UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Just-in-Time Processor Customization – on the Feasibility and Limitations of FPGA-based Dynamically Reconfigurable Instruction Set Architectures

Dissertation

von
Mariusz Grad

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

To *Anna*

Abstract

In the new millennium, the increase of processing performances in sequential processors had reached their limits due to power constrains in semiconductor technology. In consequence, the field of reconfigurable computing where, power-efficient and high-performance solutions are investigated, had attracted a lot of attention in recent years. Reconfigurable application specific instruction set processors provide the possibility of tailoring the instruction set architecture of a processor to a particular application. Although researchers presented orders of magnitude performance gains for various applications, when targeting these architectures, this time consuming and error-prone customization process is rarely utilized. While the automatic and online customization process is technically feasible and provides a promising technology for adaptive computer systems optimizing themselves according to the needs of the actually executed workload, the idea of just-in-time processor customization has been hardly investigated. Despite promising research in this topic, a number of obstacles makes this exploitation challenging. First, there are only very few commercially available silicon implementations of reconfigurable ASIP architectures. Secondly, methods for an automatic identification of custom instructions are algorithmically expensive and require profiling data that may be unavailable until runtime. Finally of all, the synthesis and place-and-route tool flows for reconfigurable logic are known to be notoriously slow.

In this work, a unique system consisting of hardware architecture and a software tool flow that addresses the two first obstacles mentioned above was designed, and, in consequence, allows for a fully automatic and online customization. The designed hardware architecture is a dynamically reconfigurable instruction set processor that allows for the just-in-time processor customization during the runtime, whereas the software tool flow is based on a virtual machine and allows to customize this architecture concurrently to the application execution and without any manual efforts. This tool flow contains a set of heuristics that reduce the runtime of methods for identifying and selecting custom instructions for the just-in-time processor customization as well as a circuit library and a data path generator of required bitstreams for the hardware customization.

This dynamic system, in contrast to the static one, has several advantages. First of all, it is fully automated and no manual efforts are required. It can optimize its operation by reconfiguring the instruction set architecture of the proces-

sor and by changing the code at runtime, which is fundamentally more powerful than a static approach. This developed system can collect the execution time, profiling, and machine level information in order to identify the code sections that are actually performance limiting at runtime and therefore, constitute the ideal candidates for hardware acceleration. Moreover, the virtual machine has the capability of executing various dynamic optimizations such as hotspot detection, alias analysis, or branch prediction to further optimize the performances. Finally, the dependencies between variables and the corresponding memory layout are available, which simplifies the task of the hardware-software partitioning between the processor and the hardware accelerator.

While the advantages of the developed dynamic system over a static one are clear, it is still controversial whether the just-in-time processor customization is a worthwhile idea under the assumption that the currently commercially available FPGA devices and tools are used. The goal of this thesis is to thoroughly study the posed question and, in particular, to investigate how the long runtimes of FPGA implementation tools limit the applicability of the just-in-time approach. While it is evident that even long runtimes of design tools will be amortized over time provided that an application-level speedup is achieved, it is so far an open question whether the total required execution time until a net speedup is achieved stays within any practical bounds. In this work, this question is answered in detail for a set of benchmark applications from both embedded and scientific computing that target our real – and not emulated system.

The distinguish feature of this thesis is the fact that it not only provides with answers to problems related to the above described obstacles from the static systems, but more importantly it presents a complete and holistic system that delivers a feature that is not found in any other work. This feature allows to dynamically customize the processor to the given currently executed workload. In consequence, this work naturally spans over several different scientific fields such as dynamically reconfigurable instruction set architectures, instruction set extension algorithms, or high level synthesis and merges them together into a single coherent system. These fields were studied in other works only in separation, in our work, they are not only coupled together into a holistic system but more importantly they provide with a unique reconfigurable system capable of the just-in-time processor customization.

Zusammenfassung

Im neuen Jahrtausend hat der Zuwachs an Rechenkapazität sequenzieller Prozessoren auf Grund von Leistungsbeschränkungen in der Halbleitertechnologie seine Grenzen erreicht. Daher hat das Feld des rekonfigurierbaren Rechnens, in dem verbrauchseffiziente und Hochleistungslösungen erforscht werden, in den letzten Jahren große Aufmerksamkeit erhalten. Rekonfigurierbare, anwedungsspezifische Befehlssatzprozessoren bieten die Möglichkeit die Befehlssatzstruktur eines Prozessors an eine bestimmte Aufgabe anzupassen. Obwohl Forscher die Leistug für viele Anwendungen um Größenordnungen verbessern konnten, wird dieser zeitaufwendige und fehleranfällige Anpassungsprozess selten für diese Architekturen benutzt. Während der automatische und online Anpassungsprozess technisch möglich ist und eine vielversprechende Technologie für adaptive Computersysteme bietet, die sich abhängig von der tatsächlich nötigen Arbeitslast selbst optimieren, wurde die Idee der 'just-in-time' Prozessoranpassung kaum untersucht. Trotz vielversprechender Untersuchungen auf diesem Gebiet, gibt es einige Hürden, die die Nutzung erschweren. Zum Einen gibt es nur sehr wenige kommerziell erhältliche Silikonimplementierungen von rekonfigurierbaren ASIP Architekturen. Zum Anderen sind Methoden zur automatischen Identifikation von zugeschnittenen Befehlen algorithmisch teuer und erfordern die Analyse von Daten, die bis zur Laufzeit nicht vorhanden sind. Ausßerdem ist es bekannt, dass die Synthese und 'place-and-route tool flows' für rekonfigurierbare Logik notorisch langsam sind.

In dieser Arbeit haben wir ein einzigartiges System, bestehend aus Hardwarearchitektur und Software 'tool-flow' entworfen, dass die ersten beiden oben genannten Hindernisse adressiert und daher eine komplett automatische und online Anpassung erlaubt. Die entworfene Hardwarearchitektur ist ein dynamisch rekonfigurierbarer Befehlssatzprozessor, der 'just-in-time' Prozessoranpassung während der Laufzeit erlaubt, wohingegen der Software 'tool-flow' auf einer virtuellen Maschine beruht und gestattet die Architektur gleichzeitig mit der Programmausführung, und ohne jeglichen manuellen Aufwand, anzupassen. Dieser 'tool-flow' enthält ein Set von Heuristiken, die die Laufzeit von Methoden zur Identifizierung und Auswahl von eigenen Anweisungen für die 'just-in-time' Prozessoranpassung reduzieren, sowie eine Schaltkreisbibliothek und einen Datenpfadgenerator für benötigte bitstreams für die Hardwareanpassung.

Dieses dynamische System, im Gegensatz zum statischen, hat mehrere Vorteile.

Erstens, ist es völlig automatisiert und benötigt keinen manuellen Aufwand. Es kann seinen Ablauf optimieren, indem die Befehlssatzarchitektur des Prozessors rekonfiguriert und der Code während der Laufzeit verändert wird, was fundamental leistungsstärker ist als die statische Herangehensweise. Dieses entwickelte System kann die Ausführungszeit, Profilierung und Maschinenlevelinformationen sammeln, um die Codeteile zu erkennen, die bei der Laufzeit tatsächlich leistungslimitierend sind und daher ideale Kandidaten zur Hardwarebeschleunigung sind. Darüberhinaus hat die virtuelle Maschine die Möglichkeit, verschiedene dynamische Optimierungen auszuführen, wie Hotspoterkennung, Aliasanalyse oder Branchvorhersage, um die Leistung weiter zu optimieren. Letztlich sind die Abhängigkeiten zwischen Variablen und der dazugehörigen Speicherbelegung zugänglich, was die Hardware-Software-Partitionierung zwischen dem Prozessor und dem Hardwarebeschleuniger vereinfacht.

Während die Vorteile unseres dynamischen Systems gegenüber einem statischen klar sind, ist es immernoch umstritten, ob 'just-in-time' Prozessoranpassung eine lohnenswerte Idee ist, unter der Annahme, dass die derzeit kommerziell erhältlichen FPGA Bauteile und Werkzeuge benutzt werden. Das Ziel dieser Dissertation ist es dieser Frage gründlich nachzugehen und im Besonderen zu untersuchen wie die langen Laufzeiten von FPGA Entwicklungswerkzeugen die Anwendbarkeit der 'just-in-time' Herangehensweise einschränken. Während es offensichtlich ist, dass sogar lange Laufzeiten der Entwicklungswerkzeuge über die Zeit amortisiert werden, solange eine Anwedungsbeschleunigung erzielt wird, ist es bisher eine offene Frage, ob die gesamt erzielte Ausführungszeit bis eine Nettobeschleunigung erzielt wurde, in praktikablen Grenzen bleibt. In dieser Arbeit beantworten wir diese Frage im Detail für ein Set von Benchmarkanwendungen aus eingebettetem und wissenschaftlichem Rechnen, die unser reales – und nicht emuliertes – System betreffen.

Die Besonderheit dieser Dissertation ist, dass sie nicht nur Antworten auf die oben genannten Hürden der statischen Systeme liefert, sondern dass sie auch ein komplettes und holistisches System präsentiert, das in keiner anderen Arbeit gefunden werden kann. Diese Eigenschaft erlaubt es den Prozessor dynamisch an den aktuell ausgeführten Arbeitsaufwand anzupassen. Diese Arbeit spannt daher selbstverständlich über mehrere wissenschaftliche Gebiete, so wie dynamisch rekonfigurierbare Befehlssatzarchitekturen, Befehlssatzerweiterungsalgorithmen und 'high-level' Synthese und verbindet sie in ein einziges, kohärentes System. Diese Felder wurden in anderen Arbeiten nur seperat bearbeitet. Hier werden sie nicht nur in ein einziges holistisches System verknüpft, sondern vor allem bieten sie ein einzigartiges rekonfigurierbares System, das die 'just-in-time' Prozessoranpassung beherrscht.

Contents

Part I

Introduction and Motivation

Chapter 1

Introduction

Computers are software programmable machines designed to automatically carry a sequence of logic operations in order to solve a given problem - an algorithm. The performances of processing depend on the computer hardware architecture and used components, in particular, on a switching element which is used in construction of most essential parts of the computer: central processing units and memories. In order to obtain the best performance over the years the switching element was always driven by the state-of-the-art available technology i.e. mechanical relays, electronic vacuum tubes, and finally transistors.

1.1 [1965 – 2004]: The Era of Frequency Scaling and Uniprocessors

In 1974, Robert H. Dennard published a paper [1] where he described a long-term pace in which the transistor-switched element would scale and improve the performance. The paper stated that within every new semiconductor technology, which approximately appears every two years, a) circuits will be 40% faster, b) power consumption will stay the same, and c) the transistor density will double, which is a fundamental reason behind the Moore's law [2]. This phenomena is known under the term *frequency scaling* and it was valid for decades - roughly until year 2004 [3]. The faster circuits were not solely responsible for performance improvements. The smaller dimensions of transistors allowed to organize and develop better architectures with multi-cycle execution, pipelining, and branch prediction as well as with multiple layers of caching mechanisms which removed bottlenecks associated with fetching data directly from the main memory. During the frequency scaling years, the processing performance of uniprocessors grew on average by 52% a year and overall up-to more than *three-orders-of-magnitude* [4]. For a single threaded software application this meant that approximately every two years the application performance doubled, and every five years it increased by more than a factor of eight, which confirmed the Pollack's rule of thumb for the microprocessor performance and area.

1.2 [2004 – present]: The Era of Power Wall and Multiprocessors

In May 2004, Intel corporation canceled *Tejas* and *Jayhawk* uniprocessor designs due to the power consumption issues known under the term *power wall*. This date marks the end of the frequency scaling period, further uniprocessors developments, and their incredible performance improvements, and starts the era of multiprocessors with a *power dissipation* as a limiting factor [5].

In the multiprocessor era, the Dennard's law is not valid any more [6]. Due to threshold voltage limitations and leakages in transistors the pace of increased transistor switching and faster circuits is not achievable any longer. In addition, while Moore's law [2] regarding increased densities of transistors in the integrated circuits (IC) is still valid, the total energy available to the IC practically limits the number of available logic transistors for computation [3]. This power wall constrain means that processor designers can not utilize all available transistors to develop computational units (cores) and operate them at full speed since this would excessively transcend the available energy budget to the chip. Thus, the power wall demands *performance-energy* efficient computational solutions.

1.3 Memory Wall

The power wall is not a single factor constraining the current multiprocessors. In 1970, the process of computation was much more expensive than the process of delivering the data from the memory. This imbalance let designers into two directions. To improve the processor performance and to improve the size-price ratio for memories - not throughput performances. Unfortunately, over the years this imbalance had flipped and lead to the *memory wall* [7, 8], which is a consequence of the *von Neumann bottleneck* where memory transfers are expensive and computation cheap. For instance, it takes more than 200 cycles to access DRAM memory from a processor whereas the most sophisticated computational operations performed on a processor, including floating point types, consume just a few clock cycles. This big disproportion had large impact on uniprocessors and for several reasons it has even larger influence on multiprocessors. Since multiprocessors have more than one computational unit and each of them requires data this significantly increases the requirements for the total memory throughput, in contrast to only a single processor.

It could seem that having more memory controllers and wiring them to memories would solve this processor-memory throughput problem. To perform this wiring task the IC would require additional I/O pins. Unfortunately, due to manufacturing constrains in the IC foundries, processor packages have limited number of them (up to 1200) and in the current state it is not possible to increase their number, which discards the idea of additional memory controllers. This creates a large memory bottleneck in the multiprocessors were computational resources are often under-utilized.

Designers try to overcome this issue by bringing data closer to computational units with larger multi-level hierarchies and sophisticated techniques for on-chip cache memories as well as with techniques which allow to hide the memory latency, such as deep execution pipelines

or out-of-order instruction processing. Unfortunately, these approaches consume a considerable amount of the IC die area and transistors that could be utilized for constructing additional computational units. Moreover, the data movement between many levels of different memories consumes energy which in times of the power wall is the most expensive resource that could be utilized to improve computational performances.

1.4 Parallelism Wall

Besides the power and memory walls in the era of multiprocessors, one of the most significant challenges is the software parallelism. In order to utilize many core systems it is required that the software application would consist of a significant portion of parallelism. According to the Amdhal's law [9], an application that has a $10/90$ ratio of sequential to parallel code can improve their performances only by $10\times$. As studied by Gustafson [10], since the execution time of the sequential code to the size is, in most cases, not proportional the performance factor is usually much higher. However, if the application is single threaded and does not contain any parallelism, it will occupy only a single core and thus it will not bring any processing performance improvements over the uniprocessor. This has tremendous meaning since in the current multiprocessor era the space of computational performance improvements is limited only to parallel applications and single-threaded applications are discarded.

The processing performances do not rely solely on a software parallelism but also on the efficient mappings to the underlying hardware. Since every application has different computational and communicational patterns [6, 11] it is hard to create a general hardware architecture with exact number of cores and perfect interconnections which would fit all of these patterns. In order to achieve low-energy low-latency high-performances computations, engineers often enrich general architectures with dedicated hardware accelerators designed for a particular application.

1.5 Brick Wall

The power, memory, and parallelism walls form a *brick wall* that is the main constraining factor of the processing performance improvements in the multiprocessor era. In the past, the newer computer architectures were always driven by the evolution process. In times of the brick wall this process is radically changed. In order to overcome the brick wall requirements, new hardware architectures have to be designed from scratch. These are times of hardware revolutions - not evolutions [6].

In order to achieve better processing performances the hardware designers of future architectures have to solve many important problems. They have to find energy efficient computational solutions and advance interconnections which not only allow to effectively communicate between multiple cores located on-the-chip but more importantly allow to efficiently access memories located on-the-chip as well as off-the-chip. In addition, hardware designers have to consider how software can be effectively mapped to hardware in order to achieve best performances. This is probably one of the biggest obstacles since the parallel applications have different structure, computational and communicational patterns [6, 11] that change with

different input data [12]. Every pattern found in these applications requires a customized hardware in order to achieve peak performance. Thus, it is impossible to efficiently fulfill all brick wall requirements for all these patterns with just a single fixed general hardware architecture [13].

1.6 The Design Solutions in Multiprocessor Era

The processing performance improvements of a processor were always driven by two factors. The major one, frequency scaling phenomena described in Dennard's paper and the minor one corresponding to improvements in a processor architecture. While in times of the power wall, the frequency scaling is not anymore driving the performance improvements, the processor customization allows to improve performances for certain applications.

The processor customization is a well known procedure to Intel, AMD, and other processor manufactures. The customization is performed in order to move computation from software to dedicated hardware and thus, to increase computational performances for certain applications. To achieve this goal manufacturers provide with supplementary instructions to processor's *instruction set architecture* (ISA) that is an interface splitting software and hardware domain. Recent Intel's Sandy Bridge AVX [14], IBM's Altivec [15], Amd's 3d!Now [16], or SSE [17] are good examples of this approach. While these extensions do not break any legacy codes they are fixed and allow to increase the computational performances only to a limited subset of applications. In times of parallel application and their complex computational and communication patterns that change with different input data, this solution is not adequate. It requires to build processor extensions for every recognized pattern. This is an impractical and inefficient approach since it would easily exhaust the budget of available logic transistors. Definitively it would not be suited for all applications and their patterns. To this end, other solution has to be applied.

Recent research at Intel Research Laboratories revealed that instead of traditional approaches with fixed resources the architectures of future will be dynamically customized systems [18]. Such architectures will allow to adapt reprogrammable hardware dynamically (online) to currently executing application and thus to overcome the brick wall constrains and to provide with optimized solutions. These heterogeneous architectures will combine simplistic fixed hardwired computational units with dynamically customized hardware accelerators that will be based on the field programmable gate array (FPGA) devices. The study of the Amdhal law in the era of multiprocessors indicated that such connected dynamically customized heterogeneous architectures outperform other architectures organized in different structures [13].

In these systems, the sequential part of the application will be executed on a fixed computational unit whereas the parallel part responsible for the performance improvements on the hardware accelerator that will be created specifically for a given application. Since these accelerators will be custom fitted for a given application they will achieve high performance-power efficiency. Moreover, their custom designed interconnections will allow to reduce the data movement. This will allow to significantly overcome the memory wall constrain.

The origin of this design is based on solutions from the mobile domain where power-efficient systems-on-chip (SoC) with hardware accelerators are designed and work at low energy budgets for decades [19]. However, while in the mobile markets there is a limited number

of fixed hardware accelerators designed to improve performances of media and cryptography applications, in the case of modern computers, the variety of applications and their computational and communicational patterns is endless. Therefore, it is not efficient to follow the same path and design a fixed set of hardware accelerators for a fixed number of applications. Instead, as indicated by research, just-in-time (JIT) hardware customization is required [13, 18]. These systems allow to create hardware accelerators on demand for the currently executing application and with assistance of a special software shift computation from the processor to the accelerator. Therefore, such systems allow to overcome the brick wall constrains by dynamically adapting their hardware architectures to a given application and to provide with optimized solution for a particular application.

1.7 Contributions

The main goal and contribution of this thesis is to study the tradeoff between the feasibility and limitations of the just-in-time processor customization process. Specifically, we answer a controversial question whether the just-in-time processor customization is a worthwhile idea under the assumption that existing commercially available FPGA and CAD tool flows are used. In particular, we investigate how the long runtimes of the FPGA implementation tools limit the applicability of the approach in future hardware architectures. While it is evident that even long runtimes of design tools will be amortized over time provided that an application-level speedup is achieved, it is so far an open question whether total required execution time until a net speedup is achieved stays within any practical bounds.

To study this question in detail and in order to deliver best quality results, instead of emulation approach, we decided to design and develop an FPGA-based just-in-time hardware architecture amenable of processor customization as well as all necessary software tools to support this online process. Thus, all results presented in this work were obtained from this developed system and were not estimated in the simulation environment. In addition, instead of syntactic benchmarks or just parts of applications, the developed system was empirically evaluated with a comprehensive set of applications obtained from the SPEC2006, SPEC2000, SciMark2, and MiBench benchmarks. These applications represent both scientific and embedded domains.

In addition to the main contribution presented above, this thesis provides with a list of other contributions that were required by and studied during the development process of our system. These contributions cover:

- an exhaustive evaluation and comparison of *instruction set extension* algorithms that allow to select application's code rich in *instruction level parallelism*,

- development and comparison of efficient pruning heuristics that allow to reduce the size of the application's code to the one which achieves highest performances when implemented in hardware,

- design, development, and evaluation of Woolcano; a state-of-the art dynamically reconfigurable application specific instruction-set processor (ASIP) architecture that is based

on a modern FPGA device; a tool flow capable of the just-in-time Woolcano's processor customization.

1.8 Novelty of this Work

The distinguish feature of this thesis that sets it apart from others comes from the fact that it not only presents new methods not studied before but more importantly that it presents a complete and holistic system that delivers a feature that is not found in other works. This feature allows to dynamically customize the processor to the given currently executed workload. To this end, this work spans over several different scientific fields and merges them together into a single coherent system. While in other works, these fields were studied in separation, in our work they are not only coupled together but more importantly they provide with a unique reconfigurable system capable of the just-in-time processor customization.

1.9 Thesis Structure

This thesis consists of three main parts:

- Part I: contains this introduction and motivation and presents the overall view of the developed system.
- Part II: covers with design and implementation details of hardware architecture and software tool flow. This part has four Chapters 3 – 6.
- Part III: provides with an evaluation of the system together with thesis conclusions.

This thesis do not provide with separate related work chapter as well as with the fundamental background regarding the construction of the FPGAs or CAD tool flow for them. Instead, when adequate, the related work is presented at the end of each topic and the informations regarding the FPGA and tool flow can be found in Sass *et al.* [20].

While it might seem that each system component focuses on different example, that is not the case. We decided to use a single application to study the functionality of this complex system. Thus, while the examples are scattered around thesis and sometimes they do not appear in the order as used in the runtime system, the system functionality can be evaluated by tracking and studying these example.

In addition, there are four appendices with supplementary informations. Since this study provides with research on a software JIT ASIP tool flow, in Appendix A, we decided to explain basic terms and processes used in compilers and software translation to readers which are unfamiliar with these topics. In Appendix B, we present a more philosophical introduction to this work, which is based on the *strange loop* paradox that is explained in the Pulitzer Price winner book "Gödel, Escher, Bach" by Douglas Hofstadter [21]. Therefore, we recommend this appendix only to readers that are familiar with this concept and are in general well acquainted with the book. In the last Appendix C other experimentally verified approaches that were used for generating the user-defined custom instructions (UDCIs) with Woolcano compiler are presented.

Chapter 2

System Overview

The main goal of this work is to study the tradeoff between the feasibility and limitation of the just-in-time processor customization system. To this end, we developed a system capable to perform this task and to deliver all necessary results. In this chapter, we provide with a general view of this system and with all necessary informations that allow to comprehend its features.

The system is built from two main parts that are described in detail in the next chapter. These are: a) the dynamically reconfigurable ASIP hardware architecture that we named Woolcano and b) the just-in-time tool flow capable of Woolcano's processor customization, which we named ASIP specialization process (ASIP-SP).

In order to provide with better understanding of the features and role in the system, in this chapter, we differentiate these two components with conventional computer architecture and compiling techniques. These comparisons together with background information allow to provide answers to the following questions:

- What are the major components of the Woolcano architecture and the Woolcano compiler?
- What are the most significant differences between the Woolcano architecture and a conventional computer?
- What are the most significant differences between the Woolcano compiler and a conventional compiler?
- What is the purpose of the developed ASIP-SP?
- What is the difference between dynamic and static software translations as well as between dynamic and static ASIP-SPs?
- What kind of parallelism is supported by our system?
- In what terms are the feasibility and limitations of the developed system expressed?

2.1 Comparison and Differences Between Systems

We start with an overview of the system which is presented in Figure 2.1. The top part of the figure represents the software domain and the bottom part the hardware domain. On the left side of the figure, the conventional computer that consists of the *von Neumann* (vN) architecture and a static compiler is presented whereas on the right side we show our developed

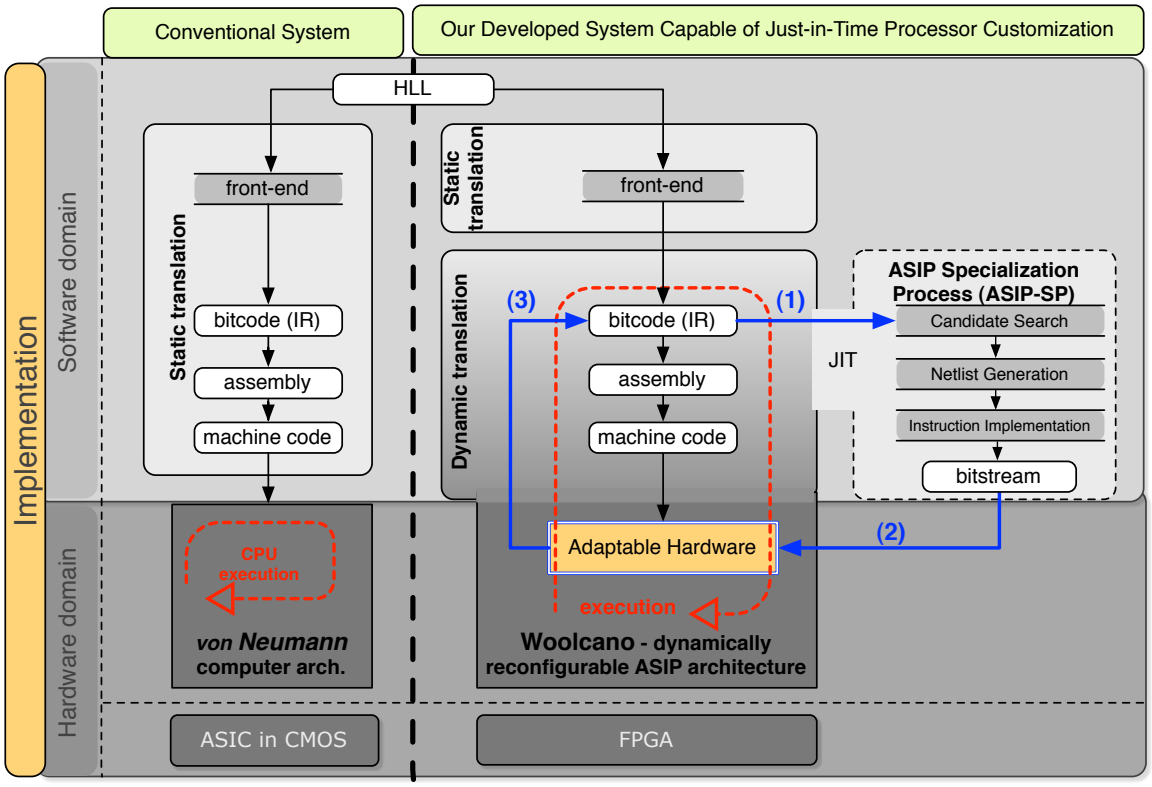

Figure 2.1: Comparison of a mainstream computer with the *von Neumann* (vN) computer architecture and our Woolcano architecture capable of a just-in-time processor customization.

system which is based on the Woolcano architecture and the Woolcano compiler that includes an ASIP specialization process.

One can observe a significant difference between these two systems in both software and hardware domains. First, in the software domain, we can observe that our system compromises not only *static translation* process as found in vN but also the *dynamic translation* processes. Almost the whole software tool flow is based on this dynamic process. Secondly, the ASIP-SP that is responsible for the hardware and software runtime adaptation is found only in our system. Next, in the hardware domain we can observe that, in contrast to the fixed-static vN computer architecture, our Woolcano architecture allows to adapt the hardware accordingly to the ASIP-SP outcomes. Finally, while the vN architecture is implemented as a hardwired application specific integrated circuit (ASIC), our architecture is supported by a reconfigurable computing technology that is based on an FPGA device.

There are two main conceptual differences between these systems that are represented with red and blue arrows. The red arrow indicates the process of computation; that is what is executed on the processor. One can see that in the case of the vN architecture the processor executes the machine code that was generated beforehand with static translation. Thus, two separated and disjointed steps are required in order to perform computation on the vN architecture, the machine code generation and the execution, respectively. This is not the case in our system that uses a dynamic translation where the machine code is generated alongside

with the execution. Therefore, for our architecture only a single step is required, in contrast to the two steps required by the vN architecture.

In addition, the dynamic translation process drives also the ASIP-SP. The usage of the dynamic translation process with ASIP-SP results in a unique feature that is represented with the blue arrow. This arrow indicates that during the program execution the adaptation of hardware occurs concurrently to the machine code execution. This is achieved with the help of three sub-steps that are presented in the discussed figure. In sub-step (1), the ASIP-SP process performs a software-to-hardware translation, that is used to adapt the Woolcano architecture in sub-step (2). Once this task is performed the source code is modified in sub-step (3) and with the help of JIT a new machine code that shifts computation from the processor to newly generated hardware accelerator is generated.

A careful reader may observe that the feature illustrated by the blue arrow provides with a paradox. In every computer, a relation between software and hardware exists where software is executed upon hardware (software-hardware relation). In our system, there exists also another relation represented with blue arrow, where hardware is modified by software (hardware-software relation). These two relations together form the so called *"strange-loop"* that has been studied in details in [21] and that inspired Appendix B.

2.2 Difference in Hardware Architecture

In this section, we introduce three topics: the Woolcano hardware architecture, the difference between the conventional and Woolcano architectures, and the difference between implementing the hardware accelerator with ASIC technology and on an FPGA device.

2.2.1 Overview of Woolcano Hardware Architecture

Woolcano is a state-of-the-art application specific instruction-set processor hardware architecture that is capable of the just-in-time processor customization. Woolcano augments the instruction set architecture of ASIC processor with UDCIs. These UDCIs are implemented in the reprogrammable hardware and are tightly integrated with a processor pipeline. Thus, they act as low-latency application specific hardware accelerators and allow to improve the processing performances and power consumption of a given application.

Number of Supported Applications

Woolcano is not constrained to any subset or number of applications. It is a dynamically reconfigurable ASIP architecture that allows for an online processor customization to any application that is currently executing. Thus, it is unlimited to any number of applications. This achievement is possible due to two features. First, Woolcano is based on FPGA that is a reconfigurable device and that allows to adapt the hardware with hardware accelerators accordingly to demands. Secondly, Woolcano is supported by a specifically developed runtime software environment that makes this task feasible.

Target Applications

It is very rare that applications are fully parallel or fully sequential and usually they are a mix of both. Since a sequential code achieves best processing performances on the ASIC processors and a parallel code on dedicated hardware accelerators, the heterogenous architectures, like our Woolcano that consists of both resources seem to be the best fit for these mixed applications. Thus, while Woolcano supports any kind of application, it targets the ones that consist of a combination of sequential and parallel codes.

Support for Legacy Models

Woolcano is an ASIP architecture that includes both processor and hardware accelerator reconfigurable resources. Since the processor is used to execute the application no changes in the source code of the application, programming language, or the software tool flow are necessary from the accelerator point of view. The development of the application is performed with familiar tool flows. The hardware accelerators are created transparently and then are automatically utilized without any manual interferences or interactions. Therefore, Woolcano allows to customize the processor for any legacy software models amenable for processor execution.

Level of Heterogeneity

Woolcano is a heterogenous system that uses two resources to perform computation: processor and hardware accelerators. The heterogeneity in this system is performed on the ISA level where the hardware accelerator supports UDCI, a subset of ISA. Thus, Woolcano does not require any changes in other areas of heterogeneity such as application binary interfaces, application programming interfaces, programming language, or memory interfaces.

Low Latency Interconnection

In Woolcano, the hardware accelerator has a direct connection to the processors pipeline. This results in a single clock cycle communication latency between the processor and the accelerator. Thus, in contrast to the bus attached hardware accelerators that consume several hundreds cycles for communication purposes, this results in a low latency interconnection.

Fine Grained Architecture

The hardware accelerators can be constructed from pre-implemented coarse grained or fine grained elements. The construction of the accelerator from coarse grained elements is faster than from the fine grained ones. This is caused by the lower level of details that needs to be investigated during the construction process. On the other hand, the fine grained elements allow for better expression to the computational and communication patterns. Thus, it is possible to achieve higher performances than with the coarse grained architectures. The Woolcano's hardware accelerators are constructed from fine grained elements and thus naturally, it is a fine grained architecture.

2.2.2 Conventional vs. Woolcano Hardware Architecture

In order to perform computations, conventional computers use commercially available processors. However, as explained in the introduction, at the end of the frequency scaling era their performance improvements are not increasing as they tend to in the past. Solutions that address this problem augment these processor with reprogrammable resources.

The reprogrammable systems, such as our Woolcano, include not only ASIC processors but also allow to adjust the surrounding hardware and create customized hardware accelerators. These accelerators target the computational and communicational patterns found in a given application. Thus, reprogrammable systems target both patterns and allow to express them in both domains, the software and the hardware one. In contrast, the conventional systems allow to utilize only computational pattern and only in the software domain.

