Lastly, a virtual memory page is relocatable and can be placed anywhere in the memory, nonetheless, this is not true for the hardware kernels. In our algorithms, we use the CCG to look at the future in contrast to the page replacement algorithms which mostly look at the pas behaviour. Furthermore, we combine the execution behaviour with another metrics from the kernel characteristics (i.e. the amount

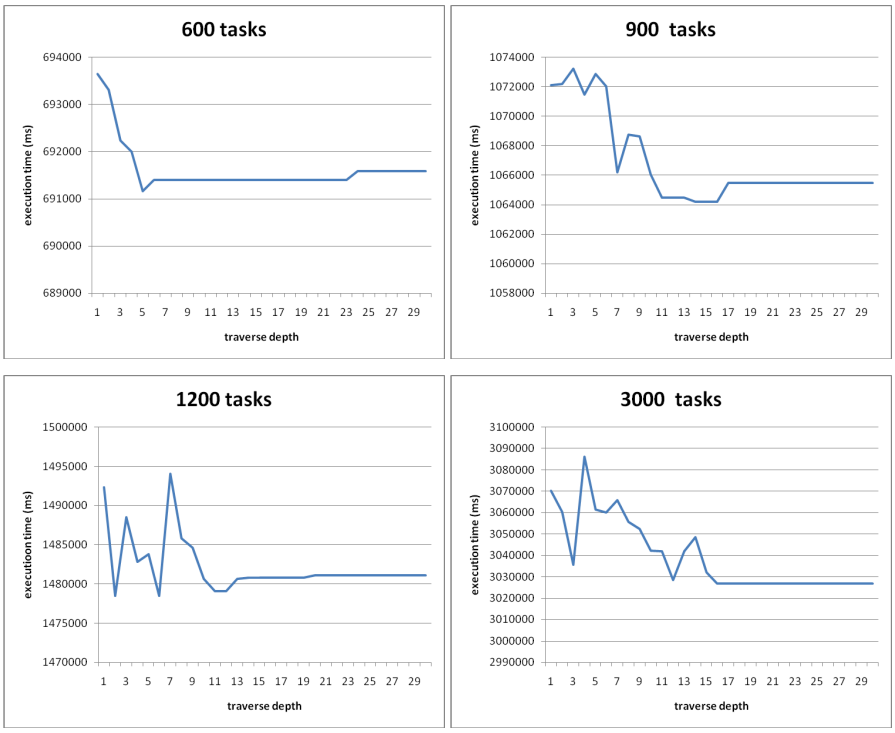

Figure 5.3: Execution Time Reduces when Travel Depth Increases

of time improvement).

The runtime scheduling procedure is a two level mechanism. The first phase is a normal scheduling policy which is performed by the operating system (e.g. a normal round robin). At the first level of scheduling, the tasks are being scheduled to run on the general-purpose processor. Apart from that, there are some points that the system needs to decide to use the reconfigurable coprocessor for compute intensive tasks. These points are either the SET or EXECUTE APIs or the internal events from the runtime profiler.

There are a few possibilities at each scheduling points. The first choice is to do nothing and continue the execution on the general-purpose processor. The other option is to choose one of the hardware implementations of the kernel from the library, configure it on the FPGA, and start hardware execution. It

should be noted that there might be other conflicting tasks already configured on the FPGA and some of them might be in the execution status (busy). In the following, we explain a few algorithms by which we address these conflicts. Each of the algorithms presented here used a different parameter to make the scheduling decisions.

5.4 Longest Distance in the Future

The Longest Distance in the Future (LDiF) algorithm replaces the tasks which will be used furthest in future based on the provided CCG. That is because such a task has the least chance of being accessed soon and can be safely replace by a more important task.

In LDiF algorithm, at each scheduling point, we define the distance to the next call for task T as follows. For application i, let d_i be the distance to the next call. This represents the number of task calls between the current execution point and the next call to the task T, in the breadth first traverse of the CCG. We use the minimum value of d_i in all CCGs as the distance to the next call for the task T.

Let us assume we are at a certain scheduling point. The scheduler is going to decide about the task T. There are n different hardware implementations $(I_1, I_2, \ldots, I_n)$ matching the task Tin the library. First, the scheduler must ensure that in the physical location of each I_i on the FPGA, there is no other busy task configured. Afterwards, it has to check if any of I_i is already configured and is in ready status. This means there is no need for the reconfiguration and the hardware execution can start right away. Subsequently, the scheduler must choose either to replace one or more of the currently configured tasks on the FPGA with one of the I_i or to continue the T's execution in software. This can be done using the replacement policy. There is a chance that no replacement takes place because it might be more efficient to keep the currently configured tasks. In this case, T runs in software.

Listing 5.3 presents LDiF algorithm. This algorithm assumes that the implementation list is sorted based on the configuration latency plus the execution time (total execution time). This assumption is because we are optimizing the execution time. As we mentioned before, other optimization can be also applied such as selecting the smaller task in size or the task with less energy consumption and so on. The replacement decision is being taken in step 6-1. In the listing 5.3, we used the distance to the next call as our decision parameter.

Listing 5.3: LDiF Scheduling Algorithm

```
LDiF (input: Implementation [] im_list) : return Implementation
   IM

//assumes im_list is sorted based on the configuration_latency
  + execution_time
//DTNC(Operation op) returns the distance to the next call of
  operation (It uses the DistanceToNextCall() function
  explained in the previous section for each CCG and returns
  the minimum values of them)

1- Scheduling begins
2- If im_list is empty return FAIL;
3- Remove the first IM from im_list;
4- Assume ToBeEvicted_list is an empty list;
5- Add all the operations, which are already configured and
  have overlaps with IM physical location to the
  ToBeEvicted_list;
6- foreach I' in the ToBeEvicted_list
        6-1- if DTNC(I'.op) < DTNC(IM.op) goto 2;
End for
7- Return IM;
8- Scheduling ends
```

We should consider the CCGs of all the running applications for calculating the distance to the next call. The algorithm listed in Listing 5.1 in Section 5.3.2, only gives the distance to the next call for each node in a single CCG. However, this is not enough due to the following reasons. First, the operations might be shared between applications and, therefore, applications can influence each

other. As a result, other CCGs should also be considered. Second, the distance should be calculated from the current execution point in each CCG. Whereas, Listing 5.1 provides the distance from the last call to the next call of the same task and not from the current execution point to the next call.

To handle the first issue, we consider the minimum value of distances to the next call in all CCGs as the distance to the next call. To handle the second issue, we define a current execution level variable for each CCG that holds the level of the current execution point. We update this value when the control transfers to the next level (e.g. a new task call happens). On the other hand, we also keep the level of the next call and the node name of the next call besides keeping the distance to the next call for each node. This way, to calculate the distance to the next call from the current execution point for each task, we only need to subtract the current execution point level from the next call's level. We should mention that, we keep a list of all the operations inside a CCG and the distance to the next call and frequency of calls in the future for member of that list. We update these values during the program execution.

The overhead of the algorithm in Listing 5.3 is dependent on the number of implementations for each task because a number of constraints have to be checked for each of them. The distance to the next call in future is calculated offline for each application and at runtime; we only need to find the minimum value of the distances for all the applications. Therefore, this will not impose a big overhead.

5.5 Least Frequency in the Future

Least Frequency in the Future (LFiF) replaces the task that will be accessed less frequent than the others in future. Replacing such a task might lead to less number of reconfiguration and therefore, to reduce the overhead involved in each reconfiguration.

Considering LFiF algorithm, for a specific task *T*, we define the frequency of

the calls in future for an application as the total number of calls to the task T in all the successors of the current execution point. Afterwards, the frequency of the calls for the task T in the whole system is the sum of the frequencies of all applications.

Listing 5.4 shows the LFiF scheduling. This algorithm works similar to the algorithm in Listing 5.1. However, in the line 6-1, it checks for the frequency of calls.

Listing 5.4: LFiF Scheduling Algorithm

```
LFiF (input: Implementation [] im_list) : return Implementation
   IM

//assumes im_list is sorted based on the configuration_latency
  + execution_time
//FqiF(Operation op) returns the expected number of calls in
  the future for operation (It uses the FrequencyInFuture()
  algorithm for each CCG and returns the sum of them)

1- Scheduling begins
2- If im_list is empty return FAIL;
3- Remove the first IM from im_list;
4- Assume ToBeEvicted_list is an empty list;
5- Add all the operations, which are already configured and
  have overlaps with IM physical location to the
  ToBeEvicted_list;
6- foreach I' in the ToBeEvicted_list
        6-1- if FqiF(I'.op) > FqiF(IM.op) goto 2;
End for
7- Return IM;
8- Scheduling ends
```

5.6 Least Frequency in the Past

Listing 5.5 presents the algorithm for the Least Frequency in the Past (LFiP). In this algorithm, we use the information from the past to predict the future.

This information is provided by the runtime profiler, which is discussed in the Chapter 7. We also have something similar to the *nearness degree* for the past information. This means that we only look at the recent behaviour of the applications. In contrast to the future that the *nearness degree* is based on the number of calls, we look only a certain time back in the past. For example, we count the number of call to a particular task only in the past 20 seconds.

Listing 5.5: LFiP Scheduling Algorithm

```
LFiP (input: Implementation [] im_list) : return Implementation
   IM

//assumes im_list is sorted based on the configuration_latency
  + execution_time
//FqiP(Operation op) returns the number of calls in the Past
  for the operation (It reads this information from the
  profiler frame which will be discussed in the chapter 7)

1- Scheduling begins
2- If im_list is empty return FAIL;
3- Remove the first IM from im_list;
4- Assume ToBeEvicted_list is an empty list;
5- Add all the operations, which are already configured and
  have overlaps with IM physical location to the
  ToBeEvicted_list;
6- foreach I' in the ToBeEvicted_list
        6-1- if FqiP(I'.op) > FqiP(IM.op) goto 2;
End for
7- Return IM;
8- Scheduling ends
```

5.7 Expected Time Improvement

The Expected Time Improvement (ExTI) algorithm tries to estimate the possibility of the acceleration by a task in the future. Considering ExTI algorithm, a time-improvement heuristic is defined by two parameters: the first is the

reduction in task execution time, which is obtained by comparing the hardware execution time of a task on FPGA and the software execution time on the general-purpose processor and considering the task configuration time. The second parameter is the distance to the next call that is the number of task calls between the current execution point and the next call to the same task. This parameter is discussed in Section 5.4. We actually combined these two parameters to come up a better heuristic.

Listing 5.6 shows this algorithm. This algorithm is very similar to the LDiF. It only adds one further improvement to that algorithm and that is checking how good one implementation compare to the others is. The LDiF always takes the fastest implementation if possible. However, the ExTI takes the fastest implementation considering the implementation's configuration time as well. For example, a task might have two implementation with the execution times of 10 and 12 milliseconds and configuration time of 8 and 5 milliseconds respectively. In this example, LDiF takes the first implementation as that one is the fastest. However, the ExTI takes the second one. Because, the second one is a better option considering the configuration latency and execution time together. Nevertheless, if an operation is being executed several times after configuration, the LDiF shows a better performance.

Listing 5.6: ExTI Scheduling Algorithm

```
ExTI (input: Implementation [] im_list) : return Implementation
   IM

//assumes im_list is sorted based on the configuration_latency
  + execution_time
//TImpr(implementation IM) returns the time improvement for IM

1- Scheduling begins
2- If im_list is empty return FAIL;
3- Remove the first IM from im_list;
4- Assume ToBeEvicted_list is an empty list;
5- Add all the operations, which are already configured and
  have overlaps with IM physical location to the
```

```
  ToBeEvicted_list;
6- foreach I' in the ToBeEvicted_list
        6-1- if TImpr(I') > TImpr(IM) goto 2;
End for
7- Return IM;
8- Scheduling ends
```

5.8 Evaluation

In this section, we present the evaluation results of the proposed algorithms. We first describe the workload we used for the evaluation and the application scenario. Then, we present the results of the experiments.

5.8.1 Workload for Evaluation

The workload for evaluation is obtained from an interactive multimedia Internet based testing application. This application can serve a large number of applicants simultaneously. We have identified eight multimedia applications that consume most of the server computation time through profiling the exam server. The identified kernels are described in the following paragraphs.

Jpeg-Encoder and Jpeg-Decoder: Jpeg is a standardized compression method for the images. Jpeg is a lossy compression method, meaning that the output image is not exactly identical to the input image. Two kernels are derived from the Jpeg; Jpeg-Encoder does image compression and Jpeg-Decoder does decompression.

Epic-Encoder and Epic-Decoder: These compression algorithms are based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. Extremely fast decoding of epic makes it suitable to be employed for portable embedded systems.

Mpeg2-Encoder and Mpeg2-Decoder: Mpeg2 is the standard for digital video transmission.

Table 5.1: Kernel Specifications (ms)

Benchmarks	Software execution time	Implementation one		Implementation two		Implementation three	
		Conf. time	Exec. time	Conf. time	Exec. time	Conf. time	Exec. time
Epic-Decoder	19.87	11.04	5.98	6.39	8.53	5.82	8.56
Epic-Coder	11.87	4.87	3.99	2.66	4.93	2.49	5.22
Mpeg2-Decoder	77.35	5.83	2.01	4.11	2.34	3.64	2.43
Mpeg2-Ecoder	10.39	7.51	1.19	5.68	1.82	4.87	1.94
G721	42.42	10.6	3.99	6.39	4.23	5.82	4.64
Jpeg-Decoder	68.39	11.72	7.56	9.13	8.11	8.72	8.63
Jpeg-Encoder	169.33	13.78	29.25	11.49	31.98	10.98	35.23
Pegwit	166.06	12.35	34.56	6.47	32.35	5.88	36.34

G.721: is a standard for speech codec that uses the Adaptive Differential Pulse Code Modulation (ADPCM) method and provides toll quality audio at 32 Kbps.

