Vlad-Mihai Sima

Compiler Assisted Runtime Adaptation

Compiler Assisted Runtime Adaptation

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 30 januari 2012 om 12:30 uur

door

Vlad-Mihai SIMA

Master of Science
University Politehnica of Bucharest
geboren te Boekarest, Roemenië

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. H.J. Sips
Copromotor:
Dr. K.L.M. Bertels

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft, promotor
Dr. K.L.M. Bertels,	Technische Universiteit Delft, copromotor
Prof. dr. R. Tripiccione	UNIFE, Italië
Prof. dr. ir. K. De Bosschere	Universiteit Gent, België
Prof. dr. F.M.T. Brazier	Technische Universiteit Delft
Dr. A.D. Pimentel	University of Amsterdam
Dr. ir. L. Jóźwiak	Technische Universiteit Eindhoven
Prof. dr. C. Witteveen,	Technische Universiteit Delft, reservelid

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Vlad-Mihai Sima

Compiler Assisted Runtime Adaptation
Delft: TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica - III
Ph.D. Thesis Technische Universiteit Delft.

Met samenvatting in het Nederlands.

ISBN 978-90-72298-25-6

Subject headings: compiler optimizations, runtime optimizations, reconfigurable computing.

To my family and to the friends I have all over the world

Compiler Assisted Runtime Adaptation

Vlad-Mihai Sima

Abstract

IN this dissertation, we address the problem of runtime adaptation of the application to its execution environment. A typical example is changing the processing element on which a computation is executed, considering the available processing elements in the system. This is done based on the information and instrumentation provided by the compiler and taking into account the status of the environment. The work focuses on heterogeneous multicore embedded architectures. We address three aspects of application optimizations: hardware software mapping, memory allocation and parallel execution. For each aspect, an algorithm is developed and, using a suitable application, it is tested on the hardware platform. The programming paradigm on which this work is based is the Molen programming paradigm, extended and adapted for our specific platform and operating environment.

The hardware software mapping algorithm objective is to choose at runtime, on which processing element it is more efficient to execute a function. For the memory allocation we propose an algorithm, that using compile-time gathered information and the current execution environment, decides on the best allocation for memory, at runtime. For dealing with parallel applications we developed an algorithm that selects the best trade-off between area and speedup and decides on the number of concurrent units that execute.

The experiments were performed on an embedded multicore heterogeneous platform, namely the hArtes Hardware Platform (hHP). This platform contains an ARM processor as General Purpose Processor (GPP), an Atmel Magic Diopsis Digital Signal Processor (DSP) and a Xilinx Virtex4 Field Programmable Gate Array (FPGA). The applications used to validate the algorithms are real life applications from the multimedia field: a video encoder/decoder and a wavefield synthesis application. The mapping algorithms obtains improvements between 5% and 43%. We showed this is an adaptable algorithm, that will adapt the execution in case the execution overhead increases. The memory allocation algorithm obtained a speedup of 18% on the selected application. For this algorithm we show that the solution is within 14% of the optimal solution, computed using Integer Linear Programming (ILP). The scenario based selection of parallel computations, is between 21% to 92% better than existing solutions.

Acknowledgments

I would like to thank here all the people that encouraged me and supported me to pursue the difficult endeavor of obtaining a PhD. I would start with my university adviser Prof. Dr. Irina Athanasiu, which convinced me during my master thesis that I can and should continue my studies. Even if she can not be present among us today, she lives through her students, that gained with her not only technical knowledge, but also important life lessons. Next, I would like to thank Prof. Dr Stamatis Vassiliadis, whom, although I met only a couple of times, I feel had a great impact, even if indirect, on my path in life. From the first meeting I had in Delft with him and Dr. Koen Bertels, I felt inspired to come and help to solve some of the intriguing questions in our field. Fate decided that I will not work with him, but I am sure that his influence manifests itself through all the people that knew him.

Would like to give a special thanks to Dr. Koen Bertels, who supported me during this challenging period in my life. His dedication to work and ability to overcome all the obstacles were an example that I will always try to follow. I am grateful to everybody that helped me improve the quality of the thesis by providing comments and suggestion. This include the committee members: Prof. Dr. H. J. Sips, Assoc. Prof Andy Pimentel, Assoc. Prof. L. Jóźwiak. Then, my colleagues that helped in the various stages of the writing: Anca Molnos, Carlo Galuzzi and Ozana Silvia Dragomir. Also, a special thanks to the people that helped with the Dutch language: Roel Meeuws and Roel Seedorf.

For all the answers they provided me during the years, I would like to thank Dimitris Theodoropoulos and Lu Yi. For all the discussion related to various problems, I will thank Razvan Nane, Yana Yankova and Elena Moscu Panainte. Also, would like to thank for the support to Lidwina Tromp, Erik de Vries and Eef Hartman.

The work in this thesis was also made possible by the hArtes EU project, so a thank is due to all the great partners and to their efforts toward building a successful project.

My life in Delft would have been very hard and boring without the many good friends I made here. Would mention, in totally random order: Ana, Catalin, Aki, Bea, Li, Dejan, Noe and many others I would not list here.

The final thanks go to my parents who always supported me and encouraged me to start this adventure.

Vlad-Mihai Sima Delft, The Netherlands, January 2012

Table of contents

List of Tables

List of Figures

List of Listings

List of Acronyms and Symbols

ADAT	Alesis Digital Audio Tape
AMAP	Adaptive Mapping Algorithm
AMMA	Adaptive Memory Mapping Algorithm
AMMAe	Adaptive Memory Mapping Algorithm Extended
ARM	Advanced RISC Machine
BCE	Basic Configurable Element
CCU	Custom Computing Unit
CPU	Central Processing Unit
DCM	Digital Clock Module
DFG	Data Flow Graph
DMA	Direct Memory Access
DSP	Digital Signal Processor
DWARV	Delft Workbench Automated Reconfigurable VHDL Generator
ELF	Executable and Linkable Format
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
gcc	GNU Compiler Collection
GPP	General Purpose Processor
GPU	Graphics Processing Unit
hHP	hArtes Hardware Platform
ILP	Integer Linear Programming

JTAG	Joint Test Action Group
KPN	Kahn Process Networks
LLVM	Low Level Virtual Machine
MAL	Molen Abstraction Layer
MEMAL	memory allocation lists
RISC	Reduced Instruction Set Computing
RSSA	Runtime Scenario Selection Algorithm
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random Access Memory
SD	Secure Digital
SIMD	Single Instruction Multiple Data
SPM	Scratch Pad Memory
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
VLIW	Very Long Instruction Word
WFS	Wave-Field Synthesis
XREG	Exchange Registers

1 Introduction

EMBEDDED systems are a constant presence in our daily life. These systems support and fulfill essential roles in our modern society. From transport infrastructure, to providing communication facilities, allowing doctors to make better diagnostics, entertain people, almost any aspect of our life is related to them. Embedded electronic devices proved to be an invaluable asset in many aspects of life. Starting from the first microprocessor, the Intel 4004 developed in 1971, the spread of such devices has been exponential. The exponential use is mirrored by an exponential growth in the number of transistors present on a chip. This number doubled almost every 2 years as predicted by the Moore's law - an empirical law postulated in 1965 by the Intel co-founder, Gordon E. Moore. To take full advantage of the large number of transistors and, thus, the available resources, applications constantly evolve and have higher computational demands. Every year, new applications are developed, which require more and more computation power, as, for example, newer video and audio protocols, executing on smaller devices, that consume less power. To show a representative example, nowadays, the volatile memory available in a mobile phone is in the range of hundreds of megabytes, where, 20 years ago, the memory available for a standard desktop system was in the range of megabytes.

The increase in frequency played an important role in the overall performance improvement. The increasing number of transistors was exploited to increase the memory capacities and to move the memory closer to the processing units. Nevertheless this trend is changing: due to the physical limitations it becomes much harder to further increase the frequency. Other paths to improve performance have to be explored by the engineers developing the embedded systems of the future. In modern devices, the main direction of development is heterogeneous multicore computing where, one or multiple specialized cores are used in order to outperform by orders of magnitude the performance of a single core General Purpose Processor (GPP). One of the disadvantages of heterogeneity is that each core has to be designed with an application field in mind. Tasks with a high level of parallelism can take advantage of the multicore and increase by orders of magnitude the throughput of the system. Of course, some platforms might choose to implement only heterogeneous computing, with dedicated cores for each type of task, while others might choose to implement only homogeneous multicores. *It is the author's belief that by combining the two approaches, the best results can be obtained.*

Performance is not the only aspect which needs to be taken into consideration when dealing with embedded systems. Some systems have high deployment costs and may require additional infrastructure work that can not be performed often. Examples are rail signaling systems and surveillance systems in buildings or metro among others. As a result, upgrade capabilities of these systems need to be taken into account at design time as well. Traditionally this was a task accomplished by software upgrades, relying on the possibilities of the GPP, which is many different tasks with a good level of performance. If software upgrades are not possible, a newer system needs to replace the old one. Over the last years, Field Programmable Gate Array (FPGA)s [75] have become very popular and can be used to address this problem. By using this technology, hardware components can be upgraded in place, extending in this way, the life span of existing products. Additionally, this improves reliability, as bugs in the hardware design can be corrected even after the deployment of the system.

The development of software for such systems would be very hard without the availability of tools and operating systems. The role of tools is to help the developer to focus on the relevant aspects of the problem, while the tools take care of all the specific details related to the platform in use. During the development of a product, many aspects play an important role. For example, the translation of the high-level description of the problem to an implementation, the verification of the implementation, the profiling of the application. For each of these tasks, tools have been developed to ease the work of the developers. At runtime, it is the responsibility of an operating system to manage the various resources and, possibly, the multiple applications running concurrently. This further increases the level of flexibility during the design of applications, and it facilitates even more the work of the developers.

In this thesis, we propose improvements to the interaction between the compile-time tools used by developers, the operating environment and the hardware platforms. The purpose of these improvements is to take full advantage of all the possible optimizations, that might be hidden if the analysis is performed only at compile-time, only for one hardware platform or only by the operating system.

1.1 Problem Overview

Existing computing platforms come in families. This is mainly due to the considerable efforts involved during the development of a platform, and different needs for different computing areas. These families share a general architecture and processing elements, but are customized for different scenarios. For example, some of these platforms are low-cost models with less functionality, or they are hardened platforms, which can work in a wider temperature range. Having a family of platforms has both advantages and disadvantages. An example of such an advantage is that the expertise for developing for a family of platforms can be reused. A disadvantage is that the process of choosing a variation and then the mapping of an application becomes more complex. Some examples of variations that can make the mapping problem more complex are: the presence or absence of a specific processing core, the different operating frequencies of the various components, the different sizes for memory and/or caches, and the different input/output ports available.

The developers, usually divide the applications in several parts and assign each of them to a processing core, while managing the communication needed between the different elements. This process is usually based on an analysis performed on a specific platform variant and with a particular data set. The tools provide guidance for each aspect of the process and allow the designer to perform critical choices. Usually, the adaptability to platform variants is not the primary concern. This happens because at the moment of development time, the different variants might not even be available on the market. All of this can result either in an increased development time or in an inefficient use of resources. Additionally, in dynamic systems, the interaction between applications is a deciding factor in the overall system performance. This can be managed manually, by the developer or the system administrator. Special policies can be implemented in the operating system to cope with each type of situation. Anyhow, with a manual approach, such efforts are targeted to instances of the problem and do not give a comprehensive solution. A more comprehensive solution would be to provide tools producing applications which adapt to the platform and the execution environment. By using compile-time information, these tools can improve the efficiency and they can reduce the time spent in the development process.

In this dissertation we will address specific sub-problems of the embedded systems development. These are specific because of the architecture choice (heterogeneous) and optimization methods used (runtime). A high level overview of these problems is:

- Mapping - what is the most efficient resource to map a computation on, taking into account the current state of the execution environment?
- Communication - how can we reduce communication between the various processing elements?
- Parallelism - how can we exploit the parallelism provided by heterogeneous platforms?

Each of these problems will be outlined in more details in Chapter 3. Besides these problems, the field of embedded computing faces several other problems among which we would mention scheduling, energy consumption optimization, data representation optimization, choosing the model of computations, etc. Although these problems are important for the field, we focused on that set of problems that were the most stringent for our platform and applications.

It should be noted that the work in this thesis was done in the context of hArtes project [16], a project focused primarily on performance and toolchain development rather than, for example, power efficiency. The hArtes project was an Integrated Project funded by the European Union, whose purpose was to lay the foundation for a holistic approach used when developing complex embedded systems. A high-level sketch of the content of hArtes project is given in Figure 1.1. The work in this thesis influenced mostly the modules/tools represented in green.

1.1.1 Dissertation Scope

An application can be optimized in many ways. In this dissertation, we only focus on optimizations involving compile-time analysis coupled with runtime decisions. We consider that a platform can change in time and thus runtime optimizations decisions are beneficial. The platform changes could include hardware upgrades or additional software installation. As we focus on embedded devices, the runtime decision algorithms that we develop have to run in a constrained environment and, therefore, need to be computationally fast. Summarizing, we focus on algorithms with the following chrachteristics:

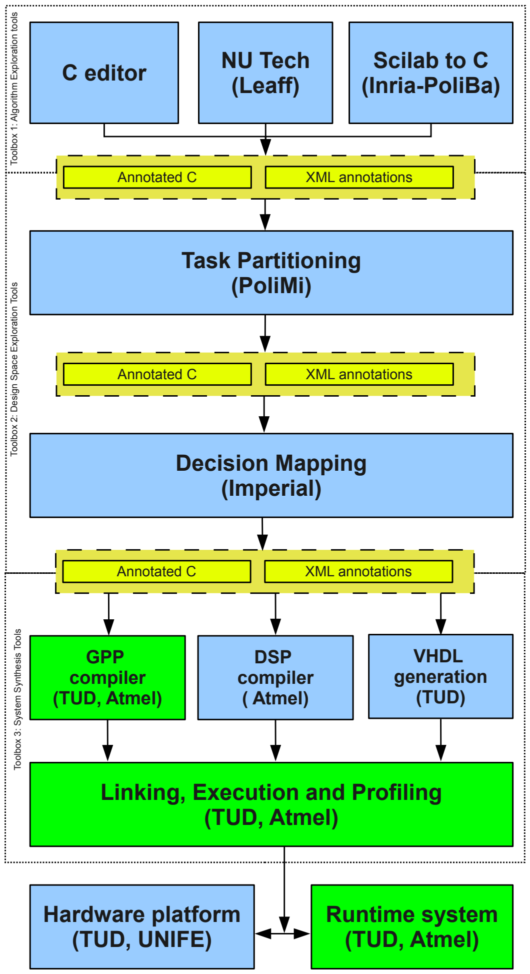
Toolbox 1: Algorithm Exploration tools
C editor
NU Tech (Leaff)
Scilab to C (Inria-PoliBa)
Annotated C
XML annotations
Toolbox 2: Design Space Exploration Tools
Task Partitioning (PoliMi)
Annotated C
XML annotations
Decision Mapping (Imperial)
Annotated C
XML annotations
Toolbox 3: System Synthesis Tools
GPP compiler (TUD, Atmel)
DSP compiler (Atmel)
VHDL generation (TUD)
Linking, Execution and Profiling (TUD, Atmel)
Hardware platform (TUD, UNIFE)
Runtime system (TUD, Atmel)

Figure 1.1: The hArtes project toolchain and platform

- The algorithms need a significant analysis of the application at compile-time, which can only be performed by a compiler. Examples are, the identification of parameters, that influence the control structure of the function, or parameters only used to access data.

- Optimization decisions can not be taken at compile-time. This can happen due to multiple reasons. We outline the main two reasons in the following. First, the execution time of a function cannot be completely determined at compile-time, where, for example, loop bounds or conditional flags are not known. Second, the execution environment can change at runtime, in a way that is not easily controllable. Examples of this are multiple applications running concurrently, hardware changes in the platforms, or changes in operational parameters (battery level).

- The algorithms are affected by the heterogeneity of the platform. Examples of this include different types of processing elements or non uniform memory hierarchies.

One set of applications which benefit from the optimization algorithms with the above characteristics are complex media applications that run on embedded systems, for example in-car audio and video systems.

Additionally, for experimental purpose, the validation of the proposed algorithms assumes the following:

- The platform used follows the Molen machine organization (outlined in Chapter 2).
- The tools used to compile the applications for such a platform adhere to the Molen programming paradigm.
- Access to the source code of the compilers and operating system of the platform is available.
- The applications can be executed on the target platform and profiled using real data.
- The designer can manually control the tools to a certain extent assuming a semi-automatic procedure.

We furthermore exclude from the scope of the thesis a number of topics, not because they are not important but would make the complexity of the problem unmanageble. More specifically we do not look at: hardware/software co-design, partitioning, run-time reconfiguration area management, scheduling, power optimizations, etc. Some of these issues were studied in the hArtes project but were never included in the final toolchain. This thesis focuses primarily on some of the operational and implementable results of the hArtes project.

1.1.2 Contribution of the thesis

The main contributions of the work proposed in this dissertation are the following:

- The definition and development of a mapping algorithm able to identify on which processing element a particular computation can be computed in the shortest amount of time. Compared to existing mapping algorithms, the mapping decision is delayed until runtime, when, by using data gathered at compile-time, a decision that will minimize the execution time can be made. We will show later in this dissertation that this organization allows an algorithm to take advantage of some optimization possibilities lost in case the analysis is performed only at compile-time, even if the analysis is done by an expert developer. In terms of experimental results, the application of the mapping algorithms results in improvements between 5% and 43% for the whole application, when compared to a static decision algorithm which can only decide for a specific computation to either run it in hardware or software. The wide variation is due to the fact that the application execution time is strongly linked to the input data, for example, in our case, videos containing a variable level of motion. Additionally, we outline the behavior of the algorithms in case the characteristics of the platform change at runtime. The characteristic that we analyze is the overhead involved in invoking a processing element. This situation might arise if, for example, other applications use the same communication resources between the GPP and the FPGA. Our finding is that the obtained performance gradually degrades and it is close to the maximum performance achievable given the circumstances.

- The development of a memory allocation algorithm targeted to heterogeneous platforms. This algorithm uses the application execution history and the characteristics of the computations, to decide, at runtime, the best memory allocation in the current memory hierarchy.The use of the memory allocation algorithm results in an overall speedup of 18% of the H.264 video encoding application. We present two variations of the same algorithm. The first can be executed very fast, while the second provides better application performance. By using synthetic applications, we have shown that both algorithms are within 14% and respectively 5% of the optimal solution, computed using an ILP approach for the same problem. The application speedup obtained for the synthetic application was between 2x and 6x.

- The development of a scenario based selection algorithm for parallel applications. This algorithm decides which combinations of parallelism is better, in case multiple applications or threads are competing for the same resources (i.e. FPGA area). For the selection of the scenario used in a system with multiple threads applications, we first give an ILP formulation of the problem. Then, with the ILP as a reference we give a runtime algorithm which is within 7% of the ILP solution and is better than existing solution by 21% to 92% in terms of application performance.

- The description of the integrated toolchain that uses these algorithms to improve performance on a heterogeneous platform, based on the Virtex4 FPGA. The problems that appeared during the development of such a framework are discussed and the chosen solutions are presented in detail.

1.2 Dissertation Organization

The work proposed in this dissertation is organized in chapters. Chapter 2 analysis the existing related work. The three main domains for our work, namely computing platforms, toolchains and execution environments, as well as the Molen programming paradigm and machine organization are described in detail. The chapter continues with the analysis of the mapping and the memory allocation algorithms. Finally, we present the work related to the parallel execution of applications on heterogeneous reconfigurable architectures.

In Chapter 3, we present the work done in the context of the hArtes project. The work includes supporting the development of a custom heterogeneous board and developing a toolchain that supports such a board. First, we present the hardware platform and then continue with the essential ideas behind the toolchain. We will also present a thorough analysis of the applications used as motivational examples for the algorithms presented in this thesis. These are real-world applications, and the evaluation process is the same process that was used during the development of the hArtes toolchain. All the information is collected at runtime, by using realistic input data. Finally, the chapter provides a list of problems that will be addressed in the following chapters.

In Chapter 4, we present a mapping algorithm, which decides if a computation should be performed on the GPP of the platform, or on any other of the processing elements. The computations used as a motivational examples are dynamic by nature and their execution time varies a lot depending on the parameters with which the computation is called. These are not the typical applications that were implemented in the embedded systems in the past, but with these devices becoming more and more used, more high level applications have to be targeted using automated tools to embedded platforms. Building flexibility into applications lowers the development cost associated with re-targeting to a new platform. Based on the execution time obtained for various input data, a runtime module decides if a function should be executed either on one co-processing element or on the GPP. To do this, it uses a 'software' cache in which the execution time for each processing element is stored. Then, based on this information, it makes the decision on where to run the computation. The benefit of such an approach is that it can adapt to various conditions changes, like, for example, frequency lowering of a processing element in case of low power, or even a complete shutdown of some of the processing elements.

A runtime memory allocation algorithm that relies on instrumented code is presented in Chapter 5. When allocating a memory block, for a complex memory hierarchy there can be several memories where the block can be allocated. The decision of which memory to place the block to obtain the best performance is not an obvious task. The memory block will be used during the application execution by one or more computations. At their turn, each of those computations can use multiple blocks. For each computation, multiple implementations for different processing elements can be available. However, in order to run on some processing elements, all the used memory blocks have to be in the scratch pad memory of the element before the computation can start. The result is that transfers between memories have to be performed during the execution, which will affect the total execution time. By tracking at runtime the memory used in each computation, we can determine an efficient way of allocating it to the various scratch pad memories. By using a simple persistence mechanism, we can improve subsequent application executions by performing future allocation, based on the saved profiling data.

We continue in Chapter 6, by presenting the issues that emerge when trying to map an application with multiple threads to a heterogeneous reconfigurable platform. An important difference from a platform with multiple identical cores is that, for a reconfigurable platform, the number of cores and memory organizations can be different and vary over time.

Conclusions are presented in Chapter 7, where we summarize the main contributions of this thesis and we discuss the relation between the presented algorithms. Finally, we propose a list of open questions and future research directions.

2
Background and Related Research

IN this chapter, we define the research context of the work presented in the following chapters. More specifically, we discuss the relations between the problem analyzed and the relevant concepts involved, such as platforms, toolchains and operating systems. We present the machine organization and the programming paradigm on which the following chapters are based on. Finally, we present the related research in the field.

2.1 Research context

The problems presented in the following chapters are part of a complex research context, which in the following is discussed from three points of view: the hardware platforms, seen at the level of processing elements and memories, the applications that have to run on those platforms and, finally the toolchain which has to transform the application into binary code suitable for a platform. We address each of these aspects separately.

2.1.1 Computing Platforms

Computing platforms have evolved in time from basic platforms with Central Processing Unit (CPU) and one type of memory to complex systems with multiple heterogeneous components inter-connected by various elements such as buses and networks-on-chip. These heterogeneous components fulfill many roles, and, usually provide an interface between the various peripherals while increasing performance. A classical example of components used to increase performance are the Graphics Processing Unit (GPU)s [26], which represent a highly customized type of Single Instruction Multiple Data (SIMD) processors. These processors are specialized for different types of computations which they perform much faster than to classical CPU. Another hardware component used for processing is the Field Programmable Gate Array (FPGA) [75]. One of its biggest advantages over hardwired circuits is its capability to change the hardwired connections even after it has been shipped to customers. This means improvements and bug fixes are possible both for the hardware and the software part of the system. Over the years, memory systems evolved as more transistors could be integrated easily on the same die. Caches became larger and multilevel cache hierarchies appeared [70] [33]. Nevertheless, the increase in number of transistors had also made communication inside the chips a serious bottleneck [89]. One solution to this problem is the development of networks-on-chip capable of providing support for fast data transfers [76]. Another solution, used also in this thesis, is to optimize the memory allocation and thus reduce the need of transfers.

The Molen Machine Organization

Well-known problems for heterogeneous computing platforms are the integration of different components and the programmability of the system. In our research, we are following the Molen machine organization to address these issues [82]. This organization is generic and not restricted to any particular architecture. Its main purpose is to allow a virtually unlimited number of extensions of a base General Purpose Processor (GPP) to be implemented with the least amount of effort compared to redesign manually the system to include each new extension. The Molen machine organization is presented in Figure 2.1.

This architecture is build on the processor - coprocessor model. A core processor, which is a GPP, executes the programs and, for certain computations, it invokes other processing elements. In Figure 2.1, we see a generic processing element that could be, for example, a FPGA or a Digital Signal Processor (DSP). The ”Arbiter” block represents the component responsible for the identification of the special Molen primitives, and it is responsible for redirecting them to the processing elements. An Exchange Registers (XREG) block is tightly connected between all the processing elements and the GPP for fast data exchange. The tightly integration of the XREG block enables a fast communication path between the GPP and the different processing elements, avoiding the high latencies imposed by the memory accesses.

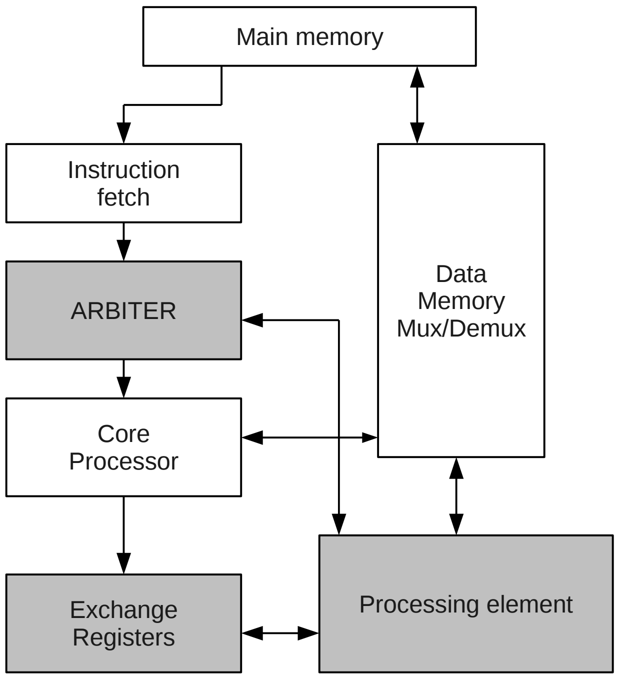

Figure 2.1: The Molen machine organization.

2.1.2 Toolchains

The notion of a tool to help developers to use computing system appeared as early as 1954 [7]. Compilers were the first tools that appeared. The compiler's task is to translate the code from a high-level description to a low level description, specific to a machine or platform in use. Later, as the development process become more complex, monolithic tools were divided in sub-tools addressing specific tasks. They evolved, for example, into frameworks for life-long program analysis and transformation as, for example the Low Level Virtual Machine (LLVM) [52], compilers that target multiple architectures as the **GCC!** (**GCC!**) [34], assemblers, and linkers among others. The connection of all these tools together, resulted in the creation of a toolchain. Nowadays more and more tools are added to toolchains, in order to improve the development process. With increasingly complex architectures and applications, the toolchains are seen today as one of the key enabler of a specific platform and even architectural paradigm [47].

The traditional compilation toolchains consider the architecture to be fixed, without significant changes during the lifetime of the product. With the current platforms, this is not always the case. The best examples are the reconfigurable architectures in which most of the components of the platform can change in different ways. The development of separate toolchain for each possible combinations is not a viable approach. On the other hand, the development of a toolchain for a specific platform means that optimization opportunities are limited. By loosing optimization opportunities, the platform can not be fully utilized, which, in turn, means that resources are lost, and the economic viability of the whole system is less than possible. A solution to these problems is to extend the current toolchains in such a way that adaptability to different platforms can be taken into consideration. This, usually, involves modifications at multiple levels. For example, by inserting new annotations at the programming languages level, or by providing a linker able to link the different processing elements.

One partial solution to the problem of developing tools for different architectures and paradigms are the reconfigurable compilers frameworks. These are compilers frameworks that allow developers to build compilers for a new architecture in a short amount of times. In this category we mention the open source compiler **GCC!** [34] which targets a large sets of architectures, the CHESS compiler which focuses on fixed point DSP processors [80], RECORD compiler [55] and CoSy compiler development system [24].

The Molen Programming Paradigm

The Molen programming paradigm was developed in order to efficiently exploit the Molen machine organization. This programming paradigm relies on five primitives that link the processor of a system with the heterogeneous cores [82]. Although special consideration was given to FPGAs in its initial design, the paradigm can be successfully employed for other types of processing elements as well as as shown later in Chapter 3. The programming paradigm relies on the sequential consistency model for memory access.

A set of predefined instructions added to the instruction stream of the GPP manages the computations executed on the different processing elements. These five primitive instructions are SET, MOVET, MOVEF, EXECUTE and BREAK. The original proposal included two SET instructions, namely a partial SET and a complete SET. Anyhow we consider them as one entity as their role is similar. In the following, we summarize in short the role of each instruction:

- SET - it configures/loads the computation on the processing element. This is an information the runtime system can use to prepare in advance for the actual execution;
- MOVET/MOVEF - it transfers the parameters/result to/from the XREG.
- EXECUTE - it starts the execution.
- BREAK - it checks if the execution has finished.

A detailed discussion on these primitives in the context of research is presented next, in Chapter 3.

2.1.3 The Applications

Another aspect that has to be taken into account during the analysis of the problem is represented by the applications, which are the input of toolchains. Despite a lot of research in the field of programming languages and modeling languages many applications for embedded systems are still written in plain C [83]. This is because the programmers need a lot of control on the inner workings of the system, to obtain performance and to adapt to new platforms, without relying on complex and, possibly, unreliable middle-ware. The companies are reluctant to rewrite the code in a new language that works for some niche platforms as the adaptation of the legacy code to new architectures can be problematic.

The C language was developed between 1969 and 1973 as a programming language for system software [74]. Due to the fact that at the moment of its conception the architectures were rather different from today architectures, two concepts are totally missing from C, which are needed when porting an application to modern embedded systems. These concepts are mapping (of memory and processing) and parallelism. This can be seen as an advantage as programs remain generic, but also as a disadvantage as programs must be adapted to meet performance. Additionally, even if not present in the language, several solutions exist to augment existing C applications with this kind of information.

To summarize in one phrase our problem we can say: how can we implement complex C applications on heterogeneous platforms, which may evolve in time, without loosing flexibility or performance?

As porting an application to a new platform involves many problems, in the following we limit our analysis to a limited subset: partitioning, memory allocation, and scheduling. Our solutions are specific to each objective discussed, although the main idea is the same for all: to integrate parts of the optimization algorithms in the application and to delay the decision-making process, when possible, until runtime. Then, when the exact architecture is known, the best decision can be taken giving the best performance for the application.