This software-to-hardware shift found in reprogrammable systems, allows to change the *control flow* computing paradigm found in conventional systems to the *data-flow* [22, 23] or even *reduction* [24] one. In consequence, this results in large processing performances and power consumption improvements.

2.2.3 Technology Difference: ASIC vs. FPGA

Every UDCI is represented by a digital circuit that acts as an application specific hardware accelerator. As illustrated in Figure 2.2, these digital circuits can be implemented either in ASIC or in FPGA. Both technologies are based on the complementary metal-oxide-semiconductor (CMOS) technology that is a major technology used for constructing processors or memories. Bellow we provide with differences between these technologies.

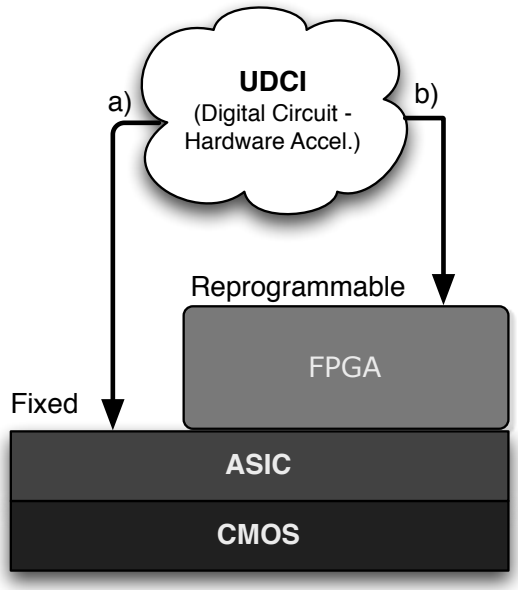

Figure 2.2: Difference in UDCI when implemented in ASIC and FPGA.

Difference in Hardware Reprogrammability (Reconfiguration)

The ASIC circuit once implemented in CMOS technology cannot be altered at a later stage – it is fixed and it does not allow for reprogrammability. This is a case for a processor used in vN architecture found in Figure 2.1 and case a) in Figure 2.2.

The reprogrammability of the hardware is possible in FPGA devices. From the abstract point of view, FPGA is an ASIC that is implemented in CMOS and that on top of the device adds a special *reprogrammable layer*. This layer is constructed from fine grained CMOS elements and it provides with two coarse grained resources: a) reprogrammable logic and b) reprogrammable switches. The logic allows to establish functionality of the digital circuit and switches to wire the logic together. Both resources are flexible and allow for digital circuit to be *"loaded"* into the hardware.

Support of JIT processor customization

Since ASIC is a fixed technology that does not allow for a hardware reconfiguration it cannot be used for an online JIT processor customization. Moreover, due to manufacturing processes the hardware accelerators implemented in ASIC are dedicated only to a single application. These obstacles are not found in FPGAs that allow for hardware reconfiguration with many different hardware accelerators. Thus, they support an unlimited number of applications and are a perfect match for the online JIT processor customization.

Difference in Circuit Characteristics

While any digital circuit can be implemented in FPGA and in ASIC with equivalent functionality, the granularity of used components and elements to perform this task is different. The circuit implemented as ASIC in CMOS will be constructed from fine grained elements. The same circuit when mapped to FPGA will be implemented with coarse grained resources (a+b) found in reprogrammable layer. In consequence, both circuits will have different characteristics when implemented with either technology. The circuit when implemented in FPGA will result in:

- lower maximum operating frequency (lower performances by $3.2\times$),
- higher resource area ($40\times$),
- higher power consumption ($12\times$)

comparing to the equivalent ASIC circuit [25]. While it might sound that the usage of FPGAs is doomed in favor of ASICs – this is not the case. All these technological disadvantages found in FPGA are occupied by the reprogrammability feature that is required in the future computer hardware architectures.

Difference in Usage

The reprogrammability makes FPGA a more accessible device than the ASIC technology. In consequence, due to practical reasons such as:

- high costs (reaching millions of dollars),
- long development and manufacturing process (often counted in years),
- requirements for large product volumes (counted in tens of thousands)

the development of ASIC circuits [26, 27] is rarely used in contrast to FPGA [28].

2.3 Difference in Software Tool Flow

The background informations regarding the compiler construction, the differences between static and dynamic translation techniques, interpretation, the just-in-time compilation, and used terminology are presented in Appendix A.

The process of translation in both systems is represented in Figure 2.1 with black arrows and it is in charge of a movement between the software stack - abstraction layers. In both systems, it starts from the high level language (HLL) and ends at the machine code that is directly fed into the processor for the execution. The most significant differences between a conventional compiler and the Woolcano compiler rely on two facts. First, our Woolcano compiler is a dynamic, often known as runtime or online system whereas the conventional one is a static system. The difference between them is explained in more detail in Appendix A.8. Secondly, our tool flow was extended with an ASIP-SP process that was specifically developed for the Woolcano architecture. This section briefly presents that process.

2.3.1 ASIP Specialization Process

The ASIP-SP tool flow is in charge of a) hardware and b) software runtime adaptation. The outcome of this tool flow is represented with the blue arrow in Figure 2.1 where sub-steps (1) and (2) are associated with task a) and sub-step (3) with task b). The ASIP-SP was designed and developed specifically for our dynamically reconfigurable Woolcano architecture and thus, it is not available for the vN system.

Hardware Runtime Adaptation (HRA)

The hardware runtime adaptation consists of a data path synthesis tool that is amenable for a *software-to-hardware* translation. The distinctive feature of our HRA is that it was designed in a way that not only allows to maximize a given circuit metric such as performance, area, or power consumption but also to minimize the overall runtime. In other words, this tool flow tries to achieve best results in the shortest runtime; the key feature for the just-in-time system. The details of this behavior are presented in Chapter 5.

Software Runtime Adaptation (SRA)

Once the bitstreams are generated with the HRA process and the Woolcano's hardware is reconfigured, the SRA process comes into action. The main aim of this process is to shift the control flow (computation) from the processor into a newly created and loaded hardware accelerator. To this end, the interpreted byte code is modified. This task, known from static systems as the *binary translation*, is performed with a developed *communicational pass* and with the help of the virtual machine features. It is studied in detail in Chapter 6.

2.3.2 Difference in Static and. Dynamic ASIP Specialization Process

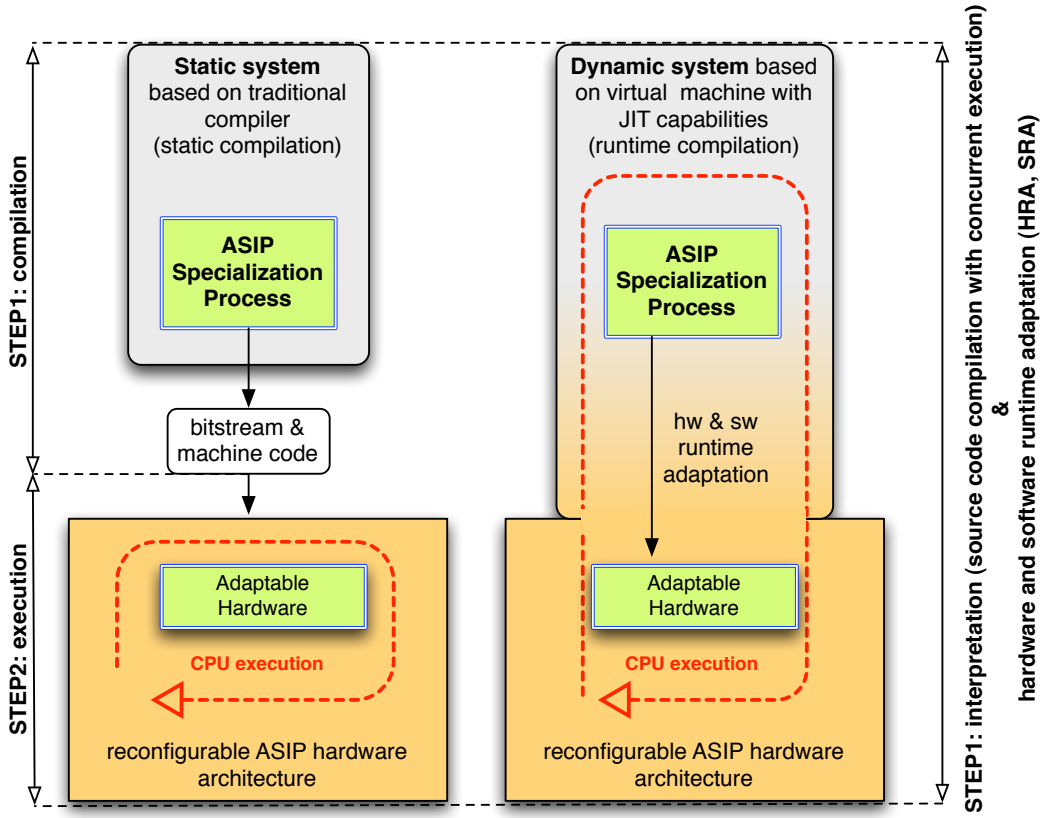

Figure 2.3: Overview of the ASIP specialization process for the conventional static and dynamic systems.

In this section, we highlight the differences between the static and dynamic (runtime) ASIP-SP tool flows. The difference between the static and dynamic translations is explained in Appendix A.8.

Figure 2.3 illustrates the difference between a conventional static ASIP-SP and a runtime system with a just-in-time ASIP-SP support. As previously described, ASIP-SP consists of a) hardware and b) software runtime adaptation processes. So far ASIP-SP has been applied almost exclusively to static systems, where steps a) and b) occur offline before the application is executed.

The main advantages of executing ASIP-SP as a part of the runtime system are:

- the system can optimize its operation by reconfiguring the instruction set architecture of the processor and by changing the code at runtime, which is fundamentally more powerful than a static ASIP specialization.
- the system can collect execution time, profiling, and machine level information in order to identify the code sections that are actually performance limiting at runtime; these sections are ideal candidates to be accelerated with UDCIs.
- the virtual machine has the capability to execute various dynamic optimizations like

hotspot detection, alias analysis, or branch prediction to further optimize the performance.
- the dependencies between variables and corresponding memory layout can be obtained, which simplifies the task of the hardware-software partitioning between the processor and the hardware accelerator.

2.4 The Level of Supported Parallelism

During the hardware runtime adaptation process, the Woolcano compiler translates the software to the hardware domain that is an exhaustive parallel resource. Thus, the selected source code indicated for a transformation into the hardware accelerator has to be rich in parallelism. Only then the hardware will be fully utilized and will provide with significant application performance and power improvements.

The Woolcano compiler aims at selecting a code that is rich in bit-level and instruction-level parallelism. For these levels of parallelism special algorithms exist that allow to automatically find and extract parallelism from the application. Until now such algorithms do not exist for the task-level parallelism and it is unsure if they will ever exist for imperative programming languages. Thus, the task-level parallelism has to be explicitly encoded by the programmer with the help of special frameworks.

The level of parallelism that is supported by our system is constrained by the availability of these algorithms. Since Woolcano is a runtime system and is dependent on these algorithms the bit-level and instruction-level parallelisms are currently supported.

2.5 Feasibility vs. Limitations

The system developed in this thesis allows to achieve high performances by moving parts of the application into dedicated hardware accelerators. These accelerators are created during the application runtime. Since the process of the hardware development is notoriously slow this limits the feasibility and applicability of this approach. Therefore, while the feasibility of this approach is expressed in terms of performance improvements, the limitations are expressed in terms of time required to generate these improvements.

2.6 Related Work

This work is built on research in three areas: reconfigurable ASIP architectures, ISE algorithms, and just-in-time compilation, which have mostly been studied in separation in related works. This work integrates all of these approaches into a consistent methodology and a tool flow.

ASIP Hardware Architectures

From the hardware perspective, this work does not target the static but reconfigurable ASIP architectures such as our Woolcano architecture [29] or comparable architectures like CHI-

MAERA [30], PRISC [31], or PRISM [32]. These architectures provide programmable functional units that in order to implement arbitrary UDCIs can be dynamically reconfigured during the runtime. The survey on reconfigurable instruction set processors published in year 2000 is available in Barat *et al.* [33].

ISE Algorithms

Research in the field of ISE algorithms for the ASIP architectures is extensive; a recent survey can be found in Galuzzi et al. [34]. However, the leading state-of-the-art algorithms for this purpose have an exponential algorithmic complexity which is prohibitive when targeting large applications and when the runtime of the customization process is a concern as it is the case for our just-in-time processor customization system. This work leverages our preliminary work [35] in which new heuristics for effective ISEs search space pruning were studied. It was shown that these methods can reduce the runtime of ISE algorithms by two orders of magnitude.

Binary Translation

The goal of this work is to translate software binaries on-the-fly into optimized binaries that use application-specific custom instructions. The binary translation is used, for example, to translate between different instruction sets in an efficient way and has been used, for example, in Digital's FX!32 product for translating X86 code to the Alpha ISA [36]. The binary translation has also been used for cases where the source and target ISAs are identical with the objective to create a binary with a higher degree of optimization [37,38]. These are found especially in the Java JIT compiler, Microsoft .NET MSIL Framework, or Virtual Machines [39].

Beck et al. [40] presented work on binary translation of Java programs for a custom reconfigurable ASIP architecture with coarse–grained reconfigurable data-path units. They show that for a set of small benchmarks an average speedup of $4.6\times$ and power reduction of $10.9\times$ can be achieved. The identification and synthesis of new instructions occur at runtime; however, the paper does not specify what methods are used for the instruction identification and what overheads arise from the instruction synthesis.

This work is conceptually similar to these approaches as it also does not translate between different instruction sets, but optimizes binaries to use specific UDCIs in a reconfigurable ASIP. This kind of binary translation has hardly been studied so far. One comparable research effort is the WARP project [41]. The WARP processor is a custom system-on-chip comprising a simple reconfigurable array, an ARM7 processor core, and additional cores for application profiling and place-and-route. This work differs from WARP in several ways. The main difference is that we target a reconfigurable ASIP with programmable processing units in the CPU's data-path whereas WARP uses a bus-attached FPGA co-processor that is more loosely coupled to the processor. Hence, this work allows to offload operations at the instruction level where WARP needs to offload whole loops to the accelerators in order to cope with longer communication delays. Further, WARP operates on the machine-code level and reconstructs the program's higher-level structure with decompilation while this work relies on a higher-level information that is present in the virtual machine. Finally, WARP assumes a custom system-on-chip and this work targets commercially available standard FPGAs.

Part II

Design and Implementation

Chapter 3

Woolcano Hardware Architecture

This section presents the Woolcano dynamically reconfigurable ASIP processor hardware architecture. This architecture is based on the Xilinx Virtex-4FX and leverages its Auxiliary Processing Unit (APU) and partial reconfiguration capabilities to provide dynamically reconfigurable functional units (RFUs) for implementing UDCIs. Woolcano is a state-of-the art dynamically reconfigurable application-specific instruction set processor. It consists of a fine grained RFUs which are tightly coupled to the data-path of the processor and which use the same general purpose register file as the main processor. While previous research on processors with RFUs has been conducted predominantly with simulation, the Woolcano architecture with its associated tool flow allows for exploring dynamic instruction set extension in practice.

3.1 Introduction

ASIPs [23] augment the instruction set of a general processor with additional UDCI for accelerating specific application classes. These UDCI are tightly integrated into the processor pipeline and typically perform purely combinational operations on values taken from the register file. This instruction set extension approach is popular for performance critical embedded systems. It allows to improve the performance and power consumption of the most important application kernels while the architecture remains a flexible general-purpose processor as opposed to a fixed-function ASIC.

Dynamic Instruction Set Extension Architecture

While ASIPs are typically implemented in silicon, they just as well can be mapped to FPGAs. Some ASIP design tools explicitly target FPGAs as emulation platforms [42, 43]. However, instruction set extension approaches generally assume that the UDCI are selected and fixed during the design phase and are not changed at runtime. When implemented in reconfigurable logic the assumption of static UDCI is, however, overly restrictive and does prevent us to capitalize on the benefit of dynamic reconfiguration.

The Woolcano extends the idea of static instruction extensions and present the processor architecture for dynamic instruction set extension. This architecture is comprised of a static part that implements the fixed instructions set and an interface for application-specific UDCI,

which can be replaced at runtime using partial reconfiguration.

This capability enables novel modes of operation, such as configuring UDCI when loading a new application, replacing UDCI during the execution of the application, or even generating new UDCI on-the-fly. In contrast to related work on new processors architectures with reconfigurable functional units, which have been evaluated primarily with simulation, this approach leverages the commercially available Xilinx Virtex-FX FPGA architecture.

Outline

In following sections, the Woolcano reconfigurable processor architecture is introduced. The architecture bases on the PowerPC 405 processor core that is embedded as a hard-core in the Xilinx Virtex-4 FX FPGA. For interfacing the UDCI the PowerPC's APU interface is used, which is discussed in section 3.2. For changing the implementation of a custom instruction at runtime the partial reconfiguration methodology is used, which is briefly discussed in section 3.3. Finally, the Woolcano architecture is presented in section 3.4.

3.2 APU Interface

Starting with the Virtex-4 FX family, Xilinx has introduced the Auxiliary Processor Unit (APU) in their Virtex product line with embedded PowerPC cores. The APU provides a direct low-latency high-bandwidth interface between the pipeline of the PowerPC 405 core and the reconfigurable FPGA fabric. The APU interface allows for attaching user-created, high-performance hardware accelerators, denoted as fabric co-processor modules (FCM). Figure 3.1 illustrates how the APU is integrated into the processor pipeline.

For accessing the FCM, the PowerPC instruction set is extended with three additional instruction classes that are decoded by the APU but executed by the FCM:

a. **User-Defined Custom Instructions:** Provide a flexible way of adding hardware accelerator to the processor. To this end, the instruction set provides eight reserved opcodes (`udi0fcm-udi7fcm`). The behavior of each UDCI can be controlled through a corresponding configuration register in the APU. The instructions can be configured to read up to two integer operands from the register file. Optionally, each instruction can write a result back to the register file and can set carry and overflow flags. Instructions that write a result are called non-autonomous and cause the processor to wait for a programmable delay until the instruction has completed. Instructions without results (autonomous) complete immediately without stalling the processor. Further, UDCIs may disable interrupts during execution and restrict execution to the privileged mode of the processor.

b. **Floating-Point Instructions:** In addition to UDCIs that can be used for arbitrary functions, the APU provides specific support for decoding standard PowerPC floating-point operations. Since the PowerPC core of the Virtex-4 FX does not provide an integrated hardware floating-point unit, floating-point arithmetic must be emulated in software, which causes a large performance penalty. For accelerating floating-point intensive applications, a floating-point unit implemented in the FPGA logic can be attached as a

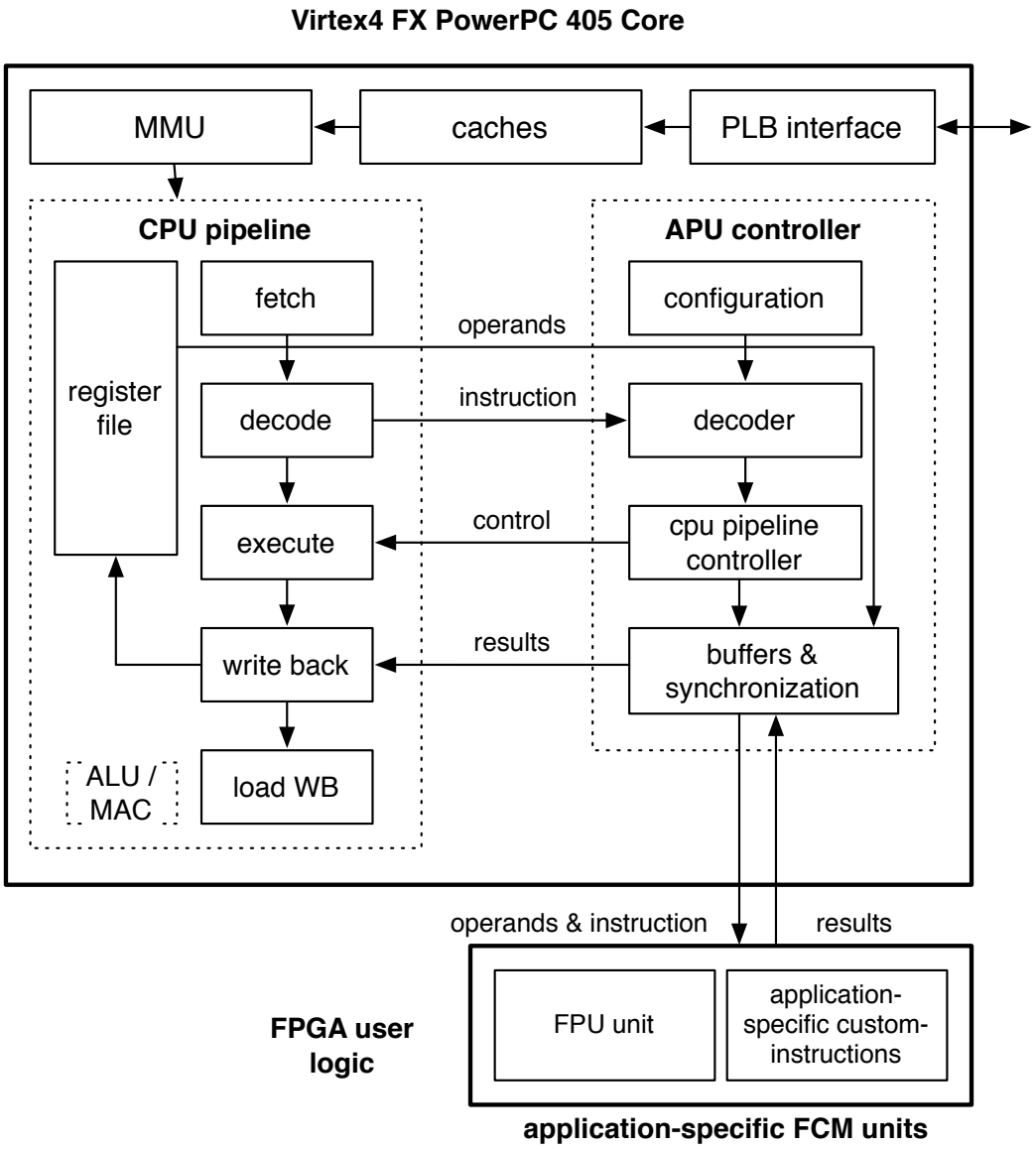

Figure 3.1: APU architecture

FCM module. Xilinx provides suitable FPU cores as a part of the embedded development kit (EDK) tool suite.

c. **APU load/store instructions:** Finally, APU load and store operations allow for transferring data blocks of up to four words between the memory and an FCM unit. This is the fastest way to transfer large data blocks between memory and the FCM.

The Woolcano architecture is build on the APU's UDCIs. This allows for a very tight integration of hardware accelerators with the processor pipeline and provides a much lower delay than memory mapped or bus-attached co-processors.

3.3 Dynamic Partial Reconfiguration

Dynamical partial reconfiguration [44] is an FPGA configuration method supported by the Xilinx Virtex FPGA family. It allows to change specific regions of the fabric during runtime, while the other parts of the hardware design continue to run unaffectedly.

Tool Flow

Partial reconfiguration is not supported by the standard Xilinx implementation tools but requires a modified version of the tools. These tools are available from Xilinx through an early access program (EAPR).

Preparation Steps

For supporting partial reconfiguration the hardware design has to be prepared beforehand. First, the design has to be partitioned into a static part and parts that can be reconfigured during runtime. These parts are denoted as partial reconfiguration regions (PRR). When synthesizing the static part of the design, specific placement constraints [45] are applied. These prevent static logic to be placed in the PRR.

Circuits that shall be loaded into the PRR are named partial reconfiguration modules (PRM). They are created in separation from the static design. During this step the same constrains are used in order to place them in PRRs. For ensuring proper communication and synchronization of the static design and the dynamically reconfigurable module two mechanisms have to be used: First, all clocks by the PRM have to be accessed via global clock buffers (BUFG). Second, data signals have to be passed via uni-directional communication channels denoted as Bus Macros (BMs), see Figures 3.4 and 3.5.

Partial Reconfiguration Bitstream Files

The result from this tool flow is a number of configuration bitstreams: a bitstream for the static part of the system and one bitstream for each PRM. The bitstream for the static part of the system is used to initially configure the FPGA at startup. During runtime, the contents of the PRR can be exchanged by loading the bitstream of a PRM either using the external configuration port of the device, or via the internal configuration access port (ICAP) that can be attached to the PowerPC core.

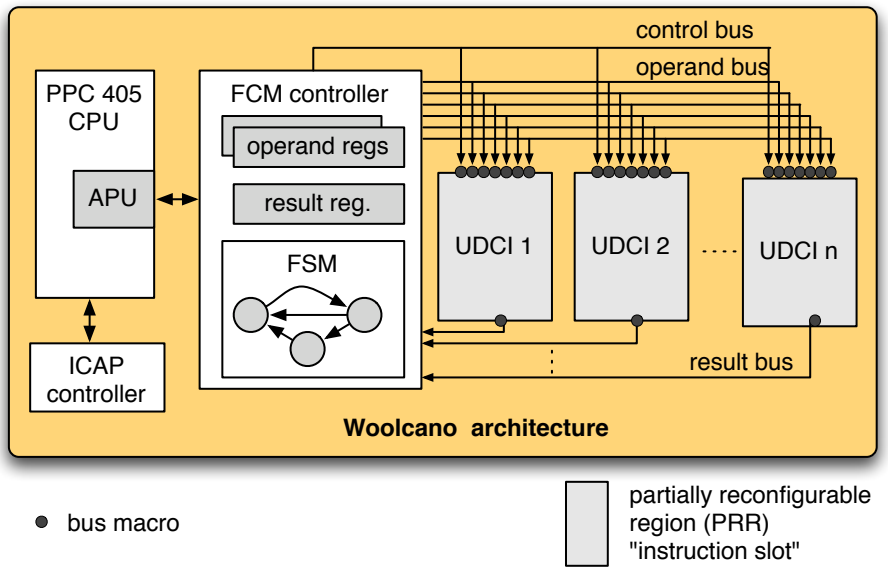

Figure 3.2: Schematic of the *Woolcano* reconfigurable processor architecture capable of just-in-time processor customization with instruction set extensions methodology.

3.4 Woolcano Processor Architecture

Figure 3.2 shows a schematic of the Woolcano reconfigurable processor architecture for just-in-time processor customization. The main components of the architecture are the PowerPC core, the ICAP controller, the FCM controller, and the partial reconfiguration regions for implementing UDCI which we denote also as *instruction slots*. The FCM controller implements the interface between the processor core and the UDCI. It forwards the inputs to the instruction slots via the operand bus and, after the custom instruction has finished computing, transfers the output back to the processor via the result bus. The control bus is used for sending control information to the UDCI, e. g., activation or abort signals.

UDCIs

The Woolcano processor architecture uses the UDCI of the PowerPC 405 architecture as follows: `udi0fcm` is used for transferring two 32bit data words to the operand registers in the FCM controller. `udi0fcm` is implemented as an autonomous instruction which immediately transfers control back to the processor after execution. `udi1fcm`–`udi7fcm` also transfers two words to the operand register file, but additionally trigger the execution of the custom instruction. These instructions are non-autonomous, i. e., they stall the execution of the processor until the result of the custom instruction is available and can be returned to the processor.

Bus Macros

Bus macros are placed at the interface between the instruction slot and the operand, control, and result busses for enabling partial reconfiguration of the instruction slots. The instruction

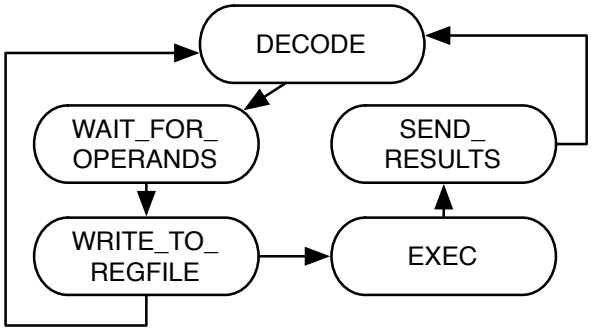

Figure 3.3: Finite State Machine of FCM Controller

slots can be reconfigured via ICAP or the external configuration port of the FPGA.

FCM Controller

As the FCM controller is a central component of the Woolcano architecture its implementation is discussed in more details. The FCM controller connects UDCI slots to the APU interface of the PowerPC 405 core. Its main function is to implement the APU protocol for transferring data and for dispatching instructions. The architectural constraints of the APU allow only for two input and one output operands to the UDCI. This restriction limits the amount of data a UDCI can operate on, which in turn limits the achievable speedup. To circumvent this limitation, the FCM core implements internal operand registers for supplying the UDCI with additional operands.

Finite State Machine

The FCM controller is implemented as a finite state machine, see Figure 3.3. Starting from the initial state DECODE, the controller waits for the notification from the APU that a predefined UDCI has been decoded. If the decoded instruction is a data transfer instruction, the controller receives the data in the WAIT_FOR_OPERANDS state, stores the received operands in the input operand registers in state WRITE_TO_REGFILE, and finally returns to DECODE again. If the decoded instruction is a custom instruction dispatch, operands are received, written to the register file, and the EXEC state is entered. In this state, the input operand registers are placed on the operand bus and the appropriate custom instruction is activated via the control bus. Finally, the results are sent to the APU in the SEND_RESULTS state. In any state except EXEC the controller can be interrupted by the APU, which resets the controller back to the DECODE state.

UDCI Slots

The number of UDCI slots as well as the their input and output operands are compile-time configurable architecture parameters denoted as $C_{\max}$, $in_{\max}$, and $out_{\max}$, respectively. Since the all inputs and outputs to the instruction slots must be fed through Xilinx bus macros,

the size and geometric placement options of the bus macros limits the number of the input operands and results.

3.5 Noteworthy Implementation Details

In the following the most important implementation details concerning the dynamic partial reconfiguration are presented. Figure 3.4 contains the excerpt from the user constrain file (UCF) where locations for three different resources including the clock, bus macros, and the PRM regions are manually setup on a physical chip. The first resource corresponds to the global clock which is wired to the PRM with the help of a global clock buffer (BUFGCTRL). The BUFGCTRL ensures that the clock will be available in the PRM region with the low-skew dedicated routing facilities. This is probably the most important entry in the UCF file since all synchronous circuits depend on it.

The second entry ensures proper communication with the PRM for the data and control signals. To this end, the 8-bit BMs are placed on the edges of the PRM are used. Since the PowerPC is a 32-bit architecture 4 BMs are used in order to send 32-bit data. The `ra` and `rb` suffices represent the input data channels to the PRM which are used to transfer the input data operands. The `RES` suffices indicate the computational result shifted back to the FCM. In addition, there are two BMs placed for the control signals going in both ways between the PRM and FCM.

The last entry in the UCF corresponds to the location of the PRM region. The PRM region is built from the slices, RAM modules, DSP hardware macros, and FIFOs. The visual representation of the PRM and the different resources can be found in Figure 3.5.

3.6 Work Related to Woolcano Hardware Architecture

Architectures supporting reconfigurable UDCI, e. g., PRISM [46], PRISC [31], DISC [47], CHIMAERA [30], or OneChip [48] have been extensively studied in the 1990's. The survey published in year 2000 is available in Barat *et al.* [33]. The prime objective of these projects was to evaluate the feasibility and potential of reconfigurable functional units connected to data path of the processor.

Simulation Based Approach

Simulation-based design space exploration has been used to study speedups, the performance of processor/RFU interfaces, the effect of instruction encoding, etc. None of these architectures has been implemented in VLSI, also limitations in the logic capacity of early FPGA architecture has prevented building single-chip FPGA prototypes. Only recently a processor with reconfigurable functional units has been commercialized with the Stretch S5 architecture [49].

```
# tunnel global clock to UDCI slot via BUFG
INST "global_clock" LOC = BUFGCTRL_X0Y29;

# place busmacros at the boundry between reconfigurable UDCI slot and static part
INST "bm_fcm2sl0" LOC = SLICE_X16Y46;
INST "bm_sl0fcm_RES3" LOC = SLICE_X16Y20;
INST "bm_sl0fcm_RES2" LOC = SLICE_X16Y22;
INST "bm_sl0fcm_RES1" LOC = SLICE_X16Y24;
INST "bm_global2sl0_ra_0" LOC = SLICE_X16Y44;
INST "bm_global2sl0_ra_1" LOC = SLICE_X16Y42;
INST "bm_global2sl0_ra_2" LOC = SLICE_X16Y40;
INST "bm_global2sl0_ra_3" LOC = SLICE_X16Y38;
INST "bm_global2sl0_rb_0" LOC = SLICE_X16Y36;
INST "bm_global2sl0_rb_1" LOC = SLICE_X16Y34;
INST "bm_global2sl0_rb_2" LOC = SLICE_X16Y32;
INST "bm_global2sl0_rb_3" LOC = SLICE_X16Y30;
INST "bm_sl0fcm" LOC = SLICE_X16Y28;
INST "bm_sl0fcm_RES0" LOC = SLICE_X16Y26;

# define the placement of the UDCI slot region
AREA_GROUP "pblock_alu_fp_0" RANGE=SLICE_X18Y0:SLICE_X67Y47;
AREA_GROUP "pblock_alu_fp_0" RANGE=RAMB16_X1Y0:RAMB16_X4Y5;
AREA_GROUP "pblock_alu_fp_0" RANGE=DSP48_X0Y0:DSP48_X0Y11;
AREA_GROUP "pblock_alu_fp_0" RANGE=FIFO16_X1Y0:FIFO16_X4Y5;
AREA_GROUP "pblock_alu_fp_0" MODE=RECONFIG;
# AREA_GROUP "pblock_alu_fp_0" PLACE=CLOSED;
```

Figure 3.4: Excerpt from User Constrains File (UCF) which defines the placement of hardware components.