Pegwit: is a program for public key encryption and authentication. It uses an elliptic curve over GF(2255), SHA1 for hashing, and the symmetric square block cipher.

In order to implement the profiled applications, we use the C code of the programs in the mediabench [75]. For each kernel, we have three different implementations. After obtaining the bit stream of the hardware implementations, their configuration times are calculated as follows [76].

$$configurationtime = sizeofbitstream/FPGAclockfrequency \quad (5.1)$$

The experiment workload is obtained from an interactive multimedia Internet based testing application [55]. The timing is based on the software or hardware execution of the tasks on the Xilinx Virtex-5 FPGA Family. The software execution time is on a hard PPC. The configuration time is the time for partial reconfiguration of each task. The numbers are generated using the REC-BENCH tool [55]. These numbers are used as the input to our simulator. All the numbers are in milliseconds.

Table 5.1 lists the information about these kernels and their implementations.

Table 5.2: Workload Set-ups

	Number of applications	*Number of tasks*
Set-up 1	12	858
Set-up 2	24	1660
Set-up 3	36	2419
Set-up 4	48	3206
Set-up 5	60	4097

The software execution time of the kernels is computed when running on the GPP. We can compute the proportion of the acceleration of hardware execution to the software execution from this information.

In general, each application in the workload is a type of multimedia test such as reading, listening, speaking, or writing. Each application includes a number of tasks. Therefore, the application-mix depends on the ordering of the multimedia tests and the number of tasks in the application. The system simulates multiple executions of the applications in the multimedia tests. For example, an application workload can have five reading tests, six listening tests, five speaking tests and two writing tests. The start times of the applications are different. The operation reuse in the application depends on the workload and the similarity between the tasks in the application workload.

We obtained the workload by running the examination server in five different set-ups (12 applicants (858 tasks), 24 applicants (1660 tasks), 36 applicants (2419 tasks), 48 applicants (3206 tasks), and 60 applicants (4097tasks)). The server's operations have been logged and the workload is extracted from these logs.

The log includes the name of the task, the execution time of the task (software-only), and the arrival time for each task. The workload is generated per applicant per set-up. Therefore, for each set-up, we exactly know how many applicants there are (number running process in the server), how and when the kernels have been called. The information about the workload set-ups is shown in Table 5.2.

Table 5.3: The Tasks Execution Time and Number of Executed Tasks on RPs in each Set-up

Scheduling algorithms		Set-up 1	Set-up 2	Set-up 3	Set-up 4	Set-up 5
Software only	Execution Time (ms)	135654.08	260508.60	381329.44	501860.74	641478.23
	Number of tasks executed on the RPs	0	0	0	0	0
LFiP	Execution Time (ms)	126883.12	226219.53	248866.52	334082.77	618196.87
	Number of tasks executed on the RPs	194	656	1092	1388	622
LFiF	Execution Time (ms)	91230.78	176467.80	267377.39	360294.81	455492.04
	Number of tasks executed on the RPs	546	978	1380	1718	2276
LDiF	Execution Time (ms)	71727.74	143546.82	216114.94	290748.26	363356.18
	Number of tasks executed on the RPs	704	1332	1836	2135	3212
ExTI	Execution Time (ms)	68648.15	137524.57	202793.31	241565.90	342166.00
	Number of tasks executed on the RPs	742	1352	1970	2564	3412

5.8.2 Evaluation Results

The simulations for a number of test takers are performed in order to evaluate the performance of the proposed scheduling algorithm. We extended the CPUSS CPU scheduling framework [77] for evaluation. We performed simulation mainly because we did not have access to the hardware platform which supports partial reconfiguration in the way that we need. We really wanted a platform in which we can partially reconfigure the FPGA for as many tasks as the scheduler decides. To get an impression of the FPGA size, in the worst case, the FPGA can accommodate only one large size task based on our measures and in the average case it can accommodate 4 tasks. We assume that the tasks are rectangular shaped.

We compared the four algorithm (LDiF, LFiF, LFiP and ExTI) presented in the previous sections. We have different set-ups in our validation experiments where the number of participants and thus the number of tasks vary. Each cell of Table 5.3 contains the total execution time of the application and the number of executed tasks on RPs in each scheduling algorithm. As illustrated in the table, all algorithms have shorter execution times than software-only execution of the tasks which is an obvious result of using reconfigurable accelerators.

The numbers in Table 5.3 certify that the LDiF and ExTI are performing better than the other algorithms. The reason that ExTI outperforms the LDiF is that ExTI, besides looking into the future, considers the reconfiguration latency for decision-making. However, as it is mentioned before, the LDiF outperforms ExTI in the scenarios in which a task will be executed several times with short intervals.

Additionally, the results show that the frequency of calls does not perform as good as the distance to the next call. The reason is that although a kernel might be used several times in the future, it does not mean that those uses are in the near future. For example, a task might be accessed 10 times during 10 seconds from which one access is in the first one second and the rest are in the last 2 seconds. Using highest frequency algorithm suggests keeping this task on the FPGA, however seven seconds between the calls might be enough time to remove this task from the FPGA and bring it back later.

The reason for better performance of future frequency over the past frequency is also obvious. Looking at the past frequency is a kind of heuristic to predict the future and it is not always accurate. Using the CCG in the future frequency algorithm, we have a better heuristic with more accuracy.

Figure 5.4 presents the percentage of the tasks, which have been executed on the RPs. This Figure shows that the algorithms with better performance were able to execute more tasks on the RPs. Therefore, they showed a better acceleration.

Figure 5.5 shows the obtained speed up. ExTI showed a speed up of two that is the best comparing to the other algorithms. As shown in Figure 5.5, the speedup

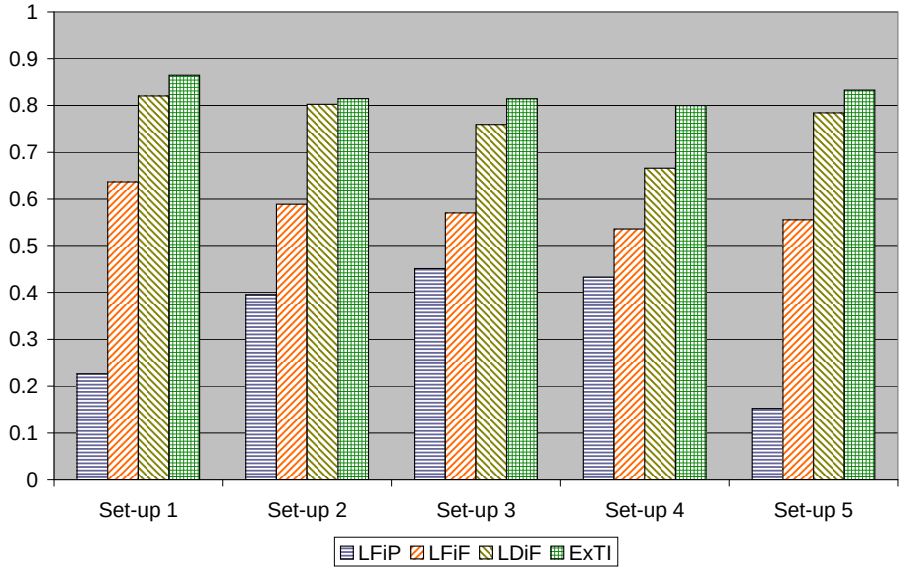

Figure 5.4: Percentage of the Tasks Executed on RPs

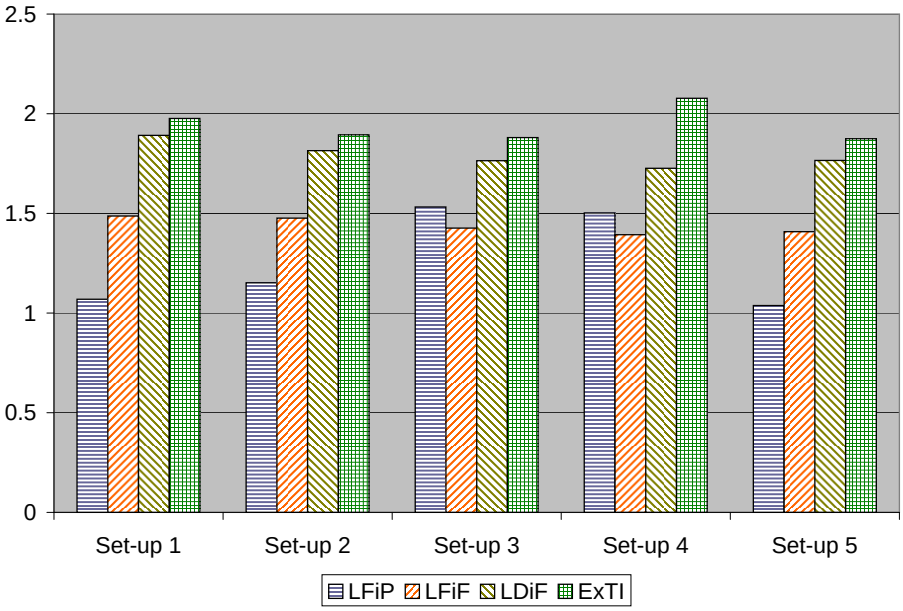

Figure 5.5: The Obtained Speedup

in all the setups almost behaves the same for all the applications.

One might argue that the obtained speedup of around 2 is not as impressive as it should be. There are some reasons behind this number. First of all as we used the processor coprocessor paradigm with only one GPP, the GPP load is a bottleneck in our system. Most parts of the code which are not compute intensive are executed on the GPP and we can not exploit any parallelism in those parts. Furthermore, we use the library to load the kernels binary from which increases the overhead of the system. Operating system overhead (for example the system calls overhead) is also another source of system overhead. However, we believe that the imposed overhead is actually the price we pay to achieve a more flexible system. As a future work, we plane to experiment the processor coprocessor paradigm with more than one GPP. Employing more GPPs can help to exploit the parallelism in none compute intensive parts of the code.

5.9 Conclusion

In this chapter, we presented the scheduler as a part of the proposed runtime system. We showed how we use a combination of design time and runtime scheduling in order to optimize the system performance. At the design time, we use the compiler to perform static task scheduling assuming single thread of execution. Then at the runtime, the runtime scheduler performs the actual task scheduling. The runtime scheduler can use the information provided by the runtime profiler. It can also use the information transferred from design time in the form of a CCG.

We also presented a number of scheduling policies performance of which is shown by experiments. Based on the obtained results, looking into the future using CCG can improve the system performance quite considerably. We used three different parameters to base our scheduling decisions on them. We showed that Expected Time Improvement and Longest Distance in the Future are very good heuristics for the scheduling. In fact, combining the future behaviour

with the amount of time we can save by a faster kernel is shown to be the best heuristic.

6

Fuzzy Real-time Scheduling

One of the main uses of heterogeneous multi-core systems is the time-critical applications. In such systems, applications need real-time guarantees for execution times. We provide scheduling mechanisms for soft real-time multi-core systems in this chapter. In the presented approaches, the decision parameters are modelled as fuzzy variables and using a Fuzzy Inference System (FIS) the scheduling priorities are determined.

It should be mentioned that in contrast to the scheduling algorithms presented in the previous chapter, the scheduling algorithms that are presented in this chapter are targeting periodic real-time workloads. The tasks in such workloads have to be executed in a predefined amount of time that is their deadlines. We assume our system as a soft real-time system. If a task misses its deadline, the scheduling algorithm will not execute that task anymore and such a task is considered as a failed execution task. In a hard real-time system, if a task misses its deadline, the whole system fails.

The scheduling algorithms presented in the previous chapter execute all the tasks since they do not have any deadlines. Therefore, the obtained results of the experiments from the algorithms here are not comparable to the results of the algorithms of the previous chapter. For example, the overall execution time of the workloads with real-time constraints is smaller than the same workloads without real-time constrains since some of the tasks never execute in the real-

time case.

Although in this chapter we considered the cores to be identical, the main concept can be employed for the heterogeneous cores as well. For example in the algorithms based on the laxity, the laxity is the same for all the cores as they are identical. However, when the cores are not identical, the laxity of each task should be calculated for each core separately. This is because each core has a different computing power and as a result the task execution time on that core is different from the others. This means that a task might be considered to miss its deadline on one core and at the same time to be still feasible for execution on another core (probably a faster core).

6.1 Introduction

Real-time systems are vital to industrialized infrastructure such as command and control, process control, flight control, space shuttle avionics, air traffic control systems and also mission critical computations [78]. In all the cases, time has an essential role and having the right answer too late is as bad as not having it at all.

In the literature, these systems have been defined as "systems in which the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced" [79]. Such a system must react to the requests within a fixed amount of time which is called deadline.

In general, real-time systems can be categorized into two important groups: hard real-time systems and soft real-time systems. In hard real-time systems, meeting all deadlines is obligatory. While in soft real-time systems, missing some deadlines is tolerable.

In both cases, when a new task arrives, the scheduler is to schedule it in such a way that guaranties the deadline. Scheduling involves allocation of resources and time to tasks in such a way that certain performance requirements are met.