2.2 Related Research

Recently, frameworks [25] [3] were proposed to make applications more adaptable to the platforms. The framework presented in [25] proposes a system that follows the cycle ”sense, reason, react”. The system is organized as a decision engine coupled with software and hardware components. When using an FPGA, the framework can hide the details of the implementation from the developer. Examples of such implementation detail is the partial reconfiguration capability. Nevertheless, the exact decision engine has to be implemented on a per application basis. One such example is presented in [29], where a pulse detection system uses the reconfigurability of the FPGA to adapt to changes in the environment. The framework presented in [3] adapts the thread model to heterogeneous embedded systems by defining a Hardware Thread Interface. This type of interface enables platform independent user level semantics that promote thread migration across the system.

Independent of the frameworks mentioned, there is a large volume of research related to the problem this dissertation addresses. Classifying this work will help in providing a clear image of the related research. As our work is focused on runtime optimization, we consider a classification based on the moment at which the optimization is done, coupled with a classification based on the area in which the optimization is applied.

From a time point of view, optimizations can be divided in off-line (compile-time) optimization and on-line optimization. Then, according to the problems defined in the previous section, we have related work dealing with the memory optimization in embedded systems and we can divide this optimization into memory related problems (cache and scratch pad memory), partitioning problems and parallelism and scheduling problems.

2.2.1 Memory Related Problems and Solutions

An ideal processing element would have an infinite amount of memory with instant access. However in reality, as the access time of the memory decreases, its cost increases and size decreases. The typical scenario is that in a system, the fastest memories are the smallest [39]. Various mechanisms are used to map parts of the application data to the memories present in the system. In the following we will explain the techniques used and their application to the case of reconfigurable heterogeneous hardware.

Cache Memories

Cache memories are a mechanism used to hide the long access time needed to access external memories [39]. This is achieved by relying on the spatial and temporal locality of the data accessed by a program. One of the biggest advantages of the caches is that they improve the execution time in a transparent way. The programmer does not need to make any modification in the source code of the application to take advantage of a cache. This advantage comes at the cost of a larger area occupied on the chip.

When used together with an FPGA, caches can enable speedups of up to 7.8x (geometric mean) and up to 4.1x reduction in power when compared to a general purpose processor [71]. This is achieved by performing a static alias analysis on the code, with the help of the 'restrict' keyword. The obtained results are good as the authors target HPC applications that do not contain complex aliasing. As the analysis is performed at compile-time and relies on static analysis, it is not able to deal with the complex situation encountered in our test applications. In our case, depending on the input data, different memory blocks are used during the program execution.

When compared to scratch pad memories, caches can be less efficient in terms of power and performance. This is shown in [9], where, by using an instruction simulator and power estimator, it is shown that Scratch Pad Memory (SPM) are on average 40% more energy efficient than caches. This is a an implication of the fact that a cache contains more transistors compared to an SPM of the same size, needed to implement the various extra logic, for example, tag arrays.

Virtual Memories

Virtual memory is a technique invented to manage multiple levels of memories, especially when a program can not fit completely in the first level of memory. It organizes the memory in pages and provides address translation between the virtual addresses used by the processor and the real address where the data can be found. Other advantages of virtual memory are protection of processes and sharing of memory and program relocation [39]. For Virtex and Altera platforms, in [86], Vuletic et al. present a unified virtual memory manager. This manager allows both hardware and software processing elements to use the benefits of virtual memory when accessing memory.

Scratch Pad Memory

SPMs have the same function as caches, although they are managed by the application developer/toolchain directly. If they are used efficiently, they perform better than caches in speed, energy efficiency, and in the area occupied [2] [9].

For the specific case of loop nests, where dependencies can be analyzed at compile-time, [48] provides an algorithm to design a custom SPM hierarchy. In case the memory hierarchy is already given, an algorithm that transforms the loop to better fit the hierarchy is presented. Both algorithms rely on the availability of information about dependencies in the loops, and can not be used for generic C code.

For specific cases, static solutions exist which provide significant performance improvements (more than 6x) by using a careful allocation of memory resources on the FPGA [13].

A low level partitioning algorithm between several SPM memories of the system is presented in [92]. The application model is a Data Flow Graph (DFG), where each node represents a coarse grained task. A special edge is inserted in the graph to model a loop carried dependency. Each node accesses a set of variables. By using two algorithms, HAFF (High Access Frequency First) or GVP (Global View Prediction), the algorithm maps each variable to one of the SPMs in the system, and then schedules the tasks in the DFG on each processing element. Additionally, they utilize a loop pipeline scheduling algorithm (RSVP) to reduce dependencies for scheduling. However, in the paper they do not consider heterogeneous architectures, and external memories are not included in the presented partitioning and scheduling schemes.

A big part of the previous work on SPMs, usually considers just one processing element, like [6], which gives an Integer Linear Programming (ILP) formulation of the problem of mapping program variables to the SPMs. The first class of variables considered is the class of global variables. Two variants of the algorithms are presented for the stack variables. The stack variables differ from the global variables because they have a limited lifetime. The first variant of the algorithm considers the stack of one procedure as one ’aggregated’ variable. The second approach allows each variable to be allocated to a different memory. The authors compare the execution time using a SPM capable of holding all the application variables to the execution time using an SPM 20% smaller than the previous one. The execution time for the small SPM is only 1.5x bigger compared to the use of a bigger SPM, which shows the effectiveness of presented techniques.

[40] presents a static method to determine the memory bank where a variable should be placed, based on the number of accesses and conflicts with the other variables. The proposed algorithm supports a complex memory hierarchy, which can be composed by a combinations of caches and application managed SRAMs. For caches, a 90% hit rate is assumed by the partitioning algorithm. In a similar way to the previous approach [6], the information used as input to the partition algorithm is provided by a detailed profiling, which gives the number of accesses for each variable. For the studied application the authors test that by changing the input data, the results are not affected more than 5%. Nevertheless, this does not hold for any application, as we will show in Chapter 3.

In [79], an algorithm is presented that determines the program points where local (stack) variables have to be transferred to/from main memory to the SPM. Program points candidates are the start of each procedure and the start of every loop in the program. Timestamps are given to each node, based on the order of their execution. Next, by using an iterative algorithm, the set of variables that are brought to the SPM and the set of variables that are written back to the DDR are determined for each program point, based on a cost model. This algorithm does not take into account dynamically allocated memory, and it does all the computations at compile-time. If the system configuration changes, the algorithm has to be applied again and the application recompiled.

In contrast to previous work, [14] instruments just the dynamic memory management primitives, such as *malloc* and *free*, along with all accesses to memory. Then, the developers collect data by providing the application with relevant inputs. An algorithm identifies block transfers and determines for each block transfer if it is better to perform it using DMA or using normal memory transfer. The disadvantage of this approach is that it targets only dynamic variables and it relies on static information.

A method at the boundary between static and dynamic allocation is presented in [62]. The authors perform a load time optimization that places the stack data in one of the memories using information computed at compile-time. The method starts with a profiling phase where, for each variable, a frequency per byte is measured using multiple input sets. The second phase consists of compiling and linking, which do not embed the actual variable addresses in the code, as these will be determined at runtime. The last phase is represented by the embedded loader, which determines the size of the SPM and, then, decides where each variable should be placed.

A dynamic SPM stack manager that reduces power consumption is presented in [49]. This manager operates at the granularity of function stack frames. The main advantages are that it does not need profile information, and it can adapt to different SPM sizes. Based on the stack frame sizes and on the SPM size, it can determine for which function calls the SPM stack manager calls needs to be inserted. These calls manage the swap-in/swap-out of stack frames to/from the SPM memory. Another dynamic SPM manager is presented in [23]. This work targets multi-application environments, but assumes there is only one processing element.

By providing both the heap and the stack management, [22] describes an iterative algorithm that determines the allocation of stack, global and heap variables and introduces transfer code when necessary. The method uses only compile-time heuristics to determine which is the best place to allocate blocks and where to transfer them from fast SPM to slower DDR.

In [65], the authors focus on multiple applications that share a SPM. The architectural abstraction is represented by a multi-core system, with identical cores that share a SPM. The problem is that the SPM must be shared between all the applications running in the system. To solve this, they present an approach, which relies on compile-time information to decide how to divide the SPM. The compile-time information is represented by the loop type and the frequency of use for each array. At runtime, based on the system status and on the amount of space requested by each application, the SPM manager divides the existing space among the applications. A second level of distributing the space is performed in the application, which distributes the space between the arrays. This work considers identical computing cores, so it can not be used if differences in execution times exist between cores.

All the solutions presented so far are software based solutions. The other possibility is to implement management solutions directly in hardware. These solutions are similar to cache solutions, although the granularity at which the management of memory is performed is coarser than for caches.

For example, [69] uses a MMU to manage the mapping to SPM of the stack of the program. The MMU tries to keep always mapped to SPM the top of the stack. For this purpose, it uses a fault mechanism. The memory outside the range mapped to SPM is protected and the MMU receives a fault if any of these addresses are accessed. Depending on the type of access, the SPM performs a transfer or a new mapping. This solution works for an architecture with a single processor and it heavily relies on the idea that the most accessed data is the one present in the top stack frame.

In [18], a hybrid system is proposed composed of SPMs and data cache. By using profiling, a dynamic call graph of the application is obtained, together with the number of memory accesses. ILP formulations are then used to determine which memory pages should be placed in the SPM to minimize the energy consumption. This information is inserted before and after each function call and, at runtime, a SPM Manager loads the appropriate pages. The conclusion of the authors is that the best combination of data cache size, SPM size and page size is dependent on the application.

The addition of a DMA engine together with high-level functions to access scratch pad is proposed in [28]. This method has the advantage that the processor does not need to waste cycles transferring data from the main memory to SPM. A big disadvantage is that it requires both a rewriting of the applications and a change of the hardware platform.

2.2.2 Partitioning Related Problems and Solutions

Partitioning is the process in which it is decided, either by the toolchain, by the runtime or by the developer, which parts of the applications are executed on which processing elements. The processing elements are either GPPs that execute software programs or hardware accelerators. Of course, this involves more than just deciding the processing element on which a function will execute as, for example, that function needs its data in an appropriate memory, the processing elements execution needs to be controlled, etc.

An important part of the work done in the hardware/software partitioning field considers static partitioning done at compile-time. A mathematical formulation of the problem is presented in [4]. The authors formulate two problems, by using a graph to represent the application where the hardware and software costs are associated to the nodes and the edges represent the communication costs. The first problem consists of mapping each node to software or hardware while minimizing the total cost, seen as a linear combination of the hardware, software, and communication costs. This problem can be solved in polynomial time. The second problem consists of mapping each node to software or hardware, while minimizing the total cost, and keeping one of the cost below a set bound. This is showed to be a NP-complete problem and two heuristics are proposed to solve the problem. The results of this work can not be directly applied in the general case as both the model of the application and the platform are generic. For example, it is not always possible to model a hardware node using only one cost factor, as a hardware node in a reconfigurable system has multiple characteristics such as area, energy usage, and performance.

Simulated annealing was used as a heuristic in [12] to solve the hardware/-software partitioning problem. In this work, the application is modeled as a direct acyclic graph. The platform is modeled as a software processor and a hardware unit connected by a system bus. One important limitation of this architecture is that the authors assume that the two processors can not execute simultaneously.

Ant colony optimization was used in [87] to partition direct acyclic graphs to heterogeneous multicore platforms. The optimization objective used is critical path execution time of the task graph under the constraint of fixed area.

In [59], a simulated annealing approach is used to determine the spatio-temporal mapping of an application onto a heterogeneous architecture. The application is also described as a directed acyclic graph. The architecture is a typical processor/co-processor architecture, where the processor is represented by the reconfigurable array. Additionally, the authors assume that the reconfigurable array supports contexts to speedup reconfigurations. The existence of contexts implies the need of a temporal partition.

[57] uses as an input an application represented as a directed acyclic graph, and then creates a hierarchical decomposition on it in order to speedup the mapping algorithm execution.

The problem of allocating the area inside the FPGA is addressed in [68]. By assuming that multiple operations need to be configured in the FPGA, and the sum of their areas is greater than the total available area, some of them need to be reconfigured. This problem is formulated as an ILP problem, with the objective of minimizing the reconfiguration area. This has the effect of improving the total execution time as, in such a system, the reconfiguration time represents an important overhead. In this work, the communication costs are not considered during the mapping.

By taking into account more aspects of the system, [61] solves the mapping problem using a mixed ILP algorithm. The application is represented as a Hierarchical Control Data Flow Graph that, besides data and control dependencies, includes timing constraints. The architecture is considered to be highly heterogeneous with FPGAs, DSPs and GPPs. The objective of the mixed ILP formulation is to minimize the resources employed, while satisfying the timing constraints. Dealing with two types of reconfigurable fabric, the work in [31] proposes a mapping algorithm that relies on static analysis and dynamic analysis to map kernels to either an FPGA or a coarse grain reconfigurable array. As other works, once a decision is made at compile time, it can not be adjusted at runtime.

Another approach that minimizes the resources used while satisfying constraints of execution time and power consumption is presented in [46]. In this work, by using a simple architecture model, where software and hardware can not work in parallel, the knapsack algorithm is used to determine the best area utilization, given the execution time and power constraints. An algorithm, based on dynamic programming that computes the exact solution, is also presented. The disadvantage of this approach is that it can only be applied to small problems as, for bigger problems, the memory used by the algorithm is larger than the available memory in current computer systems.

Different other problems were considered when doing the partitioning, such as area allocation [68], granularity selection [38], and scheduling [59] [36].

A common characteristic for all these approaches is that they rely on the fact that the profile information and execution trace are available at compile-time and they optimize just for one specific set of cases [68] [59] [38].

From the runtime and operating system point of view, the work in [20] focuses on online scheduling for tasks that are already mapped to hardware. By using a cache and software dispatch is proposed in for the cases when the contention on the hardware resources is too high.

An online hardware/software partitioning for image processing was proposed in [72]. However, as in other works, the algorithm used 'performance profiles' that have to be computed at compile-time. A similar problem is described in [42] where the problem is defined considering all the information, such as execution time on the processor, execution time when an FPGA coprocessor is used, the area used on the FPGA by each coprocessor, among other, are known. Several online heuristics are given and compared to an optimal, offline algorithm. In [30], multiple applications are considered and an algorithm is given to select the most efficient set of functions while taking into account function speed and area constraints. Once the selection is made, the decision is changed when the application gets into a new execution phase.

Applications can also be described with other models of computation, for example Kahn Process Networks (KPN). For this cases different mapping algorithms exists, like for example [77] which proposes a runtime mapping algorithm for reconfigurable architectures. The algorithm requires the application to be expressed as a KPN and takes into account the available area and reconfiguration time to execution time ratio.

2.2.3 Parallelism and Scheduling Related Problems and Solutions

The problem of automatic parallelization is a difficult problem when the input is a sequential programming language, such as, for example C. One solution to this problem is to specify the parallelism in the source code. Several paradigms and corresponding API-s, the most prominent ones being message passing represented by MPI API [60] and shared memory represented by OpenMP API [64] and OpenCL [50]. Research indicates that both OpenMP and MPI paradigms can be used to achieve the same performance on shared memory platforms [51]. From the programming effort point of view it is shown [41] that the shared memory paradigm is more efficient when dealing with novice programmers.

When configuring multiple instances of the same computation, one important overhead is the reconfiguration time. Algorithms to reduce the overhead introduced by the reconfiguration time are presented in [37], [66] and [67]. They all have an application model of a control flow/call graph and they introduce prefetch instructions to try to hide the reconfiguration time by making the reconfiguration in parallel with the computations executed on the main processor.

Task chains are the focus of the work of [10]. A task chain models an application were processing is done by independent tasks, and each tasks uses as input the output of the previous task in the chain. The architecture model assumes a reconfigurable area organized in columns. The application is composed of several tasks in the chain that process data. It is assumed that the data can be processed at once by any number of parallel, identical tasks. By using these assumptions, a heuristic algorithm is given, which provides a schedule of the configuration on the FPGA of tasks, and for each task provides the amount of data that it has to process. Due to the restrictive application model and assumptions about the data, these algorithms can only be used for limited number of complex applications.

The problem of splitting the data between multiple instances of the same computation is addressed also in [85]. Compared to the previous work, the data transfer is considered in two cases: when the data transfer is performed after the configuration has finished and when data transfer can be performed in parallel with the configuration. Based on a number of parameters, such as reconfiguration time, available bandwidth, computation time, available area, etc., algorithms are given which determine the best possible load-distribution. Similarly to the previous approach, this algorithm can be applied only to one call of an accelerator, as it does not take into account the global properties of the application.

The configuration of multiple instances will affect the management of reconfigurable area. Spatio-temporal allocation algorithms are given in [78] and [27] for a set of dependent tasks. The algorithms assume that the tasks are organized as a directed acyclic graph and all the needed information is known at compile-time. By using a rectangular task and area model, several problems are formulated and solved using a novel method named 'packing classes'. The problems include finding the smallest needed chip area given a time bound and finding the smallest time given an area bound. The communication is considered part of the total task execution time.

All the approaches described until now consider static analysis. In [63], runtime algorithms are presented that schedule the execution and reconfiguration of tasks using a tile based model of the reconfigurable area. Two of the algorithms presented process events sequentially, but can overleap reconfiguration with the actual execution. The other two algorithms presented allow also parallel execution of multiple computing units. The application model is a directed graph.

A methodology that addresses the dynamic nature of embedded systems at both design time and runtime is presented in [32]. This methodology exploits the different execution behavior of an applications for different data sets. Based on the representative data sets, provided manually by the application designer, it can decide at runtime which mapping is the best for the current system state. An application of this methodology is presented in [58], where scheduling is performed both at compile-time and at runtime, based on the scenarios identified in the wavelet subdivision surfaces application. The scenarios are represented by the 3D-scene that has to be rendered and the position of the camera. As their use-case contains 3 virtual scenes and there are 4 possible camera positions for each scene, the total number of scenarios identified is 12. The main issue with this approach is that it has to identify apriori all the possible scenarios and it will be difficult to adapt to new scenarios.

Bandwidth is also very important when considering parallelism in applications. On a reconfigurable device multiple computing units can be configured to run in parallel and the result is that communication with the memory becomes the true bottleneck. This problem is studied in [11]. The focus of this work is to provide an algorithm for resource allocation, while taking advantage of all the capabilities of modern devices, such as independent clock domains and runtime reconfiguration. The main idea is to balance the usage of the system bus by reducing the frequency of some of the computation units. It is shown that the execution time is minimum when all the computing units finish the work at the same time and each of them has a different clock frequency. The application model used is a task chain, suitable for image processing applications.

2.3 Summary

The problem of building complex computer systems, using more and more heterogeneous architectures, has to be solved via improvements in the toolchains provided for these platforms. The work in the next chapters focuses on improving the state of the art by combining compile and runtime information and providing fast decision algorithms to make applications flexible and adaptable to dynamic changes in the execution platform. As a result, in this chapter we reviewed the related research in the fields of partitioning algorithms, memory allocation, parallelism and scheduling.

3 Research Context

THE main target of the work presented in this dissertation is identifying and solving some of the issues that arise when developing applications for heterogeneous hardware platforms. We do this work in the context of the hArtes project, as it offers the necessary infrastructure for such an endeavor. The hArtes project addresses research and development issues of embedded systems, namely, it investigates hardware/software integration and its main objective was to develop a holistic approach for developing a heterogeneous embedded system. In order to improve the toolchain, the problems emerging during the development must be clearly identified. In this chapter we present the hardware platform used during the experiments, the programming paradigm followed by the program, the platform and the structure of the hArtes toolchain. Then, we focus on the analysis of the applications, with the purpose of identifying the possibilities for improvements of the toolchain.

The hardware platform is presented in detail in Section 3.2. We describe the high-level organization of the platform and the used components and conclude by identifying the issues in this approach. Closely related, Section 3.3 presents the Molen abstraction layer, which represents the implementation of the Molen programming paradigm in the context of the hArtes platform. The Molen programming paradigm is a paradigm based on the sequential consistency model that allows multiple processing elements to act as coprocessors to a central General Purpose Processor (GPP). In order to take full advantage of the platform, the applications have to express and take advantage of the parallelism provided by the platform. At programming language level parallelism can be expressed in several ways, each way having different possible implementations. As we involve in the process also the Molen paradigm, we have studied in the next section what adjustments have to be done either to Molen paradigm or the parallel implementations of the applications when using Molen paradigm. For this purpose we focus on two ways of expressing and implementing parallelism, namely OpenMP and OpenCL. After that, Section 3.4 presents the hArtes toolchain. The toolchain goal is to generate a semi-automatic best fit mapping of any application on the platform. This way, it is possible to provide a rapid development trajectory from application coding to a working reconfigurable embedded computing system. By having a semi-automatic process the developer can always choose among various solutions other than the one proposed by the toolchain. The issues emerging during the development of the toolchain are highlighted, together with the implemented solutions. The chapter continues with a detailed analysis of the real-life applications provided in the context of the project. We present the used methodology to analyze the applications and observe the problems that appear. In Section 3.6 we give an overview of the identified problems, which are further analyzed in the following chapters. In Section 3.7 we summarize our contributions. Finally, Section 3.8 concludes the chapter.

3.1 Methodology overview

Our methodology for analyzing applications includes:

- Identify the hardware platform on which the applications have to be executed.
- Select the suitable real life applications.
- Profile the selected applications, on both the desktop computers and the real platform, if possible.
- Identify from the profile information hot-spots, which take a significant portion of the total application execution time.
- Select a function or only a part of a code, for mapping to another processing element. This should be done only if this move will bring a performance improvement.

3.2 hArtes Hardware Platform (hHP)

We will introduce in this section the hArtes Hardware Platform (hHP). The hHP was designed to represent a reference target for the hArtes toolchain, while providing computing resources for several high-performance applications in the audio domain. We present, briefly, the main characteristics of this platform. The emphasis is put on aspects such as the performance capabilities of the platform, which affect the development of the toolchain, and optimization algorithms.

The requirement for high-performance audio means that particular attention has to be paid to the audio input and outputs from the platform. In order to keep flexibility, a Field Programmable Gate Array (FPGA) is used to handle, at least partially, the massive number of input and output channels. For further processing, an obvious choice is a Digital Signal Processor (DSP) processor that can take advantage of the specialized instruction set. As both FPGA and DSP are computation oriented, a GPP processor is added to coordinate the activities of the platform.

The board design was developed by University of Ferrara, in cooperation with ATMEL Roma and TU Delft. ATMEL Roma provided a DSP daughter board that was integrated in the hHP. TU Delft supported the University of Ferrara in designing an infrastructure that supports the Molen machine organization.

3.2.1 General Platform Description

The basic independent block of the system includes a Reduced Instruction Set Computing (RISC) GPP ARM9, a DSP processor (ATMEL mAgic) and an application-specific reconfigurable block (Xilinx Virtex4 XC4VFX100 FPGA. This block is called, in hArtes terminology, a Basic Configurable Element (BCE). The ATMEL mAgic processor is a Very Long Instruction Word (VLIW) processor, that can execute 15 operations per cycle and supports floating point operations. The Xilinx Virtex4 XC4VFX100 FPGA is a high performance model from the Virtex4 family, having 42,176 slices and 6,768 kB of Block RAM.

In case these resources are insufficient for an application, an extension mechanism was designed, which allows multiple BCEs to be connected together.

For the actual hardware implementation, two BCEs were put on one hardware board, the hHP. The architecture of the hHP is shown in Figure 3.1. The current boards contain two BCEs each, and multiple BCEs can be chained if required by the application. The BCE-s are independent, and can run any selected thread or application mapped by the hArtes toolchain. In case an application running on one BCE needs to transfer data to an application running on another BCE, the data can be transferred through a high bandwidth direct

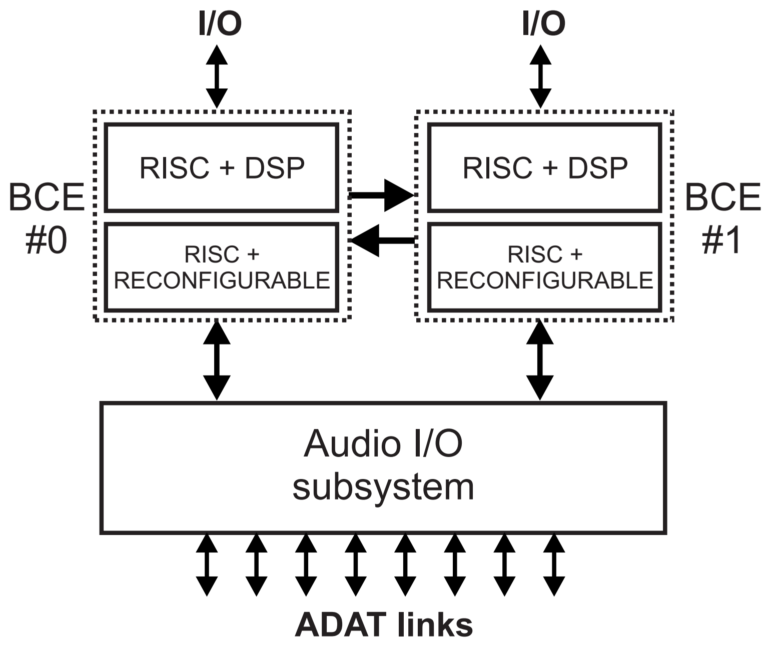

Figure 3.1: Top level architectural structure of the hArtes Hardware Platform. The system has two independent heterogeneous and configurable processors, which communicate among each other and with an audio I/O subsystems supporting several ADAT channels.

data link (approximately 200 MB/sec).

For massive data streaming, dedicated hardware is available on the board, namely the “Audio I/O subsystem” block in, Figure 3.1. Eight input and eight output ADAT Lightpipe interfaces are available on the board. ADAT Lightpipe is a standard for the transfer of digital audio that uses fiber optic cables and has Toslink connectors at either end. Each ADAT interface supports 8 audio channels. As a result, the hHP supports 64 input channels and 64 output channels. Of course, driving 64 speakers requires additional hardware, which demultiplexes the optical cables into multiple electric cables suitable for ordinary speakers.

Figure 3.2 provides a detailed overview of a BCE. The two main blocks are a RISC/DSP processor, the D940HF produced by Atmel, and a reconfigurable processor, the Xilinx Virtex4 XC4VFX100) [19]. The RISC processor is an ARM926EJ-S ARM processor, running at a frequency of 160MHz. It has a 16 kB instruction cache and a 16 kB data cache. The DSP processor is running at 80 MHz. It has 200 kB SRAM memory, a data cache of 16k x 40bit words,

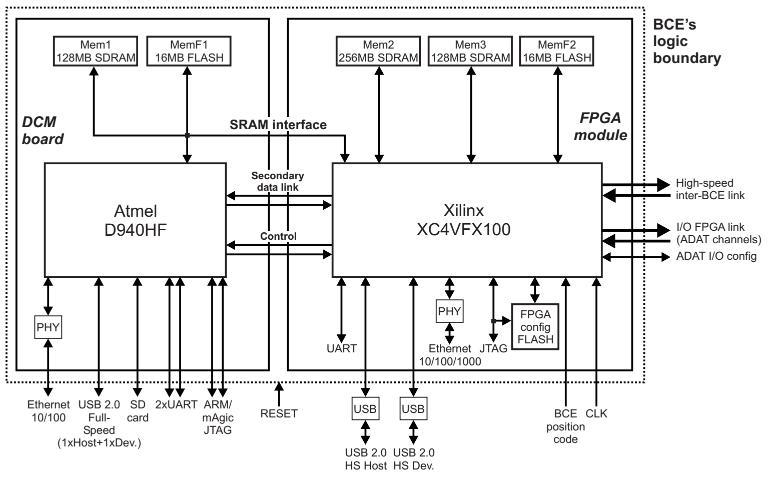

Figure 3.2: Detailed block digram of the Basic Configurable Element (BCE) of the hArtes Hardware Platform. The BCE is the basic building block of the platform, supporting several processing architectures. One or more BCEs work in parallel to support a hArtes application.

and an instruction cache of 8k x 128 bits instructions [5].

The memory is organized into private memory blocks and shared memory blocks. “Mem1” and “Mem3” in the figure are private to D940HF and to XC4VFX100, respectively. “Mem2” is a shareable memory bank that can be shared on several levels. It can be used by all computing elements on a BCE, and by processors on different BCEs, through the high speed inter-BCE link. A Flash memory is present to drive the initial configuration of the FPGA. This memory can be programmed using a Joint Test Action Group (JTAG) connector at design time [1]. For the filesystem of the operating system a Secure Digital (SD) card reader is available. Several standard input/output interfaces are also present on a BCE, from which we mention Ethernet (both for the ARM processor and for FPGA), Universal Serial Bus (USB) and Universal Asynchronous Receiver/Transmitter (UART) (used mainly for debugging purposes).

Figure 3.3 shows a picture of a physical platform in which we clearly distinguish the elements listed above. A physical board contains two BCEs, one on the left and one on the right. The big daughter board in the middle right part of the hHP contains the FPGA (Xilinx Virtex4-FX400). The smaller daughter board contains the D940F processor block. The 16 ADAT connectors (8 inputs and 8 outputs) are visible on the top of the board. The physical size of the

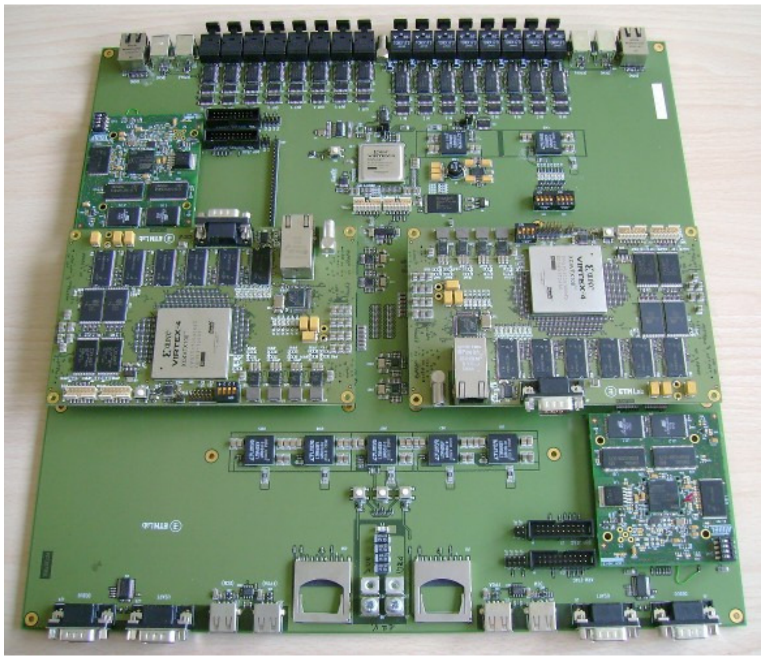

Figure 3.3: Picture of the hArtes Hardware Platform. The two BCEs use two daughters boards each, one for the D940HF processor and one for the FPGA based infrastructure. The ADAT interfaces and several standard I/O connectors are clearly visible at the top and at the bottom of the picture, respectively. [83].

board is 370mm x 370 mm. Although this might not be suitable for embedded use, our focus was on fast prototyping, and, therefore the layout was not optimized for area. If components are packed closer together, the dimensions of this board would reduce dramatically. A schematic overlay showing all the elements is presented in Figure 3.4.