Bus Attached Co-processors

Later, research in reconfigurable processor architectures has focused primarily on hybrid processors with reconfigurable co-processors, e. g., GARP [50], PipeRench [51], or MorphoSys [52]. A number of these co-processor architectures have been also implemented in VLSI. These approaches are orthogonal to processor customization with instruction set extensions. Instead of exploiting customized parallel operations on the instruction level, these co-processor approaches map complete application kernels to the co-processor. Due to the higher latency of the processor/co-processor communication, acceleration at the instruction level is difficult to achieve with these architectures.

APU Interface

The Woolcano approach uses the APU interface of the PowerPC core in Xilinx Virtex-4FX FPGAs. This interface allows for attaching a floating-point unit and for extending the instruction set with static UDCIs. Despite the availability of UDCIs since Virtex-4FX and extensive research in processor customization with instruction set extensions for ASIPs, the UDCI have been hardly used in scientific research. The most common use of the APU is to attach the Xil-

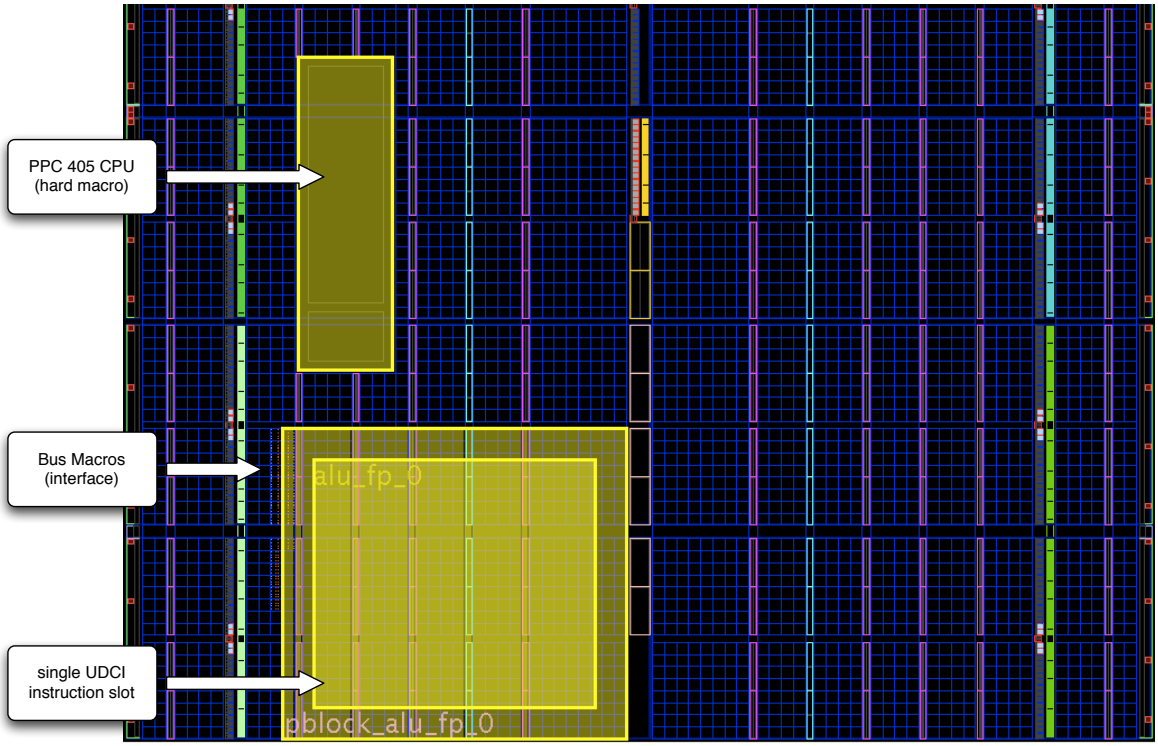

Figure 3.5: Floor Plan excerpt generated for the UCF found in Figure 3.4, targeting Woolcano prototype with single UDCI based on Xilinx ML403 starter kit featuring V4FX12 FPGA Device.

inx FPU core, e. g., [53, 54]. To the best of our knowledge, only the work of [55, 56] and [57] studied the use of the APU to attach a hardware accelerator that is not a general FPU.

Chapter 4

Woolcano Compiler

The Woolcano hardware architecture presented in the previous chapter is a dynamically reconfigurable ASIP architecture. It allows to dynamically customize the processor with instruction set extensions and to adapt the ISA to the given application during the runtime. In order to perform this task the special compiler is required. We name it Woolcano compiler.

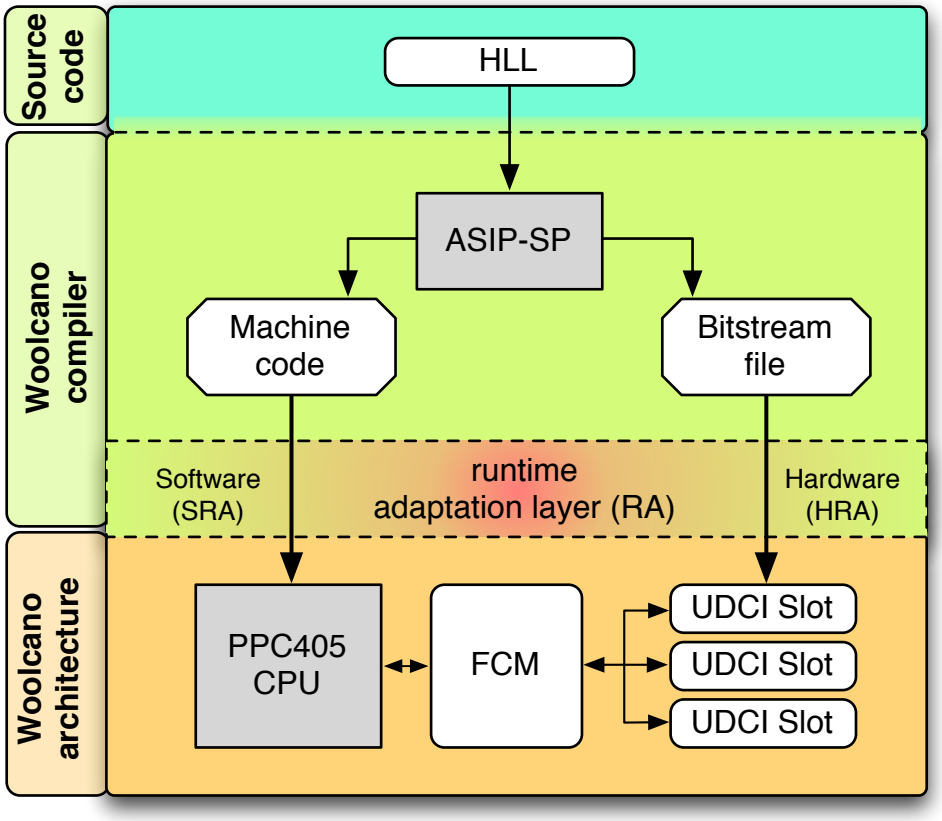

Figure 4.1: Woolcano Compiler

The concept of the Woolcano compiler is presented in Figure 4.1. It provides with a)

runtime adaptation layer and b) ASIP-SP process that is used to generate the machine code and the bitstream file. The combination of both (a+b) results in a compiler capable of software and hardware runtime adaptation.

4.1 Runtime Adaptation Layer

The runtime adaptation layer is a mechanism that allows to execute given compiler optimization, analysis or transformation during the runtime of the application. Thus, this feature is not available in ahead-of-time static compilers and is only found in interpreters and virtual machines. These machines compile and execute the code in the same process and in consequence have access to values of pointers and variables, statistics about executed code, and machine informations. These informations enable a new set of adaptive optimizations that are not found in static compilers. As the name suggests, these are adaptive optimizations which allow to utilize these informations in order to adapt the code and to improve the processing performances and the power consumption.

The Woolcano compiler itself does not implement the runtime adaptation layer. Since our compiler is based on Low Level Virtual Machine (LLVM) compiler framework the runtime adaptation layer was inherited. The details about the design and implementation of the LLVM and its features can be found in Appendix A.

4.2 ASIP Specialization Process

The ASIP-SP process is compiler's transformation pass. As previously stated and illustrated in Figure 4.1 it was designed to perform two tasks that are studied in two following chapters.

- First task is to generate proper machine code that allows to shift the control flow from the processor to the hardware accelerator and to provide with bidirectional communication channel. This channel is used for sending the data from processor to the accelerator and for receiving results. To perform this work, the machine code uses `udi0fcm-udi7fcm` instructions described in Section 3.4. The engineering challenge involved in this design and implementation is to develop a compiler able to generate correct and coherent code with these instructions.
- Second task is to generate partially reconfigurable bitstream files with hardware accelerators. These bitstream files are used to reconfigure the reprogrammable hardware and change the functionality of the UDCI slots found in the Woolcano architecture.

4.2.1 Software and Hardware Runtime Adaptation

The ASIP-SP is a compiler transformation pass that was designed and implemented as adaptive one; that is to work under the runtime adaptation layer. Thus, it can be executed during application's runtime and utilize various available online informations. To this end, the first task of ASIP-SP that adjust the machine code is referred to as software runtime adaptation. Consequently, the second task is named hardware runtime adaptation.

While the adaptive transformations found in state-of-the art runtime systems is always constrained to the software domain, to best of our knowledge our hardware runtime adaptation layer is the first and only transformation that opens doors towards adaptation in the hardware domain.

4.2.2 Overview of Implementation Details

The ASIP-SP is a complex process that is constructed from several developed and commercially available tools. While the full design and implementation details are covered in next two chapters, in Figure 4.2 we briefly illustrate major components of ASIP-SP.

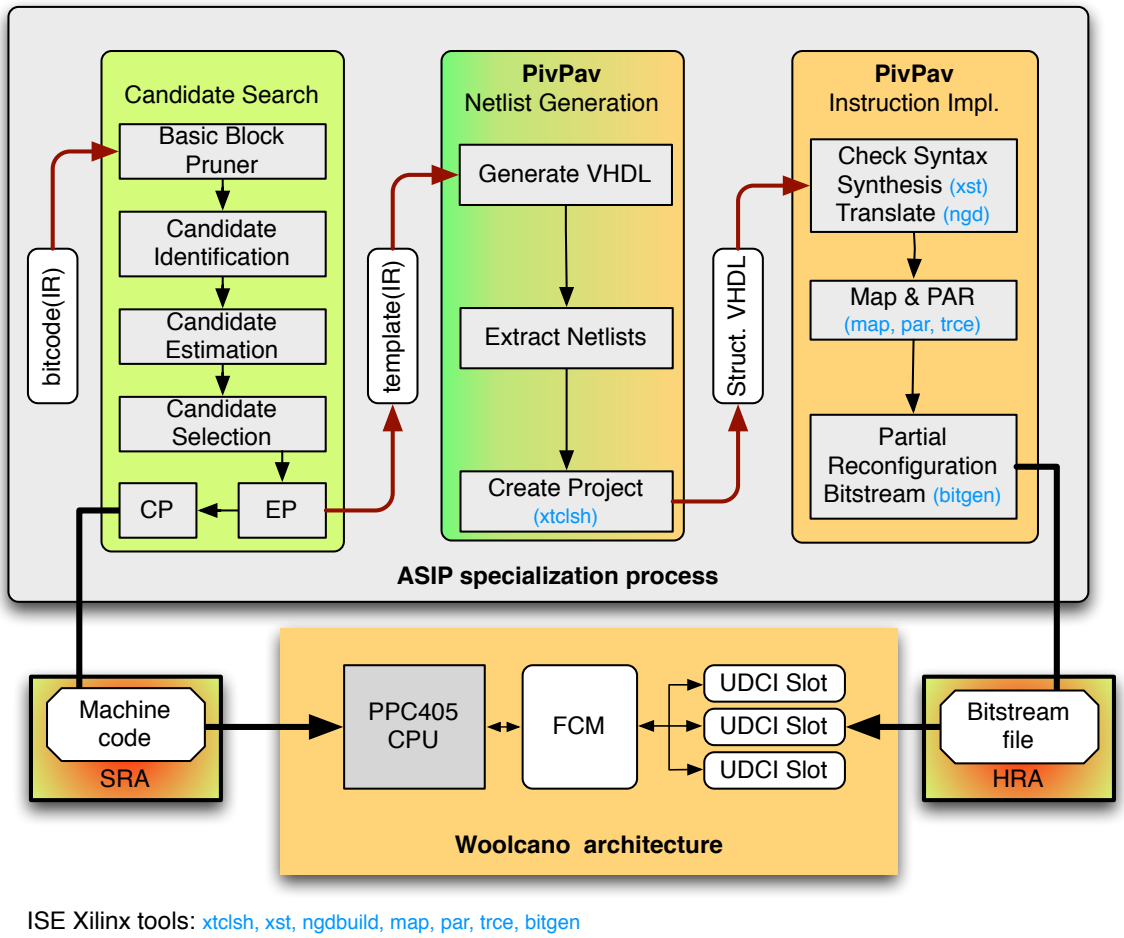

Figure 4.2: Implementation details of the ASIP specialization process.

The ASIP-SP process comprises three main phases: *Candidate Search*, *Netlist Generation*, and *Instruction Implementation* that aim to achieve different goals.

Candidate Search

During the first goal, *candidate search*, the *candidate identification* process is executed that aims at finding software regions in the application's bitcode that are rich in instruction level parallelism (ILP). To this end, the ASIP-SP compromises a set of instruction set extension algorithms. These algorithms have exponential complexity that result in long runtimes; they last from a few seconds up to days. Since these long runtimes limit the feasibility of any runtime system they are preceded with *basic block pruning* heuristic algorithms. These algorithms are able to reduce the ISE algorithm's search space to only a few basic blocks that will bring the best performance improvements. In consequence, the runtime of the whole tool flow is significantly reduced. Once a set of candidates is found, only the best ones are selected within the *candidate selection* processes. This process is depended on estimation metrics generated by *candidate estimation* process. These metrics allow to compare the performances of the candidate when executed on processor and as a hardware accelerator. Once these tasks are performed the *communication pass* (CP) and *extraction pass* (EP) come into action. These passes are responsible for the outcomes of the software runtime adaptation. The EP extracts the selected candidates code into separate function whereas the CP adapts the machine code and guarantees proper communication between processor and hardware accelerator. The EP is studied in Section 5.7 whereas the CP in Section 6.4 where the case study is provided in Figure 6.2.

Netlist Generation

The second goal, *netlists generation*, transforms spotted temporal software regions rich in instruction level parallelism into spacial hardware description circuit accelerators. This task is performed with the help of the PivPav tool that we developed specifically for this purpose. PivPav uses data path generator in order to *generate VHDL* structural code. This tool iterates over template's data path and translates every instruction to a matching hardware operator and next it wires these operators together. To this end, it *extracts netlists* of instantiated hardware operators from a pre-synthesized circuit library. This library is a part of the PivPav tool and is used as *netlist cache* in order to speedup the hardware runtime adaptation process. Moreover, the library consists of thousands of different arithmetical and logical operators. Since these operators are functionally equivalent but different in the number of pipeline stages, area, power consumption, and performances the right selection of them allows to optimize the hardware accelerator under different criteria. In addition, this tool uses intermediate representation as a source input and thus, it is agnostic to the HLL; see Appendix A for explanation of compiler terminology.

Once the VHDL code is generated the Xilinx FPGA CAD project is created with the help of the *xtclsh* shell. Finally, PivPav creates FPGA CAD project for Xilinx ISE, sets up the parameters of the FPGA, and adds the VHDL and the netlist files.

Instruction Implementation

The last goal corresponds to the *instruction implementation*. In this phase, the Xilinx FPGA CAD tool flow is used in order to convert previously created project into the partial reconfig-

uration bitstream format that is used to reconfigure the Woolcano architecture afterwards. For this task, the Xilinx ISE tools are used. They are labeled with blue color and are executed and controlled by the PivPav tool. The bitstream files are mandatory for the hardware runtime adaptation layer.

4.3 PivPav Tool

As presented in previous section the PivPav tool is a key component of the ASIP-SP. It was specifically designed and developed for this purpose. It is solely used to implement the netlist generation and the instruction implementation phases of the ASIP-SP. In addition, it is used for estimation purposes in the candidate search phase where it acts as the metric circuit database. In this section, we briefly present major concepts of this tool whereas the details can be found in [58].

4.3.1 Overview

Essentially, PivPav provides an application programming interface (API) to a library of circuits that are kept in a database. For each circuit, an extensive set of metadata is made available through an API. While PivPav is agnostic to the kind of circuits that are stored in the library, it provides an API to commonly used circuit generators such as Xilinx Coregen [59] and FloPoCo [60]. These generators can be controlled from PivPav and allow to automatically create required hardware operators in background . Besides circuit generators, customized circuits that were manually created can be also added to the database.

In addition to the circuit library and circuit generators, PivPav contains a benchmarking framework. This integrated framework allows for processing and measuring the circuits with FPGA CAD tools under a variety of implementation constraints. This process characterizes each circuit with more than 90 different metrics.

4.3.2 Goals

The main purpose of PivPav is to provide a software infrastructure for storing and retrieving circuits and meta information about the circuits from the database. These informations represent circuit latency, maximum operating frequency, power consumption, input and output interfaces, etc.

4.3.3 Use Case

A prime use case example is an ASIP-SP netlist generation phase, which was also the use case that triggered the development of PivPav. In netlist generation, complex data paths are assembled from circuits representing basic operators, where each operator can be available in a large variety of implementations that are functionally equivalent, but differ in hardware size, speed, latency, etc; see Table 4.1 as an example. For finding the optimal data path implementation, a design space exploration of the basic operators is required, which is ideally

supported by PivPav. The use case examples based on ASIP-SP process are illustrated in Figure 4.3.

Circuit	Latency	Initiation interval	Pwr . cons.	Max FRQ after PAR	FF	LUT	Slice	BUF	DSP
	[cyc]	[cyc.]	[mW]	[MHz]	no.	no.	no.	no.	no.
Integer Operators									
add_561	4	0	1244.7	169.3	66	70	53	98	0
add_558	1	0	1242.8	107.4	66	32	16	98	0
mul_376	5	0	1259.2	190.7	65	17	11	97	3
mul_403	1	0	1260.1	43.4	65	749	376	97	1
sub_19	18	0	1246.4	179.1	66	124	95	98	0
sub_1	0	0	1237.0	104.5	97	32	16	97	0
div_104	50	1	1344.6	98.8	65	-	-	112	0
div_158	43	2	1321.6	41.0	65	-	-	104	0
Floating Point Operators									
fpadd_1183	7	0	1264.0	139.0	66	556	326	103	4
fpadd_1161	1	0	1296.3	31.1	66	377	250	103	5
fpmul_1115	10	0	1316.2	136.8	66	134	150	103	4
fpmul_1136	1	0	1289.8	40.3	66	76	46	103	0
fpsqrt_1011	26	1	1072.9	175.5	34	508	475	69	0
fpsqrt_691	3	1	1074.0	29.7	34	464	294	69	4
fpsub_661	16	0	1295.2	139.4	66	381	354	103	4
fpsub_675	1	0	1278.2	30.5	66	377	251	103	0
fpdiv_1595	28	19	1254.6	171.9	66	367	231	104	0
fpdiv_1209	4	4	1267.1	32.8	66	466	268	104	4
fp-float2int_630	4	0	1062.7	172	34	286	164	70	0
fp-float2int_627	1	0	1070.3	43	34	229	138	70	0

Table 4.1: Excerpt of hardware operator metrics when querying PivPav for the XC4VFX100-FF1152-10 FPGA device.

4.3.4 Stand-alone and Back End Tool

While PivPav is useful as a stand-alone benchmarking tool, its main purpose is to act as a building block in higher-level tool flows for reconfigurable computing. To this end, it was designed as a backend tool that features rich application programming interface.

4.3.5 Application Programming Interface

The PivPav features application programming interface that allows to integrate it in custom design tools. The developed API supports C and C++ programming languages. In addition, the access to the database is available for any programming language that supports open database connectivity (ODBC).

4.3.6 Open Sourced License Model

The tool is released under the GPL open source license model. This allows the community to freely:

- benchmark and test their designs,
- benchmark FPGA CAD algorithms,
- build higher level design tools that leverage the functionality of PivPav.

PivPav is available for free download and it does not require any registration.

4.3.7 Design Overview

PivPav combines three components, which are illustrated in Figure 4.3.

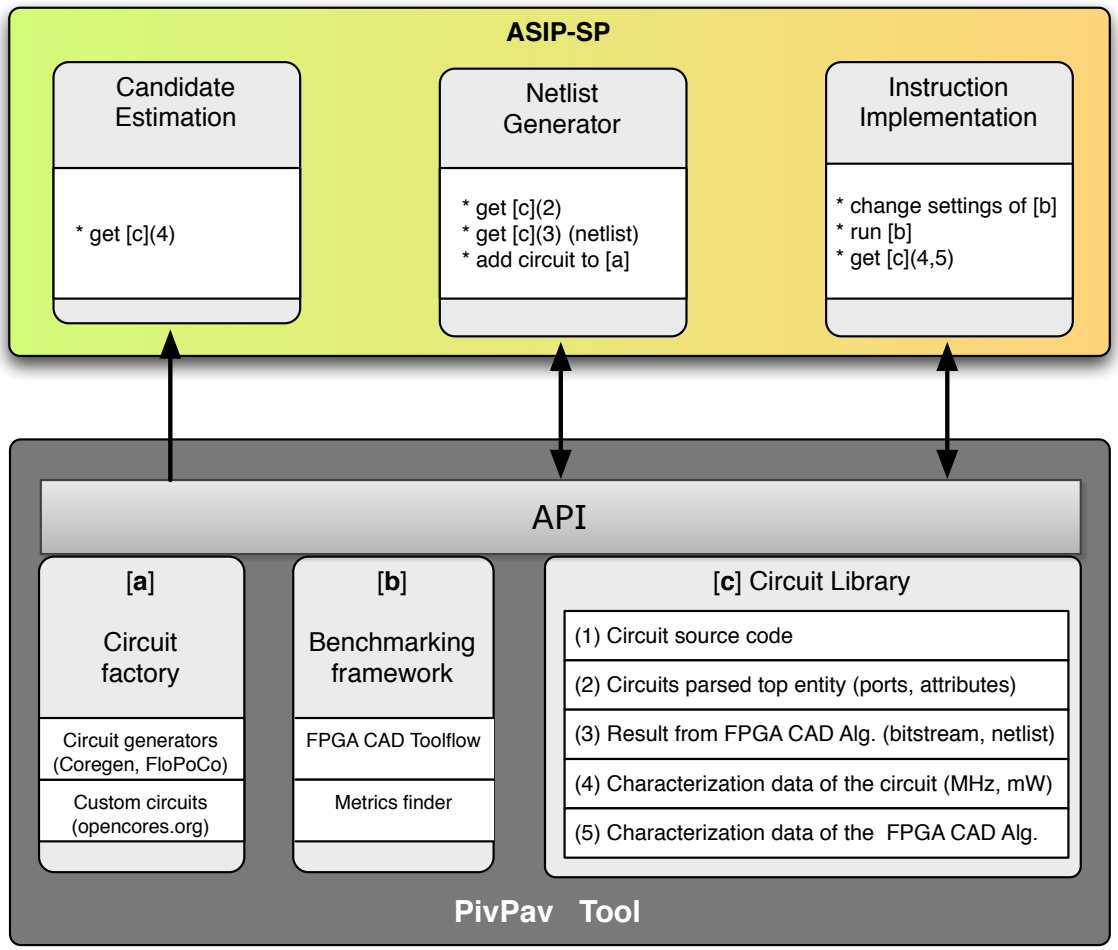

Figure 4.3: PivPav design and ASIP-SP use case.

a) The **circuit factory** is responsible for generating circuits. To this end, PivPav interfaces to the widely used circuit generators Xilinx Coregen [59] and FloPoCo [60] and supports importing custom circuits that have been generated by synthesis from hardware description languages.

b) The **benchmarking framework** is responsible for determining the performance and metadata for the circuits obtained from the circuit factory by executing the FPGA CAD tool flow and analysis tools for gathering characteristic metrics of each circuit. Overall, more than 90 different metrics are determined for each circuit. Additionally, the information about the optimization settings of the FPGA tool flow (e. g., optimization goals and effort) and the runtime and memory usage of the design tools are determined.
c) The **circuit library** is the database component that stores the circuit and the results of the benchmarking process, including the circuit's source code, netlist, parsed VHDL top entity, results files from the FPGA CAD tool flow, and the CAD tool flow characterization data.

4.3.8 Contributions

PivPav is a unique tool that was developed and studied in details in Grad *et al*. [58]. During our research we tried to re-use already existing tools for the development of our system. However, in this case we could not find any tool with required functionality, especially the open sourced one. This lead and triggered the development of the PivPav tool.

There are two main contribution of PivPav to the reconfigurable community. First contribution corresponds to the unique functionality of the tool that allows for consistent access to the circuit library, metric informations, netlists, circuit generators, benchmarking facilities, etc. Second to the open sourced model. Both features allow to utilize this standalone and backend tool by the reconfigurable community in various ways that we presented on our case.

4.3.9 Work Related to PivPav

Circuit libraries play an important role for increasing the design productivity by allowing designers and design tools to reuse existing circuits as pre-generated components. Libraries providing different implementation alternatives further allow for selecting the optimal circuit from a choice of functionally equivalent circuits, which provide different performance vs. area vs. latency trade-offs. While the circuit libraries used by HDL synthesis tools are typically not directly accessible by the designer, there FPGA tool vendors offer standalone circuit generators, such as Xilinx Coregen [59] or Altera MegaWizard [61] which generate circuits tailored for the respective FPGA architectures. Other circuit generators, such as FloPoCo [60] or GRLIB [62], support multiple FPGA families or generate architecture independent circuits.

PivPav differs from these tools by integrating a programmable circuit library, benchmarking infrastructure, and a circuit factory in a single tool. It allows to retrieve a large set of meta data about each circuit, such as performance, area, power consumption, latency, etc., which provides the necessary input data to tools that estimated the performance of a larger circuit that is composed of elements retrieve from PivPav. Additionally, PivPav allows for programmatically connecting individual circuits to form larger data paths and to generate the corresponding, structural HDL code. While all performance-driven high-level synthesis tools are likely to use a similar library internally, PivPav is the only open-source tool that offers this functionality.

Chapter 5

Hardware Runtime Adaptation

5.1 Introduction

In this chapter, we describe the parts of the ASIP-SP that allow to achieve the hardware runtime adaptation. Thus, it includes all components and tools that were presented in Figure 4.2 with an exception to the Communication Pass. These pass is presented in the next chapter that deals with software runtime adaptation. The order of presented components is chronological to the appearance in Figure 4.2. The study of the functionality of all components is based on the raytracing algorithm that is presented bellow.

5.2 Raytracing Example

Figure 5.1 presents a source code excerpt from the Codermind raytracing algorithm [63]. The top part (a) of the figure corresponds to the ANSI C code where a single function `hitSphere()` is presented. This function computes the scalar product of two 3-dimensional vectors that correspond to ray/object intersections. The bottom part (b) of the figure represents the equivalent IR code that is obtained with LLVM frontend and is used to feed the ASIP-SP. It corresponds to the most-left vertical box named *bitcode* in Figure 4.2.

5.3 Basic Block Pruning

Basic Block (BB) Pruning is the first process executed in the ASIP-SP outlined in Figure 4.2. Pruning uses a set of algorithms which act as *filters* to shrink the search space for the subsequent processes by rejecting or by passing certain BBs; definition of BB is presented in Section A.4. This decision is based on the data obtained from *program analysis* which provides information about loops, the sizes of BBs, and the contained instruction types. In addition, the ASIP-SP makes it possible to discard dead code by running the filters only for the code which was executed at least once.

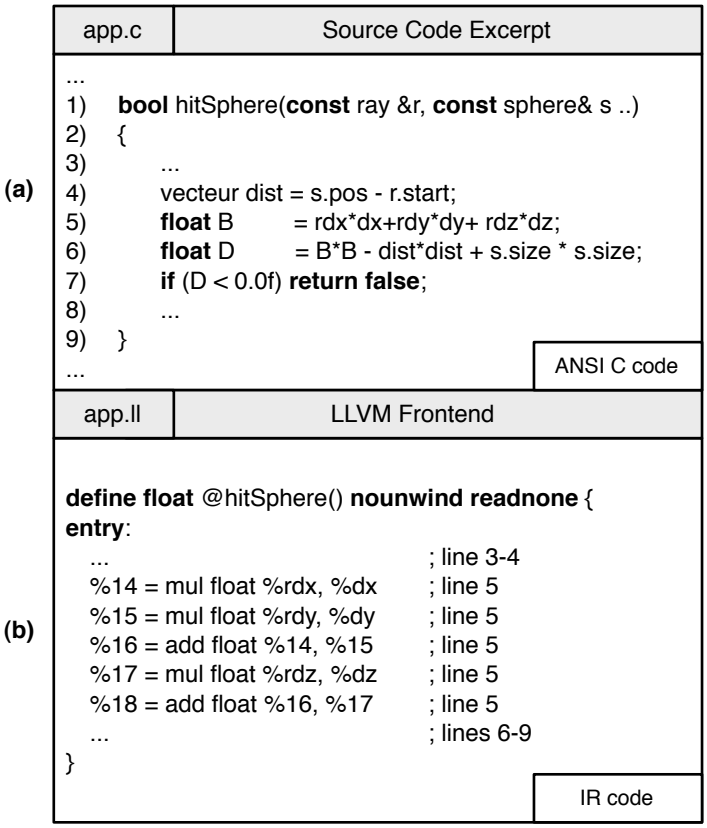

Figure 5.1: Excerpt from the raytracing algorithm and equivalent IR code.

5.3.1 Formal Definition

The objective of the pruning process is described by the following function:

$$\max\left(\frac{\text{metric function}}{\text{runtime of candidate identification}}\right). \tag{5.1}$$

The pruning aims to maximize the ratio between a *metric function* to the time spent in the candidate identification process. The *metric function* is defined in Condition (5.14) and is equivalent to the *application performance* gain. The denominator of the equation takes into the account only the *runtime of the candidate identification* since in comparison to this runtime the runtime of the *candidate estimation* and *selection* processes are insignificant.

In order to maximize the objective function it is beneficial to reduce the value of the denominator and increase the value of the nominator. To this end, pruning algorithms are developed around a hypothesis which predicts the location of the most beneficial application parts.