These tasks can be classified as periodic or aperiodic. A periodic task is a kind of

task that occurs at regular intervals, and an aperiodic task occurs unpredictably. The length of the time interval between the arrivals of two consecutive requests in a periodic task is called period.

Another aspect of scheduling theory is to decide whether the currently executing task should be allowed to continue or it has had enough CPU time for the moment and it should be suspended. A preemptive scheduler can suspend the execution of current executing request in favour of a higher priority request. However, a nonpreemptive scheduler executes the currently running task to completion before selecting another request to be executed. A major problem that arises in preemptive systems is the context-switching overhead [80].

We need to use a deadline aware scheduling algorithm in a multi-core system in order to meet all the deadlines. Such a scheduling algorithm decides to execute each task on which core based on the performance requirements and timing constrains.

Multi-core scheduling techniques in real-time systems fall into two general categories: partitioning and global scheduling [81]. Under partitioning, each core schedules tasks independently from a local ready queue. Each task is assigned to a particular core and is only scheduled on that core. In contrast, all the ready tasks are stored in a single queue under global scheduling. A single system-wide priority space is assumed: the highest priority task is selected to execute whenever the scheduler is invoked. Partitioning is the favoured approach because it has been proved efficient and reasonably effective when popular single processor scheduling algorithms such as Earliest Deadline First (EDF) and Rate Monotonic (RM) are used [82]. However, finding an optimal partitioned schedule is an NP-hard problem.

In the global scheduling scheme, processors repeatedly execute the highest priority tasks available for execution. It has been shown that using this approach with common priority assignment schemes such as Rate Monotonic (RM) and Earliest Deadline First (EDF) can give rise to an arbitrarily low processor utilization [82]. In this approach, a task can migrate from one processor to another during execution.

Although, these algorithms focus on timing constraints but there are other implicit constraints in the environment, such as uncertainty and lack of complete knowledge about the environment, dynamicity in the world, bounded validity time of information and other resource constraints. In real world situations, it would often be more realistic to find viable compromises between these parameters. For many problems, it makes sense to partially satisfy the objectives. The satisfaction degree can then be used as a parameter for making a decision. One good method to achieve this is the modelling of these parameters through fuzzy logic. The same approach is also applied on single processor real-time scheduling in [83–85].

6.2 Fuzzy Inference System

Fuzzy logic is an extension of Boolean logic dealing with the concept of partial truth, which denotes the extent to which a proposition is true. Whereas, classical logic holds that everything can be expressed in binary terms (0 or 1, black or white, yes or no), fuzzy logic replaces Boolean truth-values with a degree of truth. The degree of truth is often employed to capture the imprecise modes of reasoning that play an essential role in the human ability to make decisions in an environment of uncertainty and imprecision.

Fuzzy Inference Systems (FIS) are conceptually very simple. They consist of an input, a processing, and an output stage. The input stage maps the inputs (e.g. frequency of reference and recency of reference) to the appropriate membership functions and truth-values. The processing stage invokes each appropriate rule and generates a corresponding result. It then combines the results. Finally, the output stage converts the combined result back into a specific output value [86].

The membership function of a fuzzy set corresponds to the indicator function of the classical sets. A curve defines how each point in the input space is mapped to a membership value or a degree of truth between zero and one. The most common shape of a membership function is triangular, although trapezoidal and bell curves are also used. The input space is sometimes referred to as the

universe of discourse.

As discussed earlier in this section, the processing stage, which is called inference engine, is based on a collection of logic rules in the form of IF-THEN statements. Typical fuzzy inference systems have dozens of rules. These rules are stored in a knowledge base. An example of a fuzzy IF-THEN rule is IF laxity is critical then priority is very high, which laxity and priority are linguistics variables and critical and very high are linguistics terms. Each linguistic term corresponds to membership function.

An inference engine tries to process the given inputs and produce an output by consulting an existing knowledge base. The five steps toward a fuzzy inference are fuzzifying inputs, applying fuzzy operators, applying implication methods, aggregating all outputs, defuzzifying outputs.

Fuzzifying the inputs is the act of determining the degree to which they belong to each of the appropriate fuzzy sets via membership functions. Once the inputs have been fuzzified, the degree to which each part of the antecedent has been satisfied for each rule is known. When the antecedent of a given rule has more than one part, the fuzzy operator is applied to obtain one value that represents the result of the antecedent for that rule. The implication function then modifies that output fuzzy set to the degree specified by the antecedent. Since decisions are based on the testing of all of the rules in an FIS, the results from each rule must be combined in order to make a decision. Aggregation is the process by which the fuzzy sets that represent the outputs of each rule are combined into a single fuzzy set. The input for the defuzzification process is the aggregated output fuzzy set and the output is a single value. This can be summarized as follows: mapping input characteristics to input membership functions, input membership function to rules, rules to a set of output characteristics, output characteristics to output membership functions, and the output membership function to a single-valued output.

There are two common inference processes. The first is called Mamdani's fuzzy inference method proposed in 1975 by Ebrahim Mamdani [87] and the other is TakagiSugeno-Kang, or simply Sugeno, method of fuzzy inference introduced

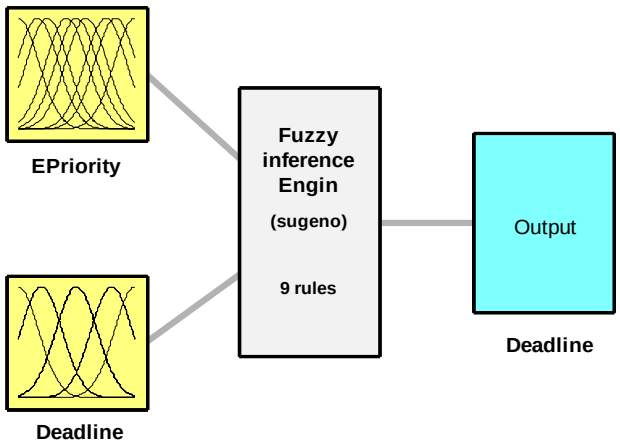

Figure 6.1: Inference System Block Diagram

in 1985 [3]. These two methods are the same in many respects, such as the procedure of fuzzifying the inputs and the fuzzy operators.

The main difference between Mamdani and Sugeno is that the Sugeno output membership functions are either linear or constant but Mamdani's inference expects the output membership functions to be fuzzy sets.

Sugeno's method has three advantages. First, it is computationally efficient, which is an essential benefit to the real-time systems. Second, it works well with optimization and adaptive techniques. These adaptive techniques provide a method for the fuzzy modelling procedure to extract proper knowledge about a data set for computing the membership function parameters that best allow the associated fuzzy inference system to track the given input/output data. However, in this chapter we do not consider these techniques. The third, advantage of Sugeno type inference is that it is well-suited to the mathematical analysis.

6.3 The Proposed Fuzzy Model

The proposed model is shown in Figure 6.1. In this model, the input stage consists of two linguistic variables. The first one is an external priority which

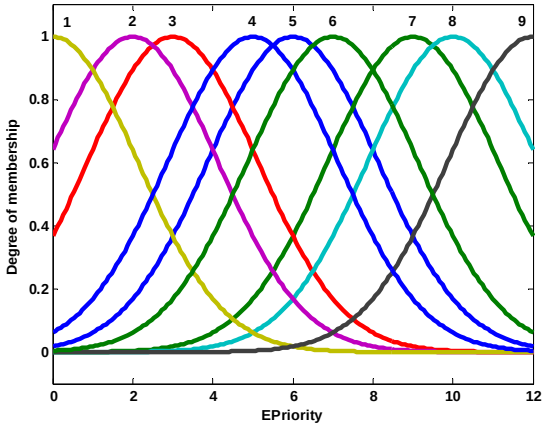

Figure 6.2: Fuzzy Sets Corresponding to the External Priority

is the priority assigned to the task from the outside world. This priority is static. One possible value can be the tasks interval, as rate monotonic algorithm does [88]. For Figure 6.1, the other input variable is the Deadline. This input can easily be replaced by laxity, wait time, or so on, for other scheduling algorithms. Each parameter may cause the system to react in a different way. The only thing that should be considered is that by changing the input variables the corresponding membership functions may be changed accordingly.

For the simulation purposes, as it is discussed later, four situations are recognized: First, by using laxity as a secondary parameter and, second, by replacing the laxity parameter with deadline and both of them are considered in partitioned and also global scheme. In fact, four algorithms are suggested: Fuzzy Global EDF (FGEDF), Fuzzy Global MLF (FGMLF), Fuzzy Partitioned EDF (FPEDF), and Fuzzy Partitioned MLF (FPMLF).

The output of the system is execution priority that determines which task has to be executed first. The input variables mapped into the fuzzy sets as illustrated in Figures 6.2 and 6.3. An expert determines the shape of the membership function for each linguistic term. These shapes can be tuned to optimize the obtained results. Each diagram in Figures 6.2 and 6.3 represents a membership function. The y-axis of Figures 6.2 and 6.3 is the degree of membership which

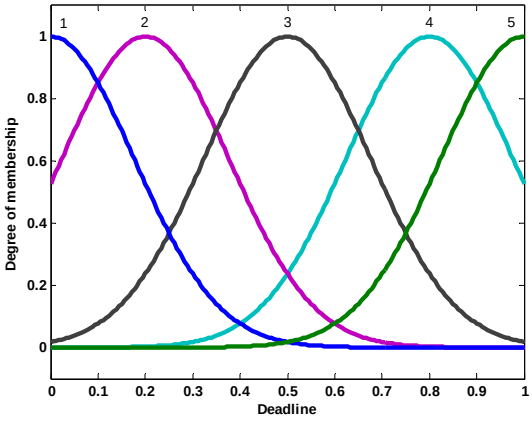

Figure 6.3: Fuzzy Sets Corresponding to Deadline

shows the degree that each values in the x-axis belongs to each membership function. The x-axis in the Figure 6.2 represents the external priorities values. The external priority is a discrete variable. However traditionally, membership functions are shown as non-discrete variables. The x-axis in the Figure 6.3 represents the normalized values (between 0 and 1) of the deadlines. As an example, deadline 0.5 belongs to the membership function 3 with the degree of 1 and belongs to the membership function 2 with the degree of 0.2.

Fuzzy rules try to combine these parameters as they are connected in the real world. Some of these rules are mentioned in the following. To make the rules more readable, we used English worlds for the membership functions. For example, the word *critical* for the deadline represents the membership function 5. Similarly, the word *sufficient* represents the membership function 3.

1. If (EPriority is high) and (deadline is critical) then (Priority is very high)
2. If (EPriority is normal) and (deadline is critical) then (Priority is high)
3. If (EPriority is very low) and (deadline is critical) then (Priority is normal)
4. If (EPriority is high) and (deadline is sufficient) then (Priority is normal)
5. If (EPriority is very low) and (deadline is sufficient) then (Priority is very low)

In fuzzy inference systems, the number of rules has a direct effect on its time complexity. So, having fewer rules may result in a better system performance. A fuzzy inference system implements a nonlinear mapping from the input space to output space. The output is determined as a weighted mean value over all the rules. A detailed discussion of the mapping can be found in [3].

6.4 The Proposed Algorithms

The FGEDF algorithm is shown in Listing 6.1. FGMLF is much the same with FGEDF just by replacing the word deadline by laxity.

Listing 6.1: FGEDF Algorithm

```
Loop // System is running for ever
 For each core in the system do the followings:
  1- for each ready task T //a task that is not running
     1-1- If (T has enough time to meet its deadline)
        1-1-1- Feed its external priority and deadline into the
           inference engine.
        1-2-2- Consider the output of inference module as
          priority of task T.
     End if
  End for
  2- Execute the task with highest priority until a scheduling
    event occurs (a running task finishes, a new task arrives)
  3- Update the system states (deadlines, etc)
 End for
End loop
```

The FPEDF algorithm is shown in Listing 6.2. The difference between Listings 6.1 and 6.2 is that in the latter, we have an extra condition in line 1-1. With that condition, the algorithm checks that the task T has not been executing on another core. Therefore, it only considers the tasks that have been executing on the same core or the newly arrived tasks. FPMLF is much the same with FPEDF just by replacing the word deadline by laxity.