3.2.2 Hardware Platform Issues

The hardware platform developed for the hArtes project was designed specifically for audio processing. The initial design decisions were taken considering the use of a FPGA for the audio processing of input or output. For this reason, a good transfer bandwidth was ensured between the audio buffers and the FPGA. The connection between the FPGA and the ARM or the main memory was not considered to be crucial. The implication is that there is no DMA available when transferring between main memory and FPGA Scratch

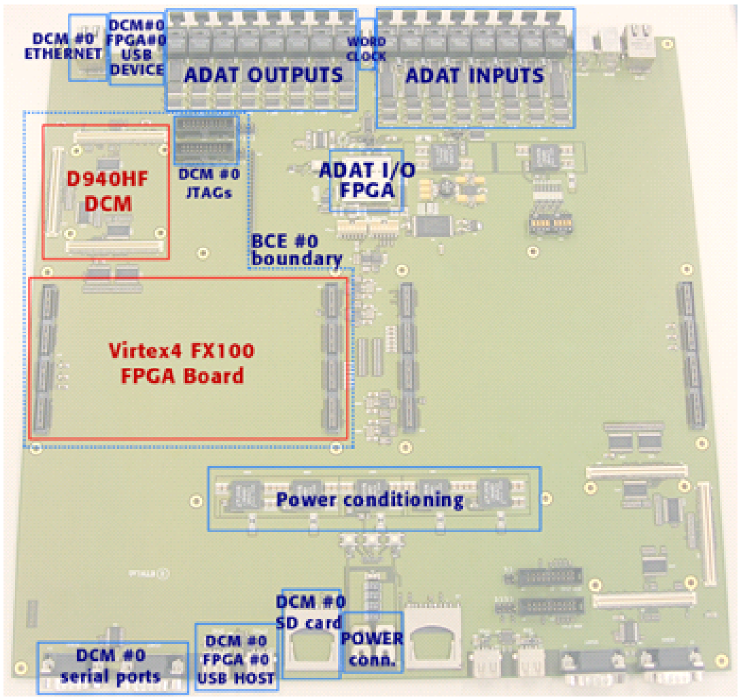

Figure 3.4: Overlay of components over a picture of the hArtes Hardware Platform.

Pad Memory (SPM). Even if the absence of DMA is acceptable for reading or writing of audio data, this limits the efficiency of the FPGA when dealing with generic applications or applications that have their input data stored somewhere else than the audio buffers.

This is not a core architecture problem. Rather, it is a design decision, determined by the objective of the hardware board and its uses. We performed a series of experiments in order to evaluate the memory speed.

Since the DMA was not available to access the FPGA memory, all the transfers between the main memory and the FPGA local memory had to be performed by the GPP. Table 3.1 illustrates the results obtained when transferring a block of 32 kB between various memories. As we can see, the transfer speed between the Synchronous Dynamic Random Access Memory (SDRAM) and the FPGA SPM is 2.85 times slower than the transfer speed between SDRAM and SDRAM. Experiments have shown that the bandwidth is the same for both larger and smaller memory block sizes.

The slow transfer speed between SDRAM and SPM, 24.28 MB/s compared to

SDRAM to SPM	1287 μs
SPM to SDRAM	1349 μs
SDRAM to SDRAM	451 μs
Transfer speed SDRAM to SPM	24.28 MB/s
Transfer speed SDRAM to SDRAM	69.29 MB/s

Table 3.1: Transfer speed of a memory block of 32 kB size between SPM and SDRAM for hArtes Hardware Platform.

the transfer speed from SDRAM to SDRAM, 69.29 MB/s, imposes constraints on the type of kernels that can be accelerated, when executed on the FPGA compared to the GPP.

For example, let us consider a kernel that has as input data and output data size of 16 kB. According to Table 3.1, the total time needed for memory transfer is $1287/2 + 1349/2 = 1318\mu s$. We are interested in the speedup including the overhead of transfer, when the computation time varies. We consider the computation time to be measured on the ARM processor. The computation speedup is the ratio between the computation time on the ARM processor and the computation time measured for the same kernel on the FPGA. In Figure 3.5, we plot the total speedup for a set of computation times, ranging from 0 to 10000 μs and assuming computation speedups between 2x and 8x. We can see that obtaining a considerable speedup for a computation will not help if the execution time is in the range of the memory transfer overhead. For example, for a computation speedup of 7 and a computation execution time of 3000 μs, the total speedup is less than 2x. Only when the computation time is 5 times larger than the memory overhead, significant total speedup can be obtained.

3.3 The Molen Abstraction Layer

The Molen programming paradigm targets machines that adhere to the Molen machine organization [82]. As described in Section 2.1.1 the Molen machine organization is based on the processor-coprocessor model and it allows the processor to control the execution of the coprocessor via a set of fixed primitives. By using these primitives, a virtually unlimited number of operations can be implemented as accelerated components. Started as an extension for reconfigurable architectures, Molen can be used for any heterogeneous architecture like the hArtes hardware platform.

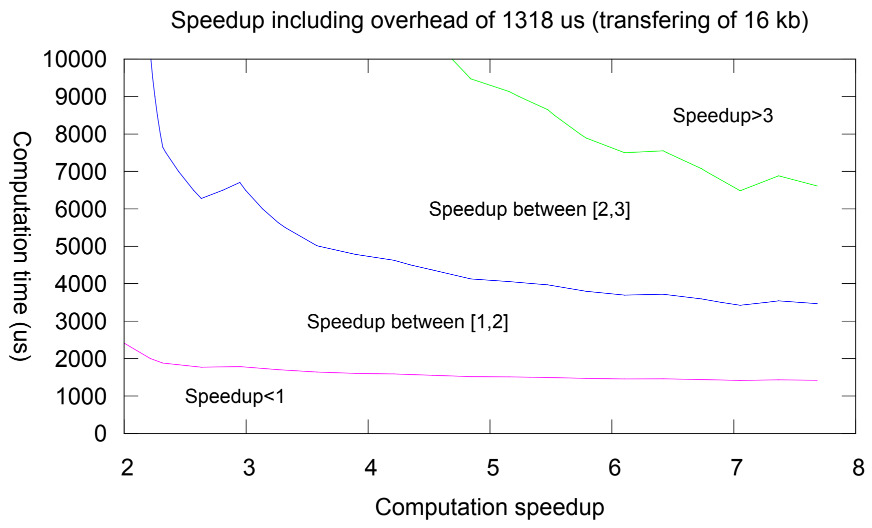

Figure 3.5: Possible speedups including memory transfer of 16kb function of speedup and execution time, for the hArtes hardware platform.

3.3.1 The Molen Programming Paradigm

The Molen programming paradigm is a paradigm based on the sequential consistency model that allows multiple processing elements to act as coprocessors to a central GPP. The paradigm was developed for tightly coupled processor-coprocessor systems. The instruction set of the GPP is modified to add instructions for controlling the coprocessor. The instructions used are *partial set*, *complete set*, *execute*, *break*, *set prefetch*, *execute prefetch*, *movtx* and *movfx*.

The instructions *set prefetch* and *execute prefetch* are used when the coprocessor has the capability of prefetching the binary data needed to run a computation. In the case of FPGAs, the binary 'program' is represented by a bitstream, and, in case of a DSP, this binary data is represented by an Executable and Linkable Format (ELF) file [56]. The FPGA on the hHP platform does not have the capability of prefetching bitstreams. The DSP has a local code memory which allows it to load all the needed data when the application starts. As a result, there is no need for *set prefetch* and *execute prefetch* in our scenario.

The instruction *partial set* is meant to reduce the reconfiguration time, by configuring the common parts of multiple computations at the same time. As stated already, the DSP has enough local memory to accommodate all the needed data. The FPGA can take advantage of such a feature. In any way, the identification of common parts at Register Transfer Level (RTL) level between multiple applications is a complex problem. In the context of the research presented in this dissertation, we do not support this feature. As a result, we will only use one complete set instruction instead of two set instructions.

To summarize, our platform, needs the following instructions: *complete set*, *execute*, *break*, *movtx* and *movfx*. As the platform does not support tight integration between the GPP and the coprocessors, we utilize the term "primitive" to describe these instructions in the rest of the thesis. We will discuss each of them in detail and we will show how we modified and extended them in the context of the hArtes project.

The SET Primitive

The SET primitive's role is to start the configuration of the processing element with the operation that has to be executed. This represents different actions for different processing elements. For a FPGA, it represents a reconfiguration of the area available for custom functionality. For a DSP processor, it represents the loading of an executable file into the memory. These operations can take a different amount of time depending on the characteristics of the processing element and on the operation. Thus, by having an independent primitive to manage the configuration allows a compiler to perform scheduling of reconfigurations. A good scheduler, by making a configuration in parallel with other useful computations would be able to hide the configuration time.

Original instruction:
SET address

Extended primitive:
SET processing element, address

The SET primitive is extended by adding, as a parameter, a processing element identification. In the original proposal this was unnecessary, as there was only one co-processing element.

The MOVET, MOVEF Primitives

The role of the MOVET and MOVEF primitives is to send (MOVET) and to receive (MOVEF) parameters and memory blocks to/from the processing element respectively.

Original instruction:
MOVET xreg, register

Extended primitives:
MOVET processing element, xreg, register
MOVET_ADDR processing element, xreg, address, size, type

Original instruction:
MOVEF register, xreg

Extended primitives:
MOVEF processing element, register, xreg
MOVEF_ADDR processing element, address, size, type

The current implementation of MOVET and MOVEF differs from the original in two aspects. First, two versions are implemented for both MOVET and MOVEF: a version for parameters representing values and a version for parameters representing addresses. The second difference is represented by the information provided to the primitives, such as the type of the parameter. This was done for two reasons:

- Each processing element can have a different memory size for a data type. This is, in part, because the C standard does not specify the exact type sizes, but rather it specifies the smallest bit size for each type. For example, the float type for the DSP processor float type size is 40 bits while Advanced RISC Machine (ARM) processor's float type size is 32 bits. When transferring between two processing elements, a conversion might be needed to convert the data from one format to another.
- Even if the GPP can access all the memory areas, the processing elements can physically access only a part of the whole memory. In case a processing element needs data from a memory area it can not access, the GPP has to explicitly transfer that memory block. The size parameter

has to be added to the MOVET and MOVEF primitives in order to be able to perform the transfer .

The EXECUTE Primitive

From the programmer's perspective the EXECUTE primitive starts the execution of one computation. This operation is an asynchronous operation. The semantics is that the operation will start on a processing element, while the execution continues on the GPP. It is the responsability of the programmer to check for the status of the execution, by using the BREAK primitive.

Original instruction:
EXECUTE address

Extended primitive:
EXECUTE processing element, address

As the current implementation includes more than one processing element, and it is designed to allow any number of processing elements, we extended the original implementation by adding the processing element parameter to the execute call.

The BREAK Primitive

The role of BREAK primitive is to synchronize the execution of the GPP with the execution of the FPGA.

Original instruction:
BREAK

Extended primitive:
BREAK processing element, address
BREAK_CHECK processing element, address

There are several differences between the primitive and the instruction:

- The instruction performs full synchronization (i.e. among all executing computations on the FPGA). As there are cases where waiting for one computation can improve the scheduling, we added the address parameter. This parameter identifies the computation that has to finish before BREAK gives the control back to the processor.

- The instruction completely blocks the GPP. This is not a problem in a single application environment but, in a multithreaded system, this is not a desirable behavior. We decided to implement the primitives in such a way that allows thread switching, even when the BREAK instruction did not finish its execution.
- In case of parallel unbalanced workloads, some computations will finish earlier, and, depending on the application, it might be possible to restart them before the others have finished (see the example in Section 3.3.3). For these cases, we extended the original BREAK instruction and created a non-blocking BREAK_CHECK primitive that returns *true* in case the computation has finished and *false* otherwise.

Source Annotations

The Molen programming paradigm can be used with any compilation flow able to partition an application in different processing elements. The partition is done between a master controller, the GPP, and the other processing elements. The C language and compilation flow was chosen for the hArtes toolchain, as it is one of the most used languages in the embedded system world. It was decided that functions offer the necessary level of abstraction to model the computations that run on the processing elements, because:

- The developers already understand the underlying concepts and they can use it directly.
- The syntax of a function definition and function call are part of the language.
- The input and output can be clearly defined.
- Existing tools can be used for profiling and debugging.

The standard directive extension mechanism of C, the pragmas [44], was used to specify which functions are to be mapped onto which processing element. Two pragmas are introduced as it can be seen in Listing 3.1 and Listing 3.2:

- On a function declaration: the semantics is that all calls to that function are translated to Molen primitives. The Molen backend compiler then provides an implementation for the corresponding processing element.

- On a function call: the semantics is that the call is replaced by the corresponding Molen primitives. The Molen backend compiler then provides an implementation for the corresponding processing element for that call. This gives more opportunities for optimization, as specific implementations can be generated for that call, for example, by taking into account constant function parameters.

The *map* pragma includes two pieces of information:

- The name of the processing element to which the computation is mapped.
- The implementation identifier: this makes a link between the implementations and the function declaration or function call to which the pragma applies.

Listing 3.1: Molen pragma on function definition, mapping function *func* to processing element Virtex4 with implementation identifier 1.

```
#pragma map call_hw VIRTEX4 1
int func(char *p, int len) {
  ...
  return v;
}
```

Listing 3.2: Molen pragma on function call, mapping that specific call to function *func* to processing element Virtex4 with implementation identifier 2.

```
int func(char *p, int len) {
  ...
  return v;
}

void main() {
  ...
  #pragma map call_hw VIRTEX4 2
  result = func(variable, 100);
  ...
}
```

3.3.2 hArtes Implementation of The Molen Programming Paradigm

Two implementations of the Molen Abstraction Layer (MAL) for the hArtes Hardware Platform were necessary: one for the Virtex4 FPGA and one for the mAgic DSP. Each implementation had its own peculiarities and we will discuss them in detail separately. The transformation from source annotations to Molen primitives will be discussed in Section 3.4.1.

The MAL Implementation for Virtex4 FPGA

Given the organization of the FPGA in the hHP, the SET primitive will perform a full reconfiguration of the reconfigurable part.

The computation that is configured on the FPGA has only access to a local scratch pad memory of size 128 kB. The primitives MOVEF_ADDR and MOVEF_ADDR manage the transfer to/from this local memory. As the hardware compiler uses the C language type sizes used by the GPP compiler, no conversion is needed when moving data to/from the FPGA part of the system.

A tight coupling with the FPGA was impossible as the GPP processor resides physically on a daughter board. The EXECUTE and BREAK primitives are implemented using the memory mapped control registers of the Molen controller found in the FPGA. EXECUTE writes a 1 at physical address 0x30000006. BREAK is implemented by checking the value found at address 0x30000007.

The internal FPGA structure is presented in Figure 3.6. The GPP processor is located in the ATMEL D940HF module. The *controller* block in the FPGA module controls the computations on the FPGA.

In Figure 3.6, we can see the hardware blocks that represent the Molen machine organization. The Reconfigurable processor component which contains the computation has access, through a memory arbiter to the D940 memory interface. The ARM, located on the ATMEL D940HF board, can access the controller which manages the execution of computations through the data exchange links.

MAL Implementation for mAgic DSP

Even if it is not essential for our work described in this thesis, we present the implementation of the MAL for the mAgic DSP to show the generality of the

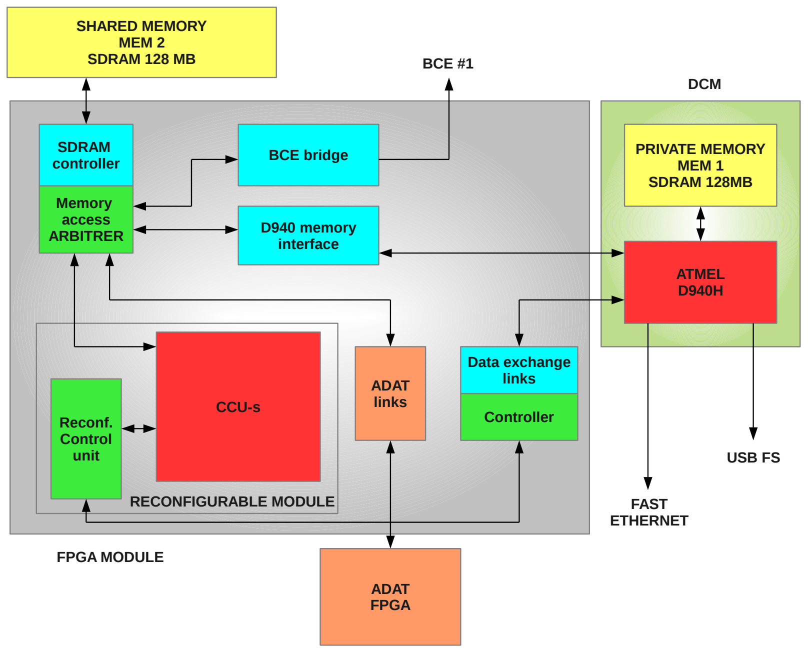

Figure 3.6: Internal organization of firmware used for Xilinx Virtex4 FPGA.

approach presented and the kind of customization that needs to be done for various platforms. The most important differences with the MAL implementation for the Virtex4 FPGA are that the mAgic DSP has general purpose processor functionality and the transfers to and from its memory are managed by a Direct Memory Access (DMA) controller.

For the mAgic processor, the code has to be loaded in a local memory before being executed. As memories are much cheaper than reconfigurable area, the local program memory can easily store all the code needed for the computations, and the loading operation will be performed at the beginning of the program. This is in contrast with the FPGA, where only a small number of Custom Computing Unit (CCU)-s can be configured at the same time.

The DMA configuration adds an additional overhead. It is more efficient to perform it once for all the memory blocks that need to be transferred. The result is that the MOVET primitives will only construct a list of blocks that need to be transferred. The EXECUTE instruction will start the transfers and the end of the DMA operation will be signalled to the mAgic, which will start the computation.

3.3.3 The Molen Paradigm and OpenMP

The Molen programming paradigm can not be used directly by the developer to express parallelism. On the other hand, the compiler can schedule Molen primitives to take advantage of the existing parallelism. When both OpenMP and Molen annotations are present in the same program, the compiler can use one of the following scenarios:

- Use Molen parallelism instead of the thread parallelism present in OpenMP, where possible. The advantage of this approach is that the overhead of thread management is eliminated (Molen overhead is much smaller than the thread creation/switching overhead).
- Generate a multithreaded application based on the OpenMP annotations and the use of Molen primitives in each thread. The Molen primitives have to be thread safe for this scenario to be supported.

We give details about each of the scenarios in the following sections.

Using Molen Primitives Directly

In order to support parallel execution we introduce OpenMP pragma's and we generate Molen primitives. Listing 3.3 gives an example where it is specified that 2 kernels can execute in parallel. For this case, the compiler understands the parallelism structure and, instead of creating and synchronizing threads, it generates and schedules the Molen primitives. The results are presented in Listing 3.4. Similar examples can be constructed for other OpenMP constructs.

Listing 3.3: A Molen pragma in the context of OpenMP sections.

```
#pragma omp sections nowait
{
  #pragma omp section
  {
    #pragma map call_hw VIRTEX4 1
    fft(p, n);
  }
  #pragma omp section
  {
    #pragma map call_hw VIRTEX4 4
    value = sad(d, l);
  }
}
```

Listing 3.4: Molen primitives generated for the OpenMP sections example.

```
SET(1);
SET(4)
MOVTX_ADDR(1, p, n);
MOVTX(1, n);
EXECUTE(1);
MOVTX_ADDR(4, p, n);
MOVTX(4, n);
EXECUTE(4);

BREAK();
```

The use of Molen Primitives in OpenMP Threads

The second option is to use the Molen primitives in the OpenMP generated code. This implies that it is possible that Molen primitives will be invoked with the same computation identifier. This is a case not explicitly covered by the parameters sent to the primitives.

This scenario was not implemented for the hHP, due to the fact that the area allocated for the computations on the FPGA was not sufficient to allow multiple kernels to run in parallel. This scenario was implemented on the next generation Xilinx platform, the Virtex5 FPGA. We give a high-level description of the implementation as an argument to the flexibility of the Molen approach.

When multiple computations are run in parallel, a computation identifier will not uniquely identify a running computation anymore, but only the type of the computation. On the ML510 platform, the reconfigurable area is partitioned in multiple slots and each slot can run a different computation. To support this, a level of indirection is added between the computation identifier and the slot number, based on the thread identifier. When invoked, SET primitive will call an atomic function *int MOLEN_get_slot(int identifier)* that will return the slot that has to be used by the current thread for that computation. The BREAK primitive will invoke an atomic function *void MOLEN_relase_slot(int slot)* that will allow other threads to use that slot.

3.3.4 The Molen Paradigm and OpenCL

The OpenCL standard defines, in an abstract way, a platform and the way of specifying applications for that platform. The platform is composed of a host processor and several computational devices each of them containing one or more computing units. These, in turn, contain one or more processing elements. This platform is considered to have four memory regions: global, local, constant and private. The management of the transfers between these regions is entirely controlled by the programmer.

The application consists of code to be run on the host processor and code for the computational devices (kernels). The host program code assembles commands into command-queues that are then executed on the computational device. Examples of commands are the setup of the computational device, the memory transfers to and from the local memories of the computational device to the main memory, the start of parallel execution of multiple instances of the same kernel.

OpenCL is more platform dependent than the Molen programming paradigm. It specifies the memory hierarchy that the platform has to implement and the programmers have to manage manually, when they develop applications.

As both paradigms have the same scope, it is natural to analyze the possibility to use either of them. We will do this in the next sections.

The Generation of Molen Primitives from OpenCL Programs

The Generation of only Molen primitives from OpenCL programs is a difficult task as OpenCL provides more information and extensions to the C language than Molen. Also, some concepts are not present at all in Molen and that implementation would transform Molen completely. As a result, we present a mapping between a subset of OpenCL and Molen. This subset allows OpenCL programmers to target Molen enabled platforms.

The implementation of command queues, events and contexts into Molen represents a significant change of the paradigm. As a result, even if they allow a great flexibility in describing an application, we choose to simplify them into in order, blocking execution with one single context.

Device information is useful to allow OpenCL applications to make adjustments depending on the different parameters of the architecture (for example cache sizes and maximum local memory available) and could be directly implemented for a Molen platform as well.

OpenCL concept	Molen equivalent/notes
Context	not available
Queues	not available
Events	not available
Memory objects	implemented internally in Molen
Program objects	transformed at compile-time
Kernel objects	transformed to pragma annotations
Vector types	not supported
Image types	not supported
OpenGL API sharing	not supported

Table 3.2: The Molen concepts and their corresponding OpenCL concepts.

The memory management is not directly specified by the Molen programming paradigm. The implementation of the Molen runtime will contain functions that operate on different memory areas.

Functions related to program loading have to be extracted at compile-time and the identifier information needed by Molen will be generated from them. The program execution functions can be transformed into calls to the kernels annotated by a pragma. In the case of *clEnqueueNDRangeKernel*, a loop containing the call will be generated.

The OpenCL extensions of vector types could be implemented in a Molen backend compiler. Without these extensions and some other minor issues, the OpenCL language is compatible with C and, as a result the current Molen backends are able to generate code from it.

All is summarized and shown in Table 3.2.

The Generation of OpenCL Functions from Molen Annotated Programs

In this scenario, a Molen annotated application exists and the task of the developer is to port it to an OpenCL enabled platform. As Molen was developed with embedded systems in mind, while OpenCL implementations exist currently for high performance oriented platforms such as NVIDIA GPU-s and Cell BE [54], an application annotated with only Molen pragmas will not be able to take fully advantage of the range of possibilities provided by OpenCL. Anyhow we present the generation process, as it shows the generality of Molen, which enables the developers to work with multiple platforms,

Molen primitive	OpenCL functions used
SET	clCreateProgramWithSource clBuildProgram clCreateKernel
MOVTX	clSetKernelArg clCreateBuffer clEnqueueWriteBuffer
MOVFX	clEnqueueReadBuffer
EXEC	clEnqueueTask
BREAK	clFinish

Table 3.3: Molen primitives and their corresponding OpenCL functions.

even if not in an optimal way.

We will assume in the following description that the initialization of the OpenCL process is already made. This assumption does not affect in any way the process of generating OpenCL functions from Molen pragmas. For a kernel invocation, the Molen primitives map almost 1-to-1 to OpenCL functions, as shown in Table 3.3

As OpenCL delays the compilation of kernels until the actual execution, the SET primitive should be extended to load and compile the program. Of course, this should be done only once per execution and the results should be stored in the internal cache. Then, for each SET invocation, a call to *clCreateKernel* will be made.

The MOVTX primitive corresponds directly to the *clSetKernelArg* function. However, in the Molen programming paradigm, the MOVTX primitive is also responsible for transferring memory to the local computing element memory so, for those operations two other OpenCL functions have to be used *clCreateBuffer* and *clEnqueueWriteBuffer*. The corresponding MOVFX primitive will use *clEnqueueReadBuffer*.

The EXEC and BREAK primitives correspond directly to similar functions that have the same semantics as in the Molen programming paradigm.

3.4 hArtes Toolchain

The hArtes toolchain's main objective is to support the entire process of mapping an existing application onto a specific reconfigurable heterogeneous system. The application has to be described either at a high algorithmic level or in a high level programming language. For the algorithm description, the developer has the choice of using a graphical entry interface or a signal processing oriented language. As high level programming language, the most obvious choice as input is the C programming language.

The toolchain is organized in several toolboxes, which provide different functionalities and, together, realize the main objective. A schematic representation of the toolchain is presented in Figure 3.7. Highlighted in green are the components improved as a result of our work. Our work resulted also in improvements to the runtime system, not represented in this figure.

The high level design alternatives (either a graphic entry interface, a signal processing or a computation oriented language, or just general purpose C language) are offered by the hArtes AET (Algorithm Exploration and Translation) toolbox. For the graphical part, NU-Tech was adopted as Graphical Algorithm Exploration (GAE) solution and Scilab was adapted as the computation-oriented language. NU-Tech [53] is a platform which supports the development of algorithms for real-time scenarios, emphasizing on the strict control over time and latencies. Scilab is a free and open source software for numerical computation, similar to Matlab [43].

The output from the Algorithm Exploration Toolbox is processed by a series of other toolboxes integrated in a coherent workspace (the hArtes framework). The toolchain workflow will guide the application developer to realize an efficient implementation of the application mapped onto the identified hardware, while giving him/her at each step full control over the process.

The available tools in the hArtes Framework perform the following tasks:

- The automated partitioning of the high-level algorithm descriptions represented as C code. By using a set of predetermined design criteria and information about available resources, the high level algorithms are divided into tasks.

- The transformation of the high-level algorithm tasks. This includes, for example, transformations to make tasks compatible with special processing elements, like an FPGA. These transformations are done to provide the design space exploration with more options.

- The design space exploration. The exploration of the mapping possibilities between the tasks available and the processing elements of the reconfigurable heterogeneous system. This is done by using either estimated or measured costs for each task.
- The mapping and generation of the code for the targeted GPPs and DSPs.
- The code manipulation of Molen primitives. This includes, for example, scheduling of lengthy operations such as reconfiguration. This manipulation has to be done after the mapping is fixed.
- The VHSIC Hardware Description Language (VHDL) code generation for the target FPGA. For this purpose we use Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) tool developed at TU Delft.
- The synthesis of the VHDL code obtained, by using vendor specific tools.
- The compilation of the C code for the GPP processors present in the system.
- The integration of all binary code generated and running on the platform.

Each activity performed on the code is performed by a separate tool. This distribution of activities improves the modularity and the manageability of the complete solution.

3.4.1 Using GPP Compiler to Generate Molen Primitives

As explained in Section 3.3.1, a transformation from source annotations to Molen primitives is performed automatically by the toolchain. This is implemented in the GPP compiler used by the platform, in our case the GCC compiler for ARM.

We implemented three new compiler passes, namely:

- *pass_fpga_replace_calls_tree* that replaces calls annotated with a pragma to Molen primitives.
- *pass_diopsis_replace_calls_tree* that has similar functionality with the previous pass, but generates the Molen primitive implementation for DSP (described in Section 3.3.2).

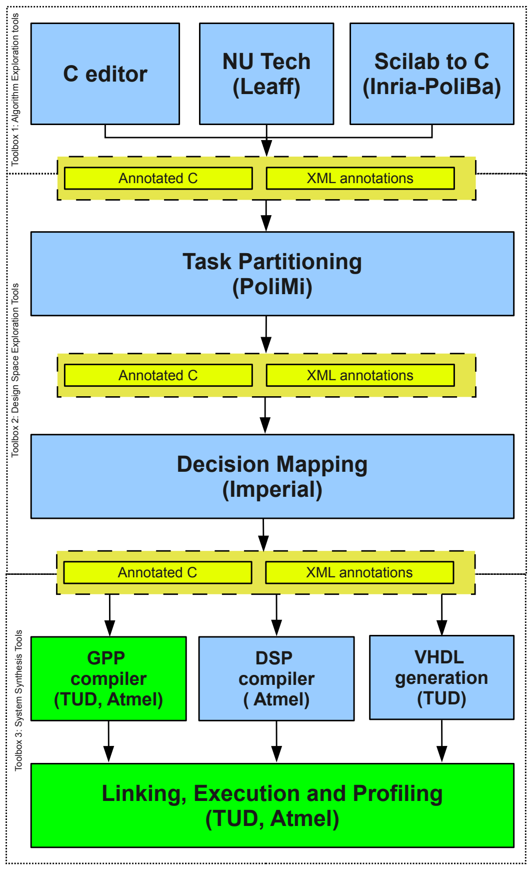
Toolbox 1: Algorithm Exploration tools
C editor
NU Tech (Leaff)
Scilab to C (Inria-PoliBa)
Annotated C
XML annotations
Toolbox 2: Design Space Exploration Tools
Task Partitioning (PoliMi)
Annotated C
XML annotations
Decision Mapping (Imperial)
Annotated C
XML annotations
Toolbox 3: System Synthesis Tools
GPP compiler (TUD, Atmel)
DSP compiler (Atmel)
VHDL generation (TUD)
Linking, Execution and Profiling (TUD, Atmel)

Figure 3.7: The hArtes toolchain overall architecture.