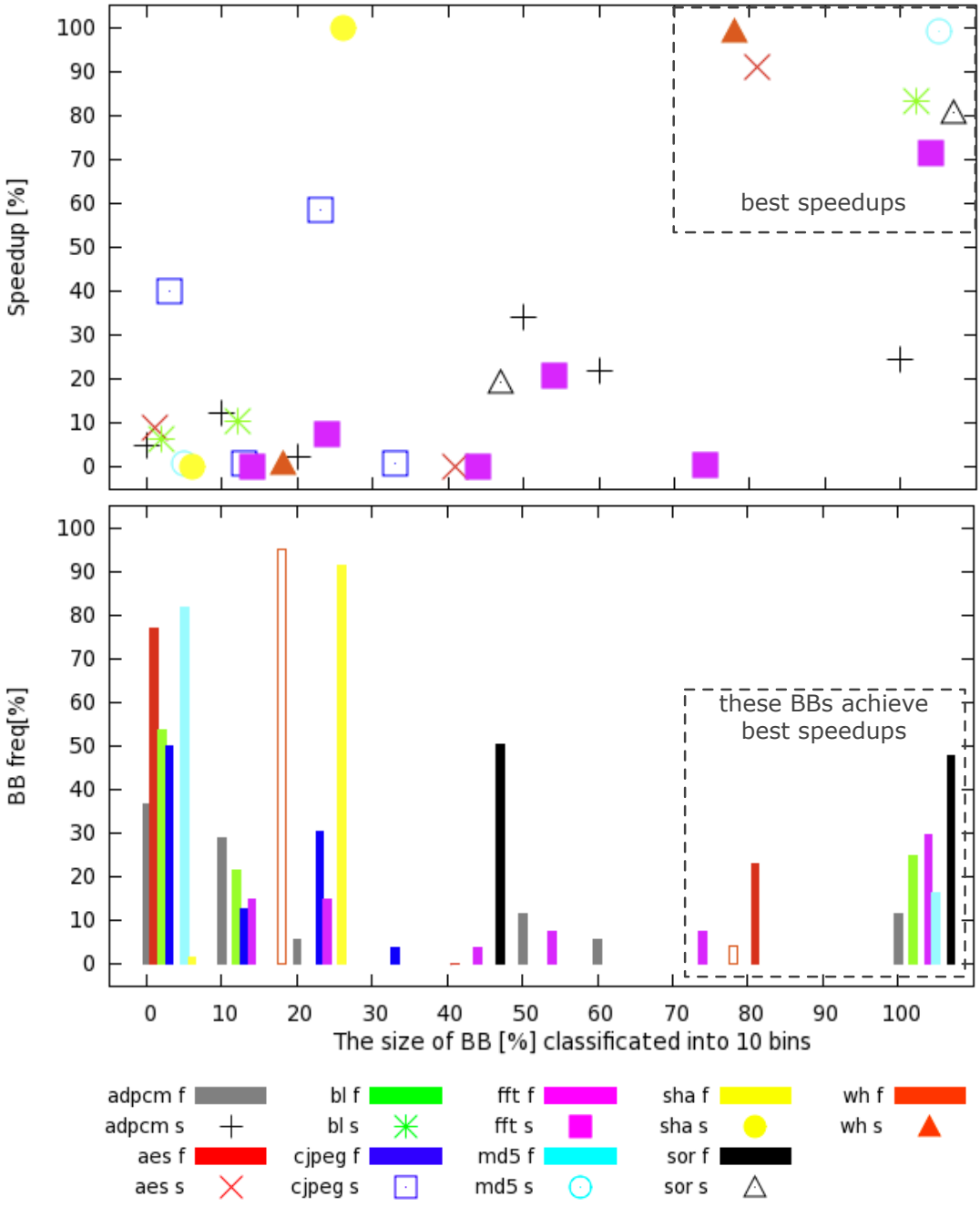

Figure 5.2: Relative relationship between BB sizes to application performance gain and to BB frequency for MaxMISO ISE algorithm (see Section 5.4). Absolute values are presented in Table D.1. s = speedup, f = frequency.

5.3.2 Pruning Hypothesis

Figure 5.2 illustrates the mutual dependencies of BB size, execution frequency, and achievable speedup for 9 different applications. For each application, all BBs that can be accelerated with custom instructions are sorted into 10 bins depending on their absolute sizes (measured in instructions) in a histogram-like manner. The first bin contains all BBs that have a size of 0–10% of maximal basic block size for this application, the second bin collects all BBs that have a size of 10–20% of the maximum BB size, etc. For each application and bin, the total execution frequencies for all BBs assigned to each bin are shown in the bottom figure. The top figure shows the overall application-level speedup when implementing the UDCI.

Best Performance Improvements

As visualized in Figure 5.2, the best speedups (70–100%) are found in the top right corner of the figure and are achieved for basic blocks in the 80–100% size bins, whereas the most frequently used BBs (up to 90%) are located in 0–30% size bins. This observation suggests that it is more significant to identify candidates in frequently used large BBs than in the most frequently used but smaller BBs. The hypothesis of our proposed pruning methodologies is to identify these highly profitable BBs and to ignore the insignificant rest of BBs.

Conclusions: Frequency vs. Size

In large BBs, ISE algorithms are able to find more and larger candidates which overall result in better speedups than accelerating more frequently executed but smaller BBs. In other words, execution frequency matters but among the most frequently executed BBs the largest BBs are the most important for achieving high overall speedups. The `aes` application is an ideal example of such a behavior. There are two red bars representing the frequencies of BBs for `aes`, the first is located at 0%, the second at 80%, both with distinctively high frequencies. However, the major speedup (91%) is achieved when accelerating the BBs represented by the second bar which are larger but less frequently executed than the first one.

Another interesting application is `sor`. There are two black bars at 40% and 100%, respectively, with almost the same frequency. The sum of these bars is equal to 100% frequency, which means that we are investigating all executed BBs in the application. Once again the second bar which represents bigger BBs has a major speedup of more than 80%.

5.3.3 Pruning Algorithms

In this subsection, a set of pruning algorithms, presented in Table 5.1, is described. Every pruning algorithm works as a filter and reduces the amount of the IR code available for the candidate identification and remaining processes. All algorithms except `n(F/S)` try to leverage the hypothesis stated above that the highest speedups are achieved for large and frequently executed BBs.

Name	Algorithm	Description
nS	n-size	select n largest BB
nF	n-freq	select n most frequently used ones
nL	n-loop	select n largest loop BBs
@nL	n-loop(@)	eliminate not used BBs and do nL
mLn*	n-(m-loop(uniq F))	do nL for every function and then mL
mSnL	n-loop(m-size)	do mS and them nL
@mSnL	n-loop(m-size(@))	eliminate not used BBs and do mSnL
mFnS	n-size(m-freq)	do mF and then nS
mSnF	n-freq(m-size)	do mS and then nF
tF	threshold = max(F)	select BBs with this threshold
tS	threshold = max(S)	select BBs with this threshold
n(F/S)	n-(freq / size)	select n BBs with largest ratio

Table 5.1: Description of the pruning algorithms. The n and m letters represent constraining numbers, the p suffix, if added to these, represents percentage of the range. The @ tells that the algorithm used JIT to indicated BBs which were executed at least once (dead code elimination), S = Size, F = Frequency and L indicates Loops.

Basic Block Constrains

The easiest way to restrict the set of BBs to the most beneficial subset is to constrain the sizes and frequencies, which correspond on specifying the n and m parameters in the mFnS and mSnF algorithms. The remaining algorithms with the exceptions described below try to mimic the same goal but without using the frequency data. This is particularly important for a JIT processor customization scenario in which the execution frequencies for BBs are unknown. They can be collected during the execution but will remain incomplete or inaccurate until the application terminates. Hence during the runtime only partial information is available, which makes the goal of JIT processor specialization challenging.

Frequently Used Basic Blocks (L)

In order to find the frequently used BBs without profiling, a program analysis pass that examines the control flow structures in the application and indicates which BBs are contained in loops was developed. This technique is used in the L algorithms. The chances of executing such BBs are higher than the others and therefore it is one indication that these BBs are frequently executed. A loop can consist of many BBs and the nL algorithm selects only the largest BBs. Whenever possible analysis pass tries to skip the `pre-header` and `latch` parts of the loop and focuses only on the computational BBs in the loop bodies. This technique can also leverage applications which use the common software design patterns to encapsulate every significant part of the code in a separate function. For such applications, the nLm* algorithm will try to find the m loop-BBs in every function and will finally select the n largest ones from this set.

Dead Code Elimination (@)

The limitation of the `nL` algorithm is that it can consider loop-BBs which are not used at all during execution. For instance, such loop-BBs exist in the `decrypt` and `encrypt` functions of the `aes` application. During the execution of the application only one function is used, which can easily mislead the `nL` algorithm. To avoid such a case, the `@nL` algorithm was developed. It considers only BBs in functions that have been executed at least once which is an information that can be made available to the JIT systems.

Identical Execution Times

The difference between the `tF` and `nF` algorithm where $n = 1$ is that several BBs may have an identical execution frequency. In such cases, the first algorithm will consider all of them whereas the second will only consider the first BB. The same applies to the `tS` and the `nS` algorithms.

Hypothesis Opponent

Finally, the algorithm `n(F/S)` was implemented n(f/s) as a reference to test the hypothesis that the design space should be constrained to the largest and most frequently executed BBs. This algorithm favors the frequent but small BBs and hence should result in lower speedups if the hypothesis is correct.

5.3.4 Raytracing Example

Figure 5.3 represents a single BB that was passed by the pruning mechanism when analyzing the source code found in Figure 5.1(b). This passed BB corresponds to three (3–6) ANSI C code lines found in Figure 5.1(a).

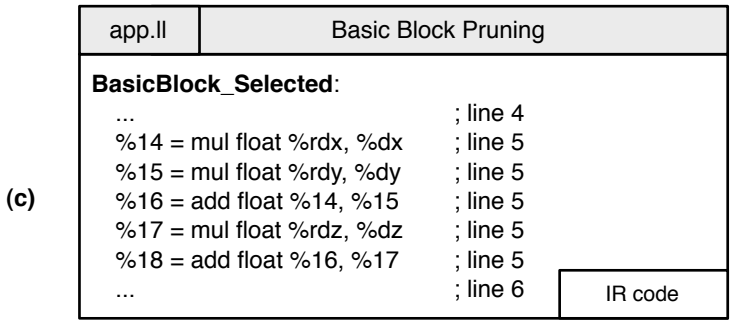

Figure 5.3: Basic Block that was selected by pruning mechanisms from the source code presented in Figure 5.1.

5.3.5 Contributions

The Basic Block Pruning for the runtime systems is a novel idea. To best of our knowledge, it has not been studied before. This is a first study in this field and is presented in Grad *et. al* [35].

5.3.6 Work Related to Pruning Algorithms

The issue of reducing the design space for candidate identification has not received any attention so far, with the exception of the *guide functions* [34]. Unfortunately, these are not well suited for the online systems, since they are based on profiling data which are not available in any online system. The lack of pruning mechanisms for the candidate identification algorithms may be caused by the fact that for a static non-reconfigurable ASIPs the runtime reduction is of less importance than for JIT ASIP-SP process which is a novel research area in itself.

While the ad-hoc heuristics have been proposed to select the basic blocks subject to custom instruction identification, e. g., all hot loops with an execution time of more than 1% of the total runtime [64] or just the most frequently executed basic block [65], no systematic study of approaches for pruning the design space has been conducted so far.

5.4 Candidate Identification

The candidate identification process identifies subgraphs in the *intermediate representation* (IR) code, which are suitable for fusing into a new UDCI that can be implemented for the Woolcano architecture. Suitable candidates are rich in *instruction level parallelism* (ILP) while satisfying the architectural constraints of the target architecture.

5.4.1 Formal Definition

Formally, we can define the candidate identification process as follows: Given a data flow graph (DFG) $G = (V, E)$, the architectural constraints $in_{\max}$, $out_{\max}$ and a set of infeasible instructions F, find all candidates (subgraphs) $C = (V', E') \subseteq G$ which satisfy the following conditions:

$$C_{\text{in}} \leq in_{\max}, \tag{5.2}$$

$$C_{\text{out}} \leq out_{\max}, \tag{5.3}$$

$$V' \cap F = \emptyset, \tag{5.4}$$

$$\forall t \in C : convex(t). \tag{5.5}$$

Here:

- the DFG is a *direct acyclic graph* (DAG) $G(V, E)$, with a set of *nodes* or *vertices* V that represent *IR operations* (instructions), *constants* and *values*, and an edge set E represented as binary relation on V which represents the data dependencies. The edge set E consists of ordered pairs of vertices where an edge $e_{ij} = (v_i, v_j)$ exists in E if the result of operation or the value defined by v_i is read by v_j.

- $C = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

- C_{in} is the set of *input nodes* of C, where a node $v_i \notin C$ with $(v_i, v_j) \in E$ for some node $v_j \in C$ is called an input node.

- C_{out} is the set of *output nodes* of C, where a node $v_o \in C$ with $(v_o, v_k) \in E$ for some node $v_k \notin C$ is called an output node.

- $in_{\max}$, $out_{\max}$ are constants that specify the input/output operand constraints for UDCIs which apply to the instruction slot implementation of the Woolcano architecture.

- $F \subseteq V$ is a subset of illegal graph nodes (IR instructions) which are not allowed to be included in UDCI. This set includes for instance all memory access instructions like *loads* and *stores* (since UDCI cannot access memory) and other instructions which are not suitable for a hardware implementation; for example, for performance reasons.

- *Convex* means that there does not exist a path between two vertices in C which contains a node that does not belong to C. In other words candidate C is *convex* if $v_i \in C, i = 0, \ldots, k$ for all paths $\langle v_0, \ldots, v_k \rangle$ with $v_0 \in C$ and $v_k \in C$ where path is defined as follows: A *path* from a vertex u to a vertex v in a graph C is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $v_0 = u$, $v_k = v$ and $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, k$. Convexity ensures that C can be executed atomically in the hardware. This property is required to make sure that an instruction can be executed as an atomic operation without exchanging intermediate results with the CPU.

Basic Block as Elementary Unit

Translating these formal definitions into practice means that the identification of UDCI candidates occurs at the level of basic blocks. BBs are suitable units for this purpose since they have a DAG structure as required by the ISE algorithms. In addition, it is feasible to enforce the convexity condition 5.5) on them. When selecting IR code for a UDCI implementation, illegal instructions, such as control flow or memory operations, must be excluded; see condition 5.4. Finally, the number of inputs and outputs of the candidate has to match the architectural constraints that are described in conditions 5.2 and 5.3. These architectural constraints are variable and are defined by the FCM controller and by the interface to the partially reconfigurable slots into which the UDCIs are loaded.

5.4.2 Supported Instruction Set Identification Algorithms

For the identification process, we implemented three state-of-the-art ISE algorithms that are considered as the most suitable for runtime processor specialization: the *SingleCut* (SC) [66], the *Union* (UN) [65], and the *MaxMiso* (MM) [67] algorithm. The most relevant properties of these algorithms are presented in Table 5.2.

Algorithm	# of inputs	# of outputs	worst-case complexity	overlapping candidates
MaxMiso	invariant (∞)	invariant (1)	O(n)	no
SingleCut	variant	variant	O(exp)	yes
Union	variant	variant	O(exp)	no

Table 5.2: Comparison of Candidate Identification ISE algorithms.

Finding Candidates Only for Selected Basic Blocks

All identification algorithms do not try to find candidates in the entire application at once. Instead, they focus on the individual basic blocks of the application. Hence, in order to prune the search space, the ISE algorithm is executed selectively only for the most promising BBs. Each ISE algorithm analyzes the basic block to identify all feasible candidates. Further, some algorithms allow to constrain the number of inputs and outputs of candidates to match the architectural constraints of the targeted ASIP architecture. Finally, all algorithms fulfill the condition 5.4.

Algorithm Comparisons

The advantage of the MM algorithm is its linear complexity $O(n)$. However, it finds only candidates which have a single output and does not allow to constrain number of inputs. Therefore, some of the generated candidates need to be discarded at additional costs later in the selection phase because they validate conditions and thus they cannot be implemented. In contrast, the SC and UN algorithms already allow for restricting the desired number of inputs and outputs for candidates during the identification phase. Finally, the SC algorithm may produce overlapping candidates. Hence, when using this algorithm an additional phase is needed in the selection process to eliminate the overlapping candidates.

5.4.3 Raytracing Example

Figure 5.4 represents the source code of the candidate (blue color) that was found by the ISE algorithm. The ISE algorithm searched for candidates in the BBs that were passed by the pruning mechanisms. The source code of this BB is presented in Figure 5.3(b).

5.4.4 Contributions

The research in Candidate Identification is based on the well known state-of-the-art ISE algorithms. In this work, we did not invent any new ISE algorithms. The contributions of this thesis to this field corresponds to: a) implementation of all algorithms under consistent framework, b) detailed algorithm comparison and analysis (algorithm runtime vs. BB-size; number-of-found-candidates vs. BB-size; achievable performance gains), c) study of the best ISE algorithms for the JIT purposes.

(d)

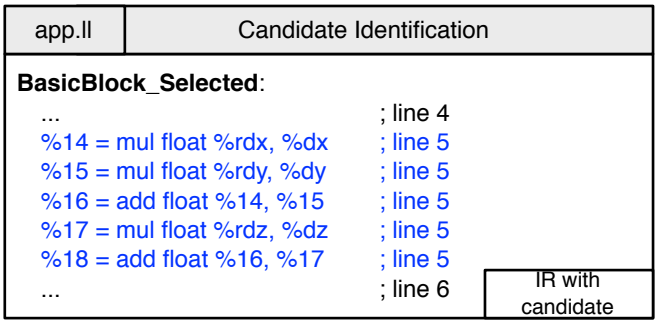

Figure 5.4: Candidate found by the ISE algorithm (blue color) in a BB that was selected by the pruning mechanism in Figure 5.3.

5.4.5 Work related to Candidate Identification

The instruction set extension algorithms have been extensively studied; see [34] for a recent survey.

5.5 Candidate Estimation

Since the number of UDCIs that can be implemented concurrently in the Woolcano architecture is limited, only the best candidates are selected for an implementation. The corresponding selection process which is described in the following section is based on the estimated performance of every candidate when executing in software on the CPU or in hardware as UDCI. Based on these performance estimates the subsequent selection process can decide whether it is affordable and beneficial to implement a candidate as a hardware UDCI.

5.5.1 Software Estimation

The Woolcano architecture consists of a PowerPC 405 CPU hard core which is used for software execution. In contrast to modern general-purpose CPUs, the PowerPC CPU has a relatively simple design. It is a in-order scalar processor with five-stage pipeline where most instructions have a latency of a single cycle. This simple design makes the task of software estimation relatively easy since the execution of the instructions in the candidate is sequential. Hence, the estimation method presented bellow is not a novel idea and it is based on research published in Gong et al. [68].

Formal Definition

We estimate the performance of the execution with the expression shown in Condition 5.6 which corresponds to the sum of latencies of all instructions found in the candidate multiplied by the CPU clock period. This estimation technique has an algorithmic complexity of $O(n)$

where n is the number of PowerPC instructions found in a candidate.

$$\begin{aligned} T_{\mathrm{sw}} &= T_{\mathrm{cpu}} \cdot L_{\mathrm{sum}} \quad [\mathrm{ns}], \\ L_{\mathrm{sum}} &= \sum_{i=0}^{n} L_i \quad [\mathrm{ns}]. \end{aligned} \tag{5.6}$$

Here:

- T_{cpu} is the clock period of used PowerPC CPU.
- L_{sum} is the sum of latencies of all instructions found in a candidate, where n is the number of instructions and L_i is the latency of the *i*-th instruction found in a candidate (the instruction latencies have been determined from the PowerPC manual [69]).

The use of the T_{cpu} in Condition 5.6 ensures that the differences in clock periods between the PowerPC CPU and the UDCI hardware implementation are taken into account.

Estimation Difference between IR and Machine Code

The presented method yields correct results only when the candidate's IR code is translated one-to-one into the matching PowerPC machine instructions. In the case of a mismatch between these two, the estimation results are inaccurate. The mismatch can happen due to a few reasons; i. e., folding a few IR instructions into a single target instruction, or because of differences in the register sets. The PowerPC architecture has a fixed amount of registers (32 general-purpose registers) whereas the IR code uses an unlimited number of *virtual registers*. For larger code, which requires more registers than available, the backend of the compiler produces additional instructions which will move data from registers into temporary variables kept on the stack. These additional instructions are not covered in the software estimation process. For such cases it has been shown that the estimation inaccuracy can be as high as 29% [70].

5.5.2 Hardware Estimation

Since each instruction candidate can be translated to a wide variety of functionally equivalent datapaths, the task of hardware estimation is much more complicated than the task of software estimation. In the following, we choose to illustrate approach used in this work by means of an actual example for the UDCI candidate that was selected by ISE algorithm from previous section; see Figure 5.4.

Estimation for Data Path Synthesis vs. High Level Synthesis

When translated to hardware, the DFG structure of the candidate is preserved; that is, instead of complex high-level synthesis [71], a more restricted and thus simpler data path synthesis (DPS) process [22, 23] is required which does not generate complex finite state machines in order to schedule and synchronize computational tasks.

Estimation Process

The first step in the hardware estimation for DPS involves translating each IR instruction (node) into a corresponding *hardware operator*, where operators may exist as purely *combinational* operators or as *sequential* operators. Sequential operators produce a valid result after one or – in the case of pipelined operators – several clock cycles. Also, functionally equivalent operators can have a large variety of different properties such as hardware area, speed, latency, power consumption, etc. Therefore, the hardware estimation tasks have to deal with three different types of datapath structures: a) combinational, b) sequential and c), hybrid datapaths, where a mix of sequential and combinational operators exists. Examples for such datapaths for the discussed candidate are shown in Figure 5.5.

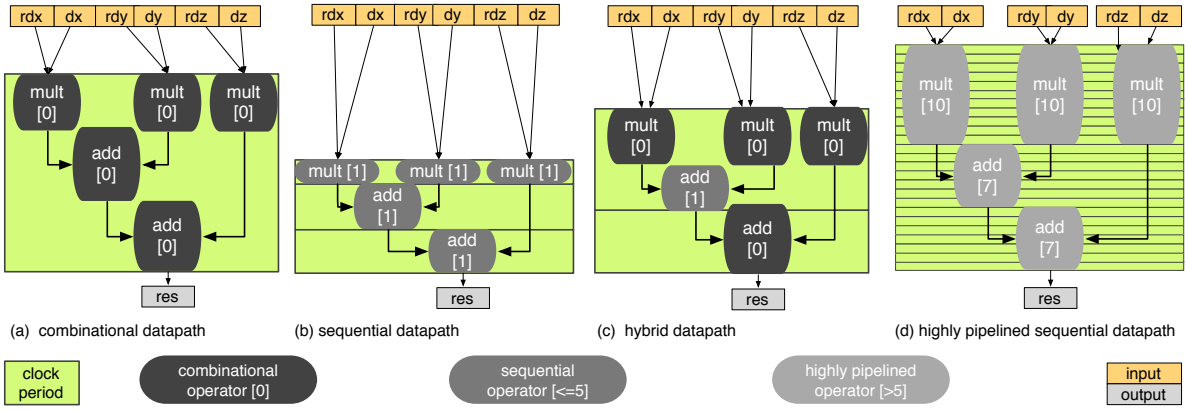

Figure 5.5: Different types of a DFG presented for a UDCI candidate that was found by ISE algorithm; see Figure 5.4.

Formal Definition

The hardware estimation process used in this work for estimating the delay of a UDCI supports all of these scenarios and is formally defined as:

$$C_{\text{hw}} = T_{\text{udci}} \cdot R_{\text{D}} \cdot P_{\max} \quad [\text{ns}], \tag{5.7}$$

$$T_{\text{udci}} = \max\{L\} \quad [\text{ns}], \tag{5.8}$$

$$P_{\max} = \max\{P\} \quad [\#]. \tag{5.9}$$

Here:

- T_{udci} corresponds to the minimal allowable *clock period*, which is visualized as the tallest green box in Figure 5.5. For combinational datapaths, scenario (a), it is equivalent to the latency of critical path, whereas for the sequential data paths, scenario (b), to the maximal latency of all operators (`add[1]` in this example). For hybrid data paths, scenario (c), it corresponds to the highest latency of all sequential operators and combinational paths; in this case, to the sum of combinational `mult[0]` and sequential `add[1]` operator latencies.

- R_{D} is an experimentally determined routing delay parameter which is used to decrease the T_{udci}. The routing delays are equivalent to the communication latencies between connected operators caused by the routing switch-boxes and wires. The precise value of R_{D} is unknown until the physical placement of the operators is performed in the FPGA; however, experiments showed that R_{D} often corresponds to about half of all circuit latencies.

- $P_{\max}$ is the maximum number of pipeline stages. It can be interpreted as the maximum number of all green rectangles covering a given DFG. For scenarios a,b,c, and d, $P_{\max}$ equals to $1, 3, 2$, and 24, respectively.

- L is a set of latencies generated in Algorithm 1 and its maximum for all operators defines the minimal allowable clock period; see Condition 5.8. The graphical interpretation of L, as presented in Figure 5.5, corresponds to a list of the height of all green boxes. Thus, combinational datapaths, scenario (a), have the highest latencies, whereas the smallest latency can be found in highly pipelined sequential datapaths, scenario (d). The latency of each operator is obtained from the PivPav circuit library with the `Latency()` function found in the algorithm in the 4th line.

- P is a set of all pipeline stages generated by Algorithm 1. P is used in Condition 5.9 to select the maximum number of stages in a given datapth. The number of pipeline stages for the operator is retrieved with `Pipeline()` function and it is presented in square brackets in the operator name; thus, the `mult[10]`-reflects the 10 stages multiplier.

Algorithm Overview

Algorithm 1 is used to compute the values of the L and P sets, which are associated with the height and the number of green boxes, respectively. In the first line of the algorithm, initialization statements are found. In the second line, the algorithm iterates and generates results for every *critical path*, which indicates that a path leads from the *input* to the *output* node. In next three lines, for each node in a given path, the latency and the number of pipeline stages are accumulated in L_p and P_p temporal variables, respectively. If the given node is sequential, a new green box is created and the current latency value L_p is moved to the set L (lines 6–8). Lines 10–15 are executed when the algorithm reaches the last node in the given critical path. Thus, lines 10–12 add an additional pipeline stage if the last node is a combinational one, whereas lines 13–14 move the values of temporal variables to the resulting sets. Since the candidates do not overlap, each node of the candidate is visited only once and therefore this estimation technique is of $O(n)$ complexity.

5.5.3 Hardware Operators

The T_{udci} and $P_{\max}$ factors in the equations depend on the characteristics of every used hardware operator; see `Latency()` and `Pipeline()` functions in Algorithm 1; metrics found in Table 5.3. Thus, the key to accurate hardware estimation is the quality of the characterization database that provides performance, latency, and area metrics for each operator. For

Algorithm 1 Hardware Estimation

```
1: L ← P ← φ
2: for p in critical_paths do
3:    for n ∈ nodes(p) do
4:       L_p ← L_p+ Latency(n)
5:       P_p ← P_p+ Pipeline(n)
6:       if Pipeline(n) ≠ 0 then
7:          L ← L ∪ L_p and L_p ← 0
8:       end if
9:    end for
10:   if Pipeline(n) = 0 then
11:      P_p ← P_p + 1
12:   end if
13:   L ← L ∪ L_p and L_p ← 0
14:   P ← P ∪ P_p and P_p ← 0
15: end for
```

some selected operators, for example, for floating point operators obtained from *IP Core Libraries*, data sheets that characterize each operator with the required performance metrics are available; see e. g., [72]. For most other operators – in particular those created on-demand by HDL synthesis tools, such as integer, logic, or bit manipulation operations – no data sheets exist. Moreover, the characterization data in data sheets are not exhaustive and do not cover all FPGA families, models, and speed grades, which is problematic, since even within one device family the performance of operators can vary significantly. For example, the data sheet for Xilinx Coregen quotes the maximum speed of a floating-point multiplier as 192 MHz, or 423 MHz respectively, for two FPGA models from the Xilinx Virtex-6 family [Table 23 vs Table 25] [72]. This huge range makes it impractical to estimate accurate performance metrics for devices that are not even tabulated in the data sheets.

CMOS vs. FPGA Hardware Operators

The lack of accurate characterization data for arithmetic operators implemented on FPGAs had an impact on related work [65, 66, 66], which has obtained these metrics for fixed CMOS architectures instead of reconfigurable FPGA devices. The metrics in related work were obtained by synthesizing given candidates using the *Synopsys Design Compiler* targeting 0.18 or $0.13\mu m$ standard-cells CMOS libraries. The performance difference between a fixed CMOS implementation and an FPGA implementation of UDCI candidates is frequently in the order of one magnitude or even more; see technology difference subsection 2.2.3. Since this work targets the FPGA-based Woolcano reconfigurable ASIP architecture this frequency difference has two implications. First, even for a static instruction set extension scenario, it is impossible to truthfully compare results with related work. Secondly, it is impossible to use the same estimation approach.

5.5.4 Raytracing Example

In order to illustrate the estimation process in detail, we show how C_{hw} is estimated for sequential (Fig. 5.5(b) and highly pipelined sequential (Fig. 5.5(d)) datapath representation of the candidate. The candidate used by this experiment was found by the ISE algorithm in Candidate Identification process and is presented in Figure 5.4.

Metrics

The metrics of used hardware operators were obtained from the PivPav circuit library; see Table 4.1. The metrics required for the raytracing example are summarized in Table 5.3. The top and bottom parts of the table show the metrics of the operators used in scenarios presented in Fig. 5.5(b) and Fig. 5.5(d), respectively.

HW Oper.	P_{p}	L_{p}	Max FRQ after PAR	FF	LUT	Slice	BUF	DSP
	#	ns	[MHz]	#	#	#	#	#
Sequential Operators (Fig. 5.5b)								
fpmul_1136	1	24.81	40.3	66	76	46	103	0
fpadd_1161	1	32.15	31.1	66	377	250	103	5
Highly Pipelined Operators (Fig. 5.5d)								
fpmul_1115	10	7.31	136.8	66	134	150	103	4
fpadd_1183	7	7.19	139.0	66	556	326	103	4

Table 5.3: Excerpt from metrics requested by the Candidate Estimation process from the PivPav circuit library for the XC4VFX100-FF1152-10 FPGA device.

Results

The estimation results found in Table 5.4 indicate that the sequential datapath is able to produce the first result almost twice as fast as the highly pipelined datapath ($1.82\times$). However, if processing many data (d) is able to fill the pipeline and work with 24 data at once, generating the results every 9.5 ns, accordingly to the formula $T_{udci} * R_d$, , whereas (b) generates a result only every 41.78 ns.

5.5.5 Contributions

It is worth noticing, that both presented estimation methods are not in themselves a novel ideas. In the software domain, the estimation techniques is relatively easy, especially that the UDCI candidate representation does not include any conditional branches. Such and other methods are studied in Gong *et al.* [68].

In the hardware domain, there are many timing analysis approaches which perform steps equivalent to the one presented above and in Ref. [73]. Therefore, the hardware estimation subsection does not contribute to the hardware timing analysis field. The novelty in this

		Fig. 5.5(b) sequential datapath	Fig. 5.5(d) highly pipelined sequential datapath
L	[ns]	24.81 and 32.15	7.31 and 7.19
T_{udci}	[ns]	32.15	7.31
P	#	3 and 2	24 and 17
$P_{\max}$	#	3	24
R_{D}	#	1.30	1.30
C_{hw}	[ns]	125.39	228.07

Table 5.4: Results of hardware estimation process for UDCI candidate presented in Figures 5.5 (b) and (d) implemented with two different sets of operators found in Table 5.3.

.

estimation approach relies in the precise characterization data that were generated with the PivPav tool. These data together with the presented methods allow to precisely estimate the hardware performances for UDCI.

5.5.6 Work Related to Candidate Estimation

There are many software estimation techniques; see Ref. [73]. A similar estimation technique based on the same methods and tools that we developed has been recently presented in Aung *et al.* [74].

In the hardware domain, the timing analysis and the performance estimation of the circuit is often performed with the high level synthesis [71]. This technique is considerably faster than estimation at the full synthesis level, yet it relies on third-party tools and it is not as accurate as the full synthesis. A simpler estimation method which operates directly on the code's DFG is proposed in [23]. Nontrivial hardware implementations often require additional circuits, e.g. finite state machines. If the delays of the components are known then the timing and performance analysis can be effectively calculated [75].

5.6 Candidate Selection

Once the set of candidates has been determined and estimation data are available, the selection process makes the final decision about which candidates should be implemented in the hardware.

5.6.1 Formal Definition

The formal definition of the selection algorithm is constructed from two tasks: a) pruning the search space for selection algorithm and b) problem definition.

Search Space Pruning

First, all the candidates that violate at least one of the constraints presented below are rejected:

$$C_{|\text{in}|} \leq in_{\max}, \tag{5.10}$$

$$C_{|\text{out}|} \leq out_{\max}, \tag{5.11}$$

$$\frac{C_{\text{sw}}}{C_{\text{hw}}} \leq threshold. \tag{5.12}$$

Here:

- $C_{|\text{in}|}$ and $C_{|\text{out}|}$ are equivalent to the constraints described in conditions (5.2) and (5.3), respectively. They correspond to architectural constraints for the number of input and output operands to the UDCI. They are applied to the ISE algorithms that are not able to perform this step themselves, such as the MM algorithm.
- C_{sw} and C_{hw} correspond to the software and hardware estimations, respectively. If $threshold = 1.0$, then there are no performance gains; when $threshold > 1.0$ there is a performance gain since it takes more cycles to execute the candidate in software than in hardware. Finally, if $threshold < 1.0$, the hardware implementation has a lower performance than the software.

After applying these conditions, the search space of the selection process is significantly reduced, since candidates that are either infeasible or would provide only low speedups are discarded. As a result, the runtime of the subsequent steps in the tool flow is considerably lower.