Listing 6.2: FPEDF Algorithm

```
Loop // System is running for ever
 For each core in the system do the followings:
  1- For each ready task T //a task that is not running
     1-1- If (T in its current execution has not been executed
       on another core) and (T has enough time to meet its
       deadline)
        1-1-1- Feed its external priority and deadline into the
           inference engine.
        1-1-2- Consider the output of inference module as
          priority of task T.
     End if
  End for
  2- Execute the task with highest priority until a scheduling
    event occurs (a running task finishes, a new task arrives)
  3- Update the system states (deadlines, etc)
 End for
End loop
```

6.5 Performance Evaluation

To evaluate the proposed algorithm, 1024 test cases with different load factors were generated. The load factor of 100 means that the system load is at the maximum. The load factors greater than 100 mean that the system is overloaded. In each test case, the number of tasks and the corresponding execution time and request interval were randomly generated. In addition, each task has been assigned a priority according to the rate monotonic principle (tasks with shorter request interval are given higher priorities). The behaviour of all the four algorithms (FGEDF, FGMLF, FPEDF and FPMLF) is compared with each other. The following performance metrics are used to compare the algorithms. Number of missed deadlines is an influential metric in scheduling algorithms for soft real-time systems. When task preemption is allowed, another prominent metric comes into existence and that is the number of preemptions. Each of the preemptions requires the system to perform a context switch which is a time

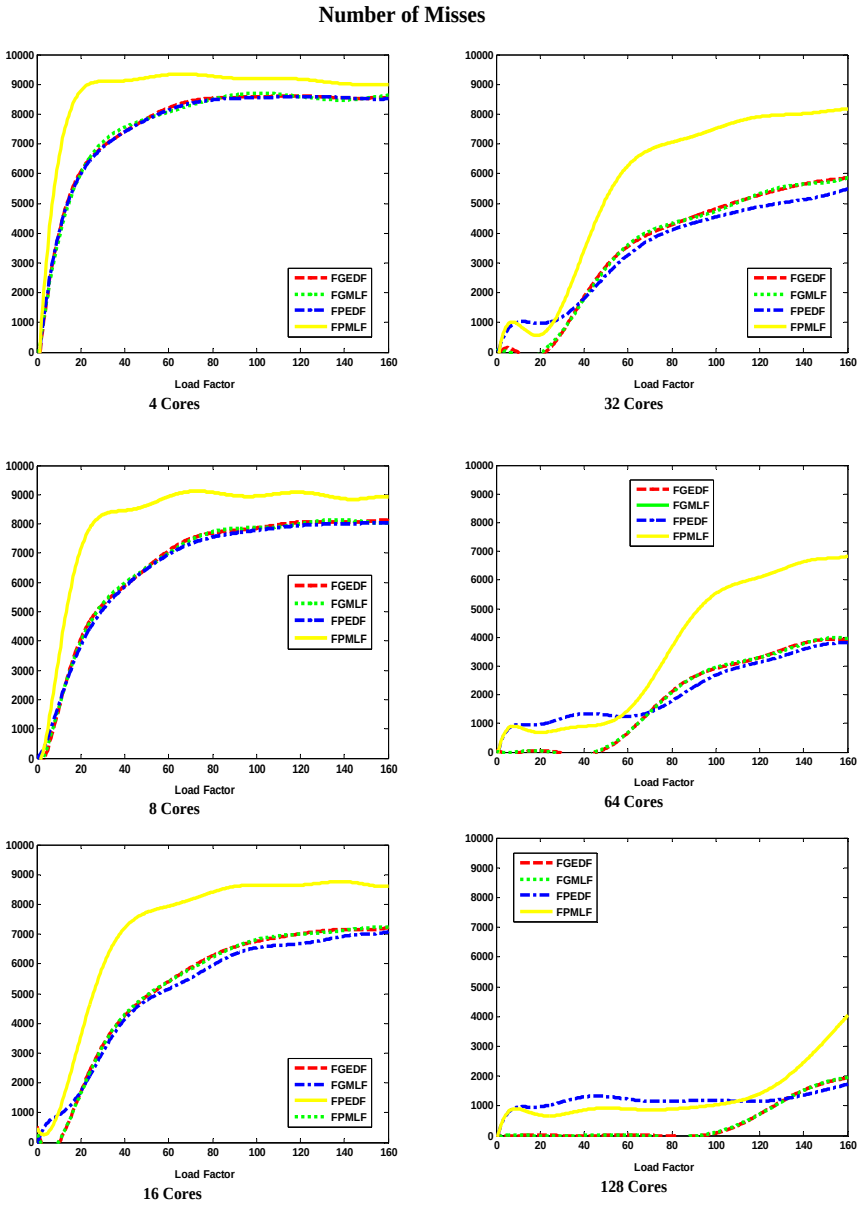

Figure 6.4: Number of Misses

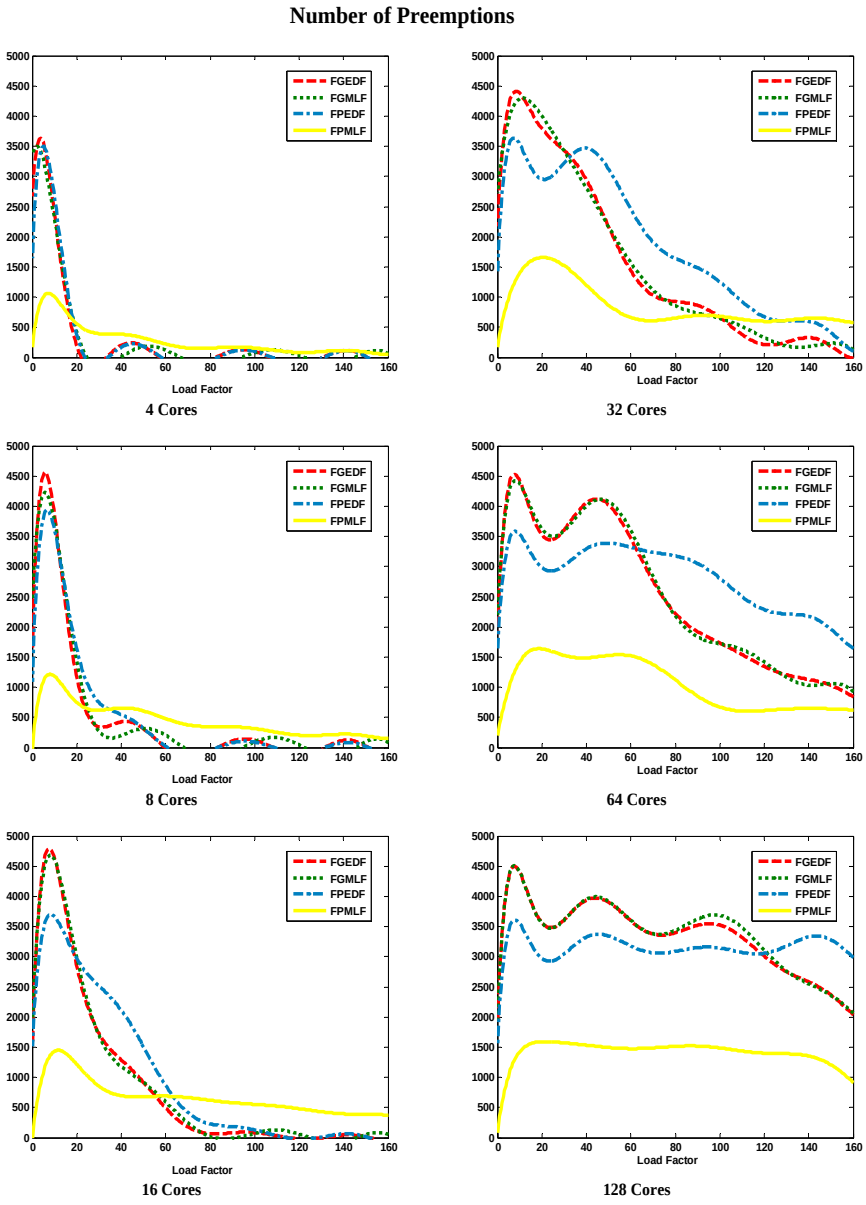

Figure 6.5: Number of Preemptions

consuming action. We performed our simulation for systems with different number of identical cores and we judge the algorithms against each other in these conditions.

As Figures 6.4 and 6.5 show, among the four algorithms FGEDF and FGMLF nearly achieve the same performance in all situations and all metrics. FPMLF performs poorly in number of misses, but its performance on the number of preemption is much better than the others especially when the number of CPUs increases.

About number of misses, as Figure 6.4 shows, the FPMLF algorithm has the most number of misses. The FPEDF algorithm has the minimum numbers of misses among the other algorithms. This figure shows that deadline is a better scheduling parameter considering the number of tasks missing their deadlines.

Comparing number of preemptions, as Figure 6.5 demonstrate, the FPMLF algorithm outperforms the others. The FPEDF algorithm reaches to a higher number of preemptions as the load factor increases. Numbers of preemptions in FGMLF are a little more than FGEDF. In this case, deadline is the better parameter.

6.6 Conclusion

This chapter has described the use of fuzzy logic in multi-core real-time scheduling. We modelled the scheduling parameters in the form of fuzzy variables and used a fuzzy inference engine to derive the scheduling priorities. We presented four different algorithms for real-time scheduling. These algorithms are based on the deadline and laxity as input parameters. Furthermore, we considered both global scheduling approach as well as the partitioning approach. As it was shown, algorithms which are based on the deadline as a fuzzy parameter outperformed the algorithms which are based on the laxity. We presented the number of misses and the number of preemption for each algorithm.

7

Runtime Profiling

In this chapter, we present the design and implementation of a runtime profiler which is responsible to produce statistics about the code running on the system. The profiler facilitates the runtime mapping process of the tasks into the reconfigurable processors. We have performed a set of experiments in order to show the overhead of our proposed profiler. The evaluation results show that the overhead imposed by the profiler is less than one percent of the total execution time and the information generated by the profiler is almost as accurate as a design time profiler such as *gprof*.

7.1 Introduction

One of the main limitations of using the heterogeneous multi-core systems is the difficulty in programming them. Usually, separate tools are used for writing software and designing hardware. Therefore, the overall design and implementation cycle is difficult and needs knowledge from both the hardware and the software. Furthermore, when moving towards multi applications, multi-tasking scenarios, it is even more difficult for the designers to deal with such systems, as the exact configuration of the system is not known at design time. These requirements necessitate the existence of a runtime system, which is responsible for operating the system and performs the resource management

[89]. The resource management of a heterogeneous multi-core system by itself is a very complicated task and is dependent on the available information for decision making [70].

One important tool required that can help with the decision making is a runtime profiler. The runtime profiler gives vital statistics about the code running on the heterogeneous multi-core system. Those statistics can then be used by the runtime system to decide which parts of the code need to be translated into hardware.

In this chapter, we present a runtime profiler, which is intended to run concurrently with the applications in the actual application execution time. Therefore, one key difference between such a profiler and traditional design time profilers such as *gprof* [90] is that it has to be very low overhead.

Design time profilers such as *gprof* are quite good and produce very useful information. However, as they might impose a considerable overhead to the application execution time, they cannot be used at runtime. Furthermore, they need expert users to operate them. On the other hand, general-purpose runtime profilers [91–93] are not as useful as they should be for employing in the proposed runtime system. This is because, the profiler has to communicate in real-time with the scheduler and other runtime system's components. As a result, the interface between them has to be very efficient.

Furthermore, there is very little related work. Among them, we can refer to the Warp processor [29] which detects kernels at runtime using a frequent loop detection profiler. The main limitations to this work are that it only profiles loops. Secondly, it deals with the addresses at physical level, while we need to deal with virtual addresses, as we want to profile all programs running in the user space. Furthermore, it only keeps counts of 16 loops at a time, while we want to keep much of all calls to the kernels in the profiled applications. DAProf [94] is also another similar work focusing only on loop profiling.

The proposed profiler profiles all the functions in the application and produces the number of calls per function and the approximate time spent in a function. It

is completely software based and runs under the Linux operating system. This makes the proposed profiler quite portable and easy to be incorporated in any runtime system.

7.2 Design Choices

The proposed runtime system is intended to completely virtualize the underlying hardware and thus relieves the program developers from the difficulties of hardware design issues. Within the proposed runtime system, the profiler communicates with the scheduler at runtime. Therefore, it needs to use data structures that allow fast writing and reading of the profiled statistics. Furthermore, it should be capable of continuously profiling the application behaviour. This is because, kernels may be swapped several times because of the limited available hardware. This is a fundamental difference between the proposed profiler and profilers used in virtual machines like Java. In such virtual machines, once a function is optimized, it no longer needs to be profiled.

Different techniques used in the literature for profiling are summarized in Table 7.1. In this table, we discuss different methods used in each type of profilers as well as the advantages and disadvantages for them. We also included some example for each type.

We have to work with executable binaries and therefore, we cannot perform compile or link time code injection like what *GProf* [90] or *ATOM* [95] do. We need to either modify the executable binary, so that it can contain and use an instrumentation code or use a hardware approach as used by the *frequency loop detection profiler* and *DAProf*. Although the hardware approach has a very low overhead, it has some disadvantages. One obvious disadvantage is that it is not very portable. A hardware design made for one system might not fit into another one. The second disadvantage is that we need to take care of the virtual addresses in the system. At the hardware level, we only see the physical addresses. In cases such as *frequency loop detection profiler* which only deals with small loops, they do not require knowing the virtual addresses. The reason

Table 7.1: Comparison of Different Types of Profilers

Profiling Technique	Method used	Advantages	Disadvantages	Examples
Instrumentation	Instrumentation Code Insertion at Compile and Link time	Platform Independent	Cannot work with executable binaries	GProf [90]
	Instrumentation Code Insertion during linking of object files	Easy to port	cannot work with executable binaries	ATOM [95]
	Interpretation of Native Binary Code and JIT Compilation	Instrumentation Code can be inserted anywhere in the code	Has relatively large overhead. Very complicated and time consuming to create such a tool.	Pin [96], Dynamo [91]
	Modification of function prologues to jump to an Instrumentation Code	Has relatively low overhead. Easy to implement and port.	Instrumentation Code cannot be called from anywhere in the code. A function prologue must be of a certain minimum size, so that it can be replaced with a jump to the Instrumentation Code.	Detours [97] and IgProf [98]
Sampling	Using Timer Interrupts	Has low overhead and is non-intrusive	Can only give statistical approximation about time spent by different parts of the code	GProf and [99]
	Using Timer Interrupts and Hardware Performance Counters	Has low overhead and is non-intrusive. Gives additional information like the parts of code causing more cache misses, pipeline stalls etc.	Same as above	OProfile [93]
	Software Based	Easy to port	Extra code has to be injected	Arnold and Ryder [92] and Profiler for IBM TestaRossa [100]
Hardware Based	Using custom-designed hardware	Has very low overhead and is non-intrusive	It is difficult to port a design to another system. Consideration has to be given to handle virtual addresses.	Frequency Loop Detection Profiler [101] and DAProf [94]

is that the small loops can be seen from the instruction bus. On the other hand, in our case, in which we deal with software implementations, we require to know the virtual addresses of the instructions in the programs.