- *pass_fpga_parallel_optimisation* that schedules Molen primitives when OpenMP pragmas are present. This transformation was described in Section 3.3.3.

These passes were added before after the frontend processing, allowing GCC to perform all it's suite of optimization on the code.

One of the objectives of the hArtes project was to facilitate an automated mapping to different processing elements, by using a shared memory paradigm. In the hArtes implementation, the GPP has access to all the memory of each processing element, but the processing elements will have access only to their local memory. In case the local memory of each processing element is not sufficient for all the data that will be processed during the application's execution, transfers have to be performed to and from the local memory.

There are multiple possible solutions for the making processing elements use application data but, as the platform is fixed, we will discuss only the software solutions:

- The memory is directly allocated in the local memory processing element memory. A discussion on this topic is presented in Chapter 5.
- The runtime library manages the transfer automatically, when certain blocks are needed to be processed by a specific processing element.

In the following we describe the extensions implemented for the second case. Even if it is a straightforward and unoptimized approach that does not perform any significant analysis or optimization, it is a good starting point to enable an application to run on our platform. Additionally, based on this implementation various optimizations can be performed in later stages.

As described in Section 3.3. two additional primitives are introduced to differentiate the MOVET or MOVEF primitives for the special case of a pointer:

- MOVET_ADDR - used for each pointer parameter sent to a processing element.
- MOVEF_ADDR - used for each pointer parameter sent to a processing element, after the kernel completed the execution.

These two primitives work together with the support of the runtime system. All the dynamic memory allocation will be made by special 'wrapper' functions that keep track of allocations (for example *hmalloc* instead of *malloc*). The replacement of the 'malloc' functions (and similar functions, i.e. *realloc* and *calloc*) is made by the source-to-source transformation tool. Then, at a later point in the execution of the program, for each address sent as a parameter to the computation, the runtime system can determine the size of the block to which that address belong.

Naturally this approach has several limitations:

- Except in parameters, addresses should not be present in the memory blocks used by the kernels. This means, for example, that structures like linked list will not be supported. Although this can be seens as a limitation, all of the kernels analyzed did not use such complex data structures.

- Without further analysis or information from the developer the transfer can be very inefficient. For example, some memory blocks are only written by the kernel while other are only read. The C language provides some information about this (for example the const keyword), but it is incomplete as, for example there is no way to specify write only locations.

3.4.2 Development and Toolchain Debugging

During the development of the toolchain, due to the complexity of the work involved we identified several problems that need to be addressed, in order to deliver a successful product. In the following, we describe two of the main problems analyzed and solved during the framework development: the enforcement of the checks on the versions of the various components and the simulation support.

Enforce Checking of Interface Between Tools

This problem can be summarized as follows: the development of a hardware system involves changing a lot of parameters. Sometimes it can happen that you use two components, one adapted for a value of a parameter of the platform and another adapted for another value. This results in incompatibilities and, more specifically, in a system failure at runtime. This means that a function will give incorrect results without any warning or error during the compilation process.

As an example, the frequency is one of the configurable parameters, for a reconfigurable hardware platform. The number of cycles needed for various operations changes based on the frequency. Floating point multiplication is an example of such an operation. In case of a low frequency, for example below 50 MHz, the hardware unit will take advantage of the long time between cycles and use combinational logic to perform operations between two clock cycles. A floating point multiplication at 50 MHz needs, to obtain the result, 4 cycles. On the other hand, if the frequency increases, less combinational logic will be used than in the previous case, and the hardware unit will have to include clocked registers and perform the computation in multiple cycles. Generating a floating point multiplication at 200 MHz results in a total number of cycles of 6. Of course, if used in a pipeline, the higher frequency hardware unit will always have a higher throughput but, in the automatically generated hardware kernels, the operations are not always used in such a way. Another possibility would be to always use the higher frequency hardware unit but, obviously this will degrade the performance of a low frequency design. By having different latencies for different frequencies, the scheduling and the area used by the kernel are affected and, therefore, these latencies have to be an input to the automated C-to-VHDL compiler.

As the development went along, it was obvious that a mechanism had to be provided to make sure the computation - which, when generated for FPGA we call CCU - was generated for the system in which it was integrated. This involved the following parameters (all except endianess are related to frequencies but manifest themselves in different parts of the system):

- Endianess.
- The number of cycles needed to read/write the memory and the Exchange Registers (XREG) memory area.
- The number of pipeline stages of each floating point component.
- The frequency at which a specific computation could run.

The solution depends on the moment at which the check can be performed. For the first three problems, the check can and will be performed at compile-time. The obvious choice was to add dummy signals to the CCU, the interface with the memory and the XREG memory area, and the floating point components. In case the CCU was generated with a different configuration, the VHDL compiler would not have been able to match the signals and would report an error.

Examples of such dummy signals are given in Listing 3.5. Each name refers to the parameter that it 'checks'. For example, the presented CCU uses a floating point (fp) single precision (sp) multiplication unit (mult) that has a pipeline of 6 cycles. The number of cycles needed to access memory is 5 (mem_cycles). The number of cycles needed to access the XREG is 3 (xreg_cycles), and so on.

Listing 3.5: Dummy signals used to check the version of CCUs.

```
check_fp_dp_mult_top_10 : in std_logic;
check_fp_sp_mult_top_6 : in std_logic;
```

```
check_fp_dp_large_3 : in std_logic;
check_fp_dp_less_equal_3 : in std_logic;
check_fp_dp_large_equal_3 : in std_logic;

check_endianess_little : in std_logic;
check_mem_cycles_5 : in std_logic;
check_xreg_cycles_3 : in std_logic
```

For the last problem, a dynamic solution was chosen. As an FPGA has a complex Digital Clock Module (DCM), which can generate different frequencies, before running the CCU special code inserted by the compiler will set a configuration register with the correct value.

Compilation Times and Maximum Frequencies

The compilation time is not the most important aspect when targeting embedded systems. Nevertheless it can still become a bottleneck in the development process. Within the compilation process, the most time consuming part is the FPGA related processing. We measure time for two stages in this compilation process: the VHDL generation and the VHDL synthesis, as they are very different in nature and developed by independent entities.

The general compilation flow, for Xilinx tools, is as follows:

- Generation, which can be either manual or, as in our case, automatic.
- Synthesis, which translates the VHDL to a netlist, which is a low level representation of hardware blocks and connections.
- Translation, which translates all the netlists and constraints file into an internal representation.
- Mapping, which maps the logic defined in the previous step to FPGA elements.
- Placing and routing, which assign physical elements to the elements identified in the previous stage.

The synthesis step can be performed by multiple tools, such as Synopsis Synplify Pro or Xilinx XST. In order to avoid possible incompatibilities between tools from different vendors, we choose to use a Xilinx only flow.

One notable aspect is that the targeted frequency for the design is set in the *translate* step. The subsequent steps will try to reach the requested frequency, but they offer no such gurantee. The determination of the highest possible frequency is a trial and error process.

The timing results are presented in Table 3.4. As we can see in the table, the initial VHDL generation takes a very short time compared to the rest of the process.

We provide a brief explanation on why we obtained these frequencies.

- Floating point operations frequencies - for each floating point component, a number of pipeline stages has to be specified as a configuration parameter. The higher the number, the higher the frequencies the component can achieve. Nevertheless this also means a larger area and an inefficient execution for lower frequencies. A solution is to increase the number of pipeline stages.

- Addition - performing a 32 by 32 addition can take a long time and can not be done in one cycle at very high frequencies. A solutions is to deduce the needed bit width reducing, for example, a 32 by 32 addition to a 16 by 16 addition, if possible, without changing the behavior of the application. Another solution is the introduction of additional pipeline stages.

3.4.3 Toolchain Retargetability

It is important to note that the proposed toolchain can be employed for different platforms. The development of such a toolchain is a very time and resource consuming, the design of the framework must be done with care to allow easy integration with 3rd party toolchains.

There are several directions that have to be taken into account:

- The use of an open and generic standard for communication between tools is the foundation for making a toolchain flexible. This was done from the beginning, when it was decided that C and XML annotations would be used as an interface between tools.

- The use of annotations (either in C or in XML) independent from the platform. A good example for such design is the OpenMP standard, which abstracts most of the details (such as the number of threads) from the users. All these functionalities should be based on functions in libraries, as changing the library is much easier than rewriting a code generator.

Kernel	DWARV time (s)	Xilinx time(s)	Frequency	Result	Reason
sad	<1	678	125		
sad		678	142		
sad		678	166		
sad		678	200		
sad		825	250	Fail	addition
satd	2	871	125		
satd		874	142		
satd		876	166		
satd		1114	200	Fail	addition
satd unrolled	3	1521	125		
satd unrolled		1521	142		
satd unrolled		1529	166		
satd unrolled		2474	200	Fail	addition
ffd	10	7038	125		
ffd		6471	142	Fail	routing
FracShift	10	900	125		-
FracShift		889	142		fp add-

Table 3.4: Compilation times using FPGA flow at different frequency constraints.

- The construction of an open build system. As specific compilers have to be used to build the final system, each with its own specificities, the development of an open build system will ease the control of the whole process for the developer.

The hArtes toolchain adheres to these principles, in general. In addition, we propose several improvements based on our experience:

- By excessively centralizing the data into one repository (XML) negative effects on the manageability of the data produced and consumed by different tools may occur. Lack of standardization of interfaces makes tool changes prohibitively expensive in effort.
- The responsibility of the different tools is not always clearly defined causing unwanted interference. For instance, register allocation should be the responsibility only of the low level compiler, otherwise errors might appear.
- By building a system that only relies on time stamps, severe penalties can happen in case files are regenerated. The build system should be able, at least for some type of files and very slow tools, to identify if the contents changed from the last code generation, for example by using hashing functions. This is especially relevant for FPGA synthesis tools, but there were cases in which the DSP compiler took more than 20 min in compilation time.

3.5 hArtes Applications

In this section, we give an overview of the applications that were used to validate the toolchain in the hArtes project. A performance analysis is performed and an initial mapping onto the platform is done, based on the results of the analysis. The profiling is done using the *gprof* tool [35].

3.5.1 Video application - The H.264 codec

The H.264 family represents a standard for video compression and decompression, developed jointly by the ITU-T and the ISO/IEC (with the name MPEG-4 AVC). The initial target of this standard was to halve the required bandwidth necessary for a given quality [88]. The first release of the standard was in 2003.

File type	Number of files	Total number of lines	Average lines per file
C source	51	33602	658.8627
Header	53	4565	86.1320
Assembly	21	11586	551.7142

Table 3.5: x264 application metrics.

From 2003, 10 revisions of the standard have been developed, which make this standard well known and studied [45].

The industrial partners in the project decided to use a free software library implementation, namely the x264 [84]. This library is actively developed and optimized for a broad range of microprocessors such as, the x86, the PowerPC (using AltiVec [21]), and the ARMv7 (using NEON).

Initial Analysis

The initial analysis was performed on the code obtained from the project source repository in the initial stages of the hArtes project. At the beginning, the hArtes hardware platform was not available. As a result, the initial tests where performed on a desktop machine, and, more specifically, an Intel(R) Core(TM)2 Duo CPU (E8500@3.16GHz) with 4 GB of RAM. The assumption was that the profile information would not substantially differ from that of the platform. More specifically, it would differ in absolute values, but not in relative values. This assumption holds for applications for which the general-purpose processors have similar capabilities. More specifically, it works for applications that do not use floating point operations as, for these operations, the embedded hardware floating point coprocessor available in the Intel Core will skew the results.

In Table 3.5, we present 3 of the main metrics that give an idea of the overall complexity of the application. We can see that a lot of code is written directly in assembly for different architectures. As we will see, this also affects the application structure and, as a result, an automated toolchain has to do more work to reveal the real structure.

Without any modifications, the profile results are summarized in Figure 3.8. Still, this profile information would not be very useful to either a human or automatic tool as, by checking the application source code we can

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 12.16     39.58    39.58 360123870     0.00     0.00  get_ref
 10.14     72.56    32.98  49361070     0.00     0.00  x264_pixel_satd_16x16
  9.92    104.84    32.28 183708960     0.00     0.00  x264_pixel_satd_8x8
  7.93    130.64    25.80 290354340     0.00     0.00  motion_compensation_chroma
  4.85    146.41    15.77    406080     0.04     0.04  x264_frame_filter
  3.72    158.50    12.09  13311540     0.00     0.00  x264_pixel_sad_x4_16x16
  3.41    169.58    11.08 230324820     0.00     0.00  x264_pixel_satd_4x4
  3.41    180.66    11.08  33480810     0.00     0.00  x264_pixel_satd_16x8
  3.40    191.71    11.05  34505700     0.00     0.00  x264_pixel_satd_8x16
  3.18    202.07    10.36  48297960     0.00     0.00  x264_pixel_sad_x4_8x8
```

Figure 3.8: The initial profile results on Intel x86 architecture for the x264 application.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.82    110.84   110.84 619761780     0.00     0.00  pixel_satd_wxh
 11.66    149.05    38.21 360123870     0.00     0.00  get_ref
  7.77    174.52    25.47 290354340     0.00     0.00  motion_compensation_chroma
  6.63    196.26    21.74 363844770     0.00     0.00  x264_pixel_sad_8x8
  6.59    217.85    21.59 112416120     0.00     0.00  x264_pixel_sad_16x16
  5.15    234.74    16.89    406080     0.04     0.04  x264_frame_filter
  3.59    246.52    11.78 108754950     0.00     0.00  x264_pixel_sad_8x16
  3.20    257.01    10.49 108532500     0.00     0.00  x264_pixel_sad_16x8
  1.89    263.19     6.18  37245600     0.00     0.00  refine_subpel
  1.87    269.33     6.14 119345190     0.00     0.00  quant_4x4
```

Figure 3.9: The initial profile results on Intel x86 architecture, without inlineing, for the x264 application.

see that, in fact, functions *x264_pixel_satd_16x16, x264_pixel_satd_8x8 and x264_pixel_satd_4x4* contain just another function call, which is inlined, thus hiding the real kernel. To avoid this, we performed the profile with inline disabled. This is done by using the GCC's option "-fno-inline-functions".

Additionally, the application is not compiled using the assembly code written for x86, to get an idea of the effort done on a general purpose processor without any specific extension (such as SSE or MMX extensions).

Without using inlineing, the results are show in Figure 3.9. We can see now that one functions already stands out, namely *pixel_satd_wxh*.

The next step was to analyze the functions that had a big impact on the total application execution time. From these functions, we note a peculiarity with the functions *x264_pixel_sad_16x16, x264_pixel_sad_8x8 and x264_pixel_sad_4x4*. These functions are not standalone functions, but functions created by macro expansion, like in Listing 3.6. We can consider the macro transformation as a hand optimization performed with a GPP architecture in mind. This might not fit as good a different architecture and prevents a tool to perform the necessary optimizations based on profile information. So, we modified by hand the application to make a new function *pixel_sad_wxh*, with the definition shown in Listing 3.7.

Listing 3.6: Initial definition for *x264_pixel_satd* functions.

```
#define PIXEL_SAD_C( name, lx, ly ) \
 int name( uint8_t *pix1, int i_stride_pix1,     \
           uint8_t *pix2, int i_stride_pix2 )    \
{                                                \
    int i_sum = 0;                               \
    int x, y;                                    \
    for( y = 0; y < ly; y++ )                    \
    {                                            \
        for( x = 0; x < lx; x++ )                \
        {                                        \
            i_sum += abs( pix1[x] - pix2[x] );   \
        }                                        \
        pix1 += i_stride_pix1;                   \
        pix2 += i_stride_pix2;                   \
    }                                            \
    return i_sum;                                \
}

PIXEL_SAD_C( x264_pixel_sad_16x16, 16, 16 )
PIXEL_SAD_C( x264_pixel_sad_16x8,  16,  8 )
PIXEL_SAD_C( x264_pixel_sad_8x16,   8, 16 )
PIXEL_SAD_C( x264_pixel_sad_8x8,    8,  8 )
PIXEL_SAD_C( x264_pixel_sad_8x4,    8,  4 )
PIXEL_SAD_C( x264_pixel_sad_4x8,    4,  8 )
PIXEL_SAD_C( x264_pixel_sad_4x4,    4,  4 )
```

Listing 3.7: Rewritten function *pixel_sad_wxh*.

```
int pixel_sad_wxh(uint8_t *pix1, int i_stride_pix1,
                  uint8_t *pix2, int i_stride_pix2,
                  int lx, int ly)
{
    int i_sum = 0;
    int x, y;
    for( y = 0; y < ly; y++ )
    {
        for( x = 0; x < lx; x++ )
        {
            i_sum += abs( pix1[x] - pix2[x] );
        }
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 30.38    110.16   110.16 619761780     0.00     0.00  pixel_satd_wxh
 26.65    206.78    96.62 693548340     0.00     0.00  pixel_sad_wxh
 10.69    245.55    38.77 360123870     0.00     0.00  get_ref
  7.19    271.63    26.08 290354340     0.00     0.00  motion_compensation_chroma
  4.55    288.11    16.48    406080     0.04     0.04  x264_frame_filter
  1.66    294.13     6.02 119345190     0.00     0.00  quant_4x4
  1.64    300.07     5.94  37245600     0.00     0.00  refine_subpel
  1.32    304.85     4.78  29886060     0.00     0.01  x264_me_search_ref
  1.23    309.32     4.47 121346040     0.00     0.00  sub4x4_dct
  0.95    312.75     3.43  34650000     0.00     0.00  ssim_4x4x2_core
```

Figure 3.10: Profile on Intel x86 processor.

```
            pix1 += i_stride_pix1;
            pix2 += i_stride_pix2;
        }.
        return i_sum;
}
```

The final profile information obtained after the above modifications is show in Figure 3.10.

At the moment the hArtes hardware platform became available, we performed also tests in that environment. The results obtained are presented in Figure 3.11.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  9.23     56.30    56.30  2018801     0.03     0.03  x264_pixel_satd_16x16
  8.35    107.20    50.90 29341194     0.00     0.00  x264_pixel_satd_4x4
  7.58    153.41    46.21  9513701     0.00     0.00  get_ref
  7.14    196.94    43.53  5916503     0.01     0.01  x264_pixel_satd_8x8
  6.75    238.13    41.19    26991     1.53     1.53  x264_frame_filter
  4.43    265.15    27.02 63421925     0.00     0.00  x264_cabac_encode_decision
  3.96    289.32    24.17  6960002     0.00     0.00  motion_compensation_chroma
  3.19    308.77    19.45  3774643     0.01     0.01  block_residual_write_cabac
  2.44    323.65    14.88  9962094     0.00     0.00  quant_4x4
  2.27    337.51    13.86  1247495     0.01     0.01  sub8x8_dct
  2.13    350.50    12.99   254811     0.05     0.48  x264_mb_analyse_intra
```

Figure 3.11: Profile on the ARM processor for the x264 application.

By comparing the two results we can see that although the same functions appear as the most computing intensive, the ratio between their execution times differs quite a bit. The difference, for the common function, is show in Table 3.6.

Function Function	Percent on Intel	Percent on ARM	Difference Difference
get_ref	12.18	7.58	62.23%
x264_pixel_satd_16x16	10.13	9.23	91.12%
x264_pixel_satd_8x8	9.87	7.14	72.34%
motion_compensation_chroma	8.01	3.96	49.44%
x264_frame_filter	5.05	6.75	133.66%

Table 3.6: Comparison of profile information between Intel x86 and ARM for x264 application.

Mapping

By using the information obtained from the previous step, it was decided to map the most time consuming two functions onto the FPGA. The first step was to assess the speedup obtained for each kernel. Not all kernels give a speedup when moved to another processing elements so even if they are high ranked in the profile list, it does not mean that they should be mapped on to the FPGA.

The normal approach to performing unit tests is to obtain the data used by one execution of the kernel and use it with the kernel run on the processing element. In our case, the kernel uses various data sizes as inputs, depending on the video that it encodes, so we could not use only one data set. The steps performed are the following:

- The automatic creation, using DWARV tool, of an FPGA implementation for both *pixel_sad_wxh* and *pixel_satd_wxh*.
- The instrumentation of the application (by hand first, later this was possible using the hArtes toolchain) to obtain all the possible data sizes used by the kernel.
- The execution of the kernels using the collected data.

By centralizing the data on several tests video [90], we obtained the results outlined in Table 3.7 and in Table 3.8. From the tables, we can observe that, for the *sad* kernel, two combinations of parameters represent 71% of the invocations. For the *satd* kernel the situation is more balanced, where four combinations of parameters represent 60% of the invocations.

Video	s1=16, s2=16 lx= 8, ly=16	s1=16, s2=16 lx= 8, ly= 8	s1=16, s2=16 lx=16, ly=16	s1=16, s2=16 lx=16, ly= 8	s1=16, s2=240 lx=16, ly= 8	s1=16, s2=240 lx= 8, ly= 8	s1=16, s2=240 lx=16, ly=16	s1=16, s2=240 lx= 8, ly=16	s1=16, s2=32 lx= 8, ly= 8	s1=16, s2=32 lx= 8, ly=16	s1=16, s2=32 lx=16, ly= 8	s1=16, s2=32 lx=16, ly=16
akiyo	0	1	0	0	8	39	36	8	1	0	0	0
carphone	1	3	2	1	12	43	13	12	4	1	1	1
claire	0	2	1	0	8	39	32	8	2	0	0	0
coastguard	1	3	1	1	12	46	12	12	5	1	1	1
container	0	1	0	0	6	57	24	6	1	0	0	0
foreman	1	3	2	1	12	43	12	12	5	1	1	1
hall	0	1	0	0	5	57	26	5	0	0	0	0
m-america	0	2	1	0	7	46	24	7	3	0	0	1
mobile	1	3	1	1	13	43	11	13	6	2	2	1
news	0	2	1	0	9	46	25	9	1	0	0	0
salesman	0	1	0	0	7	56	21	7	1	0	0	0
silent	1	2	1	1	9	52	17	10	1	0	0	0
suzie	1	3	2	1	11	43	15	11	4	1	1	1
Total	6	27	12	6	119	610	268	120	34	6	6	6
Average from total (%)	0	2	0	0	9	50	21	9	2	0	0	0

Table 3.7: Number of calls for each combination of parameters and percent from the total number of invocations for *sad*, when running on a set of sample videos.

Video	s1=32, s2=16 lx=16, ly=16	s1=32, s2=16 lx= 8, ly= 8	s1=16, s2=16 lx= 8, ly=16	s1=16, s2=16 lx= 8, ly= 8	s1=16, s2=16 lx=16, ly=16	s1=16, s2= 8 lx= 4, ly= 4	s1=16, s2= 8 lx= 8, ly= 8	s1=16, s2=16 lx=16, ly= 8	s1=16, s2= 8 lx= 8, ly= 4	s1=16, s2= 8 lx= 4, ly= 8	s1=16, s2=240 lx=16, ly= 8	s1=16, s2=240 lx= 8, ly= 8	s1=16, s2=240 lx=16, ly=16	s1=32, s2=16 lx= 4, ly= 4
akiyo	2	5	3	13	12	18	19	2	4	4	1	4	3	3
carphone	2	4	3	12	4	22	8	3	6	6	1	4	1	17
claire	2	4	2	11	9	21	18	2	5	5	0	3	2	8
coastguard	2	4	3	13	4	16	5	3	4	4	1	4	1	26
container	3	6	1	14	8	18	10	1	2	2	0	4	1	23
foreman	2	4	4	13	4	22	7	4	7	7	1	4	1	14
hall	3	6	1	15	9	22	12	1	2	2	0	4	1	15
m-america	2	4	2	10	6	25	16	2	5	5	0	3	1	13
mobile	3	6	5	17	4	23	5	5	7	6	1	6	1	2
news	2	5	3	13	7	20	10	3	5	5	1	4	1	13
salesman	3	7	2	18	9	24	10	2	4	4	0	5	1	3
silent	2	5	3	14	5	24	8	3	6	6	0	4	1	12
suzie	2	4	3	10	4	21	9	3	6	6	0	3	1	20
Total	30	64	35	173	85	276	137	34	63	62	6	52	16	169
Average from total (%)	2	5	2	14	7	22	11	2	5	5	0	4	1	13

Table 3.8: Number of calls for each combination of parameters and percent from the total number of invocations for *satd*, when running a set of sample videos.

For a general purpose architecture, it is worth to make one instance for each combination of parameters. In this case, the parameters will be constant, except the pointers to the data to process. This would allow the compiler to perform more optimizations than in the case in which the parameter is variable. An example of such an optimization is full unroll.

For a reconfigurable architecture, where area might be a concern, we choose to optimize just the original kernel and not specific instances. By having all instances configured, the available area would be exhausted.

The *satd* kernel

By synthesizing the *sadt* kernel, we obtained the results presented in Table 3.9. Running this kernel for the identified cases in the application gives the results presented in Table 3.10. From these results, we can see that the FPGA is usually faster (up to 2.85x in one case) although there are cases in which it is slower (0.81x). This is directly related to the number of iterations performed: the worst performance is obtained for the smallest number of iterations (lx = ly

Measure	Synthesis results	Percent of available
Slices	1,717	4%
DSP48s	0	0%
Frequency	141.46	-

Table 3.9: The synthesis statistics for the *satd* kernel on the Virtex4-ML410 board.

Case	Times μs			Speedup	
	FPGA	FPGA with transfer	ARM	FPGA vs ARM	FPGA w/t vs ARM
s1=32, s2=16 lx=16, ly=16	46	194	108	2.37	0.56
s1=32, s2=16 lx= 8, ly= 8	16	152	28	1.68	0.18
s1=16, s2=16 lx= 8, ly=16	26	167	56	2.11	0.33
s1=16, s2=16 lx= 8, ly= 8	16	148	28	1.68	0.19
s1=16, s2=16 lx=16, ly=16	46	186	108	2.36	0.58
s1=16, s2= 8 lx= 4, ly= 4	9	135	7	0.81	0.05
s1=16, s2= 8 lx= 8, ly= 8	16	149	28	1.68	0.19
s1=16, s2=16 lx=16, ly= 8	26	159	54	2.09	0.34
s1=16, s2= 8 lx= 8, ly= 4	11	138	14	1.22	0.10
s1=16, s2= 8 lx= 4, ly= 8	11	143	14	1.24	0.10
s1=16, s2=240 lx=16, ly= 8	26	212	60	2.31	0.28
s1=16, s2=240 lx= 8, ly= 8	16	204	31	1.86	0.15
s1=16, s2=240 lx=16, ly=16	46	303	130	2.85	0.43
s1=32, s2=16 lx= 4, ly= 4	9	140	7	0.81	0.05
s1=16, s2=240 lx= 8, ly=16	26	284	66	2.50	0.23

Table 3.10: Processing times and speedups for kernel *satd* in various scenarios.

= 4), while the best is obtained for a large number of iterations. Additionally, when taking into account the transfer of memory performed automatically between the ARM processor and the FPGA, the overall execution time is larger on the FPGA, as it can be seen in column 3. This is due to a very inefficient transfer speed between the main memory and the local FPGA memory.

By inspecting the code, we noticed that several internal loops had fixed size and used just local variables. Although memory transfer is the bottleneck in this case, we choose to optimize the FPGA version to get an idea of the possible performance improvements obtained by simple code modifications. These results are presented in Table 3.11. As we can see from the table, the performance consistently improves for some cases, with speedup up to 7.58x compared to the ARM processor.

Case	Times μs			Speedup	
	FPGA	FPGA with transfer	ARM	FPGA vs ARM	FPGA w/t vs ARM
s1=32, s2=16 lx=16, ly=16	17	165	109	6.32	0.66
s1=32, s2=16 lx= 8, ly= 8	9	144	28	2.94	0.19
s1=16, s2=16 lx= 8, ly=16	14	152	55	3.94	0.36
s1=16, s2=16 lx= 8, ly= 8	9	141	28	2.92	0.20
s1=16, s2=16 lx=16, ly=16	19	156	108	5.56	0.69
s1=16, s2= 8 lx= 4, ly= 4	9	134	7	0.79	0.05
s1=16, s2= 8 lx= 8, ly= 8	11	140	28	2.43	0.20
s1=16, s2=16 lx=16, ly= 8	12	143	54	4.64	0.38
s1=16, s2= 8 lx= 8, ly= 4	8	135	14	1.68	0.10
s1=16, s2= 8 lx= 4, ly= 8	8	139	14	1.73	0.10
s1=16, s2=240 lx=16, ly= 8	12	197	60	5.12	0.31
s1=16, s2=240 lx= 8, ly= 8	9	196	31	3.22	0.16
s1=16, s2=240 lx=16, ly=16	17	272	130	7.58	0.48
s1=32, s2=16 lx= 4, ly= 4	8	137	7	0.90	0.05
s1=16, s2=240 lx= 8, ly=16	12	270	66	5.53	0.24

Table 3.11: Processing times and speedups in various scenarios for *satd* unrolled.

Measure	Synthesis results	Percent of available
Slices	724	1%
DSP48s	0	0%
Frequency	181.71	-

Table 3.12: The synthesis statistics for the *sad* kernel on the Virtex4-ML410 board.

The *sad* kernel

By synthesizing the *sat* kernel, we obtained the results presented in Table 3.12. By checking the results, we can see this is a very simple kernel that takes very little resources on the FPGA and it can run at high frequencies. Still, due to the fact that it less computation intensive than *satd*, the total execution time, when considering memory transfers, is even lower when compared to the ARM than the ones obtained for *satd*.

Conclusions

From the analysis performed on the application, we can see that it is not sufficient to only consider the profiling information. Code transformation might give a skewed view over which functions take most of the application execution time. Additionally, bulk profiling information is not sufficient in all cases.

Rather, a way of detecting the correlation between parameters should be available in an instrumentation tool to help the designer making the best decision when optimizing. Such a method is described in Chapter 4.

As expected, unroll provides a significant improvement as it exposes parallelism for FPGA. Nevertheless, this comes at the cost of increasing the used size (by almost 4 times) and reducing the maximum execution frequencies.

3.5.2 Immersive audio - Beamforming and Wavefield Synthesis

In this section, we describe the work performed on the implementation of a multi-beam, broadband, beamforming and Wave-Field Synthesis (WFS) application. These audio algorithms can be used in audio-visual transmission scenario, like a telepresence application. On the acquisition side, a camera tracks the recording directions (for example, by tracking human faces in the recorded scene). By using the beamformer algorithm [81], the sound waves are filtered based on direction and sent through a transmission medium to the rendering side. There, the spatio-temporal properties of the recording space are reproduced using a WFS algorithm [15]. Such an utilization scenario is presented in Figure 3.12. Compared to other stereophonic approaches, the properties that are reproduced are not limited to a sweet spot, but to a much wider area, depending on the number of speakers array.

Initial Analysis

The initial analysis was performed on the code obtained from the project source repository of the hArtes project, as the code for this beamformer, developed by Fraunhofer IGD, is not open source. The host machine on which the tests were performed was an Intel(R) Core(TM)2 Duo CPU (E8500@3.16GHz), with 4 GB of RAM.

As before, in Table 3.13, some metrics are presented which give an idea of the complexity of the application. This application is simpler than the x264 application and does not contain assembly code. By being more high level, we can expect to map more easily onto a heterogeneous architecture using design space exploration tools. Most of the computation performed is performed in floating point, which is a major drawback for the simple ARM processor embedded in the hArtes platform, which does not have a Floating Point Unit (FPU) attached.

The beamforming and the wavefield synthesis parts of the application share the

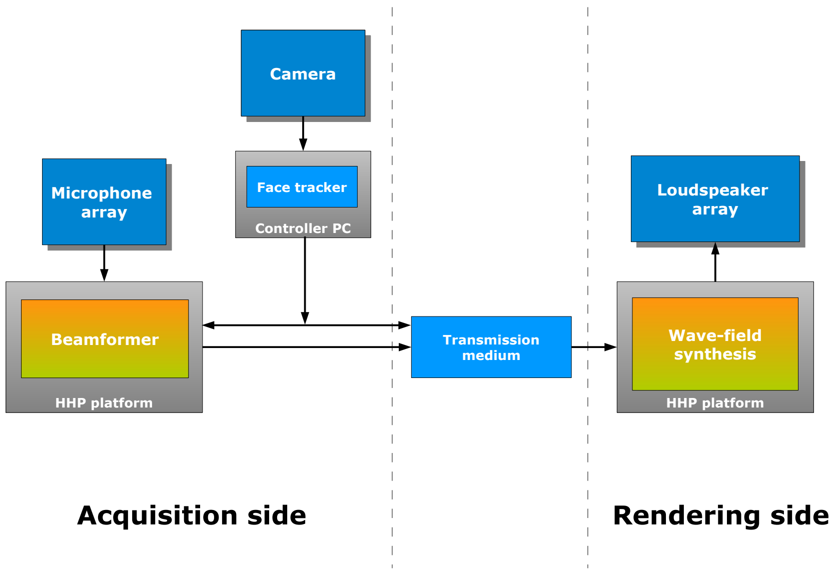

Figure 3.12: Immersive Audio Architecture.

same computational structure when dealing with the array of microphones/speakers. Additionally to this processing, the beamforming application contains a module to determine the position of the sound source. This is executed in parallel with the audio processing, and it is triggered by an external event. Although computing intensive, this module contains very simple code that relies on floating point operations, so an in-depth analysis is not required.

The application was run with the default parameters. Among them: the sampling frequency (48 khz), the number of sources (2) and the room size (15m). The results of the profile on the host are presented in Figure 3.13. We can see

File type	Number of files	Total number of lines	Average lines per file
C source	14	2350	167.8571
Header	16	510	31.8750
Assembly	0	0	

Table 3.13: The WFS application metrics.

that only one function takes almost 90% of the execution time.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 89.75      5.08     5.08    20000    0.00     0.00  fFD_RealFIR_Pair_fpga
  3.18      5.26     0.18        1    0.18     0.18  wav_store
  2.12      5.38     0.12    20000    0.00     0.00  fillAudioPairBuffer
  1.77      5.48     0.10    80000    0.00     0.00  sumAndWeightChannels
  1.41      5.56     0.08    20000    0.00     0.00  interleave
```

Figure 3.13: Profile information on the Intel x86 architecture for the WFS application.