Problem Definition

The aim of the candidate selection process is to select up to $C_{\max}$ candidates from the set of C_{input} candidates generated by the identification process that offers the greatest advantage in terms of some metric M; in this case the *application performance*:

$$C_{\text{res}} = max\left(\forall C_i \in C_{\text{input}} : \sum_i M(C_i)\right). \tag{5.13}$$

Here:

- C_{res} is the resulting set of best candidates, $C_{|\text{res}|}$ is a size of this set, and $C_{\max}$ is the architectural constraint representing the number of supported UDCIs.
- M is a metric function defined as:

$$M(C_i) = \frac{C_{\text{sw}}(C_i)}{C_{\text{hw}}(C_i)}. \tag{5.14}$$

5.6.2 Selection Algorithm

For the purpose of the ASIP specialization, we use the greedy candidate selection algorithm that is presented in Algorithm 2 and which has a computational complexity of $O(C_{|\text{input}|})$. When using the ISE SC algorithm which may produce overlapping candidates for ISE identification our algorithm rejects any candidates that overlap with any candidate that has been selected so far.

Algorithm 2 best candidate selection

while $\mathcal{C}_{|\text{res}|} \leq \mathcal{C}_{\max}$ **do**
  $c_i \leftarrow max\{M(\mathcal{C}_i) \mid \mathcal{C}_i \in \mathcal{C}_{\text{input}}\}$
  if c_i does not overlap with $\mathcal{C}_{\text{res}}$ **then**
    $\mathcal{C}_{\text{res}} \leftarrow \mathcal{C}_{\text{res}} \cup c_i$
  end if
  $\mathcal{C}_{\text{input}} \leftarrow \mathcal{C}_{\text{input}} \cap c_i$.
end while

5.6.3 Selection Metrics

The metric function is used as a policy in the greedy selection process and is responsible for selecting only the best candidates. While in Condition (5.14), the *application performance* policy is used, nothing prevents basing the decision preference on a different metric. It is worth mentioning that the PivPav tool could be used to provide a wealth of other metrics since the circuit library stores more than 90 different properties about every hardware operator. These properties could be used for instance to develop resource usage or power consumption policies. Consequently, they can be used to estimate the size of the final *bitstream* and partial reconfiguration time, the runtimes of netlist generation and instruction implementation or many other metrics. Finally, all these policies could be merged together into a sophisticated ASIP specialization framework which would:

- maximize the performances,
- minimize the power consumption,
- constrain the resource area to the sizes of the UDCI slot.

Such a combined metric can be defined as an *integer linear programming* model. While this method would allow a more precise selection of candidates based on more parameters, its algorithmic complexity is higher than $O(C_{|\text{input}|})$, resulting in runtimes that are much longer, often by orders of magnitude. Since it is important to keep the runtimes of the ASIP-SP as low as possible, the tradeoff between the gains and the costs of the metric function is an interesting research topic in itself.

5.6.4 Raytracing Example

For our raytracing example, we assume that the spotted candidate presented in Figure 5.4 and estimated in previous section (Table 5.4) has been selected for UDCI hardware implementation.

5.6.5 Work Related to Candidate Selection

In general, selecting the optimal set of UDCI under different architectural constraints is a demanding task. Related work has studied different selection approaches, such as greedy heuristics, simulated annealing, ILP, or evolutionary algorithms; see, for example, Pozzi *et al.* [76] or Meeuws *et al.* [66].

5.7 Extraction Pass

The EP together with the CP which is described in the next chapter, constitute the last component in the Candidate Search phase of the ASIP-SP tool flow; see Figure 4.2.

5.7.1 Goal

The goal of the EP is to fold the IR code that corresponds to the UDCI candidate into a separate IR function. This behavior is achieved by creating a new function and by moving candidate's instructions into it. This allows for other ASIP-SP components to operate on a candidate more easily since, instead of a set of IR instructions they, treat the candidate as an IR function.

5.7.2 Algorithm Overview

The functionality of the extraction pass is presented in Algorithm 3.

Algorithm 3 candidate extraction

$f \leftarrow \mathrm{CreateFunction}()$
$f \leftarrow \mathrm{SetFunctionType}(\mathrm{C_{out}})$
for $arg \in C_{\mathrm{in}}$ **do**
  $f \leftarrow \mathrm{AddFunctionArgument}(\mathrm{arg})$
end for
$BB \leftarrow \mathrm{CreateBasicBlock}(\mathrm{Entry})$
$f \leftarrow f \cup BB$
for $ins \in C$ **do**
  $BB \leftarrow \mathrm{MoveInstruction}(\mathrm{ins})$
end for
$BB \leftarrow \mathrm{AddTerminator}()$

Here:

- C corresponds to the selected candidate; for instance see Figure 5.4.
- C_{out} represents output arguments and is equivalent to the condition (5.3).
- C_{in} represents input arguments; see condition (5.2).

This algorithm consists of several steps. First, an IR function with C_{out} data type and arguments defined in set C_{in} is created. Next, a basic block is added to the function. Finally, all instructions that are used by the selected candidate are moved to this basic block together with terminating instruction that returns control flow from the function. While not specified in the listing the algorithm performs one more step. It replaces every C code *user* with the *call* to a newly created function.

Once the candidate's IR instructions are extracted to the separate function with the EP pass two tasks occur. First, the extracted function is passed to the Netlist Generation phase. Secondly, after the hardware accelerator is generated the extracted function is passed to the software runtime adaptation and to the CP that are described in Section 6.4.

5.7.3 Raytracing Example

Figure 5.6 presents the outcome of the EP when executed for selected C candidate from Figure 5.4.

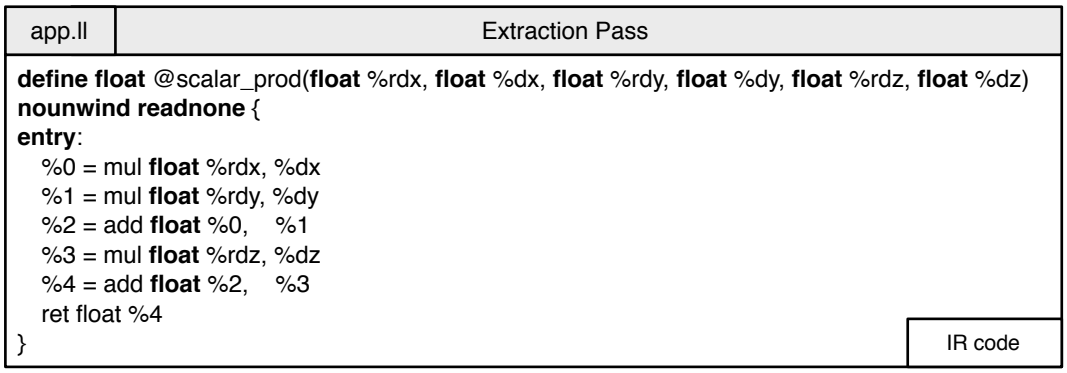

Figure 5.6: Outcome of the EP for raytracing case study.

5.8 VHDL Generator

The VHDL generator is a first process involved in the Netlist Generation (NG) phase. The aim of NG is to generate the hardware description equivalent to the specified IR function. In our system, this function is delivered by the extraction pass but in general this can be any IR function. In addition, NG is responsible for creating the FPGA CAD project.

5.8.1 Goal

As the name suggests the goal of this task is to generate the structural VHDL code from the given function. The need for the structural and not behavioral VHDL comes from the fact that it allows for design space exploration with different hardware operators. In addition, it supports all circuits that are found in the PivPav library.

5.8.2 Algorithm Overview

Algorithm 4 VHDL generator

$DFG \leftarrow$ CreateDFG(f)
for $node \in DFG$ **do**
  $node \leftarrow$ GetHWOperator()
end for
$VHDL \leftarrow$ WireOperators(TopologicalTravers(DFG))

First, the VHDL generator constructs the DFG representation of the specified IR function. To this end, it uses the *def-use* and *use-def* chains and data type informations which are conveniently available in the SSA IR. Therefore, every node of the DFG represents an IR instruction. Next, the generator maps nodes to the equivalent hardware operators. These are found in the PivPav's circuit library. Figure 5.7 shows an example of the DFG after performed mappings that was created for the raytracing code specified in Figure 5.6.

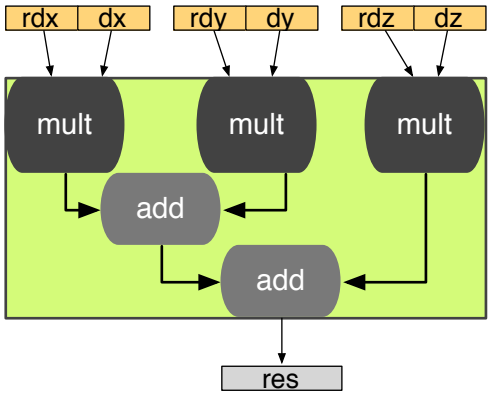

Figure 5.7: DFG generated with `CreateDFG()` for function `f` presented in Figure 5.6.

Finally, once the mappings are performed, the structural VHDL code is generated. To this end, the nodes of DFG are traversed in topological order and the PivPav's circuit library is used in order to obtain the informations about top entities of the used operators. These informations allow to instantiate and to wire operators accordingly.

5.8.3 Hardware Operator Wiring

Each operator has different entity that differ in data types and widths, naming of IO ports, polarity of reset signals for sequential operators, etc. Thus, while the operators wiring in VHDL may look like a straightforward task on the surface, connecting a wide variety of hardware operators from different sources bears a number of practical difficulties. In order to overcome these obstacles, the `WireOperators()` uses a consistent and convenient API that is available in PivPav. Otherwise the wiring task would be much more challenging.

5.8.4 Structural and Behavioral Code

The structural style of describing the hardware has a few advantages over the behavioral style.

- The structural code is straightforward for processing, which results in faster synthesis times. What is more important the structural code empowers users and gives them more control over the hardware description. This is presented in Figure 5.8 where `carry in` or `clock enable` signals can be specified, which is not the case for the behavioral code. Thus, the user is able to describe the functionality of the circuit more precisely. Moreover, since the constructs of the behavioral code are similar to the software HLL this leads to a habit of placing them in the hardware description, which results in errors.

- The structural code open doors towards the hierarchical design, which is well known method used to improve the performances of to the FPGA CAD tool. The generator uses top entities of operators to wire them together, however, nothing stands on the way to replace operators with more complex subsystems and, in consequence, start creating hierarchical designs.

- The structural style of programming allows for the design space explorations, which is shown in Section 5.11. The operators of the behavioral code are built into the synthesis tool flow and, for instance, there is no possibility to change the + adder for a different one. This is not the case for the structural style of programming where operators are not locked and can be freely replaced.

This difference between behavioral (h1) and structural (h2) code is presented in Figure 5.8. Both codes are equivalent to the DFG presented in Figure 5.7 with casted data types from `floating point` to `unsigned integer`. This data type casting operation was performed due to unavailability of the floating point operators in the behavioral description. In addition, while the behavioral code was implemented manually the structural code was generated automatically.

5.8.5 Raytracing Example

As already explained, Figure 5.8(h2) presents the structural VHDL code that was obtained from the VHDL generator. This code is generated for extracted function presented in Figure 5.6 with the difference of data types.

The code makes use of two PivPav's hardware operators: `mul_376` and `add_561`. Metrics of these operators are presented in Table 4.1. Please note that in the code these components are *black-boxed* and only declarations of these operators are shown. The implementation details are delivered by the netlists that are externally obtained from PivPav with Netlist Extraction process.

The presented code does not show the necessary wiring with partial reconfiguration *bus macros*. This step is performed in order to implement the fixed interface to the FCM controller for the partially reconfigurable slots.

app.vhd — Code Developed Manually

```
1)  entity scalar_prod is
2)          port (          clk         : in std_logic;
3)                          rdx, dx, .. : in  std_logic_vector(31 downto 0);
4)                          result      : out  std_logic_vector(31 downto 0));
5)  end entity;
6)  architecture arch of scalar_prod is
7)  begin
8)          result <= rdx * dx + rdy * dy + rdz * dz when rising_edge(clk);
9)  end architecture;
```

VHDL Behavioral Code

(h1)

app.vhd — VHDL Generator

```
1)  entity scalar_prod is
2)          port (          clk         : in std_logic;
3)                          rdx, dx, .. : in  std_logic_vector(31 downto 0);
4)                          result      : out  std_logic_vector(31 downto 0));
5)  end entity;
6)  architecture arch of scalar_prod is
7)  component mul_376 is port(   clk, ce : in std_logic;
8)                               a, b : in   std_logic_vector(31 downto 0);;
9)                               p : out std_logic_vector(31 downto 0));
10) end component;
11) component add_561 is port(   clk, ce, c_in : in std_logic;
12)                              c, d : in   std_logic_vector(31 downto 0);;
13)                              y : out std_logic_vector(31 downto 0));
14) end component;
15) signal umul1_s,umul2_s,umul3_s,uadd1_s :std_logic_vector(31 downto 0);
16) signal ce_s : std_logic := '1' ; signal cin_s : std_logic := '0';

17) begin
18) u_mul1 : mul_376 port map (  clk => clk,      a=> rdx,     b=>dx,
                                 p=>umul1_s,  ce=>ce_s);
19) u_mul2 : mul_376 port map (  clk => clk,      a=> rdy,     b=>dy,
                                 p=>umul2_s,  ce=>ce_s);
20) u_mul3 : mul_376 port map (  clk => clk,      a=> rdz,     b=>dz,
                                 p=>umul3_s,  ce=>ce_s);
21) u_add1 : add_561 port map (  clk => clk,      c=> umul1_s,   d=>umul2_s,
22)                              y=>uadd1_s,  ce=>ce_s,       c_in=>cin_s);
23) u_add2 : add_561 port map (  clk => clk,      c=> uadd1_s,   d=>umul3_s,
24)                              y=>result,       ce=>ce_s,       c_in=>cin_s);
25) end architecture;
```

VHDL Structural Code

(h2)

Figure 5.8: Comparison of *behavioral* vs *structural* VHDL code produced for DFG presented in Figure 5.7 with data types casted from `floating point` to `unsigned int`.

5.8.6 Contributions

The data path synthesis is not a novel idea and has been previously presented in De Micheli [77]. The developed VHDL generator does not purpose with a new synthesis algorithm to this field. The contributions correspond to designing and developing an open source tool that is capable of generating a structural code from IR.

5.8.7 Work Related to VHDL Generator

High level synthesis is a broad field that has been studied for decades [77]. Creating an optimal schedule with minimal delay becomes a hard problem once resource limitations are introduced. Several approximate (typically using heuristics) and exact (e.g. using ILP) methods have been proposed [78].

5.9 Netlist Extraction

After the VHDL generation the netlists are obtained from the PivPav's circuit library for every used hardware operator. For this purpose the PivPav's circuit library is used as a netlist cache. This has two advantages:

- It significantly reduces the runtime of the synthesis process since only the top entity of the generated circuit needs to be synthesized. The netlists for all the other components that are *black-boxed* are delivered externally and therefore there is no need to synthesize them.

- The second advantage results from the fact that the netlists can be swapped if only the operator's entity between them is the same. This allows to obtain the same functionality of the circuit but with different performances and resources and it opens doors towards the design space exploration.

5.10 FPGA CAD Project

In this step, the FPGA CAD project is prepared. It includes the VHDL source code of generated circuit, the netlist files of used hardware operators, and the user constrain file (UCF) which describes the precise locations of bus macros and resource areas used by partial reconfiguration slot; see Figures 3.4 and 3.5. If the UDCI shall be loadable at runtime to different slot, this process is repeated with different placement constraints for every UDCI slot. The project is prepared with the help of the PivPav tool and with the TCL scripting language which has API access able to control the Xilinx's ISE design suite.

5.11 Instruction Implementation

Once the FPGA CAD project is prepared the instruction implementation phase comes into action and generates the partial reconfiguration bitstream file that can be loaded into the UDCI slot. This task is performed automatically with the benchmarking facilities of PivPav. In addition, the PivPav tool is able to explore the impact of the FPGA CAD tool flow configurations *(design goals)* on the circuit. The results from this step for the code presented in Figure 5.8 are provided in Table 5.5 for six different design goals (`a-f`).

Implementations

The first implementation variant, denoted as Behavioral, is generated by running the FPGA CAD for the behavioral description from Figure 5.8(h1) with the Xilinx ISE design suite. This implementation is purely combinational [see Figure 5.5(a)] and uses the operators inferred by the synthesis tool instead of custom operators from the PivPav library. Additionally, two pipelined implementations, denoted as Pipelined(1) and Pipelined(2), are generated from the structural VHDL description located in Figure 5.8(h2). These implementations use circuits `add_558` and `mul_403` for Pipelined(1), and circuits `add_561` and `mul_376` for Pipelined(2). The characteristics of these basic hardware operators are shown in Table 4.1. It should be emphasized here, that the generator does not need to re-generate or re-synthesize the basic operators but the netlists for these operators are retrieved from the PivPav's circuit repository.

Comparison

Comparing the single-cycle behavioral implementation with the Pipelined(2) implementation reveals a speedup by a factor of 4.07 but comes at the price of increasing the latency to 13 cycles. The table row Pipelined(1) presents data for the same circuit built up from combinational components with pipeline registers inserted after each operator (latency = 1). While the latency increases from 1 to 3 cycles, the maximum operating frequency of Pipelined(1) is lower than for the Behavioral variant. The substantial differences in hardware resources are attributed to the fact that Pipelined(1) implements the functionality with LUTs rather than DSP blocks.

Runtime and Memory Usage

Results show that PivPav collects the runtime and memory usage for each execution of the FPGA tool flow. Further, it collects the circuit characterization data for each FPGA CAD algorithm setting. The b setting (minimum runtime) has almost the same runtime as the a (balanced) settings, also the circuit characterization data for both of these settings are exactly the same. Settings d, e, and f have the highest runtimes. This kind of explorations can lead to a better understanding of the FPGA CAD tool flows and can point out their strengths and weaknesses. What is more important, the knowledge gained from these observations can be applied to improve the quality of loop iteration. The availability of numerous metrics (in particular area, speed, latency, and power consumption), reveals the full potential of ASIP-SP for generating VHDL code for different scenarios, such as maximum performance, minimum power consumption, or minimum use of resources.

5.12 Noteworthy Implementation Details

The ASIP-SP process as well as the rest of the Woolcano compiler were developed in *C++* language, the same language which is used in the development of the LLVM compiler infrastructure.

	Goal	computation time [secs]					memory usage [MB]				circuit metrics						
FPGA Alg.		xst	ngd	map	par		xst	ngd	map	par							
						Total					Lat.	Freq.	Power	FF	LUTs	Slices	DSP
											[cyc]	[MHz]	[mW]				
Behavioral	a	8	4	10	38	60	412	321	696	724	1	67.94	875.35	47	79	62	9
	b	8	3	8	35	54	411	321	698	725		67.94	875.35	47	79	62	9
	c	7	3	37	55	102	411	322	915	733		66.92	864.87	32	94	62	9
	d	8	3	37	54	102	411	322	913	735		77.69	876.00	32	62	43	9
	e	7	3	46	93	149	411	322	936	699		154.66	862.55	47	79	66	9
	f	8	3	45	56	112	411	322	941	726		167.06	876.00	32	62	39	9
Pipelined(1)	a	14	5	13	53	85	443	340	722	804	3	53.41	889.61	372	2311	1314	3
	b	15	6	12	54	87	443	345	722	804		53.41	889.61	372	2311	1314	3
	c	15	6	74	110	205	443	345	956	782		60.90	897.54	372	2311	1329	3
	d	15	6	70	192	283	443	345	955	780		53.83	2039.16	148	2311	1233	3
	e	15	6	59	147	227	443	345	952	734		63.73	896.23	372	2311	1389	3
	f	15	6	88	249	358	443	345	949	791		55.17	2044.65	184	2311	1249	3
Pipelined(2)	a	9	4	8	37	58	413	326	701	769	13	227.38	880.37	474	237	266	9
	b	9	4	8	36	57	413	327	701	770		227.38	880.37	474	237	266	9
	c	9	4	46	66	125	413	327	919	737		**276.40**	889.32	474	237	356	9
	d	9	4	39	75	127	413	328	918	738		194.33	2031.40	250	237	169	9
	e	10	4	55	64	133	413	328	943	732		263.37	890.63	472	239	372	9
	f	9	4	50	79	142	413	328	943	731		179.05	2034.84	248	239	187	9

Table 5.5: Measured characterization data for three implementation variants of the example shown in Figure 5.8. Design Goal legend: a = balanced, b = minimum runtime, c = power optimization, d = timing performance optimization with IOB, e = timing performance optimization with physical synthesis, and f = timing performance optimization without IOB. Pipeline(1) makes use from `add_558` and `mul_403` hardware operators whereas Pipeline(2) from `add_561` and `mul_376`; see Table 4.1 for characterization data.

Libraries and APIs

The developed ASIP-SP code makes extensive usage of the C++ *Standard Template Library* (STL)[1] as it is also the case in the LLVM. The access to the PivPav tool is performed with two different APIs. To access the PivPav's *Circuit Library* presented in Figure 4.3 the *C/C++ Sqlite interface*[2] is used. The *C++/TCL interface*[3] provides access to the remaining parts of the PivPav's functionality. The implementation details can be found in [58].

The candidate identification process was implemented with the help of Boost C++ library[4]. Boost libraries are also developed in C++, peer-reviewed, open sourced, and standardized libraries. They were used to represent the data flow graphs and templates for the candidate search phase.

5.12.1 Data Flow Graph Representation

As described in Section A.3 the LLVM IR already contains representation of the program's DFG that was used to prototype the ASIP-SP process in Appendix C. Due to difficulties explained in the mentioned appendix, the DFG needed different representation, a simpler and more abstract one. To this end, Boost Graph Library (BGL) offering generic interface to the DFG properties and structure was used. It also provides with a number of graph algorithms such as topological sort algorithm that are used in the VHDL generation.

Implementation Details

The DFG implementation is based on the adjacency list graph structure representing a directed graph with bidirectional edge access. Each node of the graph has a pointer to the LLVM value which he represents. Moreover, it has a type which represents whether the value is an operand or an operator. The features of the LLVM are directly accessible in the the DFG implementation and were usually implemented by wrapping corresponding LLVM methods or by allowing direct access to them. These methods allow to query about the type of the LLVM Value or move around the *def-use* chains. In addition, the representation of the DFG can be translated to the DOT language and can be generated with Graphviz visualization software[5].

Construction

The construction of the DFG works as follows. First, the instructions of the BB are iterated with the exception to the final terminator instruction. Next, every instruction is added to the graph as a node and its type is determined. The node is configured as an output if the instruction it represents is a memory write or it used outside its basic block or in the block's terminator instruction. Once the node is added to the graph, the algorithm iterates over operands of the corresponding instruction. Operands are also added to the graph as input nodes if they are instructions contained in a different basic block, function parameters, or global values.

[1] `http://www.cplusplus.com/reference/stl/`
[2] `http://www.sqlite.org/capi3.html`
[3] `http://cpptcl.sourceforge.net`
[4] `http://www.boost.org`
[5] `http://www.graphviz.org/`

Finally, in any other case there already exists a node for the operand, edges are added from each operand to the current node.

5.12.2 UDCI Candidate Representation

The UDCI candidates are not represented as a DFG subgraphs but instead implemented with Boost `dynamic_bistset` class as `bitvectors`. For most operations the bitvector implementation outperformed the `std::vector<bool>` solution about twice. The size of the bitvector is equal to the number of the nodes in the template, and thus every bit in bitvector represents a DFG node. Setting the *i*-th bitvector bit includes the *i*-th DFG node in the template.

Characteristics

There are several advantages of this approach since operations on the template are reduced to the simple bitwise operations. This includes the joining templates together, checking whether they overlap, or adding and removing nodes to the template. This results in lower memory usage and better performances. While the bitvectors do not contain any informations about the structure of the graph, edges, or node types, these informations can be retrieved from the associated DFG.

Construction

Creating a DFG from a bitvector is straightforward operations. A node to the new DFG is added for each bitvector's bit and the type as well as the pointer to the LLVM value are copied from the parent DFG graph. Afterwards incoming and outgoing edges are added to each node, setting output nodes and adding input nodes as required.

Chapter 6

Software Runtime Adaptation

The ASIP-SP process performs two tasks: the hardware runtime adaptation and the software runtime adaptation. The hardware adaptation corresponds to generating the application specific hardware accelerators whereas the software runtime adaptation allows to utilize these accelerators. To perform the second task, the source code of the application has to be modified during the application runtime. These source code modifications will be reflected in the machine code that is interpreted by the processor. Thus, from the processor point view the aim of the software runtime adaptation is to generate a machine code with special extensions.

Goals

The generated machine code with extensions has to achieve two goals: a) change the computational control flow between the processor and hardware accelerator and b) ensure proper communication between these two devices. In order to perform these tasks, necessary changes in several modules of the Woolcano compiler were performed.

Outline

In the following sections, a draft of major changes to the Woolcano compiler is described. These changes provide with new compiler's feature that we name *pathway*. Afterwards, we describe Communication Pass that is in charge of the pathway and we use a raytracing example as a case study.

6.1 Changes Overview

The Woolcano compiler is based on the LLVM framework infrastructure that is described in Appendix A. In Figure 6.1, the green color indicates the extensions that were performed to the compiler. The large green arrow represents the pathway that is built from these extensions. In addition, the red box is used to indicate the Communication Pass which controls the pathway in order to fulfill the goals described above.

Pathway

From an abstract point of view, the compiler extensions can be seen as a compiler pathway. The start of this path is at Communication Pass located in the middle-end whereas the end is at the machine code level. The modifications to the IR provided by Communication Pass are reflected in the generated machine code that is interpreted by the processor. In consequence, all compiler modules through the pathway leads have to support it.

Methods Similar to the Pathway

Other methods that try to achieve the pathway functionality are described in Appendix C. In contrast to the pathway, they try to achieve the same functionality from lower than the IR level. For instance, they provide changes only into back end. Fulfilling the goals with these methods rather than with the pathway is a much more challenging task.

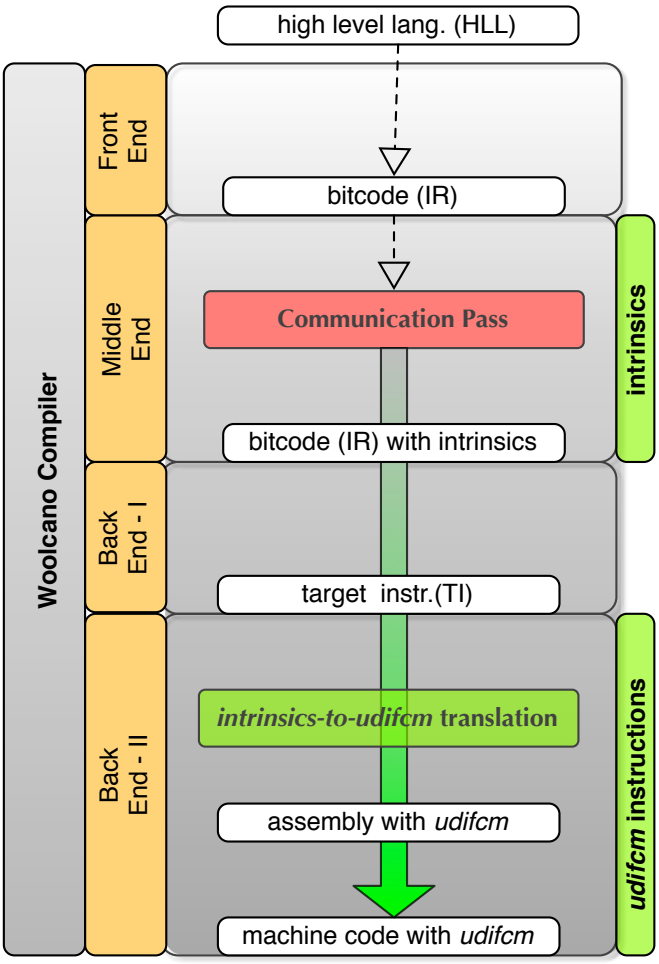

Figure 6.1: Changes to the Woolcano compiler that allow to generate a proper machine code with UDCIs.

6.2 Changes in Middle End

In the middlend of the compiler two changes were performed: an intrinsic support was added and the Communication Pass was designed and developed. The Communication Pass is an adaptive transformation that is in control of generating a machine code that fulfills the two described SRA goals. It uses intrinsically functions, which extend middle end's IR output interface. Instead of the intrinsic functions, this task can be also accomplished by adding new instructions to the IR ISA. Bellow we provide a short comparison between these two methods. The Communication Pass is described in a separate Section 6.4.

New IR Instruction vs. New Intrinsic Function

Adding a new intrinsic function is an easier, safer, and non pervasive way comparing to extending the IR ISA with new instructions, which result in a large negative impact on the middlend optimizations, transformation, and analysis passes. Either this would cause errors during passes runtimes or would outcome in an invalid IR code being generated. In order to fix these issues, each optimization and pass would have to be upgraded to support these new instructions. Since there are almost one hundred of them this task would require a huge effort and significant labor time.

Instead of this approach, we decided to use the intrinsically methodology that allows to achieve the same goals without these troublemaking issues. The middlend sees the intrinsic as a special kind of an IR function call. Thus, the functionality of all middlend optimizations is preserved.

Intrinsic Function

Intrinsic is a special function that appears in the IR code and that is treated in a different way than the normal function. All intrinsics have to be registered in the compilers backends and appropriate lowering mappings have to be provided. Once intrinsic is picked-up by the back end, these mappings are utilized and proper lower level code is generated. Intrinsic as well as the normal function are invoked with the `call` method and they can be parametrized with arguments.

For the convenience matters, we provide with two separate intrinsic functions that are named `llvm.fpga.exec()` and `llvm.fpga.write()`. Each intrinsic fulfills one of the SRA goals. The first allows to transfer the control flow from the processor to the hardware accelerator and is associated with the first SRA goal whereas the second one ensures proper communication and corresponds to the second SRA goal.

Implementation Details: Intrinsic Function

The following excerpt from the target machine description shows how the `llvm.fpga.exec2` intrinsic was added to the compiler. After declaring the intrinsic name `llvm.fpga.exec2` in lines 1+2, line 3 declares that it has one output and two inputs, all of type 32bit integer. Finally, the intrinsic is marked to have a side effect by setting the `IntrWriteMem` attribute. For the other intrinsics, similar declarations were used.

```
1. let TargetPrefix = "fpga" in {
2.  def int_fpga_exec2 : Intrinsic<
3.    [llvm_i32_ty], [llvm_i32_ty, llvm_i32_ty],
4.    [IntrWriteMem]>;}
```

6.3 Changes in Back End

During the target dependent code generation in the second back end, intrinsics are mapped onto corresponding assembly instructions. To this end, we extended the PowerPC backend with UDCIs. Bellow we present details that allow to achieve this task.

Implementation Details: UDCIs at Assembly Level

The following code snippet from the machine description shows how the machine instruction `udi2fcm` was added to the backend. Lines 1 and 2 declare the name, format of the machine instruction (`XForm_APU_RC`), and the opcode of the instruction. The opcode of the instruction is used during the JIT compilation to generate the proper machine code whereas in other situations only the mnemonic `udi2fcm` of the instruction is used to generate the assembly code. Line 3 declares that the instruction writes one and reads two general purpose registers (GPRs) . Line 4 defines how the instruction is printed in the assembly language. Finally, lines 5 and 6 specify a DAG pattern which is used during the instruction selection. The reference to the name of the `llvm.fpga.exec2` intrinsic instructs the compiler to issue `udi2fcm` instructions whenever this intrinsic is found during the code generation.

```
1. def UDI2FCM : XForm_APU_RC
2.    <4, 611,
3.    (outs GPRC:$rT), (ins GPRC:$rA, GPRC:$rB),
4.    "udi2fcm $rT, $rA, $rB", IntGeneral,
5.    [(set GPRC:$rT,
6.       (int_fpga_exec2 GPRC:$rA, GPRC:$rB))]>;
```

6.4 Communication Pass

The Communication Pass (CP) is represented with red box in Figure 6.1. The CP is plugged into the compiler at the middlend and is responsible for utilizing the developed pathway in order to fulfill the SRA goals.

Algorithm Overview

The communication pass is implemented as a compiler adaptive transformation. It locates the given function in the IR and rewrites this function in a way that fulfills the SRA goals. This behavior is illustrated in the raytracing example in Figure 6.2.