To implement the instrumentation part of the profiler in software, one possible approach is to do native binary code interpretation and JIT compiling, like *Pin* and *Dynamo*. Although this approach is very useful to create versatile profiling tools like *Pin*, it has some disadvantages. The first of which is that it is slow. Another disadvantage of this approach is that it is very complicated and time consuming to write a native binary code interpreter and JIT compiler for any processor.

In the proposed profiler, we choose to replace the function prologue with an unconditional jump to the instrumentation code. The instrumentation code contains the removed prologue as well as an unconditional jump back to the remaining part of the instrumented function. This has some advantages. First, low overhead profiling can be achieved by using an efficient instrumentation code. Second, this technique is much less complicated than the native binary code interpretation approach. Third, this technique is very portable, because we only have to deal with function prologues and not the rest of the code inside the functions.

One important issue in the instrumentation code is the way it saves the collected information. As the interaction between the scheduler and profiler has to be as fast as possible, using normal files is not an option and the data has to be kept in the memory. For this purpose, different techniques can be used. For example, one can use a */proc* file to save the information. In the proposed profiler, we propose the use of a shared memory called *Profiler Frame* and a double buffering mechanism to store and read profiled data. The *Profiler Frame* is shared between the profiler and the scheduler. Using this technique, we have been able to lower the overhead of the profiler to as low as less than one percent for typical applications.

Another contribution of the proposed profiler is in how we use a daemon that runs continuously and injects instrumentation code to the applications without any input from the user. The OS kernel is modified so that it sends a signal to the daemon whenever a new program is loaded for execution. On receiving that signal, the daemon injects instrumentation code into the application. The daemon uses the *Injector* utility for injecting code. After the instrumentation code is injected into the application, the application can perform self-instrumentation. The instrumented application automatically updates function call counts in shared memory on entering functions. Another feature of the proposed profiler is an off line program, known as the *Extractor* to optimize the code injection time. The emphExtractor extracts vital information from a program. For each program, the *Extractor* also creates a file which contains function ranges, so that the daemon can know which function was executing when a sample at a

timer interrupt was taken. This is done by using program counter value at time of the interrupt and process ID of the interrupted process. Approximate time spent by functions can be calculated in this way.

7.3 Design And Implementation

As explained in Chapter 3, the proposed runtime system is intended to make a completely virtualized and transparent hardware layer available to the program developers. In such a system, the programs are developed to be executed on the general-purpose processor and, therefore, the program developers are not bothered with complex hardware design issues. It is thus the runtime system's responsibility to find compute intensive tasks and map them into the faster cores. The profiler has to be able to profile the applications on the GPP in the first place. When a task is mapped into a different core, it has to be able to continue the profiling of the task on that core. As a result, our profiling mechanism has two aspects; the profiling on the GPP, and the profiling on the reconfigurable fabric.

As on the reconfigurable side, we use the MOLEN hardware organization and the MOLEN programming paradigm. We employ the MOLEN runtime primitives (discussed in Chapter 4) to perform the profiling. This will be discussed in more detail later in this section.

On the GPP side, the proposed profiler runs on the Linux operating system. Its task is to keep statistics of the running kernels on a general-purpose processor and present that information to the scheduler. The proposed profiler performs both instrumentation and sampling profiling. The instrumentation profiler records the number of times different functions have been called while, the sampling profiler calculate the approximate time spent per function. To instrument programs, we need some mechanism to inject code into programs. To achieve this, our code injector replaces the prologue of each function that has to be profiled with an unconditional jump to the instrumentation code. The instrumentation code contains a jump back to the rest of the profiled function

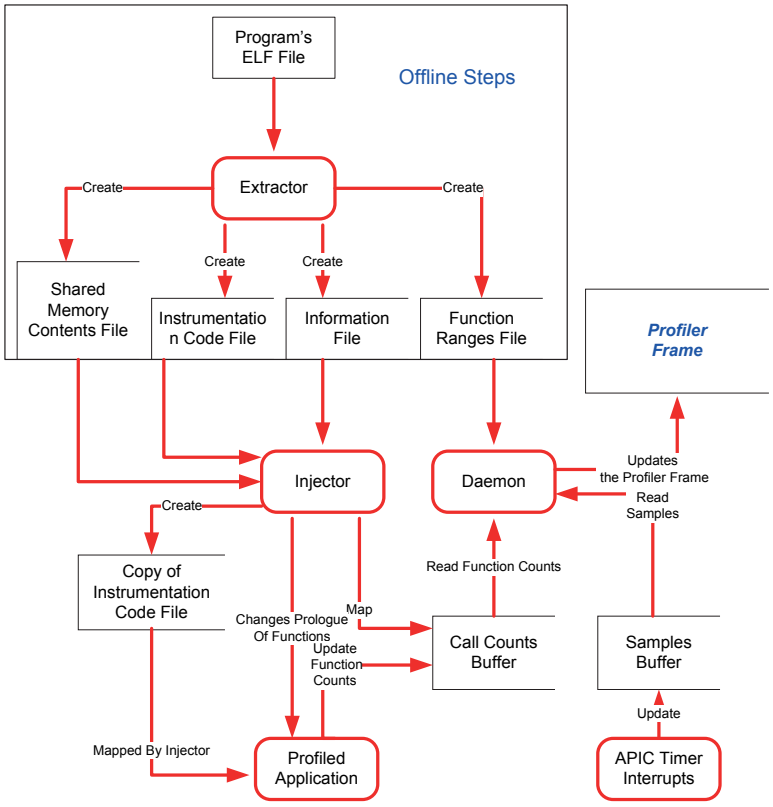

Figure 7.1: Interaction of Different Parts of the Proposed Profiler with the Profiled Application and the Scheduler

besides the prologue of the profiled function. The current implementation has been done and tested successfully on machines with x86 processors, both single-core and multi-core.

The proposed profiler consists of several different parts. The interaction between those parts is shown in Figure 7.1. The *Extractor* utility is used to create instrumentation code file besides others, from an executable *elf* file. The *Injector* utility is used to inject instrumentation code and map *Profiler Frame* into the address space of the profiled program. The *Injector* utility uses the *Ptrace* API in Linux to inject and modify code at runtime. The local APIC Timer interrupts update the *Samples Buffer*. Note that in case of processors

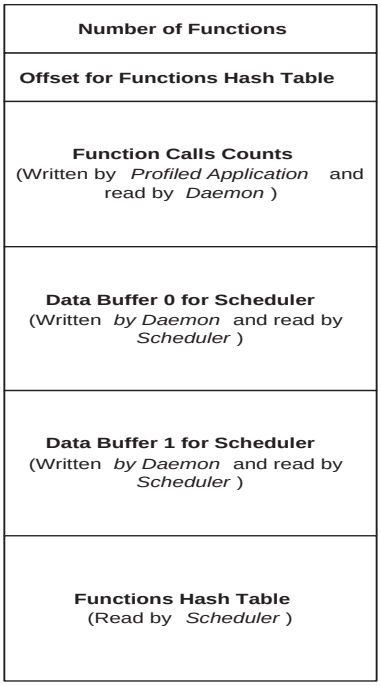

Figure 7.2: Contents of the Profiler Frame

other than x86, we can use their own local timer interrupts to update the *Samples Buffer*. Finally, the *Daemon* combines information from different places in a form that can be quickly and easily read by the scheduler. The purpose of the *Profiler Frame* is to keep the function call counts of profiled functions as well as information to be read by the scheduler. The reason we have used shared memory for the *Profiler Frame* is that it is the fastest possible method to communicate data.

One very important part of the profiler is the Profiler Frame's structure and the access mechanism to it. The Profiler Frame layout is shown in Figure 7.2 and consists of five blocks. The first block is the number of functions to be profiled. The next portion of the *Profiler Frame* is used to present data to the runtime system. The data in that portion is updated by the *daemon*. This middle part of the *Profiler Frame* employs a double buffering mechanism to prevent inconsistencies when there is a read and write conflict. To achieve that, the

daemon writes data to one buffer and the next time it writes to the other. When it completes writing data to one buffer, it changes the *index* to point to that buffer, so that the runtime system can read it from there.

Finally, the lower portion of the *Profiler Frame* contains a hash table. The keys to that hash table are profiled functions' names or IDs and values are indexes of those functions.

The *Profiler Frame* is the only part that needs to be known by the runtime system, which needs to integrate the proposed profiler. The rest of the profiler can be seen as a black box. All the generated data by the profiler is accessible from the *Profiler Frame*. The *Profiler Frame* is in fact a shared memory between the profiler and the runtime system. Therefore, each component in the runtime system that needs profiling information has to map this shared memory into its own address space.

When a kernel is mapped to the hardware, its profiling has to be continued in order to have a valid (updated) status of the systems at each point in time. Within the MOLEN programming paradigm, the mappings of the kernels to the reconfigurable fabric is being done by the SET and EXECUTE APIs [102]. This way, the profiling can be delegated to these APIs when the hardware execution is happening. Each kernel to be executed on the hardware should be invoked by the EXECUTE API from the runtime system. Therefore, in the execute phase we exactly know which kernel is executing and as a result the EXECUTE can update the corresponding values in the shared memory. This mechanism guarantees that function call counts updates in case of hardware execution. To update the values in the shared memory, we use the same code as we use in the injected codes in the instrumentation profiling.

7.4 Performance Evaluation

To test the performance of the proposed profiler, we used benchmarks from different areas. One of these is PC version of *tcf*, which is a Stationary Noise

Filter used in hArtes [103] demonstration. We used a sound file of size 19 MB as the input. Then we have *minisat2* [104], which is an industrial scale SAT solver. Note that we removed the randomness part in *minisat2*, so that our results do not vary from run to run. The input file for the *minisat* benchmark contained 630 variables and 2280 clauses. Then we have *H264/AVC encoder*, from MediaBench II benchmarks [75], which is an H264 encoder application. For the H264 encoder application, we used an input file of size 5.2 MB and bit rate of 45020 bps. Next, we have *coremark* [105], which is a free synthetic benchmark from *EEMBC* [106]. Finally, we have a benchmark that we created ourselves, known as *multiply*. That program calls a multiply function which just returns the multiplication of two fixed numbers, one billion times. Normally, such tiny functions are inlined by the compiler and, therefore, profiling would not be required for them. However, testing the profiler with such tiny functions gives us an idea of the worst-case performance of the instrumentation profiling. To avoid inlining of the multiply function, the *multiply* application is compiled with optimizations turned off.

We performed different experiments, which are discussed in the following section.

7.4.1 Instrumentation Overhead

The instrumentation overhead of the proposed profiler is shown in Table 7.2. From the table we can see that the overhead of the proposed profiler, except for the *multiply* application, is always less than 1.5 %. The low overhead for applications other than *multiply* was expected because the proposed profiler only adds three instructions to original functions for profiling. The overhead for the *multiply* application here is relatively large because it repeatedly calls a very tiny function that just returns product of two numbers.

Table 7.2: Instrumentation Overhead (secs)

	Normal	**Profiling**	**Overhead**
multiply	9.681	10.092	4.25%
coremark	12.496	12.654	1.26%
tcf	7.083	7.089	$< 1\%$
h264enc	40.774	40.865	$< 1\%$
minisat2	29.004	29.155	$< 1\%$

Table 7.3: Sampling and Daemon Overhead

	Normal	**OProfile**	**The Proposed Profiler**
multiply			
Time (secs)	9.681	9.677	9.688
Overhead	-	$< 1\%$	$< 1\%$
coremark			
Time (secs)	12.496	12.510	12.495
Overhead	-	$< 1\%$	$< 1\%$
tcf			
Time (secs)	7.083	7.121	7.126
Overhead (%)	-	$< 1\%$	$< 1\%$
h264enc			
Time (secs)	40.774	40.838	41.034
Overhead	-	$< 1\%$	$< 1\%$
minisat2			
Time (secs)	29.004	29.037	29.058
Overhead	-	$< 1\%$	$< 1\%$

7.4.2 Sampling and Daemon Overhead

We show the results achieved without performing instrumentation in Table 7.3. The purpose of not performing instrumentation in this case is to quantify the overhead imposed by sampling and the *Daemon*. The results are compared with those achieved from *OProfile*. We can see that the overhead for both *OProfile* and the proposed profiler is negligible. It should be noted here that we only used the *timer interrupt event* for *OProfile*, to make it functionally equal to the proposed profiler.

Table 7.4: Sampling Accuracy of the Proposed Profiler

Function	gprof (%)	The Proposed Profiler (%)
coremark		
crcu8	31.77	31.13
core_state_transition	30.05	31.20
core_bench_list	13.85	15.14
matrix_mul_matrix_bitextract	5.48	5.55
minisat2		
propagate	74.87	75.05
analyze	13.56	13.53
litRedundant	4.45	4.43
cancelUntil	2.85	2.92
h264enc		
SetupFastFullPelSearch	33.59	32.63
dct_luma	11.17	10.60
biari_encode_symbol	7.33	7.30
SetupLargerBlocks	3.83	3.42

7.4.3 Sampling Accuracy

In this part, we checked the accuracy of the sampling part of the proposed profiler by comparing it with gprof [90]. We executed each program five times, both with the proposed profiler and with *gprof*, so that we could get the mean values . The results are given in Table 7.4. In this table, percentages of total time spent for the functions that took the most time according to *gprof* are given. From the Table 7.4, it can be seen that the mean values for both the proposed profiler and *gprof* are almost the same. This was expected as we are using the same technique as *gprof*. The only difference is that we take our samples through the local APIC timer interrupts, so that we can take samples for multi-cores, while *gprof* uses the kernel timer interrupt and therefore cannot perform sampling for multi-cores. Since the default Linux kernel timer interrupt occurs at the rate of 100 per second, we also set the frequency of the local APIC timers interrupt to 100 per second for fair comparison.