Running the application on the board gives the results outlined in Figure 3.14. This profile includes the application functions and the floating point emulation functions. The floating point emulation functions are automatically inserted by GCC when the processor does not have native instructions to deal with the floating point operations. In the figure, there are several floating point emulation function, from which two take more that 55% of the total application execution time. These are the emulation functions for addition (*__eabi_fadd*) and multiplication (*__eabi_fmul*). As *gprof* does not provide information about which functions call these floating point emulation functions, we investigated this by using a binary instrumentation tool.

The binary instrumentation tool we used is the *pintool* which provides a rich API for instrumenting binary files under Linux. Our goal was to count how many floating point additions and multiplications were performed in each function of the application. The results, obtained after the instrumentation and execution of the application, are presented in Table 3.14. We can see that more than 90% of the floating point operations are performed in *fFD_RealFIR_Pair_fpga*. This means that, in addition to the 33.33% reported in Table 3.14, another 50% (which represent 90% of the floating point operations time in the WFS application running on ARM processor) is spent in the same function. In total, the function *fFD_RealFIR_Pair_fpga* represents more than 83% of the total execution time, a result that confirms the profile from Intel.

Mapping

fFD kernel

Using DWARV tool, and the *fFD* C code as input, the VHDL was automatically generated. Then, we first tested the *fFD* kernel separately from the application to asses the benefits it could bring. The synthesis results are presented in Table 3.15. We note that, the area occupied by the kernel is rather large, partly

Procedure	Percent of floating point additions and multiplications used
fFD_RealFIR_Pair_fpga	93.40 %
sumAndWeightChannels	6.44 %
calculateGains_pQ	0.11 %
calculateDelays_pQ	0.02 %

Table 3.14: Percent of floating point operations, from the total number of operations used, for the WFS application.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 43.75      0.42     0.42                             __aeabi_fadd
 33.33      0.74     0.32       30    10.67    10.67  fFD_RealFIR_Pair_fpga
 12.50      0.86     0.12                             __aeabi_fmul
  2.08      0.88     0.02                             __floatsisf
  2.08      0.90     0.02                             __subsf3
  1.04      0.91     0.01       63     0.16     0.16  zeroBuffer
  1.04      0.92     0.01       30     0.33     0.33  fillAudioPairBuffer
  1.04      0.93     0.01                             __adddf3
  1.04      0.94     0.01                             __aeabi_f2d
  1.04      0.95     0.01                             __aeabi_l2f
```

Figure 3.14: Profile information on ARM processor for WFS application.

due to the extensive use of floating point units (8 units, including addition, multiplication and division).

Even if the parameters used by this kernel affect its total execution time, by instrumenting the application, we observed that for a specific configuration of the application, the parameters are constant. We used this set of parameters to assess the performance gain that could be obtained for our configuration and the corresponding results are presented in Table 3.16.

Measure	Synthesis results	Percent of available
Slices	10.781	25%
DSP48s	12	7%
Frequency	122.98	-

Table 3.15: The synthesis statistics for the *fFD* kernel on the Virtex4-ML410 board.

Input size	Times μs			Speedup	
	FPGA	FPGA with transfer	ARM	FPGA vs ARM	FPGA w/t vs ARM
256	2047	3898	13169	6.43	3.38
512	4410	6965	26435	5.99	3.80
1024	9370	13671	58407	6.23	4.27

Table 3.16: Execution of units tests for *fFD* kernel on Virtex4-ML410.

Conclusions

Floating point operations are by themselves one of the most computational intensive parts of an application. Additionally, the floating point hardware units take a lot of area on the reconfigurable unit. As a result, a careful trade-off has to be made in case of parallelism between area occupancy and multiple floating point units.

3.5.3 In Car Audio - Enhanced Listening Experience

The focus of this application is to enhance the listening experience for travelers in a car. This is a hard subject due to the inherently dynamic nature of the environment, which makes it a less than ideal environment for listening. The noise, spatial and spectral properties of the reproduced field, change the rendering of the sound from the loudspeakers. The system used here as an example is a real time application with two objectives:

- The development of a complete set of audio algorithms for improving the audio quality, by taking into account some features of the cabin.
- The development of a modular system that can be either adapted to other environments or that can use other algorithms.

In the current state, the application is composed of [17]:

- A crossover network that splits the audio signal into different frequency bands.
- An equalizer that compensates the distortion caused by the resonances in the car environment.

- A phase distortion correcter, needed as each filter in the equalizer can introduce a phase distortion.

Initial Analysis

This application was developed for the Diopsis platform, so it contained library calls for that platform. The metrics that give an idea of the complexity of the application are given in Table 3.17. The complexity is similar with the complexity of the WFS application.

File type	Number of files	Total number of lines	Average lines per file
C source	17	2353	138.4117
Header	21	814	38.7619
Assembly	0	0	

Table 3.17: In car Audio Enhanced Listening application metrics.

Compared to the other applications, this application is composed of several simple kernels with some code that transfers the data to the appropriate location. By using the developer knowledge of the application, the most promising kernel for acceleration was identified to be *FracShift*.

Mapping

FracShift kernel

The kernel is represented by a delay line, followed by a floating point addition and accumulation. By using DWARV to automatically generate the VHDL, and without any modifications on the code, we obtained the results in Table 3.18. Even if there is only a modest speedup, we notice that the memory transfers are not an essential part of the total execution time. This means that, if we can optimize the computational part of this kernel, the speedup can increase dramatically.

To better understand the differences between a GPP and a FPGA, we show the code in Listing 3.8.

Listing 3.8: The *FracShift* kernel.

```
for (n=0; n<256; n++)
```

Size	Times μs			Speedup	
	FPGA	FPGA with transfer	ARM	FPGA vs ARM	FPGA w/t vs ARM
256	11017	11263	19013	1.73	1.69

Table 3.18: Execution of units tests for the *FracShift* kernel on Virtex4-ML410.

```
{
    for(i=100;i>=0;i--)
        Z[i+1]=Z[i];
    Z[0]=in[n];

    for(i=0;i<101;i++)
        tmp_dbl[i]=h[i]*Z[i];

    out[n] = 0;
    for(i=0;i<101;i++)
        out[n]=out[n]+tmp_dbl[i];
}
```

At first we can observe that there are no dependencies between the iterations of the second loop. Still, as it uses a floating point operation, it is not possible to fully unroll it, as one floating point operation takes a lot of resources, so it will not be possible to put 100 units using current FPGA technology. In the future, this might become a viable option.

Noteworthy are the first and the last loops. Both can take advantage of the huge parallelism that can be available on an FPGA. Nevertheless, to achieve this with the current infrastructure some modifications have to be done.

The first loop has an anti-dependence (iteration i reads a value written in operation i+1), which would prevent it from being parallelized. The solution to this is to use scalar expansion and to introduce for each $Z[i]$ an internal register. As the hardware compiler chooses the allocation type for a variable depending on its type: for pointers used in parameters, it will use, the memory, so the scalar replacement will no take place. By copying at the beginning of the function the Z array to a local defined Z array, scalar replacement becomes possible and the first loop will execute in 1 cycle (of course, the area used will increase).

The modified code is presented in Listing 3.9 and the corresponding results are presented in Table 3.19. We note that the optimized version provides a huge performance gain when compared to the non optimized version.

Listing 3.9: Optimized code for *FracShift* kernel.

```
#pragma map generate_hw 1
void FracShift_process (float *in, float *out,
   float *Z_out, float *tt, float *h_out)
{
     float Z[101];
     float h[101];

     float tmp_dbla[26];
     float tmp_dblb[26];

     int n,i;

     for (i = 0; i < 101; i++) {
         Z[i] = Z_out[i];
         h[i] = h_out[i];
     }

     for (n = 0; n < 256; n++) {
         Z[100] = Z[99];
         Z[99] = Z[98];
         // ... cut for brevity ...
         Z[2] = Z[1];
         Z[1] = Z[0];
         Z[0] = in[n];

         out[n] = 0;

         for (i = 0; i < 100; i += 4) {
             tmp_dbla[i / 4] = h[i + 0] * Z[i + 0]
               + h[i + 1] * Z[i + 1];
             tmp_dblb[i / 4] = h[i + 2] * Z[i + 2]
               + h[i + 3] * Z[i + 3];
         }
         tmp_dbla[25] = h[100] * Z[100];
         tmp_dblb[25] = 0;

         for (i = 0; i < 26; i += 1) {
             out[n] = out[n] + tmp_dbla[i];
             out[n] = out[n] + tmp_dblb[i];
         }
     }

     for (i = 0; i < 101; i++) {
```

Size	Times μs			Speedup	
	FPGA	FPGA with transfer	ARM	FPGA vs ARM	FPGA w/t vs ARM
256	2389	2646	19176	8.03	7.25

Table 3.19: Execution of units tests for the *FracShift* kernel with the unroll optimization on Virtex4-ML410.

```
        Z_out[i] = Z[i];
    }
}
```

3.6 Identified Problems

In the following, we summarize the problems identified and comment on the solutions.

(A) **Variable speedup based on parameter values**. This was seen in Section 3.5.1 for both kernels. One solution is to adapt at runtime based on the value of the parameters and the speedup. Such a solution is presented in Chapter 4.

(B) **Automatic memory transfers to/from scratch pad memory**. One of the issues with a new platform is that the developer has to do a lot of manual operations, even if sometimes an automatic way would be good enough from a performance point of view. Automatic memory transfer is one of those cases, in which the toolchain automatically manages the transfers based on the heap management at runtime or based on the compiler analysis at compile-time.

(C) **Memory allocation in scratch pad memories based on function speedup**. For all the application analyzed, the identification of the places where the memory is allocated and decide on an optimal allocation at compile-time is not feasible. By taking into account that the available memory can change from execution to execution it is obvious that a runtime solution is better. We analyze such a solution in Chapter 5.

(D) **The identification of the optimal balance between several parallel kernels on a reconfigurable device**. For the WFS application, making a balance between the number of kernel instances that compute the output waves and the number of kernels that compute the coefficients when needed, does not have an obvious solution. An ILP approach and a runtime algorithm that solve this problem are presented in Chapter 6.

(E) **The determination of the best parameters for specific optimizations, to take fully advantage of the reconfigurable device**. The two cases for this are the loop unroll number and the number of variables that are made local. Most of the applications are affected by this issue (like the applications presented in Section 3.5.1 and Section 3.5.3). Although a known optimization, in the context of the reconfigurable device, this has different implications, such as reducing the frequency by making routing harder.

(F) **Profile has to be done on the target platform mostly for floating point applications**. As typically running on the hardware platform is more complex than running on a desktop computer, after the performed analysis, we can conclude that for integer only applications, this is good enough for an initial performance assessment. Nevertheless this will not hold for floating point applications so, for those, profiling has to be done on the real hardware platform.

We will study the above problems on the WFS and H.264 applications as they fit into the focus of our thesis. The in car audio application described in Section 3.5.3 is also affected by the allocation problem, but, in that case, the solution would involve major modifications in the C-to-VHDL compiler. Such extension is outside the scope of the thesis.

3.7 Contributions

The work presented in this chapter was implemented collaboratively by a large team composed of multiple universities and companies. To clarify what is our contribution we will list here the major contributions:

- Support University of Ferrara in designing the hHP to conform to the Molen machine organization.
- Implement the Molen Abstraction Layer (sections 3.3.2). Involving the following: implement a runtime support library for the Molen primitives

and modify the GCC compiler to schedule Molen primitives calls when OpenMP pragmas are used.

- Modified the GCC code generation to replace annotated calls with calls to the Molen primitives. Described in Section 3.4.1.
- Develop the infrastructure needed for the generation and integration of the result of the backend compilers (GPP, DSP and VHDL) into one executable, shown as last block in Figure 3.7. This work is described in Section 3.4.
- Implemented improvements to the tool flow to allow better debug and error reporting capabilities (described in Section 3.4.2).

3.8 Summary

In this chapter, we presented the work done in the context of hArtes project. We analyzed the applications and we highlighted the main problems that have to be solved before obtaining an efficient implementation on a Molen architecture. A description of the development flow has been proposed to provide insights on the complexity of a toolchain for heterogeneous platforms. Problems identified in the flow were summarized and will be addressed in the following chapters.

Note. The content of this chapter is based on the the following paper:

K.L.M. Bertels, V.M. Sima, Y. D. Yankova, G. Kuzmanov, W. Luk, G. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, A. Michelotti, **HArtes: Hardware-Software Codesign for Heterogeneous Multicore Platforms**, IEEE Micro, October 2010, Special Issue on European Multicore Processing Projects.

4 Runtime Hardware/Software mapping

In this chapter, we propose a solution for Problem A [1] given in Section 3.6. The problem we address is the following: the execution time of some kernels varies based on their input data and on the processing elements on which they are executed. For each instance of input data a choice of processing element must be made, so that the total execution time is minimized. Considering our platform, the mapping problem is reduced to the problem of choosing between the General Purpose Processor (GPP) and the Field Programmable Gate Array (FPGA). Our proposed algorithm relies on runtime information and will improve the execution time of the application by making the best possible mapping at runtime. It will use compile-time information to guide the profiling process.

4.1 Introduction

Compared to compile-time optimizations, runtime optimizations can access also information about the running state of the application. Using this runtime information, adjustments can be made to the application mapping to take advantage of any speedup opportunity. The Molen programming paradigm gives us a clear environment in which we can develop such an algorithm. More specifically, for functions that have an execution time dependent on parameters, we propose an on-line adaptive decision algorithm to determine if the benefit of running that function in hardware outweighs the overhead of transferring the parameters, managing the start and stop of the execution and obtaining the result. In this way, the compiled application can adjust on the fly to new platforms where one or more characteristics of the platforms are different

[1] the speedup of a function depends on the values of its parameters.

from the platform for which the application was originally compiled.

The algorithm is applied on the x264 implementation of the H.264 state of the art video codec. The improvement for dynamically mapping the *satd* kernel is up to 24%. We also determine the overhead and execution time ranges in which this optimization is useful and what other factors can affect it.

In this chapter, we present a decision algorithm called Adaptive Mapping Algorithm (AMAP). Its key feature is that it takes into account particularities of a function such as the types of the parameters and the history of the execution and decides which implementation to use for the execution of that instance. One main novelty of the algorithm is that it takes this decision as late as possible, before each run, so it can make better decisions than a compile-time algorithm. After a call, the profile information obtained will be stored and will be used when taking the decision for the next executions of that function.

We describe the algorithm in the following sections, as following. We start by presenting the previous work. Then we give a detailed motivational example, based on one the application presented in Chapter 3 and specify the exact problem. Next, we continue with a detailed description of the runtime algorithm in Section 4.4. The results of the algorithm are shown in Section 4.5 and we end with conclusions in Section 4.6.

4.2 Background and Related Research

One of the main research directions for hardware/software mapping is mapping and partitioning at compile time using static analysis [4] [12] [59] [68] [61] [46]). The main drawback of compile time analysis is that a lot of relevant information is missing, leading to inefficient mapping decisions. In comparison our approach will not suffer from this limitation as the decision is delayed until runtime when all the information is available.

Runtime based approaches have been proposed, that, decide on a hardware/-software partitioning, but do it based on compile time information [72] [42] [30]. However, for these algorithms, the execution information about the application remains fixed and pre-computed at the moment of the compilation. Our approach will adapt to the changing conditions by taking the mapping decision at runtime based on the information collected during the actual execution.

Another possible issue with existing approaches is that they rely on an application model which is not directly available in existing applications, such as Kahn Process Networks (KPN) networks ([77]) or direct acyclic graph ([12], [59]). From this point of view we rely on plain C, which is a language widely used for embedded development [83].

4.3 Problem Definition

When optimizing an application for a reconfigurable platform, two of the most important steps are the hardware/software partitioning and mapping. The role of these phases is to determine which functions/tasks will be implemented in hardware and which in software. In previous work, the decision was taken at compile-time based on various attributes depending on the partitioning and mapping algorithms. Examples of such attributes and how they are obtained are the estimated execution time obtained for example from statistical models based on source code metrics, profile information obtained from running with one or multiple given inputs, hardware area available and data dependencies between tasks. The attributes are considered independent on the input data received by each kernel. The disadvantage of this approach is that, if the attribute depends on the input data an approximation has to be made. The most common approach is to use an average over the possible attribute values. For the same tasks, some of the attributes (such as execution time) can have different values based on the input data. Using only one value in the decision process of the mapping algorithm implies that some optimizations possibilities are lost.

Consider that the hardware software mapper decided that computation f should be implemented in hardware. The total execution time in hardware is represented by the following equation:

$$t_{hw\_exec} = t_{setup} + t_{exec} + t_{return} \tag{4.1}$$

The value t_{setup} represents the time needed for parameter transfer, memory setup and to start the hardware. This should include any overhead introduced, for example, by the operating system or the hardware control unit. The value t_{return} includes the time needed to retrieve the result, copy the data if necessary and stop the hardware. The value t_{exec} is the time in which the hardware unit processes the data and provides the results.

Each computation can have several parameters that allow the execution to be adapted to the current needs. For example, specifying the length of the matrices that have to be multiplied is such a parameter. If the times in Equation (4.1) are independent of the parameters, the total execution time t_{hw_exec} can be computed at compile-time. The choice of using the hardware or the software implementation can then be taken at compile-time without loosing any possibility of optimization. On the other hand, if any of the aforementioned times depend on parameter values, the decision that is taken at compile-time could be suboptimal for some cases.

As a first example of such a case, we present a function that has a variable execution time based on parameters. The code of the function is given in Listing 4.1. This code is derived from the x264 application, presented in Section 3.5.1, but it is simplified for presentation purposes. It is clear that the execution time depends on the parameters *lx* and *ly* as these two parameters control the number of iterations. We can see that, for this function, multiple execution times are possible depending on the parameters. Even more the 'speedup' (ratio between the software and hardware execution time plus overhead) is not constant. A static, compile-time, hardware software partitioner could use just the average of the execution times or the average speedup when deciding on which processing element to map a kernel. If, for example, in half of the executions the kernel will have a $2/3$ speedup and in half of the cases the same kernel will have a 2 speedup, on average, the speedup will be 1, and the mapper will decide not to map it to the processing element. This misses optimizations, as in half of the cases, mapping it to the processing element would be beneficial.

Listing 4.1: Motivational example from x264 application.

```
int pixel_sad_wxh(uint8_t *pix1, int i_stride_pix1,
 uint8_t *pix2, int i_stride_pix2, int lx, int ly) {
  for( y = 0; y < ly; y++ ) {
    for( x = 0; x < lx; x++ ) {

      ... computations ...

    }

    pix1 += i_stride_pix1;
    pix2 += i_stride_pix2;
  }
}
```

A second example, when parameters could determine the speedup, is related to the initialization overhead (t_{setup}). In order to work correctly, the kernel needs data in the local FPGA memory. This data has to be transferred from main memory. The kernel might read the data using a stride, or with another predetermined access pattern. Using this knowledge, the transfer size could be reduced, but deducing all the information necessary, such as, the stride and the total size of the memory accesses, from the source code is a complex task which is not supported by the current compilers. Without any further analysis, the simplest way is to transfer the entire block used. Accessing the data directly from main memory would introduce extremely high latencies, while providing a local cache would increase the hardware usage.

Problem statement: When both a software and a hardware implementation for a function are available, taking into account the overheads and the current input data, determine which of the two implementations lead to the shortest execution times.

The overheads are also affected by the particularities of the architecture such as the time needed to transfer the parameters, the time needed to start/stop the execution of the hardware function (t_{setup}) and the time needed to retrieve the result (t_{return}). Our algorithm takes this decision at runtime.

The main advantage of this approach is that the decision is taken based on the current state of the system. One example of an unpredictable event that changes the state of the system is the start of a different application. In this way, our algorithm solves Problem A [2] from Section 3.6.

4.4 Conditional Hardware Execution for Molen

As instrumentation and profiling in a real environment are difficult and error prone, we propose a solution that will react dynamically, when the application runs, to the changing conditions. We assume that the designer of the application, or the toolchain, have already identified a set of candidate computations for hardware execution. For this set of candidates, detailed profiling information would be needed in order to be able to take a decision at compile-time. This is not always possible because of two reasons: the behavior of the candidate functions can change depending on the parameters, and the running conditions might change because of events external to the application, such as multiple application running or power constraints that affect the system. The main idea is to save the values of the parameters for each function call together with the execution time. The next time a function is called with the same parameters an estimate can be done on whether it is more efficient to run the hardware version or the software version. The algorithm will execute both versions, until it has sufficient data to compare the implementations.

[2]the speedup of a function depends on the values of its parameters.

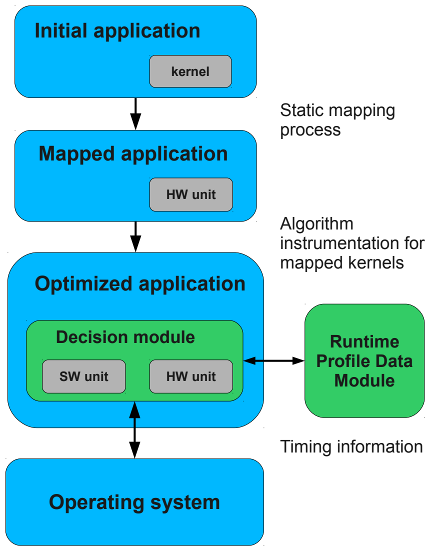

Figure 4.1: The overall structure of the AMAP algorithm.

The overall structure of the algorithm, named AMAP is depicted in Figure 4.1.

The algorithm is able to respond to changing conditions that can appear in a reconfigurable system. Also, it can immediately take into account the reconfiguration overhead as it measures the needed time for executing a hardware function. In this way, even if it is not aware of a configuration caching mechanism it will detect at runtime that one of the configurations is cached (so it can be configured much faster) and use it.

4.4.1 Selecting The Functions And The Relevant Input Data

The algorithm can be applied only to functions with a specific property. The control flow of the function should depend only on the value of the parameters, except the parameters that are pointers to memory. This means that it should not depend on values from the blocks of memory the pointers are pointing to. The reason is that the algorithm has to store the values of the parameters and compare them with the values in previous calls. If the control flow depends on the input data, storing all the input data it is not a feasible solution.

Assuming a function has the above property, the next step is to select the parameters that are included in the analysis of the algorithm. Using all parameters might make the data structure occupy too much memory and make management an overhead that would render invalid the gains obtained. Selecting too few arguments makes the algorithm inefficient. The compiler uses only the parameters that affect the control flow. These are any parameter involved in any of the *while*, *for*, *if* and *switch* instructions affect the control flow.

Listing 4.2: Rejected function example.

```
int functionA(int *a, int *b, int c, int d) {
  int i,j;
  for(i=0;i<c;i++) {
    a[i] = a[i] * d;
    if(*b==a[i])
      for(j=0;j<a[i];j++) {
        a[j]++;
      }
    }
  }
}
```

As an example, *functionA* in Listing 4.2 will not be used by this algorithm, as variable *a* is used in the *if* control structure at line 4. As it conforms with the algorithm requirements, *functionB* in Listing 4.3 will be accepted. We comment on each parameter of *functionB*:

- Parameter *a* is not used in any control structure and it is a memory block so it will not be included in the analysis.
- Parameter *b* is included as it is used in the *if* at line 4.
- Parameter *c* is included as it is used in the *for* at line 2, 5 and 11.
- Parameter *d* is included as it is used in the *if* at line 4 and 10.
- Parameter *e* is not included as it is not used in any control structure.

Listing 4.3: Parameter usage example.

```
int functionB(int *a, int *b, int c, int d, int e) {
  int i,j;
  for(i=0;i<c;i++) {
    a[i] = a[i] * e;
    if(*b==d)
      for(j=0;j<c;j++) {
        a[j]++;
      }
    }
  }
  if(d>10) {
    for(j=0;j<c;j++) {
      a[j]/=2;
    }

  }
}
```

4.4.2 Runtime Profile Data Module

The main purpose of the runtime profile data module is to provide information about past invocations of a function. The information is represented as metrics of previous calls. Our algorithm needs all the values of the parameters, so a data structure with fast insert and lookup operations is needed. Once a combination of parameters is added it can remain in memory for the complete execution of the program. Therefore the delete operation is not of importance for our purpose. Also, as all processing is done at runtime, we need a simple data structure that limits the overhead of storing and accessing the data. Taking this into account, we consider a red-black tree as the supporting data structure. The complexities of the search and insert operations are logarithmic, so a red-black tree is a structure that fits our needs.

Table 4.1: Parameter values for *satd* call.

Parameters		Steps in figure
height	width	
8	16	1,2
16	16	3,4
16	8	5,6
4	8	7,8
4	4	9,10

For a one-parameter function, the tree is straightforward as each node will contain the values of that parameter and the associated metrics. If the function has more parameters the red-black tree becomes a composition of red-black trees, where just the nodes corresponding to the last parameter have associated

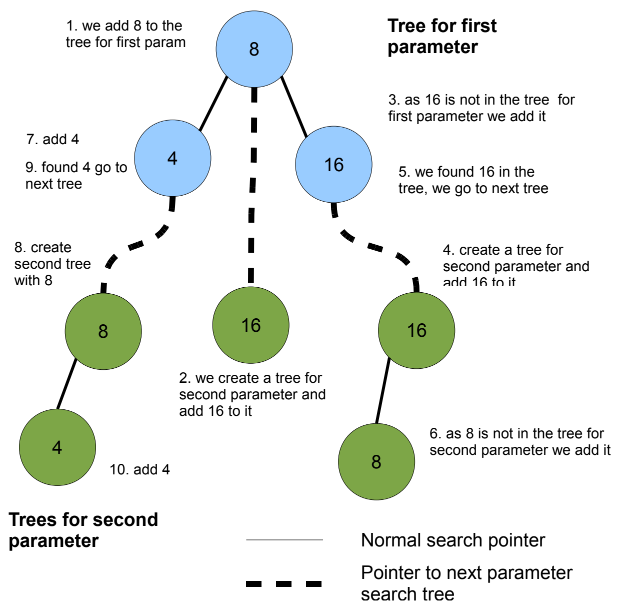
1. we add 8 to the tree for first param
8
Tree for first parameter
3. as 16 is not in the tree for first parameter we add it
7. add 4
9. found 4 go to next tree
4
16
5. we found 16 in the tree, we go to next tree
8. create second tree with 8
4. create a tree for second parameter and add 16 to it
8
16
16
2. we create a tree for second parameter and add 16 to it
6. as 8 is not in the tree for second parameter we add it
4
10. add 4
8
Trees for second parameter
Normal search pointer
Pointer to next parameter search tree

Figure 4.2: Search tree for 2 parameter *functionC*.

metrics. Let's assume we have a two-parameter function depicted in Listing 4.4. Given the parameters in Table 4.1 the resulting tree is depicted in Figure. 4.2. When the function is called, we do the steps shown in Listing 4.5 to identify the node containing information about past invocations.

Listing 4.4: Example function.