First, a list of all function parameters and their data types is constructed. PowerPC processor requires that all data sent from the APU to the FCM are integer values taken from the general-purpose register file. Data that are not passed via the general purpose register file, e. g.,

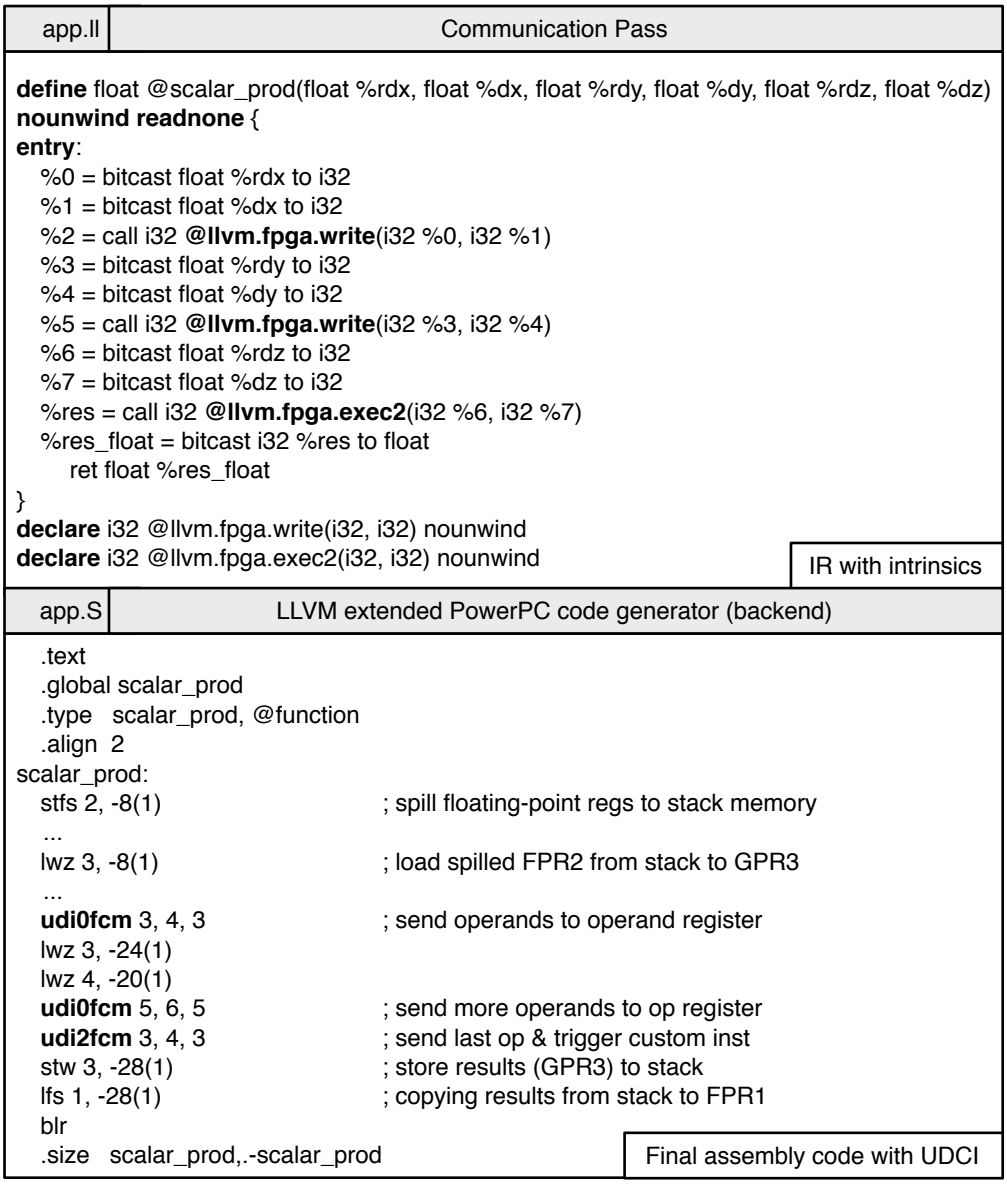

Figure 6.2: The outcome of the Communication Pass (f) and the corresponding machine code (g) when used for function presented in Figure 5.6.

floating-point data which are passed in floating-point registers, are casted to integers. This implicitly transfers them to the GPRs during the code generation. After this optional casting, the data are sent to the FCM controller (in slices of two operands) using previously described `llvm.fpga.write` intrinsic. Now the UDCIs can be issued with the `llvm.fpga.exec` intrinsic and finally, the result of the UDCIs are returned after another optional casting operation to the floating-point data type.

6.5 Raytracing Example

A demonstration of the feasibility of the CP is presented in Figure 6.2(f) for the floating-point scalar product operation shown in Figure 5.6.

Since the input function uses the floating point data types, necessary casting is performed with `bitcast` instruction for all six input arguments. These casted arguments are transferred to the hardware accelerator with the help of two `llvm.fpga.write` and one `llvm.fpga.exec` intrinsics. These intrinsics besides transferring the last two arguments trigger the hardware accelerator execution. The outcome is then casted back to the floating point data type and is returned by the `scalar_prod` function.

The assembly code that is generated by the extended LLVM PowerPC backend is shown in Figure 6.2(g). To make the code easier to understand, it has been generated without optimizations. The casting of floats to integers results in spilling the floating-point registers to the stack and loading these values back to GPRs. The data transfer and instruction execution intrinsics are translated to `udi0fcm` and `udi2fcm` assembly instructions, respectively.

6.6 Work Related to Woolcano Compiler

The RC hardware architectures have different designs, features, and limitations. Therefore, typically a single compiler is developed for the particular hardware architecture, which is also the case for the Woolcano compiler. The exception to this rule is LISA [79] which is retargetable compiler able to map C constructs for arbitrary hardware architecture. This task is performed with the usage of the architecture description language.

Overview

In this subsection, the latest compilers targeting architectures similar to the Woolcano are presented. This includes the Chimaera [30] and the commercial Stretch S5 compilers [49]. Complete survey of compilers targeting reconfigurable architectures is presented in Cardoso *et al.* [80]. Both compilers are built upon the *gcc* compiler suite framework which is rather rare approach because gcc is generally well known for being difficult to efficiently adapt to the embedded processor designs.

Stretch S5

The commercial Stretch S5 is the ongoing product started around year 2005 by a company named StretchInc whereas the Chimaera has roots in the late 90th and was mostly studied at

the beginning of the new millennium. Both compilers target the C language and while the S5 compiler is a proprietary system the design details of the compiler are unrevealed to the publicity. The difference between S5 and Chimaera compilers rely in the way the software is exposed and mapped to the RFUs. In the case of S5, this is performed manually within the help of C dialect whereas, in the case of Chimaera, this is performed automatically.

The S5 C dialect allows for the notion of bit width types and operations. These extensions are used by a programmer to mark and then rewrite the fragment of the source code amenable for the RFU. The generation of the RFU as well as the corresponding UDCI invoking the RFU is handled automatically by the compiler.

Chimaera

In the case of Chimaera, the gcc compiler suite was adapted with the *control localization* method which combines several basic blocks into a single larger one and with the *SIMD within single register - SWAR* methodology. This allows to increase the chances of finding better UDCI candidates by increasing the ILP in the basic block as well as use MMX alike extensions if the data can be compressed into the single register. The control localization method is similar to the super block formation [81, 82] approach. From this perspective the predicate execution [83] and the hyperblock formation [84] approaches have the same aims but they differ in the used methodology.

The Chimaera compiler has a fixed UDCI identification algorithm which is able to automatically spot the multiple input single output (MIMO) regions suitable for the RFU implementation. These correspond to the architectural constrains which in that case allow to read nine operands from shadow register file and to generate a single output. Thus, the *control data flow graph* (CDFG) of the UDCI only can take the form of *direct acyclic graph* (DAG). The Chimaera tool flow does not implement the RFU automatically and the feasibility of this approach was tested only in the simulation environment.

Runtime Support

In contrast to the Woolcano compiler, both compilers are working offline and they do not allow to use online methods in particular the JIT functionality to guide the process of ISA extensions.

Part III

Evaluation and Conclusions

Chapter 7

Woolcano Hardware Architecture

In this chapter, we evaluate the Woolcano hardware architecture described in Chapter 3. As a case study, we use raytracing application that was presented in Section 5.2 and was evaluated in the previous chapters. We provide with a comparison of the performances of the floating-point `scalar_prod` operation shown in Figure 5.6 for three implementation variants:

1. using software floating-point emulation,
2. with a general APU-attached floating point FPU,
3. with a dedicated UDCI; see Figures 5.8[1] and 6.2.

7.1 Configuration

The case study was implemented on a AVNET PCI Express development board that features a Virtex-4 FX100 FPGA. The Xilinx FPU core v3.0 (full variant) was attached to the APU interface, which allowed to compare the performances and area usage with the UDCIs.

Frequency, Resources, and Tools

The Woolcano architecture (including the PowerPC core) executed at a frequency of 100MHz and the caches of the PowerPC were enabled. The Xilinx FPU executed at half the system clock, as required by the FPU IP core whereas the DDR-333 SDRAM at a frequency of 200MHz. The FCM controller used 77 slices and the system used 9201 slices in total (21%). The hardware was generated using Xilinx ISE/EDK version 9.2i with SP4, IP update 2, EAPR, and Bus Macros (24.08.2008) version 8 on Fedora Linux.

Software Binaries

The software binaries were compiled with optimization level setup to `O3` and produced a code of size 55587 bytes for the floating-point software emulation and 54879 bytes for the Xilinx FPU IP core. This allowed to place binaries into the BRAM memory which had 64kB size with the heap and stack size setup to 4kB each.

[1] With the difference to the data types. The code presented in this figure uses `unsigned int` data types whereas for this evaluation we used `floating points`.

Architecture	Memory	cycles	speedup vs. sw. emulation	speedup vs. APU FPU
SW floating-point emulation	BRAM	542'562	1.0	0.04
Xilinx APU FPU v3.0		19'660	27.60	1.00
UDCI		15'166	35.77	1.30
SW floating-point emulation	SDRAM	1'051'629'842	1.0	0.03
Xilinx APU FPU v3.0		31'752'386	33.12	1.00
UDCI		26'307'650	39.97	1.21
ICAP initialization	–	18'128	–	–
Reconfiguration	–	17'604'700	–	–

Table 7.1: Performance of the scalar product benchmark for computing 80000 scalar products and reconfiguration time for an instruction slot of size 213096 bytes.

UDCI Slot and Bitstream

The Woolcano architecture was configured to a single UDCI instruction slot featuring six 32-bit inputs and a single 32-bit output. The instruction slot had a capacity of 2400 slices and comprised 12 DSP48, 24 FIFO16, and 24 RAMB16 blocks.

The UDCI bitstream was generated with the ASIP-SP. The `scalar-prod` UDCI had used 4 registers (1 cycle latency), 3 FP multipliers (4 cycles latency), and 2 FP adders (5 cycles latency). This resulted in 17 cycles latency in total and a total area of 1879 slices (4%) and maximum frequency of 260MHz. The hardware operators correspond to the ones found in the Xilinx Coregen (float-point operator library v.4.0) and were delivered from the PivPav circuit library.

Xilinx FPU IP core

The Xilinx FPU IP core consumed 1624 slices (3%) and 4 DSP48s (2%). The latency of the FPU operations was not customizable thus it was impossible to pipeline them manually. This resulted in a fixed total latency of 25 cycles (5 cycles for multiplication and 10 cycles for accumulate).

7.2 Benchmarking

For benchmarking purposes, the time was measured for the case when the `scalar_prod` function was executing 80000 times. The measurement excluded the time needed for the system and input data initialization and was performed for two different variants. When the software including the data and the instruction code was placed in either (a) the 64kB on-the-chip BRAM or in (b) the external off-the-chip SDRAM memory.

7.3 Results

Table 7.1 summarizes the results of the benchmark. Compared to the software floating point emulation, the use of the FPU provides a speedup of $27.6\times$ ($33.12\times$) for BRAM (SDRAM) implementation and the UDCI instruction provides a speedup of $35.77\times$ ($39.97\times$). The last column shows that the UDCI instruction outperforms the general FPU core by $1.3\times$ for BRAM and $1.21\times$ for SDRAM implementations.

7.3.1 BRAM vs. SDRAM

The software floating point emulation executes almost $2000\times$ slower when the software is placed in the SDRAM versus BRAM even with enabled caches. Similar ratios between SDRAM and BRAM memories are achieved for the UDCI and APU FPU case studies.

7.3.2 Reconfiguration Time

The time needed for reconfiguring the instruction slot comprises the initialization of the ICAP interface, which takes 18128 cycles, and the reconfiguration process itself dependent on the size of the bitstream. For the setup of the case study, the reconfiguration time for the UDCI slot with a bitstream size of 213'096 bytes is 17'604'700 clock cycles, which corresponds to 176ms at 100MHz and 12.09 bytes per clock cycle on a 32-bit interface. This time can be reduced by more than an order of magnitude by improving the ICAP port interface [85].

7.4 Conclusions

The Woolcano hardware architecture allows to effectively adapt the ISA of the CPU with UDCIs. The results show that this approach is feasible and that the communication latencies are sufficiently low to obtain speedups even for instructions with a moderate complexity.

Communication Bottleneck

One can also notice that the bandwidth of the communication interfaces, including the APU, limits the performances. The UDCI instruction can compute a scalar product in only 13 cycles, while the FPU requires 25 cycles running at half the clock frequency (50 cycles in contrast to UDCI), i. e., potentially a higher speedup could be achieved (up to $4\times$). Increasing the frequency of the FPU will not provide any changes in this ratio, while the Woolcano architecture can run at the frequency of system, constant ratio of 2:1 between the FPU and the rest of the system will be always achieved. Thus, it is expected that the UDCI implementation will outperform the APU FPU by a higher rate when removing the memory interface bottleneck. Unfortunately, the Virtex4FX architecture does not support pipelining for the `udifcm` and the floating-point instructions, hence it is not possible to provide better communication mechanisms.

Memory Access Latency

While the BRAM is an on-chip memory with access latency of a single clock cycle the access to the off-chip SDRAM varies between 22 and 33 cycles. This points out that in order to effectively perform computations it is necessary to decrease the latency of the memory interface as much as possible. The other solution is to decrease the amount of data and memory transfers. This arguments is often used agains the instruction-stream based von Neumann architectures.

Chapter 8

Experimental Setup

While this work targets the reconfigurable ASIPs, for example as our Woolcano architecture presented in Figure 8.1, it is currently not feasible to execute the complete ASIP-SP on an embedded reconfigurable ASIP architecture due to practical limitations. The ASIP specialization process heavily uses the LLVM compiler framework and the Xilinx ISE tools which require high-performance CPUs and desktop operating systems. These resources are not available in the currently existing ASIP architectures. Hence, we used Linux and Dell T3500 workstation (dual core Intel Xeon W3503 2.40GHz, 4M L3, 4.8GT/s, 12GB DDR3 / 1333MHz RAM) as a host computer in place of the PowerPC 405 CPU of the Woolcano architecture to execute the ASIP-SP; see Figure 8.1. This shift improves the processor performances roughly $40\times$ from $0.6 - 0.7k$ to $27k$ Dhrystone Million Instruction Per Second (DMIPS) .

The lack of the possibility to run the complete tool flow on the ASIP has a number of consequences for the experimental evaluation. Instead of running the ASIP-SP as a single process, we are forced to spilt this process into two steps. In the first step, the host computer is used to generate the partial bitstreams by executing the tasks corresponding to the upper half of Figure 8.1. In the second step, we switch to the Woolcano architecture where we use the generated bitstreams to reconfigure the UDCI slots and to measure the performance improvements.

It is also worth noticing that this two-step process has an impact on several reported measurements. First, all performance measurements reported in Tables 11.1 and 12.1, in columns *Max ASIP-SP speedups* and *ASIP ratio*, are performed for Woolcano's PowerPC405 CPU and not for the host CPU. Further, in order to compute the *break-even* time reported in Table 12.1 we used the *runtime overheads* values from the same table which were measured on the host computer. Therefore, this value is computed as if Woolcano's PowerPC CPU had the processing power of the host machine. Finally, while the ASIP specialization tool flow is capable of performing UDCI reconfiguration during runtime, in practice, we had to switch from the first to the second step manually.

The hardware limitations of Woolcano, in particular the number of the UDCI slots, in practice do not allow us to measure the performance improvements on a real system for all benchmarking applications presented in Chapter 9. Therefore, for these applications we estimate the speedups with the help of the techniques presented in Section 5.5.

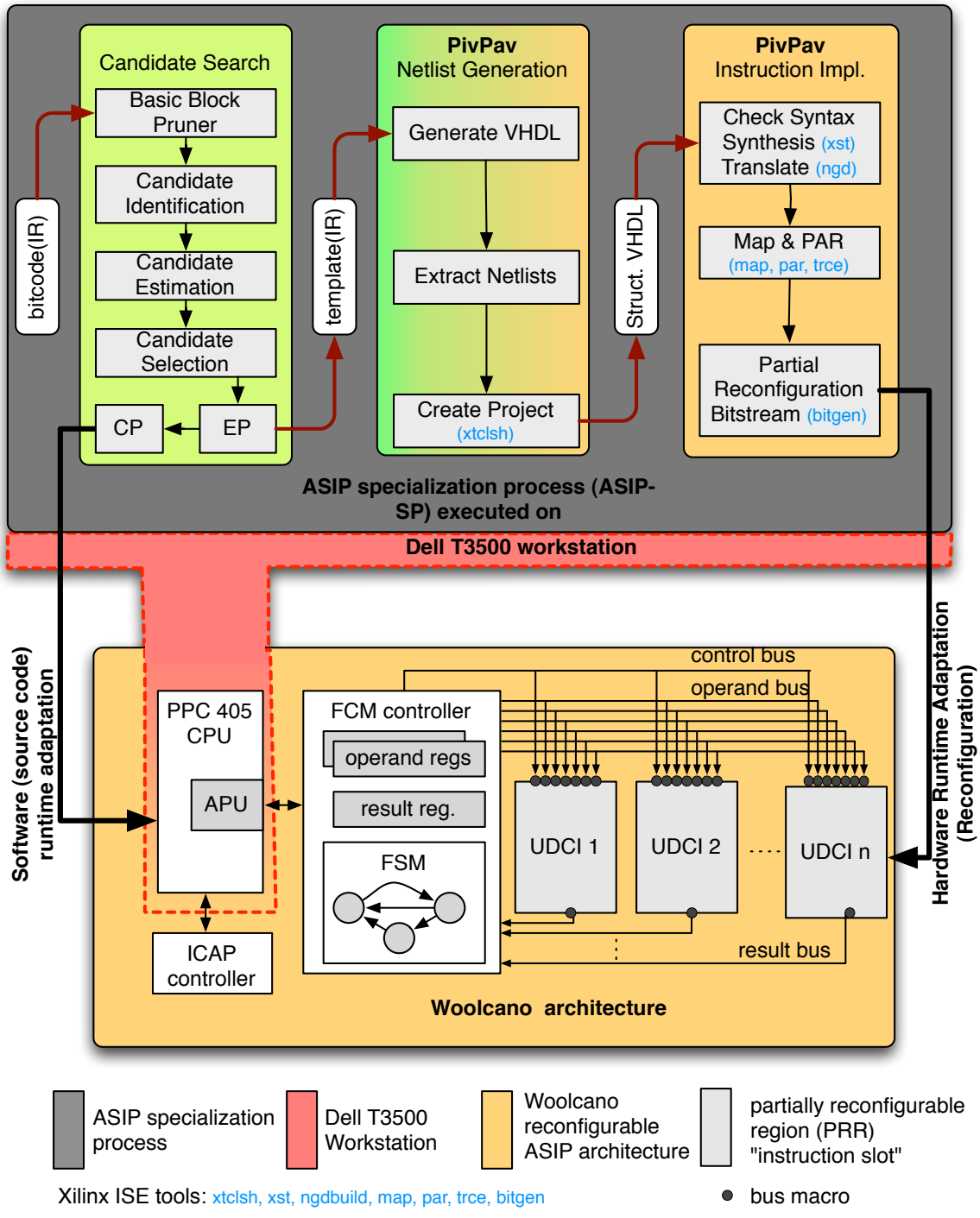

Figure 8.1: Overview of the developed tool flow and the targeted Woolcano hardware architecture. During the experimental evaluation, instead of a PPC405 CPU, the ASIP specialization process was executed on a Dell T3500 workstation.

Chapter 9

Applications for Experimental Evaluation

Table 9.1 presents the characteristics of used applications divided into two groups. The upper part of the table shows data for applications obtained from the SPEC2006 and SPEC2000 benchmark suites which represent the scientific computing domain whereas the lower part represents applications from the embedded computing domain obtained from the SciMark2 and MiBench. While the used benchmark suites count 98 different applications altogether, we could not run our evaluation on all of them due to cross-compilation errors. Hence, from the set of available applications, we selected the ones which are the most representative and allow us to get comprehensive insights into the JIT ASIP specialization methodology.

Outline

There are six sections in this chapter. Each section corresponds to a column in Table 9.1 and provides with results description.

9.1 Source Size

The second and third columns of Table 9.1 contain the number of *source files* and *lines of code* and tell that the scientific applications have on average $23.89\times$ more code than the embedded applications. This difference influences the *compilation time* shown in the fourth column which for scientific applications is $28.22\times$ longer on average, but still the average compilation time is only 6.5s.

9.2 Compilation to IR

The next three columns express the characteristics of the *bitcode* reflecting the total number of *functions*, *basic blocks*, and *intermediate instructions*, respectively. For the scientific applications, the ratio between *ins* (13065) and the *LOC* (6874) is 1.9,which means that an average single high-level code line is expressed with almost two IR instructions whereas less than one (0.8) for the embedded applications. Since the scientific applications have 24 times more LOCs than the embedded applications, this results in a $57\times$ difference in the IR instructions.

App	Sources		Compilation to IR				IR in BBs			Code Coverage			Kernel Size			Execution runtimes		
	files	loc	real	fun	blk	ins	max	avg	udci	live	dead	const	size	ins	freq	VM	Native	Ratio
	#	#	[s]	#	#	#	#	#	[%]	[%]	[%]	[%]	[%]	#	[%]	#	#	x
164.gzip	20	8605	3.89	33	1006	6925	59	6.88	29.68	38.86	44.66	16.48	5.78	400	90.34	23.71	18.47	1.28
179.art	1	1270	1.06	21	376	2164	43	5.76	21.53	42.05	28.47	29.48	9.84	213	92.45	69.92	74.70	0.94
183.equake	1	1513	1.71	15	257	2670	132	10.39	23.0	75.39	8.91	15.69	15.32	409	92.9	7.97	6.79	1.17
188.ammp	31	13483	10.10	98	4244	26647	382	6.28	25.74	19.22	70.89	9.89	3.38	901	95.81	23.18	17.24	1.34
429.mcf	25	2685	0.97	18	284	1917	77	6.75	13.09	75.9	13.09	11.01	25.77	494	98.46	23.94	24.06	1.00
433.milc	89	15042	10.88	87	1538	14260	363	9.27	32.59	61.67	34.72	3.61	10.83	1545	93.99	20.95	16.43	1.28
444.namd	32	5315	22.77	84	5147	47534	291	9.24	37.82	31.71	62.81	5.48	7.33	3486	93.64	39.94	34.31	1.16
458.sjeng	23	13847	8.49	86	3373	20531	69	6.09	21.1	48.49	49.44	2.07	44.6	9157	100.0	180.41	155.66	1.16
470.lbm	6	1155	1.36	16	104	1988	405	19.12	57.55	55.23	24.9	19.87	32.75	651	97.57	5.68	5.36	1.06
473.astar	19	5829	3.68	45	757	6010	70	7.94	27.45	78.79	5.31	15.91	6.39	384	91.3	66.00	67.68	0.98
AVG_S	24.70	6874	6.49	50	1709	13065	189.1	8.77	28.95	52.73	34.32	12.95	16.20	1764	94.65	46.17	42.07	1.14
adpcm	6	448	0.29	6	43	233	39	7.21	33.48	60.66	29.18	10.16	41.97	128	91.79	29.22	28.35	1.03
fft	3	187	0.26	10	47	297	41	6.53	42.09	58.88	30.26	10.86	44.08	134	95.98	18.47	18.49	1.00
sor	3	74	0.13	4	19	99	22	7.06	34.34	46.51	50.39	3.1	24.81	32	97.52	15.83	15.85	1.00
whetstone	1	442	0.25	12	44	285	32	6.58	34.04	32.75	36.27	30.99	10.21	29	93.27	28.66	28.50	1.01
AVG_E	3.25	288	0.23	8	38.3	228.5	33.5	6.85	35.99	49.70	36.52	13.78	30.27	80.75	94.64	23.04	22.80	1.01
RATIO	7.60	23.89	28.22	6.29	44.67	57.18	5.64	1.28	0.80	1.06	0.94	0.94	0.54	21.85	1.00	2	1.85	1.13

Table 9.1: Characteristics of the scientific and embedded applications used in experiments. AVG-S represents the averages for scientific applications and AVG-E for the embedded applications. Ratio = AVG-S / AVG-E.

9.3 IR in BBs

The ISE algorithms operate on the BBs and thus the *IR in BBs* column indicates the characteristics of these BBs in more detail. The *max* column indicates the BB with the highest number of the IR instructions and *avg* is the average number of the IR instructions in all BBs. These values in combination with the data presented in Figs. 11.2, 11.1, and 11.3 allow to understand the *runtime* of the ISE and the *number of candidates*.

For the embedded applications, the largest BBs cover on average more than 14.7% of the application whereas the largest basic block for the scientific applications covers only 1.4% of the total application. The difference between average-size BBs for the embedded and scientific applications is $1.28\times$ and results in a small average number of the IR instructions of less than 10 for both cases. The small size of BBs in our applications needs to be attributed to the actual benchmark code, the compiler, and the properties of the intermediate representation. Our experiments have shown that the size of the BBs does not change significantly for different compiler optimizations, transformations, or with the size of the application (LOC).

Feasible UDCI Instructions

The *udci* column lists the percentage of all the IR instructions which are feasible for a hardware implementation. Feasible instructions include the arithmetic, logic, and cast instructions for all data types and make up to 1/3 of all instructions of the application. Considering the small average size of BBs this means that the size of an average found candidate is only between 2 and 3 IR instructions. This emphasizes the need for a proper BB and candidate selection and stresses even more the importance of the proper pruning algorithms in oder to avoid spending time with analyzing candidates that will most likely not result in any speedup.

9.4 Code Coverage

The *Code Coverage* columns show the percentages of the size of *live*, *dead*, and *constant* code. These values were determined by executing each application for different input data sets and by recording the execution frequency of each BB. For the SPEC benchmark suite applications, the standard *test, train*, and *ref* data sets were used, whereas for the embedded ones, due to the unavailability of standard data sets, each application was tested with at least three different custom prepared input data sets. After execution, the change in execution frequency per block between the different runs was compared. If the frequency was equal to 0, the code was marked as *dead*. If the frequency was different from 0 but did not change for different inputs, the code was marked as *constant*; and if the frequency has changed, the block was marked as *live*. These frequency information were used to compute the *break-even points* in the following section. In addition, the *live* frequency information indicates that roughly only 50% of the application has a dynamic behavior in which the ISE algorithms are interested in searching for candidates.

9.5 Kernel Size

The next three columns contain data on the size of the kernel of the application and are derived from the frequency data. The kernel of an application is defined as the code that is responsible for more than 90% of the execution time. The size of the kernel is measured as the total number of IR instructions contained in the basic blocks which represent the kernel. For the scientific applications, 16.20% of the code affects 94.65% of the total application execution time and it corresponds to more than $1.7k$ IR instructions. For the embedded applications, the average relative kernel size is 30.27% and is expressed only with 80 IR instructions. These numbers indicate that it is relatively easier to increase the performances of the embedded applications than the scientific ones, since they require $22\times$ smaller UDCI instructions.

9.6 Execution Runtimes

The *VM* column in Table 9.1 represents the application runtime when executed on the LLVM virtual machine. The runtime of the application depends heavily on the input data which, in the case of the scientific applications, were obtained from the *train* datasets of the SPEC benchmark suite. Due to the unavailability of standard data sets for the embedded applications, custom-made data sets were used. For both application classes, the input data allowed to exercise the most computationally intensive parts of the application for a few or several tens of seconds. The *Nat* column shows the *real* runtime of the application when statically compiled; that is, without the overhead caused by the runtime translation. The *Ratio* column shows the proportion of *Nat* and *VM* and represents the overhead involved with the interpretation during the runtime. For the small embedded applications, the overhead of the *VM* is insignificant (1%). For the large scientific applications, the average overhead caused by the VM equals on average 14%. However, it is important to note that for some applications like *179.art* or *473.astar*, the *VM* was significantly faster than the statically compiled code by 6% and 2%, respectively. This means that the *VM* optimized the code in a way which allowed to overcome the overhead involved in the optimization as well as the dynamic just-in-time compilation.

Chapter 10

Basic Block Pruning

In this chapter, we evaluate the Basic Block Pruning algorithms described in Section 5.3. The results of the pruning algorithms are summarized in Table 10.1. They are provided for applications found in Table D.1. These applications are almost identical to the benchmarking applications presented in Chapter 9. Since the sizes of applications differ significantly, the chances for finding more pronounced computation kernels are higher for the smaller applications than for the larger ones. Therefore, they were grouped in two columns with embedded and SPEC2006 applications.

10.1 Ratio Values

The values found in the Table D.1 represent the ratio of the average speedup to the required identification times. This ratio is presented in Section 5.3 and is computed for two whole sets of applications, both embedded and scientific. Higher ratio values mean that a given algorithm achieves better results to identification time which is preferable for systems with the just-in-time CPU specialization that requires a balance between both figure of merits.

10.2 Best ISE Algorithm for JIT System: MM

The advantage of MM (36.69, 7.50) over SC (20.76, 1.48) and UN (6.95, 2.20) is clear, especially for the smaller embedded applications, where an average speedup of $3.71\times$ is achieved in 1.82ms in total. This is a $8.86\times$ improvement over the case where `no-pruning` is used and $4.16\times$ better as the `3(F/S)` algorithm. For the latter case the average speedup is $5.05\times$ and $2.06\times$, respectively, with an identification time of 175.75ms and 19.74ms. Thus, for the online system where reduction of the identification runtime is as important as the speedup, the MM linear complexity ISE algorithm delivers the best performances.

10.3 Best Pruning Algorithm for JIT System: @50pS3L

The identification runtime of MM was shortened for `@50pS3L` over `no-pruning` almost by two orders of magnitude ($96.73\times$) which caused the relative speedup to decrease only

Algorithm	Embedded applications			SPEC2006 applications		
	MM	SC	UN	MM	SC	UN
1L	7.61	2.84	0.41	0.22	0.21	0.16
2L	9.59	1.81	0.66	0.18	0.29	0.10
3L	15.68	1.17	1.03	0.08	0.28	0.05
4L	12.70	1.00	1.01	0.07	0.27	0.05
@1L	9.06	4.31	0.53	0.82	0.25	0.53
@2L	14.99	4.17	1.07	0.82	0.45	0.53
@3L	22.90	2.11	1.51	0.32	0.41	0.20
@4L	19.95	2.11	1.51	0.25	0.37	0.14
50pS1L	9.45	1.75	0.65	0.03	0.30	0.08
50pS2L	13.87	1.81	1.00	0.08	-	-
50pS3L	28.58	1.74	1.87	0.08	-	-
@50pS1L	8.92	0.67	0.61	0.00	0.24	0.45
@50pS2L	17.70	0.39	1.24	0.00	0.54	0.49
@50pS3L	**36.69**	0.72	2.34	0.01	0.62	0.19
1L2*	7.13	2.55	0.37	0.09	0.19	0.11
3L2*	10.09	1.00	0.66	0.03	0.18	0.03
1L3*	7.13	2.55	0.37	0.09	0.19	0.11
2F1S	14.99	19.80	2.04	6.89	0.68	1.86
3F1S	13.86	17.58	1.65	5.16	0.57	1.47
6F1S	20.28	12.86	1.21	3.08	0.33	0.79
6F2S	22.43	7.11	1.92	1.58	0.65	0.41
8F1S	19.39	10.84	1.18	2.40	0.30	1.18
2S1F	22.44	**20.76**	**6.95**	**7.50**	**1.48**	**2.20**
4S2F	18.86	11.62	2.27	4.22	0.87	1.25
5S3F	13.98	8.02	1.79	2.34	0.68	0.75
6S4F	11.41	5.84	1.53	1.50	0.56	0.47
tF	12.12	17.52	2.80	6.15	0.69	2.11
tS	6.02	1.67	0.38	0.05	0.19	0.08
1(F/S)	7.30	6.09	0.88	2.01	0.29	0.48
2(F/S)	9.88	3.47	0.82	0.89	0.30	0.29
3(F/S)	8.82	2.74	0.73	0.51	0.27	0.19
1S	13.01	11.63	1.21	3.07	0.40	0.92
2S	15.62	8.63	2.09	2.53	0.67	0.75
3S	13.74	7.25	2.04	2.13	0.63	0.65
1F	results equal to 2S1F					
2F	18.47	12.43	2.24	4.49	0.85	1.31
3F	14.34	9.20	1.80	2.67	0.68	0.84
no-pruning	4.14	0.76	0.19	0.05	0.00	0.00

Table 10.1: Ratio of the speedup to identification time for the pruning algorithms (Table 5.1).

to 73.47%. This result clearly indicates the importance of the pruning algorithms in any specialization process. The correctness of our assumptions about the location of the best speedups in applications (`@50pS3L` vs `3(F/S)`) is supported by the fact that the identification time is shortened by $10.86\times$ while the average speedup is $1.8\times$ higher.