Table 7.5: Overall Overhead of the Proposed Profiler (secs)

	Normal	**The Proposed Profiler**	**Overhead**
Single Application Execution			
multiply	9.681	10.112	4.45%
coremark	12.496	12.655	1.27%
tcf	7.083	7.088	$< 1\%$
h264enc	40.774	41.158	$< 1\%$
minisat2	29.099	29.040	$< 1\%$
Multiple Applications Execution			
five benchmarks	92.32	93.04	$< 1\%$

7.4.4 Overall Overhead

In the first part of this experiment, we tested our benchmark applications with all parts of the profiler working. The results are shown in Table 7.5. The results are as expected, that is all applications other than *the multiply* application have overhead of less than 1.5%. Moreover, overall overhead for all application is almost the same as that for instrumentation overhead, thus reinforcing the fact that sampling and the *daemon* have very low overheads.

In the second part of the experiment, we executed all the benchmark applications simultaneously with all parts of the profiler working. We repeated the experiment five times and took the mean values, which are shown in bottom of table 7.5. The results show that the profiling overhead is less than 1%.

Our results shows that our profiling system performs the same as *Dynamo*, which has overhead of less than 1.5% and better than the profiler presented in [92] which has average overhead of 3%. Those parts of the code that are optimized by Dynamo are never profiled again, while the proposed profiler has to continuously profile all the functions.

7.4.5 Percentage of Profilable Functions

The proposed profiler replaces prologues of to be profiled functions with a jump instruction. For that purpose, a function's prologue must be at least 5 bytes because the jump instruction in an x86 consumes 5 bytes. Most of the

Table 7.6: Percentage of Profilable Functions

Program	Opt Level	Total Functions	Profilable Functions
tcf	-O0	59	59 (100%)
h264enc	-O2	591	560 (94.8%)
minisat2	-O3	56	52 (92.9%)
coremark	-O2	40	29 (72.5%)

functions have at least 5 bytes of prologue. However, some very small functions do not. By prologue instructions, we mean instructions that prepare the stack and registers for use within a function. We showed the number of functions that are profilable for different applications in Table 7.6. We also listed the optimization levels used for compiling those applications, the purpose of which is to see if optimizations make it any harder to find profilable functions. Except for *coremark*, all applications have more than 90% of profilable functions and both *h264enc* and *minisat2* are using high level of optimizations. The reason that *coremark* has only 72.5% of profilable functions is because there are many functions of very small sizes in it.

We use only prologue and that is why we have the limitation on the size of it. One can argue that if the prologue is too small, you can consider more instructions from the top of function as a part of prologue. However, this solution is not feasible. Because, it might happen that an instruction inside a function jumps to some instruction at the top of that function. If that instruction at the top is replaced by some other instructions, the program might crash or behave differently.

7.5 Conclusion

In this chapter, we described the design and implementation of a runtime profiler, which can be used as a part of the MOLEN runtime environment. The profiler is a combination of a Sampling profiler and an Instrumentation profiler. We discussed different parts of the profiler namely the *extractor*, *injector*, *sampler*, *profiler frame* and *daemon*. Then, we showed the overhead of the proposed

profiler from different aspects. This is done by showing the overhead on Instrumentation, Sampling and the overall overhead. Besides, we compared the accuracy of the proposed profiler with a popular design time profiler. All the presented results show that the proposed profiler has very low overhead (less than 1.5%) and is as accurate as design time profilers.

8 Conclusions

In this dissertation, we presented a runtime system for heterogeneous multi-core systems. We discussed the design and the implementation of the proposed runtime system. The runtime system consists of a scheduler, a profiler, a transformer, a JIT compiler and a kernel library. A detailed description of each component was presented and the performance of the system as well as the imposed overhead of the components was discussed.

In this chapter, we first summarize the thesis and presented results. Then, we present the main contributions of the work and describe the remaining open issues and future directions.

8.1 Outlook

In this thesis, we presented our runtime support system for heterogeneous multi-core systems. The work presented in this thesis can be summarized as follows.

In Chapter 2, we gave an overview on the background information and the related work. In the beginning, several examples of the target architectures were presented. Next, we gave a short summary of the MOLEN and the MOLEN programming paradigm as well as the MOLEN design tool chain.

Then, we briefly described some of the similar approaches used in runtime support for heterogeneous multi-core systems. The motivation for the need of virtualization is discussed afterwards. At the end, we pointed out the open issues and discussed our approach towards solving those issues.

In Chapter 3, we presented our runtime system. In this chapter, we explained the structure of our runtime system and its forming components. Each component was briefly discussed. Furthermore, we explained how the whole system is structured into layers. The system is divided into four layers; the application layer, the virtualization layer, the operating system layer, and the hardware platform layer which is further divided into the MOLEN abstraction layer and the physical hardware layer. Subsequently, we explained the interaction mechanisms between different components in the runtime system. We also included two case study scenarios and showed how the runtime system should react in those scenarios.

In Chapter 4, we explained how the MOLEN programming paradigm is extended to support multitasking and multi-application scenarios. To accomplish this, we used the same idea of MOLEN set and execute instructions to abstract away the concept of a task. In this way, we decoupled the task call from the task implementation. We also proposed a binding mechanism to bind a task implementation to the corresponding task call. The high level SET and EXECUTE APIs are also presented. Finally, we showed the overhead of the proposed APIs using some experiments.

In Chapter 5, we presented our scheduler as a part of the runtime system. In this chapter, we showed how we used a combination of the design time and the runtime scheduling in order to optimize the system performance. In the design time, we used the compiler to perform static task scheduling assuming single thread of execution. Then at the runtime, the runtime scheduler performs the actual task scheduling having the scheduling decisions from the compiler as a hint. The runtime scheduler can also use the information provided by the runtime profiler. It can also use the information transferred from the design time in the form of a CCG. Following that, we presented a number of scheduling

policies as case studies and provided performance evaluations. We based our scheduling decisions on three different parameters. We showed that *Expected Time Improvement* and *Longest Distance in the Future* are very good heuristics for the scheduling procedure.

In Chapter 6, we focused on the conditions in which we have real-time constraints. We used fuzzy logic to model the real-time constrains and to improve the scheduling decisions. Using deadline as a fuzzy parameter in real-time scheduling is more promising than laxity.

In Chapter 7, we described the design and implementation of our runtime profiler, which can be used as a part of the runtime environment. The profiler is a combination of a sampling profiler and an instrumentation profiler. We discussed different parts of the profiler namely the *extractor*, *injector*, *sampler*, *profiler frame* and *daemon*. Then, we calculated the overhead of our profiler from different aspects. This is done by showing the overhead on Instrumentation, Sampling and the overall overhead. Besides, we compared the accuracy of our profiler with a popular design time profiler, *gprof*. All the presented results showed that our profiler has very low overhead (less than 1.5%) and is as accurate as design time profilers.

8.2 Contributions

The main contributions of this dissertation can be summarized as follows.

1. We introduced a comprehensive runtime system and presented a detailed discussion of its components and performance evaluation. We provided a detailed overview of the system layers and showed how each layer interacts with the others. The most important layer is the virtualization layer which consists of a scheduler, a profiler, a JIT compiler, a transformer and a kernel library.

2. We defined and implemented a new task abstraction mechanism through which the task implementation is separated from the task call. The main

idea is taken from the MOLEN programming paradigm. We extended the model in such a way that it is suitable for multitasking and multi-application scenarios.

3. We presented a detailed discussion of scheduling requirements for heterogeneous multi-core systems and some scheduling policies together with their performance evaluations. We introduced a number of new scheduling algorithms. Our scheduling decision making is based on the *distance in the future* and *frequency in the future* as well as *expected speedup*. We also introduced the configuration call graph as a viable source of information for the scheduler.

4. We employed fuzzy logic in the decision making process of the scheduler for the systems with real-time constraints. We modelled the inputs of the scheduler such as laxity and deadline as fuzzy variables and used a Sugeno inference engine to derive the scheduling priorities. In this way, we presented some new scheduling algorithms.

5. We presented a novel runtime profiler whose task is to analyze the running code and produce statistics about code execution such as the computation intensity and frequency of the execution. This profiler has to run concurrently with other applications on the end user's machine, and hence, needs to have a low overhead. Our runtime profiler has an overhead of less 1.5%.

8.3 Open Issues and Future Directions

In this thesis, we proposed a runtime system that consists of a number of components. We have implemented the scheduler and the profiler. However, JIT compilation for reconfigurable fabric is still an open issue. Such a compiler has to compile from binary to bit stream. A JIT compiler must be able to perform technology mapping, placement, and routing. This process also requires a standard binary format for the FPGAs from all the vendors. Furthermore, JIT

compilation involves introducing fast and efficient mapping, placement and routing algorithms that can be used at runtime without imposing a considerable overhead.

The transformer component is not implemented in the current work. Once a JIT compiler generates the binary for the target core, the runtime system needs to change the binary of the main thread of the application to call the new binary code instead of the old one. The transformer is responsible to do this code modification on the fly. The basic implementation mechanism for the transformer is a little bit similar to the profiler. The profiler also modifies the binary and changes the prologue of a function. However, the difference is that the profiler changes the binary at the function body, whereas, the transformer has to do this at the function call. This involves the transformer to be able to identify the function calls in the code and change them to calls to the newly generated code.

Once we have all the components, we can integrate all of them together and have a complete runtime system. So far, we have implemented the scheduler and the profiler as well as the library. Yet, we do not have the JIT compiler and the transformer.

Another open issue which is not addressed in this work is the preemptive scheduling of tasks on the reconfigurable co processors. By preemptive scheduling, we mean the ability to stop the execution of a task on the RP at some time and resuming its execution from the point it was stopped onward. This process involves mechanisms to save the hardware status and to restore it later. Furthermore, problems might arise if the task continues execution at a different place from the previous run (i.e. to continue on a different column of an FPGA). Moreover, the task migration from one core to the other (i.e. migration from the hardware to the software) is still an open issue in the current work.

Yet another open issue which is discussed in Chapter 2 is the lack of consistent and uniform comparison methodology for comparing similar works in the area of runtime systems. This seems to be a common problem in the field. In future work, we will study this problem and provide a set of qualitative and

quantitative metrics for comparison purposes. Besides, we will provide a set of standard real world application workloads that can be used as the input for comparisons.

In the MOLEN programming paradigm, we assume there is a single general-purpose processor that acts as the master and all the other processors are coprocessors. One major issue with the processor, coprocessor paradigm is that the general purpose processor can become a bottleneck as the main thread of the execution is executing there. Generally, only the compute intensive parts might be mapped into the coprocessors. This means that a large portion of the program which is not compute intensive had to run on the GPP. This is one of the reasons that the presented speed ups in the previous chapters are not very high. However within the same paradigm, if we employ more GPPs, none compute intensive parts of the applications can also be executed in parallel. The presented APIs in Chapter 4 are based on such an assumption. However, in case of the platforms with multiple master processors that each master processor can independently access any of the cooperating processors, we need some centralized or decentralized coordination techniques to handle the conflicts between the master processors. For example, the presented APIs can implement some sort of locking mechanism in order to guarantee the mutual access to the computing resources. However, this increases the overhead and might influence the system performance quite considerably.

Bibliography

[1] G. Kornaros, *Multi-core embedded systems*. CRC Press, 2010.

[2] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006, pp. 347–358.

[3] M. Sugeno, *Industrial Applications of Fuzzy Control*. New York, NY, USA: Elsevier Science Inc., 1985.

[4] H. Flows, C. Easy, C. Uses, P. Processing, and D. Processing, "Xilinx architects ARM-based processor-first, processor-centric device," *Xcelljournal*, vol. 71, 2010.

[5] T. M. Brewer, "Instruction set innovations for the convey HC-1 computer," *IEEE Micro*, vol. 30, pp. 70–79, 2010.

[6] http://www.freescale.com, 2010.

[7] I. Colacicco, G. Marchiori, and R. Tripiccione, "The hardware application platform of the hArtes project," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, September 2008, pp. 439 –442.

[8] http://www.ece.ufl.edu/announcements/news/2010/Novo-G.html, 2010.

[9] F. Ferrandi, L. Fossati, M. Lattuada, G. Palermo, D. Sciuto, and A. Tumeo, "Automatic parallelization of sequential specifications for symmetric MPSoCs," in *IESS: International Embedded Systems Symposium*, 2007, pp. 179–192.

[10] F. Ferrandi, M. Lattuada, C. Pilato, and A. Tumeo, "Performance estimation for task graphs combining sequential path profiling and control dependence regions," in *MEMOCODE'09: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for*

Codesign. Piscataway, NJ, USA: IEEE Press, 2009, pp. 131–140.

[11] W. Luk, J. Coutinho, T. Todman, Y. Lam, W. Osborne, K. Susanto, Q. Liu, and W. Wong, "A high-level compilation toolchain for heterogeneous systems," in *Proceedings of IEEE International SoC Conference (SOCC)*, September 2009.