```
void functionC(int paramA, int paramB) {
  ...
}
```

Listing 4.5: Node identification in the tree.

```
t = tree root
p = first parameter
do {
  n = find value of p in t
  if(n is not found) {
    n = add p to t
  }
  t = next parameter tree from t
  p = next parameter
} while(p <= last parameter)
```

Each of the nodes corresponding to the second parameter, have a metrics structure associated. This structure is ommited from the figure.

The metrics we will use are:

- Hardware execution time and the number of executions.
- Software execution time and the number of executions.

These metrics were chosen because the focus of the algorithm is to improve the total execution time. Different metrics could be used in case another design goal would have been chosen. Examples are effective area occupied in case the goal would be to increase the overall utilization rate of the FPGA, power consumption if the goal is minimizing power.

The decision function uses the metrics to decide if, for these parameters, it is better to use the software or the hardware implementation.

One important issue with the profile data module is that it could take too much space of the processor cache, and that would degrade performance. The solution is to allocate the data in a contiguous block that is a multiple of the size of the cache line, and limit the increase to a certain value. When the limit is reached, the cache module will not be able to accommodate new nodes. One solution to this case is to find the closest match for the new node, in terms of parameter values, and update that node's metric. We will analyze in the results section the effect of this limitation over the algorithm efficiency.

4.4.3 Decision Module

The runtime part of the algorithm is presented in Listing 4.6. For each function call, that is managed by the algorithm, this code will replace the original call and a cache will be created. The *call_hw* function must include everything needed by the hardware call, such as hardware setup and memory transfers. The functions *set_sw_time* and *set_hw_time* are the functions that update the profile data module metrics after the call. The function *get_profile* is used to retrieve the metrics structure from the profile data module, while the function *decision* represents the logic in the decision module.

Listing 4.6: The AMAP decision module.

```
1 m = get_profile(selected parameters);
2 if ( decision(m) is sw ) {
3   t = time(call_sw_f());
4   set_sw_time(m, t);
5 } else {
6   t = time(call_hw_f());
7   set_hw_time(m, t);
8 }
```

The decision function, depicted in Listing 4.6, does a comparison of the times needed in hardware and in software and returns which implementation is more efficient for the current set of parameters. After the selected implementation is executed the cache is updated. At program start there will be no profile information available. The decision module can work only if both software and hardware execution times are available. If this is not the case, it will execute, at least once, the variant for which the profile information is missing. This is done in the decision function (line 2).

4.5 Results

In this section, we present the estimated results of applying the algorithm on the x264 video codec.

The x264 video codec is the state of the art in video compression algorithm and it requires a lot of computing power. One of the most time consuming kernels of the application (around 30% of the total execution time) is a function that computes the sum of the absolute differences.

As explained in Section 3.2.2, the hArtes application platform is suitable for a certain class of audio applications due the way the bus operates - half of the time is allocated to transfers from and to audio input/output buffers and memory. Instead of the hArtes platform we used a platform based on the same

Table 4.2: Parameters and execution time on Virtex ML510 board for *satd* call.

Parameter values	Execution times μ s			Speedup
	Overhead	t_{sw}	t_{hw}	
s1=32, s2=16, lx=16, ly=16	5	31	12.9	1.73
s1=32, s2=16, lx= 4, y= 4	4.6	2.55	1.9	0.39
s1=32, s2=16, lx= 8, y= 8	4.65	7.86	4.2	0.88

architectural concept, that was developed in parallel with the hArtes platform, but was based on off-the-shelf components, namely a Xilinx Virtex-4 ML410 development board. This board contains a Xilinx XC4VFX60 FPGA. The Molen programming paradigm is implemented using the APU unit of the PowerPC and an on chip memory, which is accessed through the DCR bus. The design contains also a Flash memory reader used as external memory and an internal 256 MB DDR2 memory. For the x264 application, we implemented and tested the *satd_wxh* kernel using the Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) hardware compiler [91]. The system GPP runs at 200 MHz. The custom hardware designs are clocked at 100 MHz.

We could not efficiently execute the entire application on the board, as the PowerPC was not able to access the internal FPGA memory through its data cache in the current operating system configuration and there is no DMA present either. As modifying the operating system configuration or adding a DMA was out of the scope of this work, we chose to estimate the memory transfer times, as if the system has a functional DMA. All the data was collected from actual runs of applications with real input. As the memory transfers can not be performed by a DMA, we measured the operating system overhead, the amount of memory transferred and the hardware and software execution time. Using the memory size and known DMA transfer speed, we could estimate the memory transfer time. Then adding all the times, we obtain an estimated total execution time.

Table 4.2 lists example values of parameters for the *satd_wxh* function, together with the times needed for the execution, for each parameter set.

By applying the algorithm, we can determine the best case for any parameter set. The application compiled to include our algorithm was used on different reference videos. We did this as the trace of the execution changes based on the data, so our decision and cache module will receive different parameters for different videos. The results are listed in Table 4.3. The *HW* column represents the total execution time of the kernel in case all the calls would be executed on the reconfigurable fabric, relative to the software execution time. By applying our algorithm only some of the calls will be selected to run on the reconfigurable hardware while others will be run in software. The results of this decision are presented in the column *AMAP*. The last column represents the overhead introduced by the algorithm, respectively the percent of the total execution time spent in the decision and cache module.

Always using the hardware can degrade performance when compared to software execution. This can happen if the overhead, represented by the system

Table 4.3: Kernel execution times in different scenarios. Reference is executing all calls in software.

Video	All in HW	AMAP	AMAP execution
akiyo	82.01%	76.08%	4.05%
carphone	106.45%	87.22%	6.50%
claire	134.97%	91.87%	7.25%
coastguard	91.25%	86.71%	4.98%
container	90.18%	79.96%	4.86%
foreman	98.69%	85.36%	5.68%
hall	113.64%	88.60%	7.14%
miss-america	121.65%	91.47%	8.10%
mobile	88.95%	82.25%	4.71%
news	88.85%	80.29%	4.77%
salesman	93.54%	80.75%	4.74%
silent	93.50%	84.37%	5.16%
suzie	104.42%	86.67%	6.37%
Average	100.62%	84.74%	5.72%

call and starting/stopping the hardware unit, is comparable (or greater) to the total time spent executing the kernel. This happens for example for *claire* video for which the execution time when using only the hardware is 134.97% of the software only execution time. When applying our algorithm, not all calls are executed in hardware. This results in improvements from 43% for *claire* video to 5% for *mobile* video with an average of 15%.

As mentioned in Section 4.4.2 one notable aspect is the memory footprint of the data stored about parameters and execution times. With less memory, the algorithm will not be able to make an accurate prediction. This will affect the total execution time, as sometimes a function which takes less time in software will be executed in hardware. To test how the algorithm behaves in such cases, we tested the behavior of algorithm for different number of nodes available. As the function tested use 4 parameters, it means the algorithm needs at least 4 nodes to store profile information. We started our exploration from 5 nodes, as this is the smallest number when our algorithm can perform a runtime estimation. With 5 nodes, the tree constructed will contain 2 combinations of parameters. Then we increase the number of available nodes until the behavior does not change.

The results are presented in Figure 4.3. For our case, for a size of 10 stored nodes there is a cut off point - after which point the improvement is marginal. The big differences in execution time per frame between the data sets can be explained by the fact that the algorithm results depend on the amount of 'motion' in the video.

As expected, the algorithm behaves badly in case there are no nodes available, but improves quite fast as the number of nodes increases.

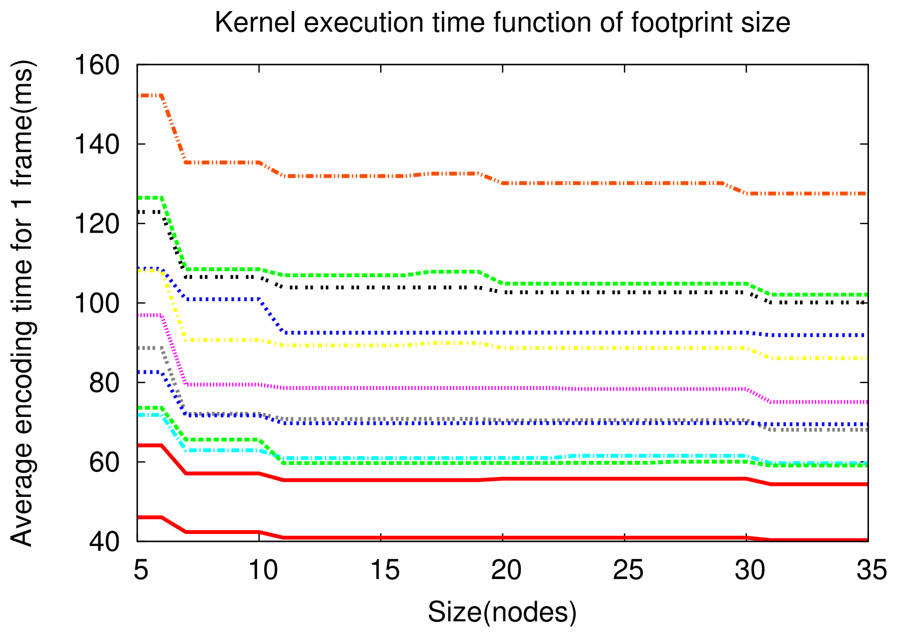

Figure 4.3: Algorithm performance for different cache sizes (each line represents a different video).

To investigate the behavior of our algorithm in extreme circumstances, we considered increasingly high overheads of executing a kernel in hardware (multiplying by 2, 3, etc. the measured overhead). The results can be seen in Figure 4.4. We compare with the execution 'completely in software'. The thick line represents the average for the percentages of the 'completely in software execution' for all the videos used in the test (same as in previous tests). As expected, as the overhead increases, the algorithm will use more and more the software version. After some point, the overhead makes running the hardware totally ineffective (in our graph, around 5x overhead), and the algorithm will use the software. The decrease in performance (after 5x, everything that is the above 100% line is a reduction in performance) is because the algorithm has to profile at least some hardware executions in order to determine that the overhead is significant and it would be inefficient to execute the function in hardware. Still, it will gracefully adapt to the new conditions and limit the performance decrease to less than 3% compared to pure software execution, even for an overhead increase of 8x.

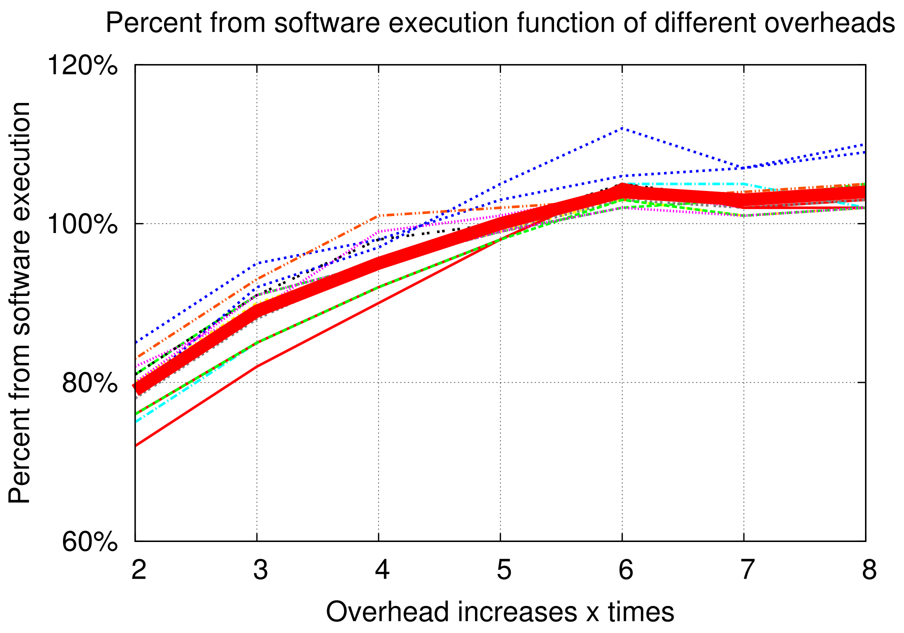

Figure 4.4: Algorithm performance for different overheads. The reference of 100% represents software execution. Each line represents a different video, the thick line represents average.

4.6 Conclusions

In this chapter, we proposed a runtime algorithm which, using profiling and compile-time information, selects between executing the software or the hardware implementation of a function at runtime. It does this based on the parameter values and the profile data gathered at runtime. Our experiments show that it can provide a significant improvement in a dynamic system, where, at compile-time, it is almost impossible to foresee all the parameters of the system. The algorithms can be seen as an extension to traditional compile-time hardware software mapping. We also studied the effect of the size of the profile data stored in the tree and examined the effect of different overheads on the performance.

Note.

The content of this chapter is based on the following paper:

V.M. Sima, K.L.M. Bertels, **Run-time decision of hardware or software execution on a heterogeneous reconfigurable platform**, Proceedings of International conference on 16th IEEE Reconfigurable Architectures Workshop, pp. 6, Rome, Italy, May 2009.

5 Runtime Memory Allocation

In this chapter, we propose a solution for problem C given in Section 3.6 when the overhead introduced by memory hierarchies can cancel the advantage given by running the computation on different processing elements. This overhead appears because the data needs to be transferred between the different memories. One solution to reduce the memory transfers is to allocate the memory in the local memory of the processing element that uses that data the most. As discussed in Chapter 3, the analysis needed to take a decision on how to perform the memory allocation, can not be performed at compile-time. In order to solve this limitation, we propose a runtime algorithm that solves this problem. Using code instrumentation, we determine what memory areas are used by functions executed on different processing elements. If not satisfied with the mapping provided by the toolchain, the developer has the option to manually specify which functions have to run on which processing element. An algorithm decides the best memory allocation, taking into account the gain obtained by running a computation on a processing element and the available scratch pad memories of the heterogeneous platform. We obtain application performance improvements of 14% on a video encoder application compared software only execution. For synthetic applications, the algorithm is within 5% of the optimum.

This chapter is organized as follows: we start by discussing the related research and giving a detailed motivational example and the problem definition. Our proposed solutions to the problems exposed is presented in Section 5.3.3. The results of the algorithms are shown in Section 5.4. In Section 5.5, we present conclusions and outline new research directions.

5.1 Background and Related Research

An efficient management of Scratch Pad Memory (SPM) can substantially improve the overall performance of embedded systems, when compared to caches [2] [9].

Again we distinguish between runtime and compile time approaches. Various approaches focus on compile time memory allocation [48] [13] [92] [6] [40] [14] [79]. In contrast to our approach, these algorithms rely on detailed compile time information.

Other approaches perform the allocation at runtime using different decision criteria. In [62] a load time optimization is performed, based on the SPM available when the application is run. [49] proposes an allocator that has as objective reducing the power consumption. [22] supports allocation of global, stack and help memory blocks and decides the best allocation taking into account this memory hierarchy. All these approaches consider a homogeneous core architecture, which is not the case of our algorithm.

5.2 Problem Definition

Let us consider the architecture described in Chapter 3. This architecture contains an FPGA and a DSP, each with its own local memory. In Figure 5.1, we give a synthetic application example. This application contains three kernels that use four memory blocks. There are two problems:

- How do we identify which kernel uses which memory block, and how many times? Static analysis can provide this information for some cases, but it will not be able to infer the complete information as the number of uses can be heavily depending on input data.
- How do we perform the mapping and allocate to the corresponding memory?

We will discuss each of these problems independently in the following sections.

Allocation Tracking

Given the application source code, and the functions annotated to be executed in hardware, a list will be maintained with all the memory blocks that are used in those functions. The function can be annotated either by the developer or automatically by the toolchain. One function can use for one call multiple memory blocks simultaneously. For each combination of memory blocks used in a kernel, we have an associated gain which represents how much time would be saved if that kernel would be accelerated in hardware. This gain can be obtained either at compile-time, by profiling the software and the hardware implementation using a specific data set, or can be computed at runtime by profiling the actual execution. We assume in the rest of the paper that we have this information provided at compile-time. The gain is computed by multiplying the execution count with the time saved by using the accelerated implementation.

Memory Mapping

The second part of the problem is to find which set of memory allocations has to be placed in the Scratch Pad Memory (SPM) instead of DDR to obtain the best performance. We assume that all the kernels execute sequentially. If parallel execution would be considered, the idea of the algorithm holds, but some of the equations have to be modified. Graphically, the problem is depicted in Figure 5.1 where the circles represent one kernel call, the rectangles represent the memory blocks and the squares represent the memory. The gain associated with a kernel invocation can be obtained just if all the memory blocks are allocated to the SPM. In our example, just $K1$ and $K2$ will run in hardware, while $K3$ will run on the GPP, because block D is allocated in DDR. With this mapping, the total gain would be of 400ms.

5.3 Memory allocation infrastructure

To solve the problems described, we present an infrastructure, composed of three modules: the allocation module, the execution module and the mapping module. A graphical representation of the infrastructure is presented in Figure 5.2. Each module has a runtime part that has to be embedded in the application. This is performed by special optimization passes that extend the compiler infrastructure.

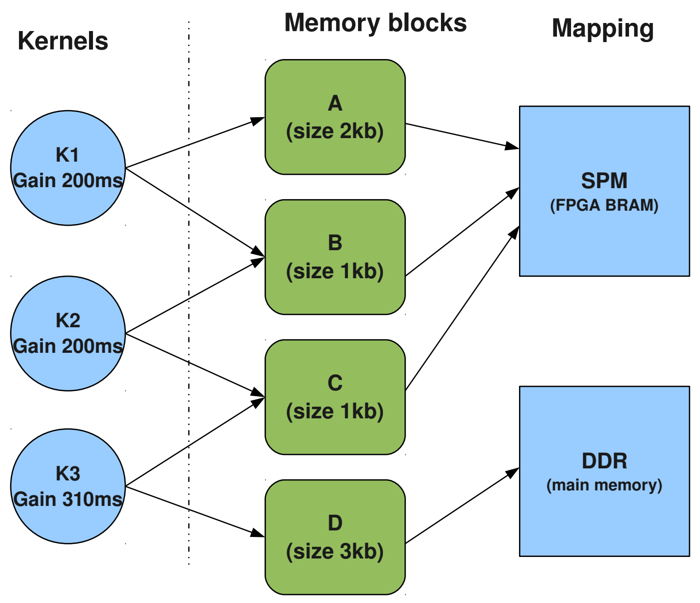

Figure 5.1: Motivational example for AMMA algorithm.

The allocation module tracks at runtime all the blocks allocated by one application by managing a list with the start and end address of the currently allocated blocks (stack, global and heap). It also decides using an algorithm where each memory block should be allocated, in scratch pad or in main memory.

The execution module, determines at runtime, for each of the functions that can be executed in hardware, if the data is allocated in SPM, and in that case uses the accelerated implementation. It also updates the gains associated with each combination of memory allocations, based on the known speedup of the current function invocation. We call the lists of memory allocations together with the associated gains, the memory allocation lists (MEMAL)s. The MEMALs are continously updated while an application runs. The lists are saved and restored across execution to provide the allocation algorithms with as much as possible information.

The algorithm that computes the best allocation, is embedded in the application, and it is executed just at specific points of the program execution. The algorithm must be called in two cases. The first case is when, at application start, the size of scratch pad memory available for the application is different

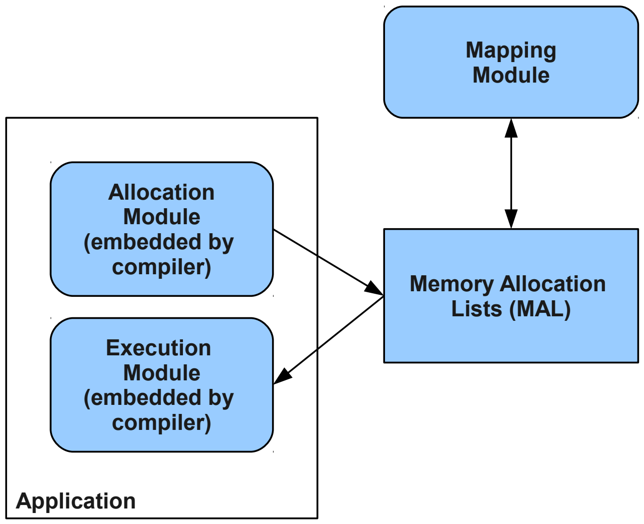
Mapping Module
Allocation Module (embedded by compiler)
Memory Allocation Lists (MAL)
Execution Module (embedded by compiler)
Application

Figure 5.2: AMMA infrastructure organization.

from the previous executions. The second case is when an allocation is no longer optimal because of a change in the input data. We did not implement this case.

It is important to mention that the influence the algorithm has on the allocations depends on the moment in time these allocations are performed by the application. The mapping algorithm influences only allocations that are made after an analysis of MEMALs. As an example, if the mapping algorithm determines that global variables should be allocated to scratch pad memory, this decision will be applied only at next application start (when the allocation is made).

Using this approach, the only application modification needed, is the annotation of the functions that have hardware implementations. Our approach is semi automatic and does not involve a manual analysis to determine exactly what allocations are needed by which kernels (and thus solves Problem A [1] from Section 3.6). We assume that the functions with hardware implemen-

[1] the speedup of a function depends on the values of its parameters.

tations are provided by the designer. The size of the available memory will be determined just at runtime, giving more flexibility to the system (and thus solves Problem B [2] from Section 3.6). If multiple applications are present, the allocation can be done in such a way that the overall system performance is improved (solves Problem C [3] from Section 3.6).

5.3.1 Allocation Tracking Module

The allocation module has two tasks:

- To manage the allocated memory blocks.
- For new allocations, place them either in SPM or DDR. This decision is take previously based on what it is estimated to be best for overall performance of the application.

Depending on the type of the memory object, different allocation mechanisms are used. These are presented in the next sections.

Global Variables

For the GPP architecture analyzed, the global variable addresses are embedded in the binary. Also, on some architectures accessing a global variable is done through a register which stores the base address of the data segment. This mechanism prevents the changing the address of a global variable at runtime to a total different memory address. To solve this problem, all the global arrays used in functions with hardware implementations are transformed to pointers and are initialized in a special initialization function *cm_global_init*. This mechanism is a well known mechanism used by the compiler when generating position-independent code. In function *cm_global_init*, there are two steps: the ranges of the arrays are added to the MEMAL for further checking and based on the allocation decided by the algorithm, each of the arrays is allocated either in main memory or in the scratch pad memory.

[2] memory needs to be transferred to the SPM in order to execute functions accelerated.

[3] based on what memory is allocated directly to the SPM functions will have a different speedup as some we will not have the transfer overhead.

```
function:
sp = cm_stack_allocate(id,size);
cm_stack_add(id,sp,sp+size);
...
sp = cm_stack_deallocate(id);
return;
```

Figure 5.3: Code added to functions for stack instrumentation (architecture dependent).

Stack Variables

As the stack management is architecture dependent, so is the part of the allocation module that deals with stack variables.

Tracking memories allocations introduces overhead. Tracking the stack for all functions would introduce unnecessary overhead as not all the stack variables can be used in an accelerated function. Instead, we perform an optimization by instrumenting just those functions which are found on a path from the root of the call graph (*main* function) to one of the accelerated functions. If function pointers are used, we will always consider that such a pointer can point to an accelerated function and instrument the function using the function pointers. Another optimization is to detect if any of the local variables in a function are used as parameters for other calls. If this is not the case, it means the local variables will never be used by accelerated functions, so the instrumentation for that function it is not necessary.

For each instrumented function, a special header is added to the assembly code. First *cm_stack_allocate* returns the address for the stack of the function, based on the decision that was taken by the algorithm. Then *cm_stack_add* is called which, based on an identifier, will add the start and end address to a list in the MEMAL. Before returning, *cm_stack_deallocate* will be called to restore the registers. A pseudo-code of the wrapper is given in Figure 5.3, where *sp* is the stack pointer for local variables.

Heap Memory Allocations

To track the dynamic memory allocations, a wrapper around standard allocation functions is provided. The wrapper has two functions: add the allocations to the MEMAL and allocate the memory according to the decision of the allocation algorithm. Each allocation has to be uniquely identified in order to be added to MEMAL. The easiest form of identification is to use the address from which *malloc* was called. This is a simple and fast solution that works in the majority of cases. The solution that would work in all cases is to use as identifier not only the address from which *malloc* was called, but also the addresses of all calling functions from *main* to the current *malloc* call. This would work with applications that wrap *malloc* themselves, but the overhead of saving and searching for the identifiers increases significantly.

5.3.2 Execution Module

The execution module is used only for the accelerated functions. Its role is to update internal data structures that track what memory areas are used by accelerated functions. Then, if all the data are placed in the scratch pad memory, it will invoke the accelerated implementation. Otherwise, the software implementation will be called.

5.3.3 Mapping Algorithms

We will present three mapping algorithms. The first algorithm, Adaptive Memory Mapping Algorithm (AMMA) has the advantage that it has the shortest execution time from all the algorithms. The extension of this algorithm, Adaptive Memory Mapping Algorithm Extended (AMMAe),is slower but provides better results for some cases than AMMA. The final algorithm presented is an Integer Linear Programming (ILP) based algorithm. It is used to assess the performance of the other two algorithms.

AMMA algorithm

The data available to the algorithm can be formally represented by:

- The set of memory blocks allocated, $A = \{a_j, j = \overline{1..b}\}$, where b is the number of blocks tracked, and a_j is one allocated block.
- The combinations in which the allocations are used by each kernel. There can be multiple combinations used by each kernel, and one allocation can be in multiple combinations. Let the set of all combinations be $S = \{C_i, i = \overline{1..n}/C_i = (\{a_j\}, k_{i_{gain}})\}$. We denote by C_i a combination of memory allocations used in at least one kernel invocation and by a_j a specific memory allocation. The time gained by allocating all the memory in set C_i to scratch pad memory and running the associated kernel accelerated is denoted by $k_{i_{gain}}$.

- The size of the available scratch pad memories, $size_i$.

The idea of the algorithm is to order the memory allocations based on the memory score then allocate them to scratch pad memory, until the memory is full. The score is computed as the sum of the gains of the kernels that use that specific allocation. This is represented for memory location a_j as $a_{j_{score}}$ and we compute it using Equation 5.1. This is a very low overhead algorithm, as it needs just one sort operation on the memory objects and one pass to fill the memory. The pseudo-code is given in Listing 5.1.

$$a_{j_{score}} = \sum_{a_j \in C_i, m_i \in Ci} k_{i_{gain}} \tag{5.1}$$

The main drawback of AMMA is that it does not take into account the fact that in order to obtain the gain, a complete group of memory allocations must be in SPM. For example, if we consider Figure 5.1, *K2* can not execute in HW unless both memory block *A* and *B* are allocated to SPM.

Listing 5.1: AMMA algorithm.

```
struct allocation {
  int size;
  int allocated;
}

struct combination {
  int *alloc;
  int gain;
}

inout: struct allocation a[b];
in: struct combination m[n];

compute_scores(a);
sort(a,descending);

alloc = 0;
for i = 0 to b {
  if(alloc + a[i].size < memory_size) {
    a[i].allocated = true;
    alloc += a[i].size;
  }
}
```

```
allocate_to_DDR(A,B,C,D)    allocate_to_SPM(A,B,C)
                            allocate_to_DDR(D)
........................    ........................
call K1(A,B)                call accelerated K1(A,B)
call K2(B,C)                call accelerated K2(B,C)
call K3(C,D)                call K3(C,D)

                            Gain : 400 ms
```

Figure 5.4: Application execution trace before and after AMMA algorithm.

Even if multiple blocks are allocated in the scratch pad memory, it might be the case that no kernel can be accelerated. For example, in Figure 5.1, if blocks A and C are allocated to the scratch pad memory, still, no kernel can be accelerated.

The complexity of the algorithm is $\mathcal{O}(b * (n + log(b) + 1))$. Computing the scores for each allocation depends linearly on the number of blocks, b, and the number of combinations of allocations, n. Hence, the computation of scores takes $b * n$. Sorting the list of b scores can be performed in $b * log(b)$. Iterating once more over the list of allocations give the third term b.

Example of Applying AMMA

Assuming we have the kernels and memory allocations in Figure 5.1 and an available memory of 5 kB, the steps of AMMA are shown in Table 5.1.

Table 5.1: AMMA algorithm example.

Iteration	Memory allocations scores				Allocated to SPM	Free SPM
	A	B	C	D		
1	200	400	510	310		5
2	200	400	-	310	C	4
3	200	-	-	310	C, B	3
4	-	-	-	310	C, B, A	1
Kernels executed accelerated: K1,K2 - gain 400ms						

The memory allocation's scores are computed only using Equation 5.1 at the beginning of the execution of the application.

The gain for memory block A is the one generated by kernel $K1$, while the gain for memory block D is the one generated by kernel $K3$. For memory blocks B and C, 2 kernels contribute to their gains. In each step, the memory block with the highest score is chosen to be allocated to the SPM. The functions that would be executed with and without AMMA applied are shown in Figure 5.4. We can see memory blocks A, B, C are allocated to SPM and $K1$ and $K2$ are accelerated.

AMMA Extension

To alleviate the limitation described in the previous section, we present an extended version of the same algorithm, which we call AMMAe, which is more complex but provides better solutions than AMMA.

We now discuss an extension to the proposed AMMA algorithm where 2 new elements are introduced. The supporting equations are:

$$s_i = \sum_{a_j \in C_i, a_j \text{ is not allocated}} a_{j_{size}} \tag{5.2}$$

$$a_{j,i_{score}} = \begin{cases} 0 & \text{, if } s_i < f \text{ or } a_j \notin C_i \\ k_{i_{gain}} \cdot \frac{a_{j_{size}}}{s_i} & \text{, otherwise} \end{cases} \tag{5.3}$$

$$a_{j_{score}} = \sum_{i}^{n} a_{j,i_{score}} \tag{5.4}$$

where, s_i represents, at the current iteration, the memory that kernel i needs in order to be possible to run in hardware.

The first element is that, the gain of a kernel is only added to the score of a memory allocation when it can fit the available remaining SPM memory. This is represented by the if statement in Equation 5.3. In the example in Figure 5.1, and considering an SPM of size 3k, the gain of kernel $K3$ will not be added to the score of either memory block B or C, because the memory required for $K3$ is, in total 4k, which is more than the available memory.

The second change to the AMMA algorithm is represented by the term $k_{i_{gain}} \cdot \frac{a_{j_{size}}}{s_i}$ in Equation 5.3. This equation multiplies the kernel gain by the proportion the current block represents of the total memory needed for kernel k_i. This way we take into account how much the current memory block consumes of the total required memory the kernel needs. This is represented by the $\frac{a_{j_{size}}}{s_i}$. The larger this factor is, the more the kernel gain contributes to the final score.

Assuming f is the free memory at the current step, the new gain formula is given by Equation 5.4. The outline of the algorithm is given in Listing 5.2.

Listing 5.2: AMMAe algorithm.