10.4 mFnS and mSnF algorithms

The `mFnS` and `mSnF` algorithms, as assumed in Section 5.3, can easily locate the most profitable BBs in the applications. The results for SC and UN are better when constraining the sizes first and the frequencies afterwards, that is `2S1F` instead of `2F1S`. In addition, the ratio of speedup to identification for these algorithms is maximal when targeting only a single BB with parameters `m = 2` and `n = 1` due to their exponential complexity. This is not the case for the linear complexity (MM) algorithm, which achieves the best results for cases with more than one BB, that is $n > 1$. The capability to process BBs faster is the reason why MM outperforms SC and UN when using the speedup over the identification time figure of merit.

10.5 Basic Blocks in Loops

Generally, the best ratios are obtained when applying the ISE algorithms only to a small number of BBs. This is problematic for the loop (`L`) algorithm which focuses on the loop-BBs because the amount of loop-BBs in an application is usually very high. Without an execution trace it is difficult to accurately identify the most frequently executed loop-BBs. In particular, for the exponential SC and UN algorithms it would be important to focus on a few or even a single BB to maintain a high speedup to identification time ratio.

10.6 Loop Algorithm: L

The `L` results for the SC and UN are not as good as for the other algorithms that make use of the frequency data (`nSmF`, `nFmS`), but they are still satisfactory. In particular for the MM algorithm which allows for targeting more than one loop-BB while keeping a good speedup to identification time ration the results are in the top range for the smaller embedded benchmarks where fewer loop-BBs exist.

Hence, the `L` algorithms work best with the linear complexity algorithms, where the inaccuracy of identifying loop-BBs is overcompensated by the fast runtimes. In particular this is the case for the `@` loop algorithms which show higher speedup to identification time ratios than the `L` algorithms that do not exclude BBs which are never executed.

The results for the `nLm*` algorithm are mediocre. Perhaps the applications do not have the necessary modular design or it was destroyed during the compiler optimizations.

10.7 Remaining Algorithms

It is interesting to see how well the `nF` and the `nS` algorithms work. The `nF` algorithm eliminates not executed BBs, which is a clear advantage over the `nS` algorithm, especially for $n = 1$. For $n > 1$, the difference is disappearing, since `nS` has a higher chance to identify used BBs. However, for such cases the `@npSmL` algorithm should be used as it performs even better.

The `mSnF` algorithm on average provides better results than the `S` and `F` separately and should be used therefore. Otherwise, `nF` should be chosen over the `nS` algorithm.

10.8 Conclusions

This chapter evaluated algorithms for pruning the design space for the just-in-time ASIP specialization process, a problem that has so far not been address in related work. The study showed that for JIT systems where reducing the runtime for identification is as important as speedup, linear complexity ISE algorithms deliver the best performance. The proposed algorithms are able to shorten the runtime for ISE identification by up to two orders of magnitudes at the cost of reducing the speedup by $1/4$.

Chapter 11

Candidate Identification

In this chapter, we evaluate and compare the ISE algorithms that were described in Section 5.4. These algorithms have been used for the UDCI candidate identification as presented in Figure 4.2. Our evaluation covers the runtime characteristics of the algorithms, the number of identified candidates for different architectural constraints, and the maximum gain in the application performance. The discussion is based on data presented in Table 11.1 that were obtained for the benchmarking applications introduced in Chapter 9.

Outline

This chapter consists of three sections which correspond to columns found in Table 11.1. These are: the ISE algorithm runtime, the candidates found, and Max ASIP-SP Speedup.

11.1 ISE Algorithm Runtimes and Comparison

The average ISE algorithm runtimes are presented in the 2nd to 4th columns of Table 11.1. As stated previously, the MM algorithm has a linear complexity and therefore is the fastest, resulting in a 0.22 s runtime for `444.namd`, which is the largest application. Due to its larger algorithmic complexity, the runtime of the UN algorithm should generally exceed the runtime of SC but this is not the case for applications which include specific types of BBs. For example, it took 3837 ms to process such a specific BB consisting of 55 nodes in the `adpcm` application, which is 99.15% of the overall runtime of the UN algorithm. In contrast, the same BB was analyzed by the SC algorithm in mere 4.7 ms.

Since both SC and UN have an exponential complexity, their runtimes are a few orders of magnitude higher than for MM. In average, MM is $96.94\times$ faster than the SC algorithm.

The identification times for BBs of similar sizes also vary significantly for the same algorithm since the number of candidates that need to be actually considered in a BB depends not only on the total size of the BB but also on the structure of the represented DFG, the number and location of infeasible instructions in the DFG, the architectural constraints, and other factors. For example, it took the SC algorithm 1707 ms to analyze a BB of `433.milc` with 102 instructions, while the analysis of a slightly larger BB in `470.lbm` with 120 instructions took only 76 ms, which is a $22.5\times$ difference in runtime. This example illustrates that it is

App	ISE algorithm runtime			Candidates found			Max ASIP-SP Speedup		
	MM	SC	UN	MM	SC	UN	MM	SC	UN
	[ms]	[ms]	[ms]	#	#	#	×	×	×
164.gzip	40.6	549.0	11170.0	1621	44177	43682	1.172	1.213	1.213
179.art	12.3	55.1	3350.0	371	3534	3513	1.526	21.414	21.414
183.equake	13.5	457.9	4351.0	672	9690	9690	2.147	25.972	25.972
188.ammp	145.7	15840	-	7547	122441	-	3.449	20.826	-
429.mcf	11.1	68.7	200.5	571	3571	3562	1.112	1.112	1.112
433.milc	78.1	5065	-	3573	59450	-	1.301	21.546	-
444.namd	227.5	35854	-	11490	125970	-	1.609	24.846	-
458.sjeng	123.7	6244.1	235195.7	5540	83173	83035	1.118	1.137	1.137
470.lbm	8.6	2777.1	-	490	18216	-	2.554	**44.622**	-
473.astar	33.4	914.8	303796653	1408	37025	32368	1.159	1.19	1.19
AVG_S	69.45	6783	30405092	3328	50724	17585	1.71	16.39	5.20
adpcm	1.7	15.0	3869.4	83	819	819	1.243	1.309	1.293
fft	1.6	9.7	33.1	87	553	552	3.1	14.413	14.413
sor	0.7	4.3	14.6	35	384	375	14.418	14.422	14.418
whetstone	1.6	9.5	64.0	69	435	435	**18.012**	18.012	**18.012**
AVG_E	1.40	9.62	995.27	68.50	547.75	545.25	9.19	12.04	12.03
RATIO	49.61	705.05	30549.59	48.59	92.61	32.25	0.19	1.36	0.43

Table 11.1: Specialization process executed for whole applications when targeting the Woolcano architecture without capacity constraints. The performance of the custom instructions has been determined with the PivPav tool. ISE algorithms: MM=MaxMiso, SC=SingleCut, UN=Union. SC and UN search spaces are both constrained to 4 inputs and 1 input.

impossible to accurately estimate the runtime of the exponential ISE algorithms SC and UN in advance when basing the estimation solely on the size of the BB.

It is worth pointing out that for keeping the search space and thus the algorithm runtimes manageable, we had to apply rather tight architectural constraints for the custom instructions (4 inputs, 1 output). When loosening these constraints, the execution times for the SC and UN algorithms rapidly grow from seconds to many hours.

Summary

The overall runtime characterization of the instruction identification algorithms is summarized in Figure 11.2 which plots the runtime of the different algorithms for varying BB sizes. Each data point represents an average which was computed by running the ISE algorithm 1000 times on each BB. The multitude of data points for a fixed BB size illustrates that the runtime of the same algorithm can vary over several orders of magnitude for BBs which have an equal size but differ in their structure as pointed out above. This effect is particularly strong for larger BBs where many variants of DAG structures exist.

For visualizing the overall behavior of the algorithms, we have also added the average runtime for the SC and the UN algorithms for each BB size. It can be observed that the variability for the UN algorithm is larger than for the SC algorithm. Another interesting observation is that although the SC and UN algorithms have a worst-case algorithmic complexity of $O(exp)$, their runtime is only polynomial $O(n^2)$ on average, which can be seen by comparing the blue

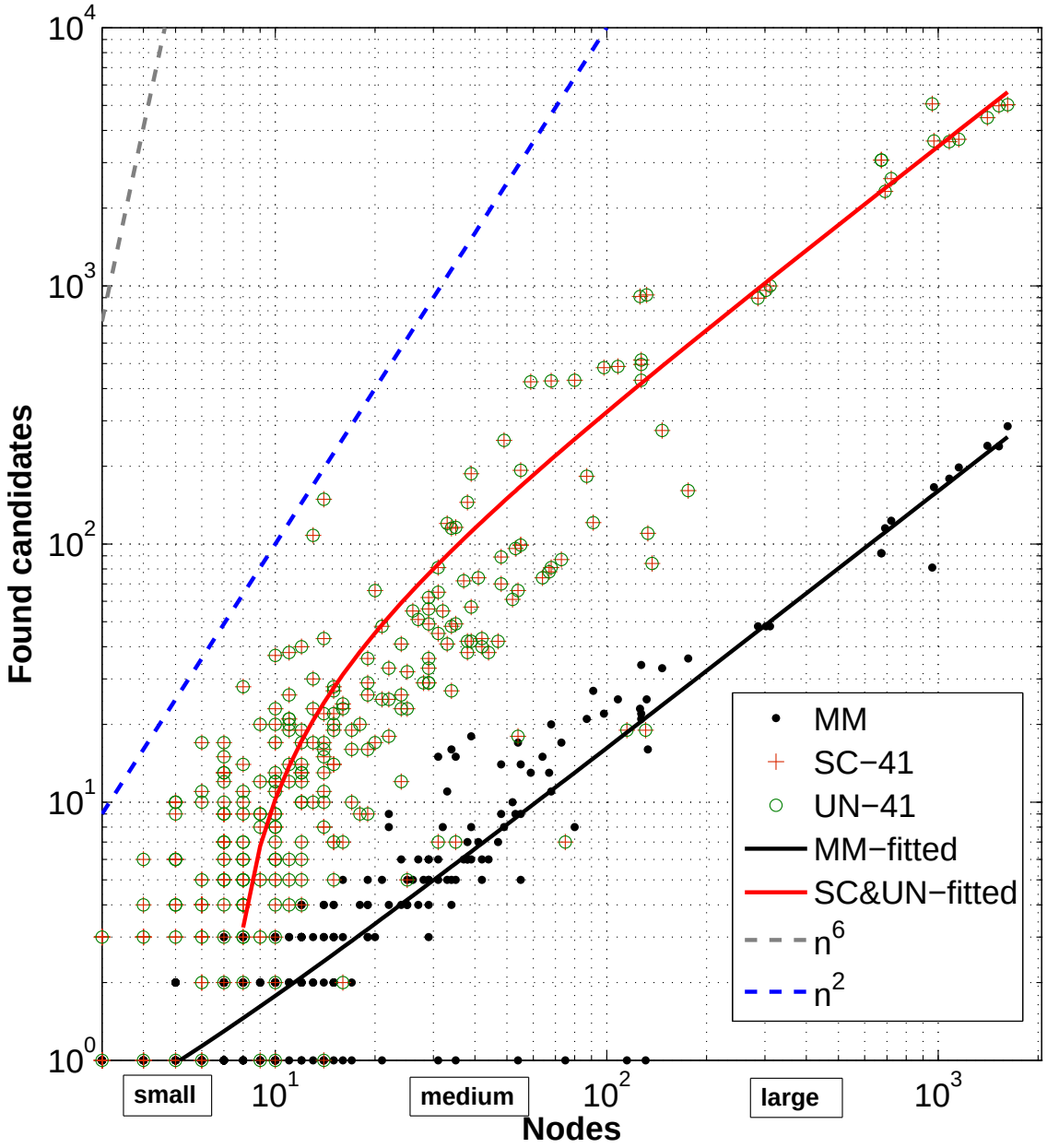

Figure 11.1: Number of found candidates by ISE algorithms for a large spectrum of BB nodes, where SC and UN are both constrained to 4 inputs and 1 output.

dotted lines with the red and green lines, respectively.

We are able to fit the runtime of the MM algorithm with a linear polynomial model which is represented with a black line which has an almost ideal characteristic (goodness of the fit: $R^2 = 0.9995$). This means that the runtime of the MM algorithm always depends linearly on the BB size $O(n)$. Unfortunately, the behavior of the other algorithms is not sufficiently regular to perform a meaningful curve fitting with similar quality.

Algorithms Runtime

In general, we can say that the MM is the fastest algorithm and outperforms the SC and UN algorithms easily in terms of runtime for small, medium, and large basic blocks. For small BBs of up to 10 instructions, the runtime difference is in the range of up to an order of magnitude; for medium inputs (10^2 instructions), up to two orders of magnitude; and for the largest BBs (10^3 and more instructions) a difference of more than three orders of magnitude can be observed. While the runtime of the MM stays on the millisecond time scale even for the largest inputs, the SC and UN algorithms work on a scale of seconds or minutes.

It is important to note that runtime of the exponential algorithms tend to literally explode when these algorithms are constrained less tightly than 4-inputs 1-outputs (41), in particular when allowing a larger number of outputs. For instance, when applying an 8-inputs 2-outputs (82) constraint, common runtimes are in the order of 10^8 ms, which is three orders of magnitude longer than for the slower 41 constraint.

In terms of runtime, SC is approximately one order of magnitude faster than UN. However, the runtime for both algorithms grows similarly for increasing BB sizes with the exception of significant outliers for the UN algorithm, for peaks with a runtime difference of three orders of magnitude can be observed for large BBs. A similar behavior was also found for architectural constraints other than 41.

Benchmarking Mode

The results presented here have been obtained using a special *benchmarking mode* of our tool flow where the instruction candidates are identified but not copied to a separate data structure for further processing. Additionally, the time needed to reject overlapping candidates for SC algorithm as well as the time needed for the MM algorithm to validate condition (5.2) presented in Section 5.4 were not included. As a result, the runtimes of the candidate identification algorithms will be in practice slightly longer when they are applied as a part of the complete tool flow.

11.2 Candidates Found by the ISE Algorithms

The number of candidates found by the ISE identification algorithms is presented in columns 5, 6, and 7 of Table 11.1. In addition, an overview of all identified candidates as a function of the BB size is shown in Figure 11.1, while Figure 11.3 presents a close up of the same data for medium-sized BBs. As illustrated by the red line in Figure 11.1, the SC and the UN algorithms generate an equal number of candidates, given that the same architectural

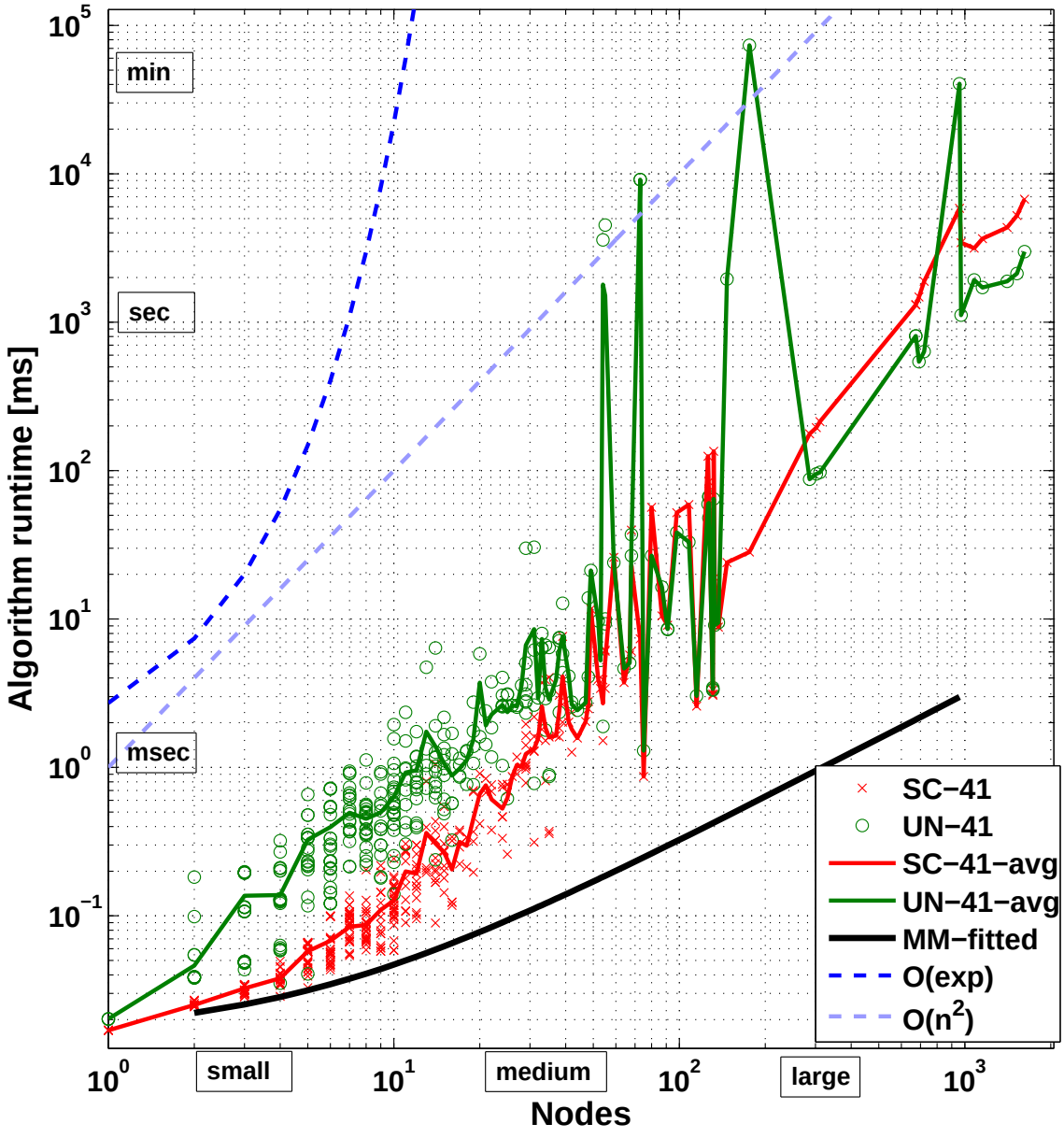

Figure 11.2: Runtimes of the ISE identification algorithm for different basic block sizes (nodes). SC=SingleCut, UN=Union, MM=MaxMiso algorithms. The label '41' means that the SC and UN search spaces have been constrained to 4 inputs and 1 output.

constraints are used and that any overlapping UDCI candidates generated by the SC algorithm are removed.

Theoretical Maximal Number of Candidates

In general, the total number of subgraphs that can be created from an arbitrary graph G is exponential $exp(n)$ in the number of nodes of G. For ASIP specialization scenarios, i. e., when architectural constraints are applied, cf.conditions (5.2), (5.3), and (5.5), [86] has shown that the number of subgraphs is bounded by $n^{(C_{\text{in}}+C_{\text{out}}+1)}$. Thus, for the 4-inputs/1-output constraints applied in this study, the search space is equal to n^6, which is represented by the gray dotted line found in the top-left corner of Figure 11.1. When applying the final constraint condition (5.4), the search space is significantly reduced from n^6 by at least a power of 4 to n^2, which is presented with the blue dotted line above all results.

Algorithm Characteristics Fitting

The *black* line represents a linear fitting for the MM algorithm, whereas the *red* line shows a second-order polynomial curve fitting for the SC and UN algorithms. The goodness of these fits represented with R^2 parameter is equal to 0.9663 for MM and to 0.9564 for the SC and UN algorithms. Therefore, it is safe to assume that the number of candidates for the 4-input/1-output constraint is limited by a second-order polynomial.

Runtime vs. Number of Candidates

Figure 11.3 shows that the longer runtimes of the SC and the UN algorithm also result in the identification of more candidates. The difference for small and medium BBs is up to one order of magnitude and increases for even larger BBs.

The data points (number of candidates) were obtained by running the ISE algorithms on the applications presented in Chapter 9. Thus, each data point is associated with a single BB and closely located data points tell that there are many BBs of similar sizes.

It can be observed in Figure 11.3 that there are less data points with large BBs than medium or small BBs. This is a consequence of the distribution of basic block sizes; i. e., most BBs found in these applications have sizes of up to 100 instructions. For such BBs, the number of feasible candidates reaches more than 10 when using the MM algorithm and more than 100 when using the SC and UN algorithms. Given that the average application has at least a few dozens (38 for embedded) or hundreds (1709 for scientific) of BBs this results in thousands of feasible candidates that are suitable for hardware implementation. In our experiments, we considered 3328 MM (50724 SC) candidates for the scientific and 68.5 MM (547.75 SC) candidates for the embedded applications. These high numbers are more than enough since an average ISA consists of around 80 core instructions for $x86$ platform and around 100 for PowerPC. If one assumes 10% modification to the ISA, it results in a task of selecting less than 10 UDCI instructions from a set of thousands of feasible candidates.

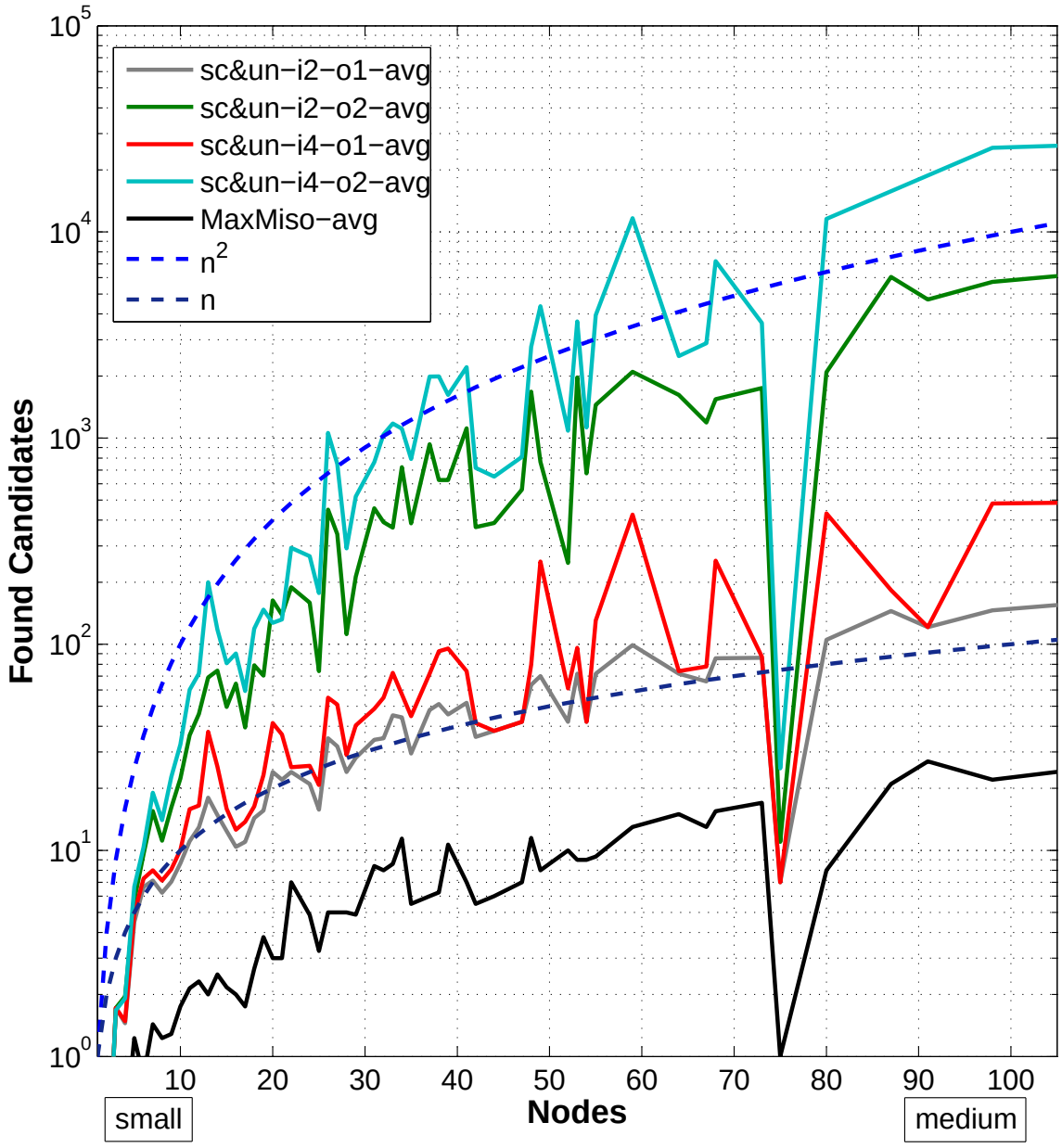

Figure 11.3: Number of found candidates by ISE algorithms for a medium spectrum of BB nodes, where SC and UN have variable constraints.

Constrains and Variant Number of Candidates

The number of found candidates depends strongly on the architectural constraints that are applied. Tighter constraints, i. e., allowing a smaller number of inputs and outputs, lead to a smaller number of candidates for the SC and UN algorithms. This behavior can be seen in Figure 11.3 where the average number of candidates is plotted as a function of the BB size for various constraints. There are two groups of constraints: the $21, 41$ and $22, 42$, between which a rising gap of one order of magnitude is established. Applying the MM algorithm to BBs with a size of 100 instructions leads to more than 10 feasible candidates whereas applying SC or UN leads to more than 100 candidates for the first set of constraints and even two-orders-of-magnitude more candidates (10^4) for the second set of constraints. This validates the second-order polynomial characteristic n^2 of the number of candidates for the SC or UN algorithms.

A similar behavior of the lines representing the average number of identified candidates is caused by the *less or equal* ($\leq$) relationships found in conditions (5.2) and (5.3). That is, the less-constrained algorithms (like 41) include all candidates of more constrained ones like 21. The area between the *red* and *gray* lines corresponds exactly to the number of additional candidates found in the less-constrained algorithms. Also, the graphs illustrate that the number of candidates depends much stronger on the number of allowable outputs than on the number of allowable inputs.

Decay

For BBs with sizes of approximately 75 instructions, we see an interesting decay from which all ISE algorithms suffer. This decay is found only in a concrete benchmark and is a result of a high concentration of illegal instructions in basic blocks of those sizes, for which only a few feasible candidates were found.

Algorithm Characteristics

Finally, it can be seen that the MM algorithm has a linear characteristic $a \cdot n$ where $a \leq 1$. The SC and UN algorithms also show a linear characteristic with $a \geq 1$ for the case of the 21 and 41 constraints, whereas for the 22 and 42 constraints, the characteristic changes to a quadratic one (n^2).

11.3 Theoretical Achievable Performance Gains

The *Max ASIP-SP Speedup* columns presented in Table 11.1 describe the upper limit of performance improvement that can be achieved with the Woolcano reconfigurable ASIP architecture and the presented ASIP-SP. These values show the hypothetical best-case potential in which all candidates found by three different ISE algorithms are implemented as UDCI custom instructions. In reality, the overheads caused by implementing all possible instructions and the limited hardware resources of the reconfigurable ASIP require pruning of the set of candidates that are evaluated and implemented to a tractable subset. Therefore, the speedup quoted in these columns should be treated only as an upper boundary on the achievable performance.

Theoretical Best Performance Improvement For Static System

The ISE algorithms have a lot to offer, reaching a speedup of up to 44.62x for SC algorithm and 18.01x for MM and UN algorithms. The average speedups achieved with MM, SC, and UN are 1.71, 16.39, and 5.20, respectively for the scientific applications and 9.19, 12.04, and 12.03 for the embedded applications. For all applications, the average speedups achieved with MM, SC, and UN are 3.85, 15.15, and 10.02, with the value of median 1.57, 16.22, and 7.85, respectively. These results clearly indicate that the SC algorithm is superior for static systems where identification runtimes are not a major concern.

Theoretical Best Performance Improvement For Runtime System

For a JIT specialization process, one needs to balance the achievable speedup with the identification time. Comparing the ratios of average speedup to identification runtime for the embedded applications results in the following ratios: 6.56 (MM), 1.25 (SC), and 0.01 (UN). These figures suggest that the MM algorithm is the most suitable for such systems. In addition, the considerable difference of 0.19 between average speedups for different application sets suggests that the MM algorithm could find better candidates in the smaller applications with more pronounced kernels and that these applications will benefit most from the JIT-based systems.

Performance Improvements and Used Technology

It is important to remember that these results were obtained for the first time for the FPGA-based Woolcano architecture and not as presented in related work for a fixed CMOS ASIP architecture. This distinction is significant since the same hardware custom instructions will achieve significantly higher speedups when implemented in CMOS technology, often by more than one order of magnitude. But at the same time, a fixed architecture will sacrifice the flexibility and runtime customization capabilities of the Woolcano architecture.

Chapter 12

Feasibility and Limitations of Just-in-Time Processor Customization

In this chapter, we study the feasibility and limitations of the just-in-time processor customization. While in previous chapters we evaluated only single components of the system, in this chapter we use all developed components in order to evaluate the whole system.

12.1 Introduction

As elaborated in previous chapters, the reconfigurable ASIP architecture is considerably faster than the underlying processor alone for both benchmark domains. Thus, the overheads of hardware and software runtime adaptation can be amortized provided that the application will be executed long enough. In this chapter, we analyze the achievable performance gains by the ASIP specialization and the runtime costs of three different phases of that process. These figures are used to compute the *break-even* which indicates for how long the application needs to be executed until the hardware generation overheads are amortized; that is, when a net speedup is achieved. These data are presented in Table 12.1 and are used to investigate the feasibility and limitations of the just-in-time processor customization. They were obtained for applications that were presented in Chapter 9.

Outline

The remaining parts of this chapter are structured as follows. First, we analyze the runtime of the ASIP-SP on the characteristic that limits our just-in-time processor customization study. Next, we investigate the maximum performance gains that characterize / reflect the feasibility of our approach. Finally, we study the introduced break-even figure of merit that demonstrates the balance between the feasibility and limitations.

12.2 Runtime of the ASIP-SP

Bellow we analyze the runtime of the ASIP-SP. Since the ASIP-SP consists of three phases: the candidate search, the netlist generation, and the instruction implementation, we subse-

quently study them in the following sections.

12.2.1 Candidate Search

We start the study from the Candidate Search phase that was presented in Section 4.2.2. This phase is responsible for finding and selecting only the best UDCI candidates from the software. The outcome of this phase is presented between 2nd and 7th columns in Table 12.1. These data were generated for the MM ISE algorithm, which was proven to be the best one for the runtime system; see Section 10.2.

Basic Block Pruning

As the candidate identification task is frequently very time-consuming, we are using our basic block pruning mechanisms introduced in Section 5.3 to reduce the search space. To this end, as presented in Section 10.3 we use the `@50pS3L` algorithm.

The third column in Table 12.1 represents the *pruning efficiency* ratio which is defined as the quotient of two terms. The first term is the ratio of the average maximum ASIP speedup to the runtime of the identification algorithm when no pruning is used. The second term is the same ratio when using the *@50pS3L* pruning mechanism. The pruning efficiency can be used as a figure of merit to describe the relative gain in the speedup-to-identification-time ratio with and without pruning.

Passed Code

The *blk* and *ins* columns represent the number of basic blocks and instructions which have been passed to the identification process. These numbers are significantly lower than the total number of blocks and instructions presented in the 6th and 7th columns of Table 9.1. That is, the pruning mechanism reduced the size of the bitcode that needs to be analyzed in the identification task by a factor of $36.49\times$ and $4.4\times$ for the scientific and embedded applications, respectively.

Number of UDCI Candidates

In the following column *can* the number of selected UDCI candidates is presented. As one can observe, this corresponds on average to 49 for the scientific and to 8 for the embedded applications.

Total Runtime of Candidate Search

The overall runtime of the basic block pruning, identification, estimation, selection, and extraction is aggregated in the *real* column. The total candidate search time is the order of milliseconds and thus insignificant in comparison to the overheads involved in the hardware generation.