[12] Y. D. Yankova, K. B. G. Kuzmanov, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: Delftworkbench automated reconfigurable VHDL generator," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, 2007, pp. 697–701.

[13] H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 2, p. 14, 2008.

[14] H. So and R. Brodersen, "File system access from reconfigurable FPGA hardware processes in BORPH," in *International Conference on Field Programmable Logic and Applications*, 2008, pp. 567 –570.

[15] H. So and R. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," in *International Conference on Field Programmable Logic and Applications*, 2006.

[16] K. Olukotun, "Towards pervasive parallelism," in *Barcelona Multicore Workshop (BMW2008)*, Jun 2008.

[17] E. Lubbers and M. Platzner, "ReconOS: An operating system for dynamically reconfigurable hardware," *Dynamically Reconfigurable Systems*, pp. 269–290, 2010.

[18] E. Lubbers and M. Platzner, "ReconOS: An RTOS supporting hard- and software threads," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL)*. Amsterdam, Netherlands: IEEE, August 2007, pp. 441–446.

[19] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *International Conference on Field Programmable Logic and Applications*, 2008, pp. 17–22.

[20] E. Lubbers and M. Platzner, "Cooperative multithreading in dynamically reconfigurable systems," in *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL)*. Heidelberg, Germany: IEEE, September 2008.

[21] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Transactions on Computers*, pp. 1393–1407, 2004.

[22] E. Lubbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, pp. 1–33, 2009.

[23] J. Kelm and S. Lumetta, "HybridOS: Runtime support for reconfigurable accelerators," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, 2008, pp. 212–221.

[24] J. Kelm, I. Gelado, M. Murphy, N. Navarro, S. Lumetta, and W. Hwu, "CIGAR: Application partitioning for a CPU/coprocessor architecture," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007, pp. 317–326.

[25] G. Wigley, D. Kearney, and D. Warren, "Introducing ReConfigME: An operating system for reconfigurable computing," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, ser. Lecture Notes in Computer Science, 2002, pp. 687–697.

[26] G. Wigley and D. Kearney, "Performance evaluations of ReconfigME," in *IEEE International Conference on Field Programmable Technology*, December 2006, pp. 309 –312.

[27] G. Wigley and D. Kearney, "The development of an operating system for reconfigurable computing," in *The 9th Annual IEEE Symposium on*

Field-Programmable Custom Computing Machines, 2001, pp. 249–250.

[28] G. Wigley, D. Kearney, and M. Jasiunas, "ReConfigME: a detailed implementation of an operating system for reconfigurable computing," in *20th International Parallel and Distributed Processing Symposium*, April 2006.

[29] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *Proceedings of the 40th annual Design Automation Conference*. New York, NY, USA: ACM, 2003, pp. 250–255.

[30] S. Haynes, H. Epsom, R. Cooper, and P. McAlpine, "UltraSONIC: A reconfigurable architecture for video image processing," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, 2002, pp. 25–45.

[31] T. Wiangtong, P. Cheung, and W. Luk, "A unified codesign run-time environment for the UltraSONIC reconfigurable computer," in *Field-Programmable Logic and Applications*, 2003, pp. 396–405.

[32] Wiangtong, C. Ewe, and P. Cheung, "SONICmole: a debugging environment for the UltraSONIC reconfigurable computer," in *Proceedings of the 2003 International Symposium on Circuits and Systems*, May 2003, pp. 808–811.

[33] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A computational model for reconfigurable devices," in *International Conference on Field Programmable Logic and Applications*, August 2006, pp. 1–4.

[34] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "Hthreads: a hardware/software co-designed multithreaded RTOS kernel," vol. 2, September 2005.

[35] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews, "Enabling a uniform programming model across

the software/hardware boundary," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 89–98.

[36] K. Kosciuszkiewicz, F. Morgan, and K. Kepa, "Run-time management of reconfigurable hardware tasks using embedded linux," in *International Conference on Field-Programmable Technology*, December 2007, pp. 209 –215.

[37] N. W. Bergmann, J. A. Williams, J. Han, and Y. Chen, "A process model for hardware modules in reconfigurable system-on-chip," in *19th International Conference on Architecture of Computing Systems, Workshops Proceedings*, 2006, pp. 205–214.

[38] J. Williams, N. Bergmann, and X. Xie, "FIFO communication models in operating systems for reconfigurable computing," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005, pp. 277 – 278.

[39] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Dynamic heterogeneity and the need for multicore virtualization," *SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 5–14, April 2009.

[40] C. Bertin, C. Guillon, and K. De Bosschere, "Compilation and virtualization in the HiPEAC vision," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 96–101.

[41] R. Buchty, D. Kramer, F. Nowak, and W. Karl, "A seamless virtualization approach for transparent dynamical function mapping targeting heterogeneous and reconfigurable systems," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, J. Becker, R. Woods, P. Athanas, and F. Morgan, Eds., 2009, vol. 5453, pp. 362–367.

[42] A. Hofmann, K. Waldschmidt, and J. Haase, "SDVMR; managing heterogeneity in space and time on multicore SoCs," in *Adaptive Hardware*

and Systems (AHS), 2010 NASA/ESA Conference on, June 2010, pp. 142 –148.

[43] S. Huang, A. Hormati, D. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," in *Proceedings of the 22nd European conference on Object-Oriented Programming*. Springer-Verlag, 2008, pp. 76–103.

[44] E. Rohou, A. C. Ornstein, A. E. Özcan, and M. Cornero, "Combining processor virtualization and component-based engineering in C for heterogeneous many-core platforms," in *Programming Models for Emerging Architectures (PEMA)*.

[45] C. Augonnet and R. Namyst, "A unified runtime system for heterogeneous multi-core architectures," vol. 5415, pp. 174–183, 2009.

[46] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 156–166, June 2007.

[47] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *Proceedings of the 17th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2008, pp. 197–200.

[48] A. Vetro, S. Yea, M. Zwicker, W. Matusik, and H. Pfister, "Overview of multiview video coding and anti-aliasing for 3D displays," in *IEEE International Conference on Image Processing*, vol. 1, October 2007, pp. I –17 –I –20.

[49] http://www.rtlinuxfree.com, 2011.

[50] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005, pp. 149 – 158.

[51] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Panainte, "The MOLEN

programming paradigm," in *3rd International Workshop on Systems, Architectures, Modeling, and Simulation*, July 2003, pp. 1–10.

[52] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Compiling for the MOLEN programming paradigm," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, September 2003, pp. 900–910.

[53] V. Manh Tuan and H. Amano, "A preemption algorithm for a multitasking environment on dynamically reconfigurable processor," *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 172–184, 2008.

[54] S. Jovanovic, C. Tanougast, and S. Weber, "A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems," in *Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE Computer Society, 2007, pp. 358–364.

[55] M. Fazlali and A. Zakerolhosseini, "Rec-Bench: A tool to create benchmark for reconfigurable computers," in *Programmable Logic Conference (SPL), 2010 VI Southern*, March 2010, pp. 187 –190.

[56] J. M. P. Cardoso and H. C. Neto, "Compilation for FPGA-based reconfigurable hardware," *IEEE Design & Test of Computers*, vol. 20, no. 2, pp. 65–75, March 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=766314.766333

[57] G. Dimitroulakos, N. Kostaras, M. D. Galanis, and C. E. Goutis, "Compiler assisted architectural exploration framework for coarse grained reconfigurable arrays," *Journal of Supercomputing*, vol. 48, no. 2, pp. 115–151, May 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1541536.1541567

[58] R. Cordone, F. Redaelli, M. Redaelli, M. Santambrogio, and D. Sciuto, "Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 662 –675, May

2009.

[59] Z. Gu, M. Yuan, and X. He, "Optimal static task scheduling on reconfigurable hardware devices using model-checking," in *13th IEEE Real Time and Embedded Technology and Applications Symposium*, April 2007, pp. 32 –44.

[60] J. Angermeier, S. Fekete, T. Kamphans, N. Schweer, and J. Teich, "Virtual area management: Multitasking on dynamically partially reconfigurable devices," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, April 2010, pp. 1 –4.

[61] F. Redaelli, M. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *Design, Automation and Test in Europe*, March 2008, pp. 519 –522.

[62] M. Sabeghi, H. Mushtaq, and K. Bertels, "Runtime multitasking support on polymorphic platforms," *SIGARCH Comput. Archit. News*, vol. 38, pp. 46–52, January 2011. [Online]. Available: http://doi.acm.org/10.1145/1926367.1926376

[63] H. Walder and M. Platzner, "Online scheduling for block-partitioned reconfigurable devices," 2003.

[64] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," 2003.

[65] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," *Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 106–111, 2005.

[66] P.-A. Hsiung, C.-H. Huang, and Y.-H. Chen, "Hardware task scheduling and placement in operating systems for dynamically reconfigurable SoC,"

Journal of Embedded Computing, vol. 3, no. 1, pp. 53–62, January 2009.

[67] M. Yuan, Z. Gu, X. He, X. Liu, and L. Jiang, "Hardware/software partitioning and pipelined scheduling on runtime reconfigurable FPGAs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 15, no. 2, pp. 13:1–13:41, March 2010. [Online]. Available: http://doi.acm.org/10.1145/1698759.1698763

[68] R. Guha, N. Bagherzadeh, and P. Chou, "Resource management and task partitioning and scheduling on a run-time reconfigurable embedded system," *Computers and Electrical Engineering*, vol. 35, no. 2, pp. 258–285, March 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1507764.1507873

[69] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "Online task scheduling for the FPGA-based partially reconfigurable systems," vol. 5453, pp. 216–230, 2009.

[70] M. Sabeghi, V. Sima, and K. Bertels, "Compiler assisted runtime task scheduling on a reconfigurable computer," in *19th International Conference on Field Programmable Logic and Applications (FPL09)*, August 2009.

[71] J. Resano, J. A. Clemente, C. Gonzalez, D. Mozos, and F. Catthoor, "Efficiently scheduling runtime reconfigurations," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 4, pp. 58:1–58:12, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1391962.1391966

[72] E. Panainte, K. Bertels, and S. Vassiliadis, "The MOLEN compiler for reconfigurable processors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 1, 2007.

[73] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Interprocedural optimization for dynamic hardware configurations," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science, T. D. Hmlinen, A. D. Pimentel, J. Takala, and

S. Vassiliadis, Eds., vol. 3553. Springer Berlin / Heidelberg, 2005, pp. 2–11.

[74] L. Bauer, M. Shafique, and J. Henkel, "MinDeg: a performance-guided replacement policy for run-time reconfigurable accelerators," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '09. New York, NY, USA: ACM, 2009, pp. 335–342. [Online]. Available: http://doi.acm.org/10.1145/1629435.1629481

[75] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *13th Annual IEEE/ACM International Symposium on Microarchitecture*, December 1997, pp. 330 –335.

[76] M. Rullmann and R. Merker, "A cost model for partial dynamic reconfiguration," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, July 2008, pp. 182 –186.

[77] G. Barnett, http://cpuss.codeplex.com/, 2009.

[78] P. Laplante, *Real-Time Systems Design & Analysis*. Wiley-India, 2009.

[79] K. Ramamritham and J. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," vol. 82, no. 1, January 1994, pp. 55 –67.

[80] J. Goossens and P. Richard, "Overview of real-time scheduling problems," in *9th International Workshop on Project Management and Scheduling*, 2004.

[81] S. Lauzac, R. Melhem, and D. Mosse, "Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor," in *10th Euromicro Workshop on Real-Time Systems*, June 1998, pp. 188 –195.

[82] B. Andersson and J. Jonsson, "Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition," in *7th International Confer-*

ence on Real-Time Computing Systems and Applications, 2000, pp. 337 –346.

[83] M. Sabeghi, M. Naghibzadeh, and T. Taghavi, "Scheduling non-preemptive periodic tasks in soft real-time systems using fuzzy inference," in *9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, April 2006, p. 6 pp.

[84] M. Sabeghi, M. Naghibzadeh, and T. Taghavi, "A fuzzy algorithm for scheduling soft periodic tasks in preemptive real-time systems," in *Advances in Systems, Computing Sciences and Software Engineering*. Springer Netherlands, 2006, pp. 11–16.

[85] M. Sabeghi, H. Deldari, V. Salmani, M. Bahekmat, and T. Taghavi, "A fuzzy algorithm for real-time scheduling of soft periodic tasks on multiprocessor systems," in *IADIS International Conference on Applied Computing*, 2006.

[86] L. Wang, *A course in fuzzy systems and control*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996.

[87] E. Mamdani and S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller," *International Journal of Man-Machine Studies*, vol. 7, no. 1, pp. 1 – 13, 1975.

[88] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[89] M. Sabeghi and K. Bertels, "Toward a runtime system for reconfigurable computers: A virtualization approach," in *Design, Automation and Test in Europe (DATE09)*, April 2009.

[90] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," *ACM SIGPLAN Notices*.

[91] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *ACM SIGPLAN Notices*, vol. 35, no. 5,

2000, pp. 1–12.

[92] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 168–179, 2001.

[93] http://oprofile.sourceforge.net/, 2010.

[94] A. Nair and R. Lysecky, "Non-intrusive dynamic application profiler for detailed loop execution characterization," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM, 2008, pp. 23–30.

[95] A. Eustace and A. Srivastava, "ATOM: a flexible interface for building high performance program analysis tools," in *Proceedings of the USENIX 1995 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1995, pp. 25–25.