```
inout: a[b] array of memory allocations
in: m[n] array of combinations of allocations

alloc = 0
do
  changed = false
  compute_scores_extended(a,b)
  sort(a,descending)
  for i = 0 to n
    if(alloc + a[i].size < memory size)
      mark a[i] to be placed in SPM
      alloc += a[i].size
      remove a[i] from a
      changed = true
      break
while (not changed)
```

The complexity of the algorithm is $\mathcal{O}(b*(n*b+b*log(b)+n))$. The do/while loop can execute at most b times. The first term represents the computations of the scores and it is done in $n*b$, as the sum in Equation 5.2 can be performed on at most b elements. The sort of the scores is done in $b*log(b)$. The third term captures the selection of the next memory block that has to go to the SPM. As it performs on iteration over the list of blocks this add n to the expression.

Example of AMMAe

Assuming we have the kernels and memory allocations in Figure 5.1 and an available memory of 5 kB, the steps of AMMAe are shown in Table 5.2.

Table 5.2: AMMAe algorithm example.

Iteration	Memory allocations scores gains				Allocated to SPM	Free SPM
	A	B	C	D		
1	133	166	177	232		5
2	133	166	410	-	D	2
3	0	200	-	-	D, C	1
4	0	-	-	-	D, C, B	1
Kernels hardware accelerated: K2,K3 - gain 510ms						

AMMAe uses Equation 5.4 to compute the gain for each memory allocation. We give examples for s_{K1} and $a_{B_{score}}$ as the rest are computed similarly. The

```
allocate_to_DDR(A,B,C,D)    allocate_to_SPM(B,C,D)
                            allocate_to_DDR(A)
........................    ........................
call K1(A,B)                call K1(A,B)
call K2(B,C)                call accelerated K2(B,C)
call K3(C,D)                call accelerated K3(C,D)

                            Gain : 510ms
```

Figure 5.5: Application execution trace before and after AMMAe algorithm.

memory needed to run $K2$ accelerated is:

$$s_{K1} = a_{A_{size}} + a_{B_{size}} = 3$$

We compute now the score associated with block B. B is used by two kernels, $K1$ ($K1_{gain} = 200$) and $K2$ ($K2_{gain} = 200$). But kernel $K1$ needs both blocks A ($a_{A_{size}} = 2k$) and B ($a_{A_{size}} = 1k$) to run, so, we add to the score of B just a part of gain of kernel $K1$ proportional to block size of B from the total size needed ($s_{K1} = 3$). The same applies for $K2$. The entire equation is:

$$a_{B_{score}} = k_{K1_{gain}} \cdot \frac{a_{B_{size}}}{s_{K1}} + k_{K2_{gain}} \cdot \frac{a_{B_{size}}}{s_{K2}} = 166$$

After each allocation iteration, all the scores have to be recomputed as the total sizes needed by one kernel will change (we consider just the additional size needed, without taking into account the already allocated blocks). After the first iteration, the total size needed by kernel $K3$ will be just the size of C, as block D is already allocated. Hence the score for C will change. The algorithm is applied until no further allocation is possible.

The instructions that would be executed without and with AMMAe applied are shown in Figure 5.5. We can see B, C, D are allocated to SPM and $K2$ and $K3$ are executed accelerated, resulting in a 25% increase in gain over AMMA.

ILP Formulation

The purpose of the ILP forumlation is to give the optimum solution and provide a comparison basis for the other two algorithms.

For each of the memory allocation we associate a 0 - 1 variable (x_j) which will be 1 in case that the memory allocation will be made in SPM. For each combination of memory allocations, that is, for each kernel invocation, we associate a 0 - 1 variable (y_i) which will be 1 in case all the memory allocations used by that invocation are allocated in SPM. Let n be the number of memory allocations combinations. Using this notation our objective function is:

$$max(\sum_{i=1}^{n} y_i \cdot k_{i_{gain}}) \tag{5.5}$$

We need a constraint linking the combinations of memory allocations with each allocation. Let c_i be the number of allocations in C_i. The idea is that y_i is 1 just if all the corresponding x_j are also 1. We can express this for each i, as:

$$y_i \leq \frac{\sum_{a_j \in C_i} x_j}{c_i} \tag{5.6}$$

Let m be the total number of memory allocations. Ensuring that all the allocation fit in SPM will be imposed by the following constraint (where MEM is the total size of SPM):

$$\sum_{j}^{m} x_j \cdot a_{j_{size}} < MEM \tag{5.7}$$

5.4 Empirical Validation

As explained also in Section 4.5, although the analysis was done for the hArtes platform, we did the empirical validation of our approach on a different but similar platform. We did this because hArtes platform is better suited for audio applications, due to the way the bus operates - half of the time is allocated to transfers from and to audio input/output buffers and memory. For our platform, the same architectural concept was implemented, but instead of a custom board design, an off-the-shelf board was used, namely a Xilinx Virtex-4 ML410 which is based on the Xilinx XC4VFX60 FPGA. The ML410 platform is similar to the hArtes platform, as they are both based on the same family of FPGA, the main difference is that the GPP and memory controller are located on the FPGA. This makes our platform better suited for this application. The memories used in our design are: a Flash memory used as external memory, an internal 256 MB DDR2 memory as main memory and a 128 kB SPM used as scratch pad memory. The kernels were implemented using the Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) tool [91]. The

General Purpose Processor (GPP) runs at 200 MHz and the hardware designs are clocked at 100 MHz. The compiler used to instrument the application was a modified GCC 4.3 Power-PC compiler.

As this is a runtime algorithm, overheads are important in measuring its performance. We present separately the overheads, measured as a percentage of the total application execution time, involved in constructing the MEMAL in Table 5.3. The videos used are the ones available at [90]. From Table 5.3 we can conclude that compared to the speedups that can be obtained by applying AMMA, the overhead incurred by it is negligible.

Table 5.3: Execution time overhead of constructing memory allocation table for stack and dynamically allocated variables.

Instrumen -tation	1	2	3	4	5	6
Dynamic	0.38%	0.37%	<0.1%	<0.1%	<0.1%	0.24%
Stack	0.36%	0.58%	0.33%	1.07%	<0.1%	0.52%

Another, similar, but less important issue is the execution time of the AMMA and AMMAe algorithms. Even if the mapping algorithm will execute rarely, we present in Table 5.4 the times spent by each mapping algorithm, obtained when testing with the synthetic applications. For the ILP algorithm, we ran just one subset of problems, because of the long execution times. The three cases are increasingly complex synthetic applications. Their generation is explained in Section 5.4.2. It is obvious that the ILP solution is not a feasible candidate for runtime execution, as it can take more than 30s.

5.4.1 H.264 video Encoder

We used the instrumentation and algorithm presented to compile and run the H.264 video codec. The execution pattern in the encoder is dependent on the

Table 5.4: Average mapping module execution times tested on the hardware platform (outside of the context of an application).

Algorithm	Average execution times (ms)		
	Case 1	Case 2	Case 3
ILP	9980	30990	37780
AMMA	21	97	239
AMMAe	52	234	522

input data. Our infrastructure will adapt to that by calling the mapping algorithm during program execution. More than 35% of the execution time is spent in two kernels: *satd_wxh* and *sad*. The kernel hardware implementations offer an average speedup of 2 and respectively 1.8. The total memory used by both functions is 603 kB. The smallest block is 256 bytes, while the largest is 49 kB. The total available scratch pad memory is 128 kB. With this setup, by applying the AMMA algorithm we obtained an application performance improvement of 14% compared to the GPP only execution. For this application and size of SPM both AMMA and AMMAe gave the optimal solution. Assuming an infinite amount of SPM the speedup that could be obtained is of 18% compared to software only execution. The application contained 17 memory allocations that were used by the two kernels, and 29 memory allocation combinations.

5.4.2 Synthetic Applications

Besides the evaluation on the H.264 application, we used benchmarks on synthetic applications, to evaluate how far is our algorithm from the optimal solution. We considered in the synthetic applications that each memory allocation has a size between 128 bytes and 256 kB. We use as reference the DWARV [91] hardware compiler which automatically generates the hardware kernels. The speedup obtained for various tests is up to 10x. We chose between 5 and 10 memory blocks and a large number of kernel (between 10 and 30) to test the algorithms in more complex situations than the one found in the motivational example. The results can be seen in Figure 5.6, Figure 5.7 and Figure 5.8.

The number of synthetic applications generated was 300. We compare against the ILP solution, which is optimal. The speedup obtained for each application depends on the amount of memory available and is in our tests between 2 and 6.

From the graph, we can see that in case there is little memory available, both algorithms perform as well as ILP. As the available memory increases, we can see that AMMAe is better than AMMA, within 4% of the ILP solution. Both algorithms converge to ILP when the available SPM-s are large enough to fit all the memory objects and all the kernels will be executed in hardware. These trends were also seen when varying the number of kernels and memory blocks (graphs omitted here for brevity), with AMMA being always within 14% of the ILP, and AMMAe being within 5% of the ILP solution.

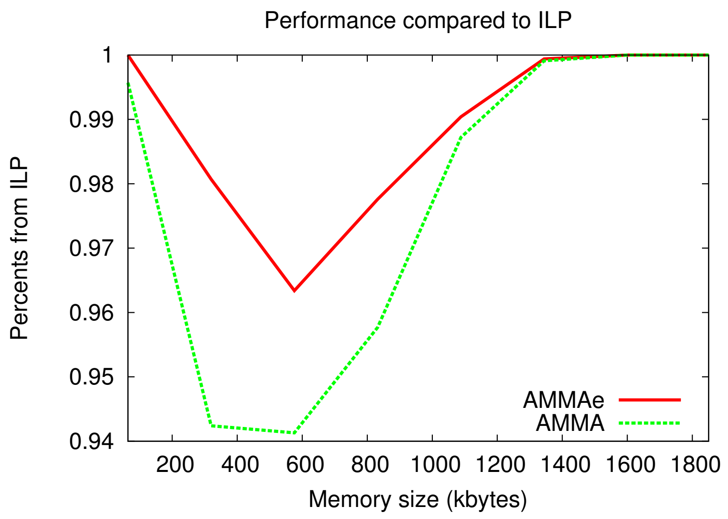

Figure 5.6: Algorithms performance for different memory sizes, number of kernels between 5 and 10.

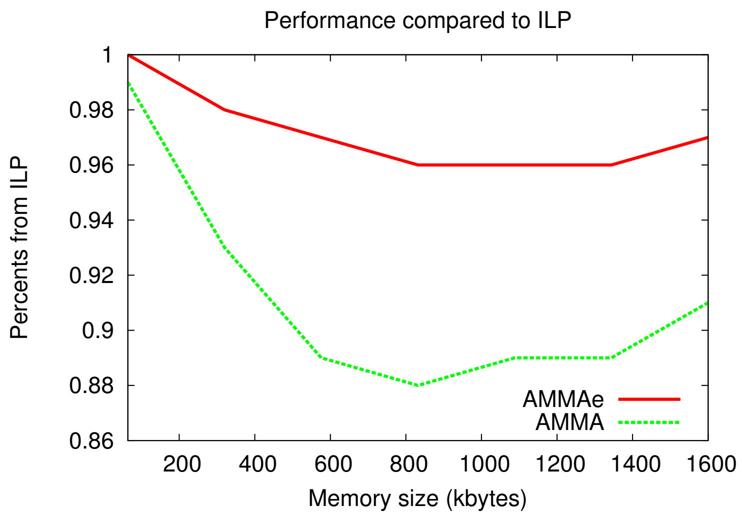

Figure 5.7: Algorithms performance for different memory sizes, number of kernels between 10 and 30.

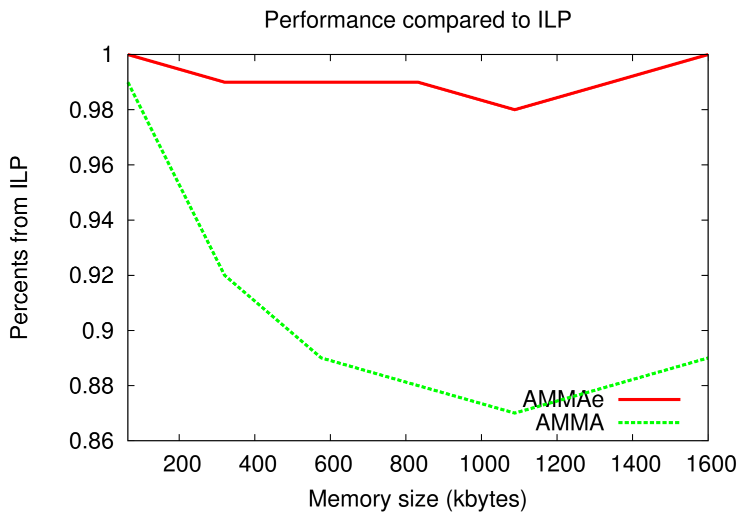

Figure 5.8: Algorithms performance for different memory sizes, number of kernels between 20 and 60.

5.5 Summary

In this chapter, we presented the compiler infrastructure, including two mapping algorithms AMMA and AMMAe, that decide, at runtime, which is the best allocation for memory. The allocation takes into account the kernels that use the memory as well as the gain that could be obtained by running those kernels on dedicated hardware components. We showed how such an infrastructure can simplify the development of applications while allowing a high flexibility by adapting to changing conditions. Our algorithm stays within 5% of the optimal solution given by an ILP formulation.

Note.

The content of this chapter is based on the following paper:

V.M. Sima, K.L.M. Bertels, **Run-time memory allocation in a heterogeneous reconfigurable platform**,IEEE International Conference on ReConFigurable Computing and FPGA, Cancun, Mexico, December 2009.

6 Scenario Based Runtime Mapping

In this chapter, we present the compiler extensions, based on OpenMP libraries, needed for supporting parallel execution on the reconfigurable Molen platform. More specifically, we start by proposing a compile-time, Integer Linear Programming (ILP) algorithm that maps parallel applications to the target platform. We assume that, for a part of the application, the designer can select from a set of hardware implementations with different area and speedup. Based on profile information, the algorithm aims to minimize the total execution time of the running threads, taking into account the limited reconfigurable area. We show that for a real application and the real hardware implementation of the kernels our ILP model is up to 1.9x better than other existing mapping approaches. We also investigate the impact of several parameters such as the size of the reconfigurable area and the number of threads on our proposed ILP model. For these factors, we determine in which range the ILP model is best applicable.

In the second part of the chapter, we extend the initial idea by developing a runtime algorithm. This algorithm, based on profile info decides at runtime which scenario to use. As this is a runtime algorithm, we consider the area will change over time.

6.1 Background and Related Research

When dealing with reconfigurable hardware and parallelism, a lot of research focused on reducing the reconfiguration overhead [37] [66] [67], the reconfiguration area management and scheduling of operations [78] [27] [63]. All these approaches address tasks organized in simplified task graphs or multiple independent applications. They do not address the efficient use of available reconfigurable area in case of multiple threads of one or more applications, when taking into account the execution time.

For single parallel applications the problem of splitting data across threads is discussed in [85]. For the same class of problems the bandwidth allocation is discussed in [11], where the application is considered to be composed of a task chain. Neither of these approaches consider multiple parallel threads and/or applications in the system, except the optimized application.

Higher level methodologies were proposed [32] [58], that identify at compile-time the full set of possible scenarios that arise during the application execution. These methodologies work in case the system is completely known at design time, compared to our proposed approach, that can adapt to a certain extent to changes in the system and applications.

6.2 Static algorithm

In this section, we present the static algorithm that selects from a set of implementations a combination that minimizes execution time. The constraint is that enough area has to be available for the implementations selected. We consider that we have all the information about speedup and area requirement as this operation is performed at compile-time.

6.2.1 Problem Definition

Our starting point for the algorithm in this chapter is the beamforming and wavefield synthesis applications presented in Section 3.5.2. In order to be applicable, our algorithm needs that the application is composed with several threads and for each thread different implementations are available. A thread can be implemented in software, in hardware or a combination of software and hardware. As the hardware parts can be implemented differently based on how much area is available, there will be multiple hardware implementations available. With each implementation we have associated a software execution time, a hardware execution time and an area requirement. For the real application evaluated this is given in Table 6.1.

Problem statement: We call a scenario an implementation for kernels executed inside an application thread. Each implementation can be more appropriate in a different context, for example when there is no reconfigurable area available, or when the fastest speed is needed. In our context, kernels have a one-to-one relationship with functions, for which the execution time

Thread	Scenario	SW Time (ms)	HW Time (ms)	Area (%)
1	$s_{1,1}$	1635000	0	0
	$s_{1,2}$	87310	179170	20
	$s_{1,3}$	87315	89585	40
	$s_{1,4}$	87320	84228	61
	$s_{1,5}$	87325	44792	81
	$s_{1,6}$	87330	43179	101
	$s_{1,7}$	87335	42114	122
	$s_{1,8}$	87340	41339	142
2	$s_{2,1}$	1570000	0	0
	$s_{2,2}$	5	71075	14
	$s_{2,3}$	10	35537	28
	$s_{2,4}$	15	26653	42
	$s_{2,5}$	20	17768	56
	$s_{2,6}$	25	13326	84

Table 6.1: Implementation scenarios for beamforming application. The various scenarios are obtained by increasing the level of parallelism used in the hardware for each task in each thread. The area is considered for platform Virtex II Pro.

is a significant percent of the total execution time of the thread. The scenario - s_j - will be characterized by software execution time t_{sw}, hardware execution time t_{hw} and area occupied a based on a specific kernel implementation. The scenarios are generated based on the available hardware implementations and profiling information. A scenario group represents a set of all available scenarios for a kernel K_i. Assuming we have m scenarios for kernel K_i: $SG_{K_i} = \{s_{i,j}, j = \overline{1..m} / s_{i,j} = (t_{sw}, t_{hw}, a)\}$.

Our problem can be formulated as follows: having a set of scenario groups $SGS = \{SG_{K_1}, SG_{K_2}, ...SG_{K_n}\}$ that are in conflict at runtime, and a total area S, determine the particular scenarios selection that will be used $SS = \{s_i / s_i \in SG_i\}$ that minimizes the total execution time of the running threads taking into account the parallel hardware execution on the reconfigurable hardware and the limited size of the reconfigurable area.

We do not have additional information except the execution times and the area occupied. We assume the software prepares the hardware executions, so the software part is executed before the hardware part. We do not assume any particular order of execution of the threads. We give an example in Figure 6.1. Software only scenarios are omitted as they would not bring any insight into our assumptions. As we do not assume any order of execution of the threads multiple schedules are possible for a chosen set of scenarios. We exemplify in Figure 6.1, the lower bound (ideal case), upper bound (worst case) and one case in between those two. Our problem is to minimize all of these possible schedules given a certain area constraint.

6.2.2 Allocation Algorithm

In the rest of this section, we present a compile-time allocation algorithm for threads that compete for the reconfigurable hardware. We assume that the profiler provides the set of kernels that simultaneously require the reconfigurable hardware and we aim to select the optimal scenario for each kernel such that the total execution time of the threads is minimized. However, taking into account the actual total execution time for the threads is dependent on the sequence in which the threads require the reconfigurable hardware and we do not have such information at compile-time. We want to minimize the upper bound of the total execution time. We assume the threads are independent and that the software processor will be fully utilized. Then for a set of scenarios with n elements, the execution time is $U = \sum_{k=1}^{n}(t_{sw_k}) + \max_k(t_{hw_k})$.

For the problem defined in Section 6.2.1, we transform it in an ILP problem as

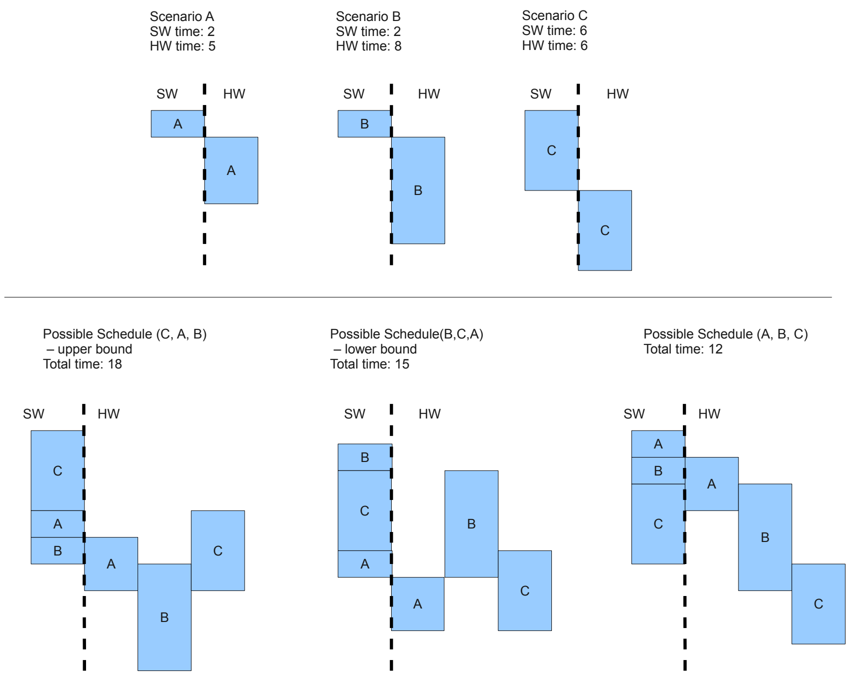

Figure 6.1: Example of scenarios (software only scenarios ommited for brevity) and schedules. As this is not fixed at compile-time, each schedule can arise during normal program execution. We give here the upper and lower bound, and also one more of the possible schedules.

follows.

0-1 selection In our case, only one scenario from a scenario group must be selected for execution. We adopt the following notations:

- n the total number of scenario groups.
- m_i be the number of scenarios in scenario group i.
- $S_{i,j}$ the j-th scenario from scenario group i.
- $x_{i,j}$ a boolean variable such that $x_{i,j} \begin{cases} 0, S_{i,j} \notin SS \\ 1, S_{i,j} \in SS \end{cases}$.
- A the total area available.

Finding the result set of scenarios is reduced to finding the values for all $x_{i,j}$.

The objective function is $\min(\sum_{i=1}^{n}(\sum_{j=1}^{m_i}(t_{sw_{i,j}} * x_{i,j})) + \max_{i=1}^{n}(\sum_{j=1}^{m_i}(t_{hw_{i,j}} * x_{i,j})))$ which can be simply expressed as:

$$\min(\sum_{i=1}^{n}\sum_{j=1}^{m_i}(t_{sw_{i,j}} * x_{i,j}) + y) \tag{6.1}$$

$$\text{for each i from 1 to n: } \sum_{j=1}^{m_j}(t_{hw_{i,j}} * x_{i,j}) \leq y \tag{6.2}$$

The constraints can be represented as a system of **linear pseudo-boolean inequalities**. The one scenario per scenario group constraint will be expressed for each $i = 1..n$ as: $\sum_{j=1}^{m_j} x_{i,j} = 1$.

The selected scenarios must fit together on the available reconfigurable hardware, thus we have the area constraint: $\sum_{i=1}^{n}\sum_{j=1}^{m_i}(a_{i,j} * x_{i,j}) \leq A$.

6.3 Results

In this section, we present the results of the algorithm for the two case studies namely the beamforming application and synthetic applications. The other applications analyzed in our work would not benefit from this algorithm as most of their computations are done in a sequence.

The **beamforming application** idea is to enhance the capabilities of sensors - in our case microphones - by jointly taking the individual signals of multiple sensors into one computation and thereby modify their spatial directivity. The application is composed of two threads: one that computes the signals for each source and one that adjusts the parameters of the computation based on the movement of the sources in space. For the computation thread, the kernel is the FIR filter, executed in parallel for all sources. The thread that performs the adjustments has a kernel represented by a matrix multiplication.

The **synthetic applications** are generated for different numbers of conflicting scenario groups. We analyzed when the number of threads ranges from 2 to 6. The upper bound for the number of threads is based on the fact that in order for the algorithm to be applied, at least one hardware implementation for each thread has to be available. As the area on the FPGA is limited, we used as an indication of kernel size the smaller kernel in our real evaluation application, which occupies 14% of the area of the FPGA as it can be seen in the second row of the second thread in Table 6.1. The number of scenarios per scenario group was chosen between 2 to 8 because of the same reason, that the hardware implementations have to fit the total FPGA area. For the speedup we use as reference the Delft Workbench Automated Reconfigurable VHDL Generator (DWARV) [91] hardware compiler which automatically generates the hardware kernels from C code. The speedup obtained for various tests is up to 10x. We assumed a speedup between 2x and 10x. The area used for the hardware scenarios was from 10% to 50%, and the total area was proportional to the number of scenario groups. The previous parameters were chosen after analyzing the beamforming applications and various hardware kernels.

As we perform a statistical analysis an important question is how big is the sample of the problems. We target a relative standard deviation of 5% for our generated sample of problems. The relative standard error represents the ratio between the standard deviation and the estimate, which in our case is the speedup obtained by an algorithm when compared to software only execution. By iteratively generating different sample sizes we observed that for a a sample size bigger than 360 problems, the relative standard deviation is below 5% for all problem instances.

At the time of the development of this work, neither the hArtes platform, nor our own ML410 based platform were yet available. Considering the same architectural concept, we tested independently the part of the applications on the available system. For the beamforming application, we have implemented and tested each kernel individually using the DWARV tool. We used Xilinx Virtex II Pro running at 300 MHz and the hardware designs clocked at 100 MHz. The results for the entire execution were estimated from the profiling information available.

We compare our algorithm, which is denoted in the rest of the section as *ILP*, to:

- A straightforward allocation solution when the total available area is equally divided between running threads - referred to as the *Equal* case.
- An adapted version of one of the algorithms proposed in [68] referred to as *FixRwSW*. The *FixRwSw* algorithm relies just on choosing from a hardware or a software version of the kernel.

As we do not consider the threads have dependencies, we can not have a predetermined execution order. For our example, sometimes thread 1 will start

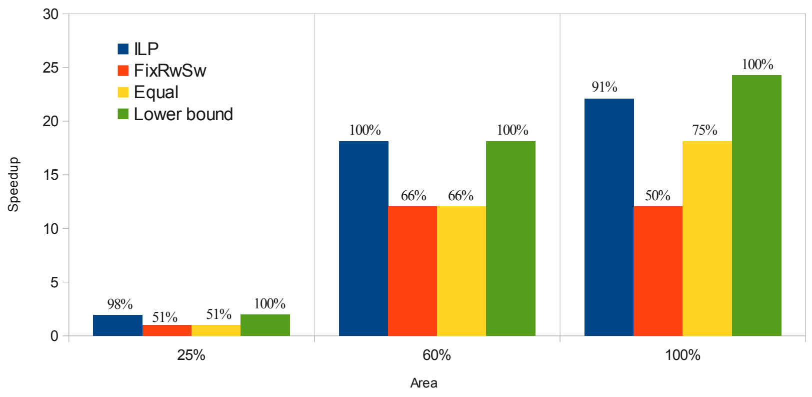

Figure 6.2: Beamforming application speedup compared to software execution. The percentage represents the percentage obtained from the lower bound case (the closest is the percentage to 100%,the better).The application contains two threads and each thread contain 8 and respectively 6 scenarios.

first and other times, thread 2 will start first. Still we need to compare different allocations from the point of view of the execution time. To do this, for all the algorithms we consider the average execution time of all possible schedules. Given the example in Figure 6.1 there are 6 schedules possible of the 3 scenarios. We also compute the absolute lower bound of the execution time - marked *Lower bound* in the graphs. This lower bound is reached for the best possible scheduling of all the threads. In the example, the lower bound is 15. The lower bound is used for comparison purposes as we can not guarantee it will be reached in all the situations.

The **results for the beamforming application** are summarized in Figure 6.2. We ran all the algorithms for different available areas on the reconfigurable fabric. In a real system the area available will depend on the amount of features that have to be included in the system. More feature will require multiple IP blocks and will reduce the available area. The speedup is compared to software only execution. Our algorithm performs better than both *Equal* and *FixRwSW*. The lower bound represents though only the best possible schedule that may appear during application execution. As we assume this is not controllable, it has to be regarded as an ideal case. In some cases the schedule does not influences the total execution time. In those cases the *ILP* algorithm can obtain the best possible result. For our example, such a case is the case when we use 60% of the area available.

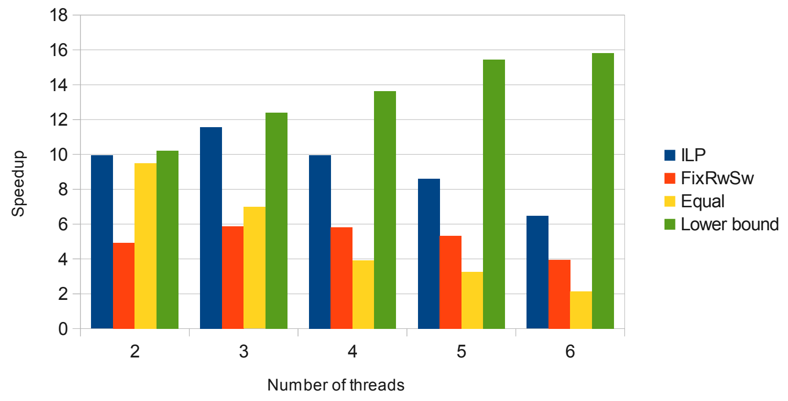

Figure 6.3: Execution time for synthetic applications for different number of threads.

For synthetic applications, the behavior of the algorithms when the number of threads is increased is presented in Figure 6.3. For all cases, the *ILP* algorithm obtains a bigger speedup than the other algorithms. Given the fact that the area is constant, adding multiple threads will decrease the overall spedup obtained.

We also investigate the impact on performance of the size of the reconfigurable hardware. For a set of 6 tasks, the results are presented in Figure 6.4. We notice that our ILP algorithm is again better than both *Equal* and *FixRwSW*. This difference increases as more area becomes available, as the ILP takes into account more possibilities than the other algorithms.

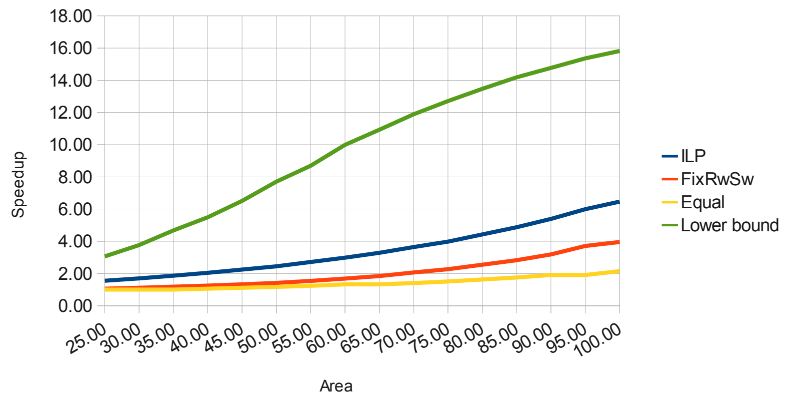

Figure 6.4: Synthetic application execution time versus area.

From the above results, we can conclude that our algorithm can be applied effectively when the unoccupied area is comparable to the area used by a scenario group in case all the iterations are implemented in parallel. However, our algorithm is not suitable for small kernels which can be dynamically configured at runtime.

6.4 Runtime Algorithm

In this section, we will present a runtime algorithm that fulfills the same role as the ILP but at runtime. There are several problems with the ILP algorithm. The first one is that it needs to have complete information about all the scenarios when it is executed. While this can be possible for small systems, in large, dynamic system, not all information will be available at compile-time. The second issue with the algorithm is that ILP is extremely computing intensive, and it is not well suited for runtime execution. To compensate both these problems we propose a runtime algorithm and show its performance is comparable to the ILP.

This algorithm will not consider partial reconfiguration is a viable option during application execution. A complete configuration will be performed, but only before the application starts and only if the system changed in a significant way. This can happen if the available area changes or more threads are needed.

Algorithm

The algorithm has the same input as described in Section 6.2.1. With the same notations, the algorithm is given in Listing 6.1. We name it Runtime Scenario Selection Algorithm (RSSA).

The algorithm is an iterative greedy algorithm. The idea is that at each step, for each scenario, we compute the efficiency of using a larger hardware scenario. This efficiency is computed as the time improvement over the additional area occupied by the new scenario. The algorithm is run until no modification is possible.

Listing 6.1: Runtime scenario selection algorithm.