[96] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[97] G. Hunt, , G. Hunt, and D. Brubacher, "Detours: Binary interception of Win32 functions," in *In Proceedings of the 3rd USENIX Windows NT Symposium*, 1998, pp. 135–143.

[98] G. Eulisse and L. A. Tuura, "Igprof profiling tool," *Computing in High Energy Physics and Nuclear Physics*, 2004.

[99] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "A dynamic optimization framework for a java just-in-time compiler," in *16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2001, pp. 180–195.

[100] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley, "Experiences

with multi-threading and dynamic class loading in a java just-in-time compiler," in *International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 87–97.

[101] A. Ross and F. Vahid, "Frequent loop detection using efficient nonintrusive on-chip hardware," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1203–1215, 2005.

[102] M. Sabeghi and K. Bertels, "Interfacing operating systems and polymorphic computing platforms based on the MOLEN programming paradigm," in *Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2010.

[103] http://www.hartes.org/, 2010.

[104] N. Een and N. Sorensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, 2004, pp. 333–336. [Online]. Available: http://www.springerlink.com/content/x9uavq4vpvqntt23

[105] http://www.coremark.org/, 2010.

[106] http://www.eembc.org/, 2010.

List of Publications

International Journals

1. M. Fazlali, M. Sabeghi, A. Zakerolhosseini, K.L.M. Bertels, **Efficient Task Scheduling for Runtime Reconfigurable Systems**, *Elsevier Journal of Systems Architecture*, Vol. 56, Issue 11, November 2010, pp. 623-632.

2. M. Sabeghi, H. Mushtaq, K.L.M. Bertels, **Runtime Multitasking Support on Polymorphic Platforms**, *ACM SIGARCH Computer Architecture News*, Vol. 38, Issue 4, September 2010, pp. 46-52.

3. M. Sabeghi, M. Naghibzadeh, T. Taghavi, **A Fuzzy Algorithm for Scheduling Soft Periodic Tasks in Preemptive Real-Time Systems**, *New Mathematics and Natural Computation Journal*, Vol. 3, Issue 3, November 2007, pp. 371-384.

International Conferences

1. H. Mushtaq, M. Sabeghi, K.L.M. Bertels, **A Runtime Profiler: Toward Virtualization of Polymorphic Computing Platforms**, *2010 International Conference on Reconfigurable Computing*, December 2010.

2. M. Sabeghi, K.L.M. Bertels, **Interfacing Operating Systems and Polymorphic Computing Platforms based on the MOLEN Programming Paradigm**, *Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture in conjunction with ISCA10*, June 2010.

3. M. Sabeghi, H. Mushtaq, K.L.M. Bertels, **Runtime Multitasking Support on Reconfigurable Accelerators**, *First International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies held within ACM ICS 2010*, June 2010.

4. M. Sabeghi, V.M. Sima, K.L.M. Bertels, , **Compiler Assisted Runtime Task Scheduling on a Reconfigurable Computer**, *19th International Conference on Field Programmable Logic and Applications (FPL09)*, August 2009.

5. M. Fazlali, A. Zakerolhosseini, M. Sabeghi, K.L.M. Bertels, G. Gaydadjiev, **Data path Configuration Time Reduction for Run-time Reconfigurable Systems**, *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA09)*, July 2009.

6. M. Sabeghi , K.L.M. Bertels, **Toward a Runtime System for Reconfigurable Computers: A Virtualization Approach**, *Design, Automation and Test in Europe (DATE09)*, April 2009.

7. M. Sabeghi, H. Deldari, **A Fuzzy Algorithm for Scheduling Periodic Tasks on Multiprocessor Soft Real-Time Systems**, *IASTED International Conference on Modeling and Simulation* , May 2006.

8. M. Sabeghi, M. Naghibzadeh, T. Taghavi, **Scheduling Non-Preemptive Periodic Tasks in Soft Real-Time Systems using Fuzzy Inference**, *9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC)*, April 2006.

Local Conferences

1. M. Sabeghi, K.L.M. Bertels, **Current Trends in Resource Management of Reconfigurable Systems**, *19th Annual Workshop on Circuits, Systems and Signal Processing*, November 2008.

2. M. Sabeghi, K.L.M. Bertels, **Toward a Run-time Support System for MOLEN Hardware Organization**, *Architectures and Compilers for Embedded Systems (ACES)*, September 2008.

3. M. Sabeghi, K.L.M. Bertels, M. Naghibzadeh, **Deadline vs. Laxity as a Decision Parameter in Fuzzy Real-Time Scheduling**, *18th Annual Workshop on Circuits, Systems and Signal Processing*, November 2007.

Other Papers
Journals

1. H. Deldari, M. Sabeghi, R. Mafi, **An Agent-based Approach to Grid Programming**, *Kuwait Journal of Science and Engineering*, Vol. 34 No.2, December 2007.

2. M. Naghibzadeh, M. Sabeghi, S. Mirshokraie, H. Abachi, **Round Data Mailer Real-Time Protocol and its Message Delivery Performance**, *International Review on Computers and Software journal (IRECOS)*, Vol. 2 No. 6, November 2007.

3. M. Sabeghi, M.H. Yaghmaee, **Using Fuzzy Logic to Improve Cache Replacement Decisions**, *IJCSNS International Journal of Computer Science and Network Security*, Vol. 6 No.3, March 2006.

Conferences

1. S. Mirshokraie, M. Sabeghi, K.L.M. Bertels, M. Naghibzadeh, **Datalife Time Analysis in RDM+ Real-Time Communication Protocol**, *IEEE International Conference on Signal Processing and Communications*, November 2007.

2. S. Mirshokraie, M. Sabeghi, M. Naghibzadeh, K.L.M. Bertels, **Performance Evaluation of Real-Time Message Delivery in RDM Algorithm**, *Third International Conference on Networking and Services (ICNS07)*, June 2007.

3. M Sabeghi, M. Naghibzadeh M., K.L.M. Bertels, **RDM+: A New Mac Layer Real-Time Communication Protocol**, *IEEE Sarnoff Symposium*, April 2007.

4. M. Sabeghi, M. Naghibzadeh, H. Deldari, **Performance Assessment of a Distributed Real-Time Control System Utilizing RDM and RDM+ Protocols for Communication**, *2nd International Conference on Future*

Networking Technologies (CoNEXT06), December 2006.

5. S. Khajoueinejad, M. Sabeghi, A. Sadeghzadeh, **A Fuzzy Cache Replacement Policy and Its Experimental Performance Assessment**, *IEEE International Conference on Innovations in Information Technology*, November 2006.

6. M. Sabeghi, M. Naghibzadeh, T. Taghavi, **A Fuzzy Algorithm for Scheduling Soft Periodic Tasks in Preemptive Real-Time Systems**, *1st International Conference on Systems, Computing Sciences and Software Engineering*,December 2005.

Samenvatting

Multi-coreprocessing platformen zijn een grote stap voorwaarts in het aanbod van high performance computing platformen. Het idee is om de prestaties te verhogen door meerdere verwerkingseenheden aan te wenden voor het uitvoeren van een taak. Dit creert echter een uitdaging voor zowel de hardware ontwikkelaars die zulke systemen bouwen, als voor de software ontwerpers die deze platformen programmeren.

Betreffende de hardware kunnen we de problemen noemen met betrekking tot interconnectie management, complexiteit van geheugenhirarchin en cache coherency. Terwijl, wat de software betreft, de meeste problemen zich voor doen in resource management, resource sharing en synchronisatie. Daarenboven is de onmogelijkheid dergelijke platformen met conventionele programmeermodellen te prrogrammeren een fundamenteel probleem aan de software kant. Dit komt grotendeels door de diepe kennis van hardware design die het programmeren van zulke platformen vereist.

In deze dissertatie pakken we de problemen aan de software kant aan door een alomvattend runtime systeem voor te stellen dat verantwoordelijk is de system resources te beheersen en alle conflicten op te lossen wanneer computing resources worden aangesproken. Bovendien biedt het runtime systeem applicatieontwikkelaars, APIs en System primitives, die platformafhankelijke details wegabstraheren en voorziet in een consistent programmeermodel. Deze primitieven koppelen het softwareontwikkelingsproces los van hardwareontwerp.

Het voorgestelde runtime systeem bestaat uit een scheduler, een profileerder, een transformeerder, een JIT compiler, een een bibliotheek van kernels. Een gedetailleerde beschrijving van de components wordt gepresenteerd. Zowel de prestaties van het gehele systeem, als de opgelegde overhead van de component is beschreven.

About the Author

Mojtaba Sabeghi was born in Mashhad, Iran, on July 16, 1981. He studied computer engineering at Ferdowsi University of Mashhad, Mashhad, Iran, where he received his BSc. and MSc. degrees in 2004 and 2006, respectively.

He joined Delft University of Technology in November 2006 as a PhD candidate, where his research work was carried out in the Computer Engineering Laboratory. His doctoral research focused on the runtime support for heterogeneous multi-core systems.

Mojtaba Sabeghi's research interests include Operating Systems, Programming Languages and Computer Architecture.

علاوه براین، سیستم زمان اجرا با فراهم آوردن واسطهای برنامه‌نویسی لازم، کلیه جزییات سخت افزاری را از دید برنامه‌نویس پنهان کرده و آنها را در سطحی بالاتر و به‌صورت انتزاعی در اختیار برنامه‌نویس قرار می‌دهد. برنامه‌نویس تنها در صورت آشنایی با این واسطها و بدون نیاز به هرگونه دانش تخصصی سخت‌افزاری می‌تواند برای هر نوع سیستم چندهسته‌ای ناهمگن برنامه‌نویسی کند.

سیستم در حالت ایده‌آل به گونه‌ای طراحی شده‌است که حتی در صورتی که برنامه‌نویس با این واسطها آشنا نباشد و برنامه‌نویسی را به طور کامل برای سیستم‌های غیر چند هسته‌ای انجام دهد، باز هم بتواند برنامه خود را روی یک سیستم چندهسته‌ای ناهمگن اجرا نماید.

سیستم زمان اجرای ارایه شده از اجزای مختلفی تشکیل شده‌است. این اجزا شامل زمانبند، پروفایلر، ترنسفورمر، کامپایلر زمان اجرا و کتابخانه می‌باشد. در این پایان‌نامه هر کدام از این اجزا با جزییاتشان ارایه شده‌اند. کارایی هر جز به همراه میزان سربار زمانی تحمیل شده به سیستم نیز مورد بررسی و تحلیل قرار گرفته‌است.

چکیده

ایده اصلی سیستم‌های چندهسته‌ای، افزایش تعداد واحدهای محاسباتی روی یک چیپ به منظور افزایش قدرت محاسباتی می‌باشد. افزایش سرعت در حقیقت با به کارگیری تعداد بیشتری واحد محاسباتی برای انجام یک کار مشخص صورت می‌پذیرد. این واحدهای محاسباتی می‌توانند از لحاظ معماری داخلی کاملا مشابه بوده و یا اینکه معماری داخلی متفاوت و بالطبع قدرت محاسباتی متفاوتی داشته باشند. در صورتی که معماری داخلی متفاوتی داشته باشند، سیستم‌های چندهسته‌ای ناهمگن نامیده می‌شوند.

واحدهای محاسباتی یا همان هسته‌ها، با همکاری و تعامل با یکدیگر هرکدام بخشی از محاسبات مورد نیاز را به صورت همزمان انجام می‌دهند و بدین ترتیب سرعت نهایی انجام کار افزایش می یابد. اما این همکاری و تعامل به سادگی قابل دستیابی نیست و پیچیدگی زیادی به سیستم تحمیل می‌کند. این پیچیدگی در هر دو بعد سخت‌افزاری و نرم‌افزاری بوجود می‌آید.

در بعد سخت‌افزاری می‌توان به افزایش پیچیدگی در مواردی مانند شبکه ارتباطی داخلی، سلسله مراتب حافظه و شفافیت حافظه کش اشاره نمود. در بعد نرم‌افزاری نیز افزایش پیچیدگی در مواردی مانند مدیریت منابع، اشتراک منابع و مسایل مربوط به همزمانی فرآیندها قابل مشاهده است.

اما مشکل اساسی‌تر برای استفاده از چنین سیستم‌هایی، عدم توانایی کافی برنامه‌نویسان در برنامه نویسی آنها می‌باشد. روش‌ها و ابزارهای متداول و سنتی برنامه‌نویسی به تنهایی پاسخگوی فرآیند برنامه‌نویسی این سیستم‌ها نیستند. در حقیقت، برنامه‌نویسی چنین سیستمی نیازمند دانش و تسلط عمیق بر مفاهیم سخت‌افزاری می‌باشد که اغلب برنامه‌نویسان از آن بی‌بهره هستند.

هدف اصلی پایان‌نامه پیش‌رو حل مشکلات نرم‌افزاری و برنامه‌نویسی سیستم‌های چندهسته‌ای ناهمگن می باشد. بدین‌منظور، یک سیستم زمان اجرای کامل و پوشا ارایه شده‌است که هدف آن مدیریت سیستم و منابع سیستمی می‌باشد. این مدیریت کاملا از دید برنامه‌نویس شفاف بوده و برنامه‌نویس به هیچ عنوان نیاز به دانستن چگونگی مدیریت توسط سیستم زمان اجرا را ندارد.

خدمات زمان اجرا

برای سیستم های چند هسته ای ناهمگن

مجتبی سابقی

TU Delft Delft University of Technology

CE

ISBN 978-90-72298-15-7