```
areaUsed = 0
modification = true
for i = 1 to length(sg) {
  solution[i] = 0;
}

while(modification) {
  scenEff = inf;
  for i = 1 to length(sg) {
    scenario old = sg.scenarios[solution[i]];
    for j = 1 to length(sg.scenarios) {
      scenario new = sg.scenarios[j];
      newEff = (old.timeSW + old.timeHW -
                (new.timeSW + new.timeHW)) /
                (old.area - new.area) < scenEff;
      if areaUsed - old.area + new.area < totalArea
         and newEff < scenEff {
         scenEff = newEff;
         modification = true;
         modify = i,j;
      }
    }
  }

  if(modification) {
     apply modification
  }
}
```

Using the notations in Section 6.2.2, the complexity can be expressed as $\mathcal{O}(n^2 * max(m_i))$.

Results

The results of RSSA are within 7% of the result obtained by the ILP algorithm. This can be seen in Figure 6.5, where we represent the time for both algorithms, when area varies.

6.5 Conclusions

In this chapter, we proposed two allocation algorithms that select between multiple implementations of the kernels taking into account the hardware parallel execution and the size of the available reconfigurable area. We compared the *ILP* algorithm to a straightforward allocation algorithm which equally divides

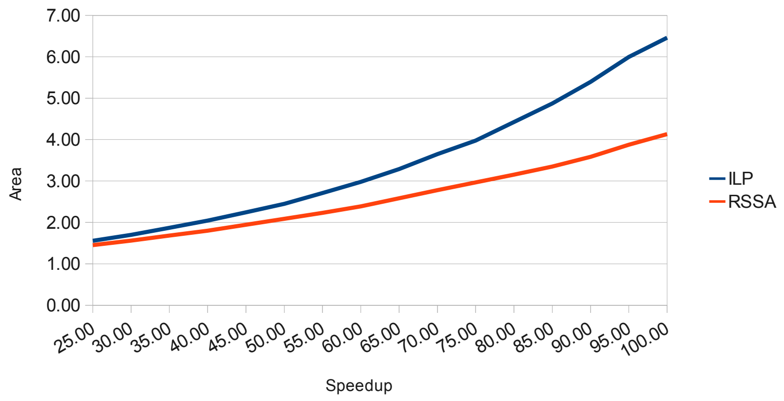

Figure 6.5: Synthetic applications speedup for different areas used of the FPGA.

the available area to the number of threads and estimated that for a real application and hardware implementation our algorithm has a double speedup. Our algorithm is close (up to 33%) to the lower bound. Finally, we determine that our algorithm performs well for those cases where the size of the reconfigurable area is not large enough to fit all the largest scenarios inside. When validating these algorithms using the beamforming application, we demonstrated that the estimated speedup obtained when applying the *ILP* algorithm, is close to the lower bound. Then, we proposed a runtime algorithm that has similar performances to the ILP solution but can be executed at runtime.

Note.

The content of this chapter is based on the following paper:

V.M. Sima, E. Moscu Panainte, K.L.M. Bertels, **Resource Allocation Algorithm And OpenMP Extensions For Parallel Execution On A Heterogeneous Reconfigurable Platform**, Proceedings of 2008 International Conference on Field Programmable Logic and Applications (FPL), Heidelberg, Germany, September 2008.

7 Conclusions

In this thesis, we addressed the problem of runtime adaptation of the application to its executing environment by combining compile time analysis and runtime optimizations that target heterogeneous multicore platforms. Performing an analysis only at compile or only at runtime can mean that, in many cases, optimization possibilities are lost. Some of the main problems that are involved in the development of an application, namely partitioning, mapping and parallelism were addressed, and solutions proposed. In this chapter, we discuss also how these optimizations can be combined, to improve even further the overall system performance. For each of the issues, we detail the future research directions.

7.1 Outlook

In Chapter 2, we presented the key concepts used in this thesis: platforms, toolchains and applications. For each choice, we discussed the requirements and the chosen solution.The Molen machine organization and Molen programming paradigm were introduced, and the main components of the toolchain were presented. Next we presented the related work that addresses the mapping, memory allocation and thread allocation problems.

In Chapter 3, we presented the work done in the context of the hArtes project. The main purpose of this project was to develop a holistic approach for developing multicore heterogeneous platforms. The hardware platform, the hArtes Hardware Platform (hHP), developed in the project, was described in detail. We discussed the design decisions and highlighted its issues. We continued by presenting the toolchain used to build an application for the platforms. Then, the applications provided by the project were analyzed in order to obtain the problems that needed to be solved.

The subsequent chapters each addresses one or more of the issues identified in Chapter 3.

In Chapter 4, we presented a dynamic mapping algorithm that allows to identify, at runtime, the best mapping based on parameter values. Our solution relied on a software cache for parameter values and execution times. The cache was created at runtime using compiler instrumented functions.

In Chapter 5, we tackled the problem of memory allocation, in the case of a platform with memory distributed across multiple processing elements with different computing capabilities. The hHP platform had many different types of memories, with different access speeds from different processing elements. This made efficient mapping of an application a challenge as it implied that specific memory transfers to the different local memories are needed. In order to reduce such a communication overhead - and thus improve performance - we presented an algorithm that tracks memory utilization from different computations at runtime. Then, at next allocation of a specific memory block, a decision regarding its destination can be made, based on the execution history.

In Chapter 6, we presented the issues that arise when the number of threads that can be executed is not fixed. In most multicore systems, the processing elements are identical and can perform computations with the same efficiency. This is not true in a heterogeneous reconfigurable system. We presented an algorithm that decides at compile-time, which is the best division of the available area, taking into account the speedup obtained for the computations mapped.

7.2 Dissertation Contributions

The contributions of this thesis are:

1. A novel mapping algorithm that decides on which processing element a particular computation can be executed in the shortest amount of time. Compared to other mapping algorithms, the decision is delayed until runtime, when using information gathered at compile-time, the best decision can be made. We show that this organization, allows the algorithm to take advantage of some optimization possibilities lost in case the analysis is performed only at compile-time (Chapter 4).

2. A novel memory allocation algorithm targeted to heterogeneous platforms. This algorithm uses the application execution history and the

characteristics of computations, to decide, at runtime, the best allocation of memory in the current memory hierarchy (Chapter 5).

3. A new allocation algorithm for multiple concurrent threads that need to be mapped onto a reconfigurable fabric (Chapter 6).

4. An analysis of the issues that emerge during the development of a toolchain for a heterogeneous multicore platform. Solutions for these issues are proposed, and their implementation is presented (Chapter 3).

7.3 Future directions

In this section, we discuss possible interactions between the algorithms presented. We also highlight the future work for each part of the work, ie. mapping, memory allocation and parallel execution.

Applying multiple algorithms

The algorithms presented in this thesis are assumed to be orthogonal to each other. We applied each of them only for one example but there is no fundamental issue in applying all of them for the same application.

Each of the algorithms addresses another aspect of executing on a heterogeneous platform. The aspect of memory management is addressed by the Adaptive Memory Mapping Algorithm (AMMA) algorithm presented in Chapter 5. The algorithm decides which blocks should be allocated directly in the scratch pad memory. Then, another algorithm - Adaptive Mapping Algorithm (AMAP) - presented in Chapter 4 decides besides on the parameter values if a call of a function should be executed accelerated or not.

As both algorithms try to adapt to the execution environment, they will interfere with each other. Any change performed by the algorithms in the allocation or the mapping can be perceived by the other algorithm as an environment change. It is an open question how fast the system will converge to a stable state where no change will be performed.

The scenario-based approach can be applied independently of the previous two algorithms as it works on a different, higher, level. In the context of multiple threads AMAP algorithm should be extended to provide a management of the available memory for each thread.

Runtime hardware/software mapping

The work presented in Chapter 4 presents the general idea and shows how it can be applied successfully to improve the execution time of an application. A software cache is used to store information about the behavior of the computations. As with any cache, a thorough analysis of the purging policies could be made. An even more powerful possibility is to combine this caching technique with other techniques such as program phase analysis.

Given the available area, the algorithm could be implemented as a hardware module. This would decrease the overhead of its execution and allow it to be used for a wider range of computations. If implemented in software, some functions with short execution time might not be managed by the algorithm as the overhead of calling the algorithm surpasses their execution time.

Combining the scenario based work with the runtime mapping is also a possibility as the latter does not analyze the global state but the state for each single kernel execution, while the former takes parallelism into account so, a combination of those algorithms can provide significant gains in regard to application execution time.

Runtime memory allocation

The application and test platform had such an architecture that even if the execution was in parallel (General Purpose Processor (GPP) and Field Programmable Gate Array (FPGA)), the memory was accessed primarily from the local cache for GPP or scratch pad memory for FPGA. Still, future architectures might have more complex memory organizations, so the impact of parallel execution on the allocation strategies should be further analyzed. Two examples of such architectures are reconfigurable partitioned caches [73] or reconfiguration cache hierarchy [8].

Exact memory utilization

Knowing the exact amount of memory used by a kernel function when executed on the FPGA could bring significant advantages. This would improve the computation of the cost of runtime mapping and improving the speedup obtained.

For now, runtime mapping uses a profiling mechanism to determine the speedup obtained when executing a function in hardware instead of software.

This speedup is affected by the amount of memory that has to be transferred from the main memory to the local FPGA memory. Introducing a cache could help in some of the cases, but, in case the size is known, making a transfer is always at least as good as the cache.

Until now, if the memory is not allocated in the Scratch Pad Memory (SPM), the size transferred is based on the size of the allocated block. But for some application where the kernel accesses a small part of the memory block this can be a bottleneck for small iteration count kernels.

Bibliography

[1] Ieee standard for reduced-pin and enhanced-functionality test access port and boundary-scan architecture. *IEEE Std 1149.7-2009*, pages c1 –985, 10 2010. 31

[2] Javed Absar and Francky Catthoor. Analysis of scratch-pad and data-cache performance using statistical methods. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 820–825, Piscataway, NJ, USA, 2006. IEEE Press. 18, 96

[3] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, Ed Komp, and P. Ashenden. Programming models for hybrid fpga-cpu computational components: a missing link. *Micro, IEEE*, 24(4):42 – 53, july-aug. 2004. 16

[4] Péter Arató, Zoltán Ádám Mann, and András Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Trans. Des. Autom. Electron. Syst.*, 10(1):136–156, 2005. 21, 80

[5] ATMEL. At572d940hf preliminary summary. 31

[6] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002. 19, 96

[7] John Backus. The history of fortran i, ii, and iii. *SIGPLAN Not.*, 13:165–180, August 1978. 13

[8] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257, New York, NY, USA, 2000. ACM. 128

[9] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM. 17, 18, 96

[10] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil Dutt. Parlgran: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 491–496, New York, NY, USA, 2006. ACM Press. 24

[11] Sudarshan Banerjee, Elaheh Bozorgzadeh, Nikil Dutt, and Juanjo Noguera. Selective bandwidth and resource management in scheduling for dynamically reconfigurable architectures. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 771–776, New York, NY, USA, 2007. ACM. 26, 114

[12] Sudarshan Banerjee and Nikil Dutt. Efficient search space exploration for hw-sw partitioning. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 122–127, New York, NY, USA, 2004. ACM. 22, 80, 81

[13] N. Baradaran, Joonseok Park, and P.C. Diniz. Compiler reuse analysis for the mapping of data in fpgas with ram blocks. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 145 – 152, December 2004. 18, 96

[14] Alexandros Bartzas, Miguel Peon-Quiros, Stylianos Mamagkakis, Francky Catthoor, Dimitrios Soudris, and Jose M. Mendias. Enabling run-time memory data transfer optimizations at the system level with automated extraction of embedded software metadata information. In *ASP-DAC '08: Proceedings of the 2008 conference on Asia and South Pacific design automation*, pages 434–439, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press. 19, 96

[15] A. J. Berkhout, D. de Vries, and P. Vogel. Acoustic control by wave field synthesis. *Acoustical Society of America Journal*, 93:2764–2778, May 1993. 67

[16] K. Bertels, V.-M. Sima, Y. Yankova, G. Kuzmanov, W. Luk, G. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, and A. Michelotti. Hartes: Hardware-software codesign for heterogeneous multicore platforms. *Micro, IEEE*, 30(5):88 –97, sept.-oct. 2010. 4

[17] S. Cecchi, A. Primavera, F. Piazza, F. Bettarelli, E. Ciavattini, R. Toppi, J. G. F. Coutinho, W. Luk, C. Pilato, F. Ferrandi, V.M. Sima, and K. Ber-

tels. The hartes carlab: A new approach to advanced algorithms development for automotive audio. In *Audio Engineering Society (AES) Convention*, 2010. 71

[18] Hyungmin Cho, Bernhard Egger, Jaejin Lee, and Heonshik Shin. Dynamic data scratchpad memory management for a memory subsystem with an mmu. In *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 195–206, New York, NY, USA, 2007. ACM. 21

[19] 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*, pages 439 –442, 2008. 30

[20] Michael Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10980, Washington, DC, USA, 2003. IEEE Computer Society. 23

[21] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, 20:85–95, March 2000. 58

[22] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, 2005. 20, 96

[23] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management in a multitasking environment. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 265–274, New York, NY, USA, 2008. ACM. 20

[24] ACE-Associated Compiler Experts. Cosy. 14

[25] S.A. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser. Generic software framework for adaptive applications on fpgas. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 55 –62, 2009. 16

[26] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, page 47, nov. 2004. 11

[27] S. Fekete, E. Köhler, and J. Teich. Optimal fpga module placement with temporal precedence constraints. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 658–667, Piscataway, NJ, USA, 2001. IEEE Press. 25, 113

[28] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 238–243, New York, NY, USA, 2004. ACM. 21

[29] M. French, E. Anderson, and Dong-In Kang. Autonomous system on a chip adaptation through partial runtime reconfiguration. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 77 –86, 2008. 16

[30] Wenyin Fu and Katherine Compton. An execution environment for reconfigurable computing. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 149–158, Washington, DC, USA, 2005. IEEE Computer Society. 23, 80

[31] Michalis D. Galanis, Gregory Dimitroulakos, and Costas E. Goutis. Partitioning methodology for heterogeneous reconfigurable functional units. *J. Supercomput.*, 38(1):17–34, 2006. 23

[32] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14:3:1–3:45, January 2009. 25, 114

[33] Binny S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 4:1–4:17, Berkeley, CA, USA, 2008. USENIX Association. 12

[34] GNU. Gnu compiler collection. 13, 14

[35] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39:49–57, April 2004. 57

[36] J. Harkin, T. M. McGinnity, and L. P. Maguire. Modeling and optimizing run-time reconfiguration using evolutionary computation. *Trans. on Embedded Computing Sys.*, 3(4):661–685, 2004. 23

[37] Scott Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74, New York, NY, USA, 1998. ACM. 24, 113

[38] Jörg Henkel and Rolf Ernst. An approach to automated hardware/-software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(2):273–290, 2001. 23

[39] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. 17, 18

[40] Jason D. Hiser and Jack W. Davidson. Embarc: an efficient memory bank assignment algorithm for retargetable compilers. *SIGPLAN Not.*, 39(7):182–191, 2004. 19, 96

[41] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society. 24

[42] Chen Huang and Frank Vahid. Dynamic coprocessor management for fpga-enhanced compute platforms. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 71–78, New York, NY, USA, 2008. ACM. 23, 80

[43] INRIA. Scilab. 2010. 48

[44] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. 39

[45] ITU. H.264. 2010. 58

[46] Wu Jigang and Thambipillai Srikanthan. Algorithmic aspects of area-efficient hardware/software partitioning. *J. Supercomput.*, 38(3):223–235, 2006. 23, 80

[47] Lech Jóźwiak, Nadia Nedjah, and Miguel Figueroa. Modern development methods and tools for embedded reconfigurable systems: A survey. *Integration, the VLSI Journal*, 43(1):1 – 33, 2010. 14

[48] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 628–633, New York, NY, USA, 2002. ACM. 18, 96

[49] Arun Kannan, Aviral Shrivastava, Amit Pabalkar, and Jong-eun Lee. A software solution for dynamic stack management on scratch pad memory. In *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 612–617, Piscataway, NJ, USA, 2009. IEEE Press. 20, 96

[50] Khronos OpenCL Working Group. The OpenCL Specification, June 1 20011. available at: `http://www.khronos.org/registry/cl/` (June. 2011). 24

[51] Géraud Krawezik. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127, New York, NY, USA, 2003. ACM. 24

[52] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. 13

[53] Leaff. Nu-tech. 2010. 48

[54] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jungho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 193–204, New York, NY, USA, 2010. ACM. 46

[55] R. Leupers and P. Marwedel. Retargetable generation of code selectors from hdl processor models. In *European Design and Test Conference, 1997. ED TC 97. Proceedings*, pages 140 –144, mar 1997. 14

[56] Hongjiu Lu. Elf: From the programmer's perspective. *NYNEX Science & Technology Inc*, page 95, 1995. 35

[57] Brennon Meals. Hierarchical decomposition algorithm for hardware/-software partitioning. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 18–23, New York, NY, USA, 2006. ACM. 22

[58] Narasinga Miniskar, Elena Hammari, Satyakiran Munaga, Stylianos Mamagkakis, Per Kjeldsberg, and Francky Catthoor. Scenario based mapping of dynamic applications on mpsoc: A 3d graphics case study. In Koen Bertels, Nikitas Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5657 of *Lecture Notes in Computer Science*, pages 48–57. Springer Berlin / Heidelberg, 2009. 25, 114

[59] Benoit Miramond and Jean-Marc Delosme. Design space exploration for dynamically reconfigurable architectures. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 366–371, Washington, DC, USA, 2005. IEEE Computer Society. 22, 23, 80, 81

[60] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. available at: `http://www.mpi-forum.org` (Dec. 2009). 24

[61] Alok Choudhary Nagaraj Shenoy and Prithviraj Banerjee. An algorithm for synthesis of large time-constrained heterogeneous adaptive systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):207–225, 2001. 23, 80

[62] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 115–125, New York, NY, USA, 2005. ACM. 20, 96

[63] Juanjo Noguera and Rosa M. Badia. Dynamic run-time hw/sw scheduling techniques for reconfigurable architectures. In *CODES '02: Proceed-*

ings of the tenth international symposium on Hardware/software codesign, pages 205–210, New York, NY, USA, 2002. ACM. 25, 113

[64] OpenMP.org. OpenMP Application Program Interface , May 2008. available at: `http://openmp.org/wp/openmp-specifications/` (June. 2011). 24

[65] Ozcan Ozturk, Mahmut Kandemir, and Ibrahim Kolcu. Shared scratch-pad memory space management. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 576–584, Washington, DC, USA, 2006. IEEE Computer Society. 20

[66] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. Instruction scheduling for dynamic hardware configurations. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 100–105, Washington, DC, USA, 2005. IEEE Computer Society. 24, 113

[67] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. Interprocedural optimization for dynamic hardware configurations. In *SAMOS*, pages 2–11, 2005. 24, 113

[68] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. Compiler-driven fpga-area allocation for reconfigurable computing. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 369–374, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 22, 23, 80, 119

[69] Soyoung Park, Hae-woo Park, and Soonhoi Ha. A novel technique to use scratch-pad memory for stack management. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1478–1483, San Jose, CA, USA, 2007. EDA Consortium. 21

[70] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of performance-optimal multi-level cache hierarchies. *SIGARCH Comput. Archit. News*, 17:114–121, April 1989. 12

[71] Andrew Putnam, Susan Eggers, Dave Bennett, Eric Dellinger, Jeff Mason, Henry Styles, Prasanna Sundararajan, and Ralph Wittig. Performance and power of cache-based reconfigurable computing. *SIGARCH Comput. Archit. News*, 37:395–405, June 2009. 17

[72] Heather Quinn, L. A. Smith King, Miriam Leeser, and Waleed Meleis. Runtime assignment of reconfigurable hardware components for image processing pipelines. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 173, Washington, DC, USA, 2003. IEEE Computer Society. 23, 80

[73] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 214–224, New York, NY, USA, 2000. ACM. 128

[74] Dennis M. Ritchie. The development of the c language. *SIGPLAN Not.*, 28:201–208, March 1993. 15

[75] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013 –1029, jul 1993. 2, 12

[76] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 667–672, New York, NY, USA, 2001. ACM. 12

[77] K Sigdel, M Thompson, C. Galuzzi, A. D. Pimentel, and K.L.M. Bertels. Runtime task mapping based on hardware configuration reuse. In *Proceedings of International Conference on Reconfigurable Computing (ReConFig 2010)*, December 2010. 24, 81

[78] M. Teich, S. Fekete, and J. Schepers. Compile-time optimization of dynamic hardware reconfigurations, 1999. 25, 113

[79] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, New York, NY, USA, 2003. ACM. 19, 96

[80] J. Van Praet, D. Lanneer, G. Goossens, W. Geurts, and H. De Man. A graph based processor model for retargetable code generation. In *European Design and Test Conference, 1996. ED TC 96. Proceedings*, pages 102 –107, mar 1996. 14

[81] B.D. Van Veen and K.M. Buckley. Beamforming: a versatile approach to spatial filtering. *ASSP Magazine, IEEE*, 5(2):4 –24, april 1988. 67

[82] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, 53(11):1363–1375, 2004. 12, 14, 34

[83] Inc. VDC Research Group. Development tools, embedded software market intelligence service. Technical report, 2010. xi, 15, 32, 81

[84] VideoLAN. x264. 2011. 58

[85] Krishna N. Vikram and Vinita Vasudevan. Mapping data-parallel tasks onto partially reconfigurable hybrid processor architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(9):1010–1023, 2006. 25, 114

[86] Miljan Vuletic, Paolo Ienne, Christopher Claus, and Walter Stechele. Multithreaded virtual-memory-enabled reconfigurable hardware accelerators. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 197 –204, 2006. 18

[87] Gang Wang, Wenrui Gong, and Ryan Kastner. Application partitioning on programmable platforms using the ant colony optimization. *J. Embedded Comput.*, 2:119–136, January 2006. 22

[88] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003. 57

[89] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23:20–24, March 1995. 12

[90] xiph.org. Xiph.org test media. 62, 109

[91] Y. D. Yankova, G.K. Kuzmanov, K.L.M. Bertels, G. N. Gaydadjiev, J. Lu, and S. Vassiliadis. Dwarv: Delftworkbench automated reconfigurable vhdl generator. In *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, pages 697–701, August 2007. 90, 108, 110, 119

[92] Lei Zhang, Meikang Qiu, Wei-Che Tseng, and Edwin H.-M. Sha. Variable partitioning and scheduling for mpsoc with virtually shared scratch pad memory. *J. Signal Process. Syst.*, 58(2):247–265, 2010. 18, 96

List of Publications

International Journals

1. K.L.M. Bertels, V.M. Sima, Y. D. Yankova, G. Kuzmanov, W. Luk, G. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, A. Michelotti, **HArtes: Hardware-Software Codesign for Heterogeneous Multicore Platforms**, *IEEE Micro*, October 2010, Special Issue on European Multicore Processing Projects.

International Conferences

1. V.M. Sima, K.L.M. Bertel, **Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform**,*Proceedings of International conference on 16th IEEE Reconfigurable Architectures Workshop*, Rome, Italy, May 2009.

2. V.M. Sima, K.L.M. Bertels, **Runtime memory allocation in a heterogeneous reconfigurable platform**, *IEEE International Conference on ReConFigurable Computing and FPGA*, Cancun, Mexico, December 2009.

3. 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 (BibTeX) (Edit DB Entry), Delft Workbench, MOLEN

4. V.M. Sima, E. Moscu Panainte, K.L.M. Bertels, **Resource Allocation Algorithm And OpenMP Extensions For Parallel Execution On A Heterogeneous Reconfigurable Platform**, *Proceedings of 2008 International Conference on Field Programmable Logic and Applications (FPL)*, Heidelberg, Germany, September 2008.

5. S. Cecchi, A. Primavera, F. Piazza, F. Bettarelli, E. Ciavattini, R. Toppi, J. G. F. Coutinho, W. Luk, C. Pilato, F. Ferrandi, V.M. Sima, and K. Bertels. **The hartes carlab: A new approach to advanced algorithms development for automotive audio**, *Audio Engineering Society (AES) Convention*, San Francisco, USA, 2010.

Books Chapters

1. Giovanni Beltrame, Fabrizio Ferrandi, Luca Fossati, Christian Pilato, Donatella Sciuto, Roel J. Meeuws, S. Arash Ostadzadeh, Zubair Nawaz, Yi Lu, Thomas Marconi,Mojtaba Sabeghi, Vlad Mihai Sima, Kamana Sigdel, **Extensions of the hArtes Tool chain** Chapter 6 in *Hardware/-Software Co-Design for Heterogeneous Multi-Core Platforms*, Springer, to be published in 2011

Local Conferences

1. V.M. Sima, E. Moscu Panainte, K.L.M. Bertels, **FPGA area allocation for parallel C applications**, *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing (ProRisc)*, Veldhoven, November 2007.

2. V.M. Sima, E. Moscu Panainte, K.L.M. Bertels, **Area allocation on reconfigurable hardware for parallel C applications**, *Architectures and Compilers for Embedded Systems (ACES)*, Edegem, Belgium, September 2007.

Publications not directly related to this thesis (international conferences)

1. 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

Samenvatting

In deze dissertatie bestuderen we het probleem om een toepassing, tijdens de uitvoering, af te stemmen aan zijn werkomgeving. Een kenmerkend voorbeeld is het verplaatsen van een berekening van de ene naar de andere verwerkingseenheid, rekening houdend met de beschikbare verwerkingseenheden in het systeem. Dit vind plaats op basis van de informatie en instrumentatie waarin de de compiler voorziet en op basis van de status van de werkomgeving. Dit werk concentreert zich op heterogene multicore embedded architecturen. De volgende drie aspecten van applicatie-optimalisaties worden in deze dissertatie besproken: hardware/software mapping, geheugentoewijzing en parallele executie. Voor elk aspect werd een algoritme ontwikkeld en, gebruikmakend van een geschikte toepassing, getest op een experimenteel hardwareplatform. De basis van dit werk is de Molen architectuur en bijhorend programmeermodel dat werd aangepast in functie van het specifieke hardware platform en de werkomgeving.

Het doel van het hardware-software toewijzingsalgoritme is om, tijdens de uivoering, de verwerkingseenheid te kiezen waar een taak het meest efficint kan worden uitgevoerd. Aangaande de geheugentoewijzing stellen wij een algoritme voor, dat tijdens de uitvoering beslist wat de beste geheugentoewijzing is op basis van informatie vergaard tijdens de compilatie en de huidige werkomgeving. Teneinde parallelle applicaties te ondersteunen, hebben we een algoritme ontwikkeld dat de beste overweging maakt tussen het benodigde chipoppervlak en de snelheidswinst door te bepalen hoeveel taken parallel genstantieerd dienen te worden.

De experimenten werden uitgevoerd op een heterogeen multicore embedded platform, om precies te zijn het hHP. Het platform bestaat uit een General Purpose Processor - ARM, een Digital Signal Processor (DSP) - Atmel Magic en een Field Programmable Gate Array (FPGA) - Xilinx Virtex4. De validatie gebeurde aan de hand van een aantal industrile multimedia-applicaties: een video encoder/decoder en een golfveldsynthese applicatie. Het toewijzingsalgoritme behaalt verbeteringen tussen de 5% en 43%. We toonde bovendien aan dat het een flexibel algoritme is, dat de uitvoering aanpast in het geval dat de uitvoeringsoverhead toeneemt. Het geheugenallocatie-algoritme behaalde een snelheidswinst van 18% voor de geselecteerde applicatie. Voor dit algoritme tonen we dat de voorgestelde allocatie zich binnen 14% van de optimale allocatie op basis van Integer Linear Programming (ILP) bevindt. De selectie van parallele berekeningen gebaseerd op scenarios, is tussen de 21% en 92%.

Curriculum Vitae

Vlad-Mihai Sima was born on 21st of August 1980 in Bucharest, Romania. He recevied his B.Sc. from University ”Politehnica” of Bucharest, Faculty of Automatic Control And Computers in 2004. The graduation project was carried out during a 6 months internship at MDCR (Motorola Development Center Romania) and the subject was the retargeting of a compiler to the ARM processor. He obtained his M.Sc from University ”Politehnica” of Bucharest, Faculty of Automatic Control And Computers in 2005 with the thesis subject on size optimization for embedded architectures. He worked as a compiler engineer between 2004 and 2006 at Freescale, Romania.

In 2006 he started his Ph.D. studies at Computer Engineering group, TU Delft, where he worked in DelftWorkBench project under the supervision of Prof. Koen Bertels. He was involved in multiple projects related to reconfigurable architectures like Morpheus, hArtes, rcosy, IFEST and Reflect. The work was focused on compiler optimization and runtime adaptation based on compiler provided information.

His current research interest include: Compile Design and Optimization, Hardware/Software Co-Design and Embedded Systems.