App	Candidate Search: *@50pS3L*					ASIP	Runtime Overheads				Break-even	
	real	pruner	blk	ins	can	ratio	const	map	par	sum	factor	time
	[ms]	effic	#	#	#	x	[m:s]	[m:s]	[m:s]	[m:s]	x	[d:h:m:s]
164.gzip	1.44	71.79	2	100	19	1.00	56:22	13:02	18:28	87:52	754051	206:22:15:50
179.art	1.05	23.37	3	79	9	1.01	26:42	8:58	13:20	49:00	1869	1:12:18:13
183.equake	2.25	8.33	2	244	11	1.00	32:38	7:56	16:12	56:46	2808847	259:02:28:33
188.ammp	3.27	52.29	1	382	92	1.41	272:58	102:12	142:49	517:59	2321	0:14:56:39
429.mcf	1.05	28.2	1	77	5	1.00	14:50	4:06	7:48	26:44	771744	213:20:05:55
433.milc	6.6	26.71	2	673	9	1.00	26:42	6:44	15:08	48:34	2343546	568:06:08:05
444.namd	7.68	57.43	3	776	129	1.03	382:45	117:24	178:04	678:13	14423	6:16:00:48
458.sjeng	1.8	184.11	3	121	8	1.00	23:44	6:56	12:58	43:38	1150851	2403:01:35:57
470.lbm	10.62	2.43	3	961	179	2.53	531:07	181:51	308:24	1021:22	17427	1:03:29:48
473.astar	2.25	38.2	3	184	33	1.00	97:54	29:46	46:59	174:39	6740636	5149:02:19:14
AVG_S	3.80	49.29	2.30	358	49	**1.20**	146:34	47:53	76:01	270:28	1460572	881:00:33:54
adpcm	0.84	5.59	2	61	8	1.08	23:44	6:00	10:34	40:18	563	0:04:34:10
fft	0.78	3.78	2	75	14	2.40	41:32	11:44	20:56	74:12	367	0:01:53:07
sor	0.24	2.21	1	22	2	1.00	5:56	4:48	10:12	20:56	92	0:00:24:19
whetstone	0.54	7.7	2	49	9	15.43	26:42	11:34	25:52	64:08	142	0:01:08:04
AVG_E	0.60	4.82	1.75	52	8	**4.98**	24:28	8:31	16:53	**49:53**	291	**0:01:59:55**
RATIO	6.33	10.23	1.31	6.95	5.99	0.24	5.99	5.62	4.50	5.42	5019	10580

Table 12.1: The runtime overheads for the ASIP-SP process.

12.2.2 Netlist Generation

The tasks discussed in this section are represented in Figure 4.2 by the second phase of the ASIP-SP.

Generate VHDL

The *Generate VHDL* task is performed with the PivPav data path generator which produces the *structural VHDL* code. The data path generator traverses the data flow graph of the candidate and matches every node with a VHDL component. This is an $O(n)$ complexity algorithm that results in constant time operation and requires 0.2 s per candidate.

Netlist Extraction

The *extract netlist* task retrieves the netlist files for each hardware component used in the candidate's VHDL description from the PivPav database. This step allows a reduction of the FPGA CAD tool flow runtimes, since the synthesis process needs to build only a final netlist for the top module.

Create FPGA CAD project

The next step is to *create the FPGA CAD project* which is performed by PivPav with the help of the *TCL* scripting language. After the project is created, it is configured with the FPGA parameters and the generated VHDL code as well as the extracted netlist files are added.

Total Runtime of Netlist Generator

On average this process took 2.5 s per candidate, making this the most time-consuming task of the netlist generation phase. The average total NG runtime is presented in the *C2V* column of Table 12.2 and amounts to 3.22 s. As the standard deviation is only 0.10, this time can be considered as a constant.

12.2.3 Instruction Implementation

Once the project is created it can be used to generate the partial reconfiguration bitstreams. These files contain the FPGA implementation of the UDCI and hardware accelerator.

Check Syntax

This step is performed with the FPGA CAD tool flow which includes several steps. First, the VHDL source code is checked for any *syntax* errors and the runtime of this task is presented in the second column of Table 12.2. On average it takes 4.22 s for candidate to perform this task. Since the *stdev* is very low (0.10) we can assume that this is a constant time too.

Synthesis and Translate

Once the source code is checked successfully the *synthesis* process is launched. Since all the netlists for all hardware components are retrieved from a database there is no need to re-synthesize them. The synthesis process thus has to generate a netlist just for the top-level module which on average took 10.60 s. The runtime of this task does not vary a lot since the VHDL source code has a very similar structure for all candidates and changes only with the number of the hardware components. After this step all netlists and constraint files are consolidated into a single database with the *translate* task, which runs for 8.99 s on average.

Mapping and Place-and-Route

In the next step, the most computationally intensive parts of the tool flow are executed. These are the *mapping* and the *place and route* tasks which are not constant time processes as the previous tasks, but their duration depends on the number of hardware components and the type of operation they perform. For instance, the implementation of the shift operator is trivial in contrast to a division. The spectrum of runtimes for the *mapping* process ranges from 40 s for small candidates to up to 456 s for large and complex ones, whereas the *place and route* task takes 56–728 s. There is no strict correlation between the duration of these processes; the ratio of *place and route* and *mapping* runtimes vary from $1.4\times$ for small candidates to $2.5\times$ for large candidates.

Partial Reconfiguration Bitstream Generation

The last step corresponds to the partial reconfiguration bitstream generation. Our measurements show that this is again a constant time process depending only on the characteristics of the chosen FPGA. Surprisingly, the runtime of this task is substantial. On average, 151 s per candidate are spent to generate the partial reconfiguration bitstream. This runtime is constant and does not depend on the characteristics of a candidate. In many cases, the bitstream creation consumed more time than all other tasks of the instruction synthesis process combined (including synthesis and place-and-route). The runtime is mainly caused by using the Early Access Partial Reconfiguration Xilinx 12.2 FPGA CAD tools (EAPR). In comparison, creating a full-system bitstream that includes not only the custom instruction candidate but also the whole rest of the FPGA design takes on average just 41 s when using the regular (non-EAPR) Xilinx FPGA CAD tools.

12.2.4 Summary of Constant Runtimes

In Table 12.2, we summarize the runtime of the processes which cause constant overheads that are independent of the candidate characteristics. These are *Candidate to VHDL translation (C2V)*, *Syntax Check (Syn)*, *Synthesis (Xst)*, *Translation (Tra)*, and *Partial Reconfiguration Bitstream Generation (Bitgen)*. The total runtime for these processes is 178.03 s and is inevitable when implementing even the most simple custom instruction. The *Bitgen* process accounts for 85% of the total runtime.

	C2V	Syn	Xst	Tra	Bitgen	Sum
	[s]	[s]	[s]	[s]	[s]	[s]
Average	3.22	4.22	10.60	8.99	151.00	178.03
Stdev	0.10	0.10	0.23	1.22	2.43	

Table 12.2: Constant overheads involved in the ASIP-SP. *C2V* corresponds to the *Netlist Generation* phase in Figure 4.2. Syn, Xst, Tra, and Bitgen are the FPGA CAD tool flow processes and correspond to the syntax check, synthesis, translate, and partial reconfiguration bitstream generation processes, respectively, which can be found in the third phase in Figure 4.2.

12.2.5 Total Runtime of the ASIP-SP

The overall runtime involved in the FPGA CAD tool flow execution is presented in the column *Runtime Overheads* in Table 12.1. The column *const* represents the runtime of constant processes shown in Table 12.2. The column *map* stands for the mapping process, the column *par* for the *place-and-route*, and the values in the column *sum* add all these three columns that aggregate the total runtime involved in the generation of all candidates for a given application. The candidate's partial reconfiguration times were not included in these runtimes since they consume just a fraction of a second [29].

Average Runtime of the ASIP-SP

On average it takes less than 50 minutes (49:53 min) to generate all candidates for the embedded applications but more than 4:30 hours (270:28 min) for the scientific applications. One can see that this large difference is closely related to the number of candidates and that *sum* column grows proportionally with the number of candidates. This behavior can be observed for example for the *444.namd* and the *470.lbm* applications, which consist of 179 and 129 candidates, respectively. The total runtime overhead for them is more than 11 hours (678:13 min) and 17 hours (1021:22 min), respectively and is caused primarily by the high constant time overheads (*const*).

This observation emphasizes the importance of the pruning algorithms, particularly for the large scientific applications. We can observe the difference for the embedded applications where a smaller number of candidates exists. On average, the *const* time drops for the scientific applications from 146:34 min to 24:28 min; that is, by a factor of $5.99\times$, which is exactly the difference in the number of candidates *(can)* between the scientific and the embedded applications.

12.3 Maximum Performance Improvements of the ASIP-SP

The column *ASIP ratio* represents the speedup of the augmented hardware architecture when all candidates selected by Candidate Search are offloaded from the software to hardware UDCIs. In contrast to the maximum performance shown in the 8th column in Table 11.1, which assumes that no pruning methodology is used and *all* candidates are moved to hardware, the average speedup drops by 30% from $1.71\times$–$1.20\times$ for scientific applications and by 46%

from $9.19\times$–$4.98\times$ for the embedded ones. Comparing the *fft* with the *470.lbm* applications illustrates the main difference between the embedded and scientific applications. Both applications have a similar speedup of $2.40\times$ vs. $2.53\times$, respectively, but differ significantly in the number of candidates that need to be translated to hardware to achieve these speedups (14 vs. 179 candidates). This correlates with the previously described observation that scientific applications have a significantly larger kernel size.

12.4 Break-Even Times

In this section, we analyze the break-even time for each application; that is, the minimal time each application needs to execute before the overheads caused by the ASIP-SP are compensated. Therefore, the break-even figure of merit reflects the balance between the feasibility and limitations of our system. If the break-even time is short this means that the reconfigured processor provided with significant performance improvements over the static processor and the runtime of the ASIP-SP was short. Otherwise, if the break-even times are long it means that the ASIP-SP required long time to generate the hardware accelerators and these did not provide with good processing performance improvements.

Algorithm Overview

A simplistic way of computing the break-even time would be to divide the total runtime overhead (*sum* in Table 12.1) by the time saved during one execution of the application, which can be computed using the *Execution runtimes - VM* from Table 9.1 and the *ASIP Ratio* (speedup) from Table 12.1. This computation assumes a scenario, where the size of the input data is fixed and the application is executed several times.

Scaling Input Data with Code Coverage Informations

We have followed a more sophisticated approach of computing the break-even time, which assumes that more input data is processed instead of multiple execution of the same application. Hence, the additional runtime is spent only in the parts of the code which are *live* while code parts that are *const* or *dead* are not affected. To this end, we use the information about the execution frequency of basic blocks and the variability of this execution frequency for different benchmark sizes which we have collected during code coverage profiling; see Section 9.4. The resulting break-even times are presented in the last column of Table 12.1.

Embedded vs. Scientific Break-Even Times

It is evident that there exists a major difference in the break-even times for the embedded and the scientific applications. While the break-even time of the embedded applications is in the order of minutes to a few hours, the scientific applications need to be executed for days to amortize the overhead caused by UDCI implementation (always under the assumption that *can* candidates are implemented in hardware). The reason for these excessive times is the combination of rather long ASIP-SP runtimes *(>4:30h)* and modest performance gains of $1.2\times$. As described above, the long runtimes are caused by implementing many candidates.

One might expect that this large number of UDCIs should cover a sizable amount of the code and that significant speedups should be obtained, but evidently this is not the case.

Covering the Application Kernel with UDCIs

The reason for this is that the custom instructions are rather small, covering only 6.9 IR instructions on average. Although there are many custom instructions generated, they cover only a small part of the whole computationally intensive kernels of the scientific application, which has a size of 1764 IR instructions on average. Adding more instructions will not solve this issue since every candidate adds an additional FPGA CAD tool flow overhead.

In contrast, the break-even point for the embedded applications is reached more easily. On average, the break-even time is five orders of magnitude lower for these applications. In contrast to the scientific applications, the custom instructions for embedded application can cover a significant part of the computationally intensive kernel. This results in reasonable performance gains with modest runtime overheads. For an average embedded application, a $5\times$ speedup can be achieved, resulting in a runtime overhead of less than 50 minutes and a break-even time of less than 2 hours.

Average Size of UDCIs

The difference between the scientific and embedded applications is not caused by a significant difference in the number of IR instructions in the selected candidates. The scientific applications have on average 7.31 instructions per candidate, while the embedded applications have on average 6.5 instructions per candidate.

Average Size of Basic Block

Since we cannot decrease the size of the computational kernel, we should strive for finding larger candidates in order to cover a larger fraction of the kernel. Unfortunately, this turns out to be difficult because the reason that the candidates are small is that the BBs *(blk)* in which they are identified are also small. The average basic block has only 7.64 (5.94) IR instructions for a scientific (embedded) application (see Table 9.1).

Selecting Large Basic Blocks with Pruning Algorithms

The pruning mechanism we are using is directing the search for UDCIs to the largest basic blocks; hence, the average basic block that passes the pruning stage has 155.65 instructions for a the scientific and 29.71 for embedded application (see Table 12.1). However, even these larger blocks include a sizable number of the hardware-infeasible instructions, such as accesses to global variables or memory, which cannot be included in the hardware UDCIs. As a result, there are only 7.31 instructions per candidate in the scientific application which causes high break-even times for them.

This observation illustrates that there are practical limitations for the ASIP-SP when using code that has been compiled from imperative languages.

Chapter 13

Reduction of ASIP-SP Runtime

In this section, we propose two approaches for reducing the total runtime overheads and in turn also the break-even times: partial reconfiguration bitstream caching and acceleration of the CAD tool flow.

13.1 Partial Reconfiguration Bitstream Caching

As in many areas of computer science, caching can be applied also in the context of our work. Much like virtual machine cache for the binary code that was generated on-the-fly for further use, we can cache the generated partial bitstreams for each UDCI. To this end, each UDCI needs to have a unique identifier that is used as a key for reading and writing the cache. We can, for example, compute a signature of the LLVM bitcode that describes the candidate for this purpose. The cached bitstreams can be stored for example in an on-disk database.

13.2 Acceleration of the CAD Tool Flow

A complementary method for reducing the runtime overheads is to accelerate the FPGA CAD tool flow. There are several options to achieve this goal. One possibility is to use a faster computer that provides faster CPUs and faster and larger memory or to run the FPGA tool concurrently. Alternatively, it may be possible to use a smaller FPGA device, since the *constant* processes of the tool flow depend strongly on the capacity of the FPGA device. We have used a rather large *Virtex-4 FX100* device, therefore switching to a smaller device would definitely reduce the runtime of the tool flow. Another option would be to use a memory file system for storing the files created by the tool flow. As the FPGA CAD tool flow is known to be I/O intensive, this should speed up the tool flow. Finally, we could change our architecture to a more coarse-grained architecture with simplified computing elements and limited or fixed routing. It has been shown that it is possible to develop customized tools for such architectures which work significantly faster [87].

	Faster FPGA CAD tool flow[%]									
Cache	0	10	20	30	40	50	60	70	80	90
hit [%]	[h:m:s]	[h:m:s]	[h:m:s]	[h:m:s]	[h:m:s]	[h:m:s]	[h:m:s]	[h:m:s]	[h:m:s]	[h:m:s]
0	01:59:55	01:49:02	01:36:55	01:24:48	01:12:41	01:00:34	00:48:27	00:36:20	00:24:14	00:12:07
10	01:47:44	01:36:58	01:26:11	01:15:25	01:04:38	00:53:52	00:43:06	00:32:19	00:21:33	00:10:46
20	01:32:59	01:23:41	01:14:23	01:05:05	00:55:47	00:46:29	00:37:11	00:27:53	00:18:36	00:09:18
30	01:28:09	01:19:20	01:10:31	01:01:42	00:52:53	00:44:04	00:35:15	00:26:27	00:17:37	00:08:49
40	01:13:08	01:05:49	00:58:30	00:51:11	00:43:53	00:36:34	00:29:15	00:21:56	00:14:37	00:07:19
50	01:01:00	00:54:54	00:48:48	00:42:42	00:36:36	00:30:30	00:24:24	00:18:18	00:12:12	00:06:06
60	00:48:50	00:43:57	00:39:04	00:34:10	00:29:18	00:24:25	00:19:32	00:14:39	00:09:45	00:04:53
70	00:35:12	00:31:41	00:28:10	00:24:38	00:21:08	00:17:36	00:14:05	00:10:33	00:07:02	00:03:31
80	00:29:19	00:26:23	00:23:27	00:20:31	00:17:35	00:14:39	00:11:43	00:08:47	00:05:51	00:02:56
90	00:14:07	00:12:42	00:11:17	00:09:53	00:08:28	00:07:03	00:05:39	00:04:14	00:02:49	00:01:24

Table 13.1: The average *Breaking Events* for the embedded applications with enabled cache and faster FPGA CAD tool flow

13.3 Extrapolation

In Table 13.1 we calculate the average *breaking-even time* for the embedded applications when applying these two reducing ideas. When the cache is disabled and we do not assume any performance gain from the tool flow, the first value is equal to the *AVG_E* row and the last column in Table 12.1. One can note also that these values do not scale linearly because we consider the frequency information for basic blocks; see Section 9.4.

Cache Hit Ratio

For this evaluation, we varied the assumed cache hit rate to be between 0%–90%. That is, for simulating a cache with a 20% hit rate, we have populated the cache with 20% of the required bitstreams for a particular application, whereas the selection whose bitstreams are stored in the cache is random. Whenever there is a *hit* in the cache for a given candidate, the whole runtime associated with the generation of the candidate is subtracted from the total runtime; see *sum* column in Table 12.1. The values in the *Faster FPGA CAD tool flow* columns are decreasing linearly with the assumed speedup.

Faster FPGA CAD Tool Flow

If we assume that the FPGA CAD tool flow can be accelerated by 30% and that we have 30% cache hits, the average break-even time drops almost by half ($1.94\times$), from *1:59:55* h to *1:01:42* h. This means that the whole runtime of the ASIP-SP could be compensated in a bit more than one hour and for the rest of the time the adapted architecture would provide a performance gain by an average factor of $5\times$. These assumptions are modest values since the *cache hit* rate depends only on the size of the cache and our Dell T3500 workstation could be easily replaced by a faster one.

Improving Processor Performances

The Woolcano hardware architecture contains an embedded PowerPC 405 that allows to achieve approximately $0.6 - 0.7k$ DMIPS (1.52/MHz). Since this processing power was not adequate to the requirements of the ASIP-SP, experiments were performed on Dell T3500 Workstation; see Chapter 8. The processor found in this machine (*Intel W3503 Xeon*) allows to achieve up to $27k$ DMIPS. While this results in $40\times$ difference in processing power, the state-of-the-art from-the-shelf processors like *Intel Core i7 990x* allow to achieve up to $159k$ DMIPS. Under the condition that this processor would be fully utilized this would lead to further $5.8\times$ break-even times improvements.

Chapter 14

Conclusions and Outlook

This thesis study the feasibility and limitations of a process capable of the just-in-time processor customization that targets dynamically reconfigurable instruction set architectures. In particular, this thesis provides an answer to the controversial question whether the just-in-time processor customization is a worthwhile idea under the assumption that the currently commercially available FPGA devices and tools are used. To this end, we have developed a holistic and unique system capable of the just-in-time processor customization and in order to provide a trustfully answer to the posed question, we utilized it to customize a whole sets of benchmarks from the embedded and scientific domains. The measured results included runtime of the tool flow as well as the application-level performance gains, which allowed to compute the break-even time and to study the feasibility, limitations, and economics of the developed system. These characteristics are presented in Figure 14.1 and were investigated for three different computing domains represented by the embedded, personal, and high performances systems. While there are many differences between these systems for the conclusion purpose we consider only the properties of the utilized applications and the performances of a single processor found in these systems.

Application Properties

The properties of the applications that are taken under consideration are specified in the top part of the table shown in Figure 14.1. They consist of the usual kernel runtime, size, complexity, and available instruction level parallelism where complexity is a metric that relates to the software to hardware translation process. Thus, it corresponds to the number and the position of the control flow statements found in the source code, operations and usage of different memory structures, and the source code modularization. These informations have a direct impact on the basic block size, and in consequence, on the available instruction level parallelism, sizes of UDCIs, and kernel coverage.

The embedded applications have to fit into the constrained environment thus, they have small and not complex kernels defined usually in the same function and, in consequence, it is possible to cover this kernel with a small number of rather large UDCIs. For scientific applications, we assume that the kernels were optimized by the developer and therefore, their sizes and complexity is moderate. The average application and kernel runtime for both embedded and scientific domains is long and counted in hours, which helps to compensate the

break-even.

The kernels of the mainstream applications that are utilized by a personal computer are swapped frequently and thus, have short runtimes. Moreover, we assume the that emphasis was not put on the optimizations but on functionality, which results in large and complex kernels.

14.1 Developed System

Our developed system consists of the hardware architecture and the software tool flow that allow for hardware and software runtime adaptation and, in consequence, for a fully automatic and online just-in-time processor customization process. The designed hardware architecture is a dynamically reconfigurable instruction set processor that allows for the online during runtime customization, whereas the software tool flow is based on a virtual machine and allows to customize this architecture concurrently to the application execution and without any manual efforts. This tool flow contains a set of heuristics that reduce the runtime of methods for identifying and selecting custom instructions for the just-in-time processor customization as well as a circuit library and a data path generator of required bitstreams for the hardware customization.

The most significant parts of the software toolflow, including the candidate identification, estimation, selection, and pruning mechanisms were not only described with precise formalisms but were also experimentally evaluated. In particular, we discussed and compared characteristics of three state-of-the-art instruction set extension algorithms in order to study the candidate identification mechanism in detail. This study included not only algorithm runtimes, number of found UDCI candidates, their properties, and impact of algorithm constraints on the search space, but more importantly the achievable maximum performance gains for various embedded computing and scientific benchmark applications.

Simultaneously, this work has explored the potential of our Woolcano reconfigurable architecture, the ISE algorithms, and basic block pruning mechanism for them as well as the PivPav estimation and data path synthesis tools.

14.2 System Feasibility

The study has shown, that for the embedded applications, an average speedup of $5\times$ can be achieved with a runtime overhead of less than 50 minutes. This overhead can be compensated if the application executes for two hours or for one hour when assuming a 30% cache hit rate and a faster FPGA CAD tool flow. If we take under the consideration that the usual embedded application has a long runtime and executes at least for a few hours than we can definitively state that the just-in-time processor customization is a very promising and applicable approach for these applications. This behavior is illustrated with the green color in Figure 14.1.

Extrapolating these results to a broader general use case, we claim that we see potential of the just-in-time processor customization process for applications that contain relatively small and simple kernels and that utilize them for longer periods of time. This claim is valid for the most demanding case where applications are developed with the mainstream imperative pro-

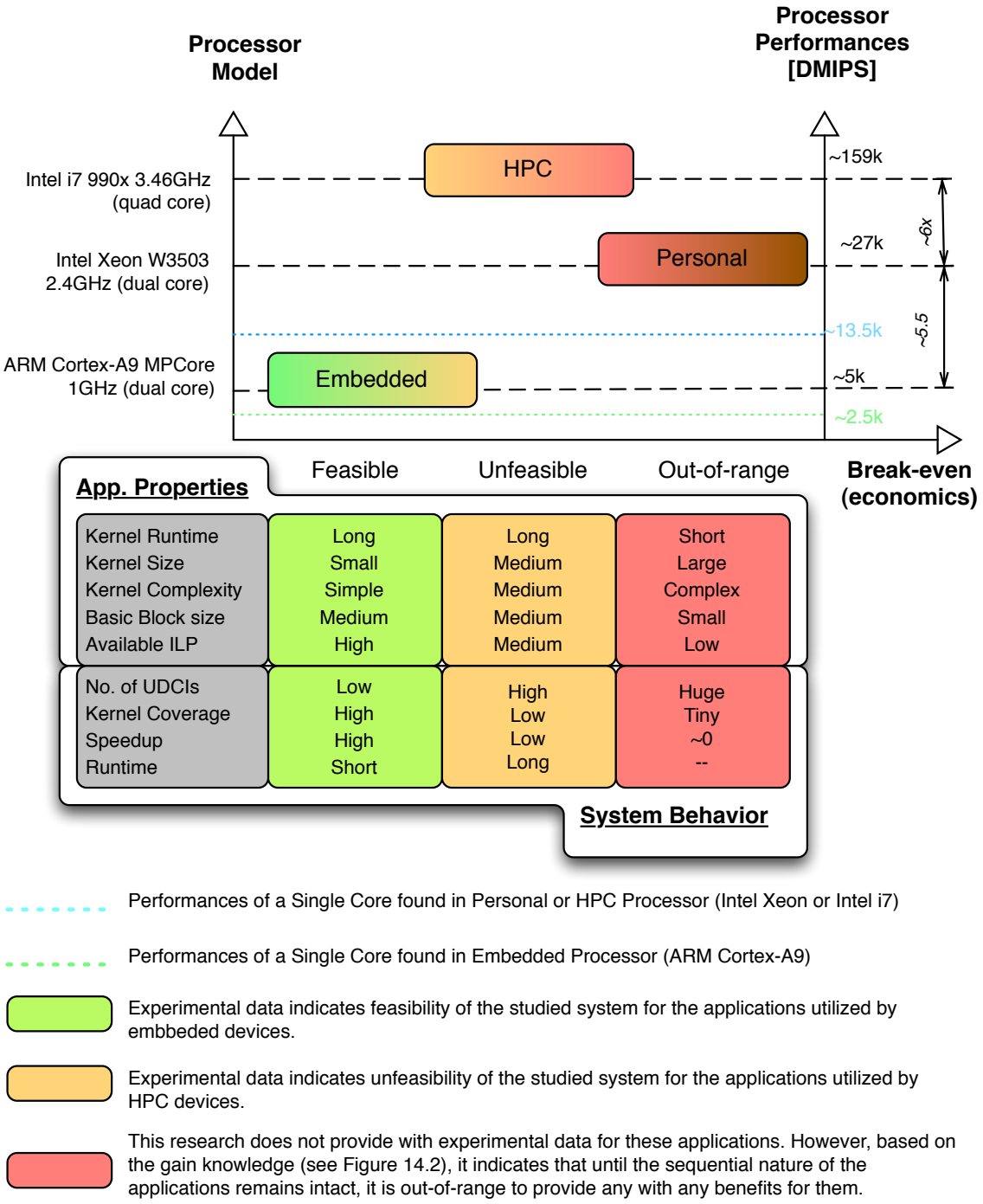

Figure 14.1: Economics of the just-in-time processor customization when extrapolated for the embedded, personal, and HPC computing domains.

gramming languages, they do not explicitly expose any kind of parallelism, and their source code is not tailored in any way to utilize features of our system.

14.2.1 Applicability and Technological Barrier

The developed system is practically applicable without any further major changes under the condition that the embedded processor will increase its performances by a factor of $5.5\times$. This number is calculated as a difference between the embedded processor that is built-in into the state-of-the-art Virtex-7 FPGA and the general purpose processor that we used for experiments. The difference between them is presented in Figure 14.1 where the ARM Cortex-A9 embedded processor is capable of 2.5k DMIPS for a single core (5.0k DMIPS for two cores) whereas Intel Xeon W3503 used in experiments is capable of 13.5k DMIPS (27.0k).

The applicability of our system increases further if we assume that the system can utilize many computational units simultaneously. In our experiments, all computations including the virtual machine interpretation with ASIP-SP were performed on a single computational unit. For that case, the usage of HPC processors such as Intel i7 would bring further improvements by additional factor of $6\times$ and, in consequence, would further increase the applicability of our system.

14.3 System Limitations

The just-in-time processor customization is a methodology from which a wide range of computing systems would benefit. Unfortunately, while our research indicated benefits for the embedded systems at the same time it demonstrated that the customization process is unfeasible for the other ones, which have far more demanding application properties. This is caused rather by the sequential nature of applications than by gaps or shortcomings in our methodology. In particular, our study showed that larger and more complex software kernels of the scientific applications, represented by the SPEC benchmarks, do not map well to hardware UDCIs targeting our hardware architecture and lead to excessive times until the break-even point is reached. This behavior is presented in Figure 14.1 with the orange color and it addresses the HPC systems. There are two factors that cause this behavior which are presented in Figure 14.2: relatively long total runtime and low applications-level performance gains.

14.3.1 Long Total Runtime

The total runtime required for the UDCI candidates implementation is indicated with the green color in Figure 14.2 and is very long (> 4:30 h) for the investigated applications. The reason for this is that in order to cover a significant portion of the kernel, the number of UDCI candidates indicated for hardware implementation is very high and the runtime of the implementation process for each of them is very long. Only the implementation processes that have constant runtimes (see Table 12.2) account for almost 3 minutes. This is an inevitable runtime for every UDCI candidate and it does not depend on their sizes, which on average contain 6.9 IR instructions.

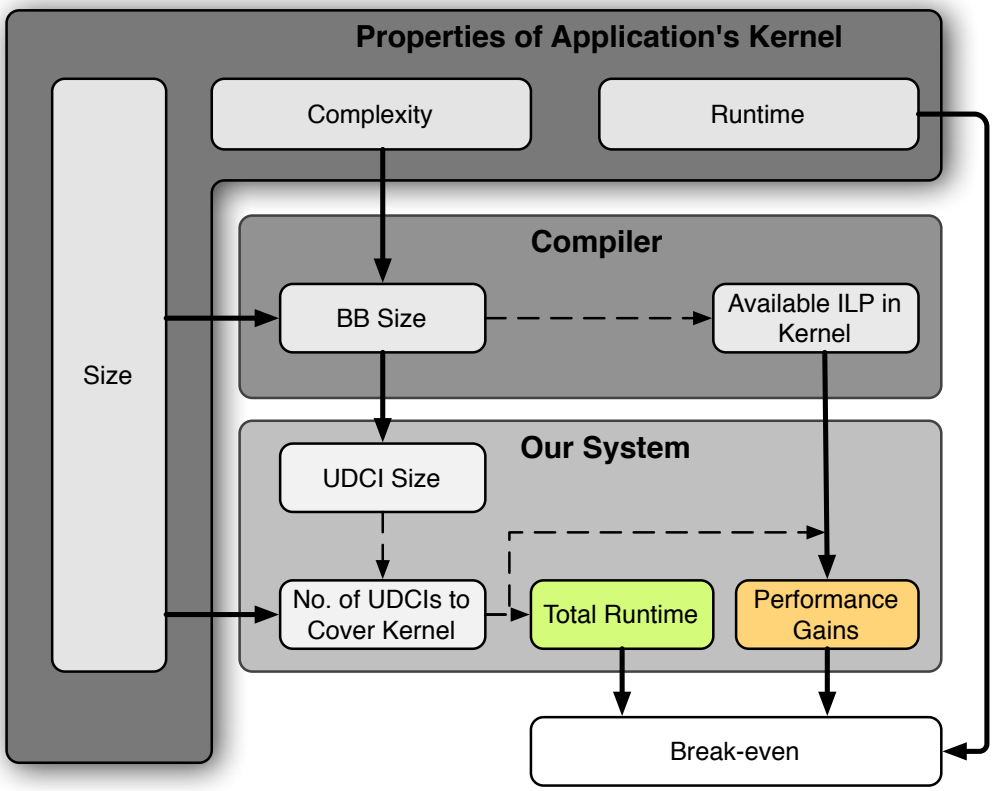

Figure 14.2: Dependencies between the application properties, compiler, and just-in-time processor customization process.

14.3.2 Low Performance Gains

The performance gains, represented with the orange color in Figure 14.2, are strongly dependent on the available instruction level parallelism in the kernel and the kernel coverage by UDCIs. Since the kernel for these applications is relatively larger than for the embedded applications (80 vs. 1960 IR instructions) and number of UDCIs is insufficient to cover it significantly in the end, only $1.20\times$ application-level performance gain was achieved.

14.3.3 Further Limitations

Figure 14.1 presents three different computer systems and applications that target them. While it is feasible to provide with benefits for the embedded applications (green color) and unfeasible for the scientific applications (orange color), the figure points also a third category of applications that are the most demanding and target personal computers (red color). These applications are used to provide an outlook and to extrapolate the gain knowledge presented in Figure 14.2 to a broader spectrum of applications.

For these demanding applications, we indicate that, unless the nature of the sequential applications will change, the just-in-time processor customization is out-of-reach. It is due

the fact that the runtime of our system can be amortized under the condition that the significant application-level performance gain is achieved and for these applications we indicate a speedup close to zero. Therefore, whatever the runtime is, short or long, this does not make any difference since it will be never amortized if the system does not provide with any performance gains.

14.3.4 Origins of Limitations

The root of the problem is caused by the many times criticized [88–91] obsolete sequential computing paradigm based on the von Neumann computer architecture and, in consequence, the nature of applications even when utilized on the state-of-the-art devices and developed under recent programming frameworks; see Figure B.6. These coherent sequential applications put many obstacles for the parallel computing paradigm infiltration.

In our system, this is reflected with very small basic block sizes. In consequence, this affects the sizes of the UDCIs as well as the available instruction level parallelism [92] and in the end, it provides with out-of-reach break-even values.

The reason behind the small basic blocks can be found in the properties of the intermediate code generated by LLVM when compiling imperative C or C++ code. This constrain does not apply only to LLVM, but in general, it corresponds to a wide group of compilers. It relates to dominate frontier algorithms that translate the control flow operations found in HLL source code to the equivalent Φ instruction in the SSA IR code. The number of Φ instructions as well as their position determine the sizes of the basic blocks. Thus, the sizes of the basic blocks are directly influenced by the number and the location of the control flow statements in the source code, which we represent under the *kernel complexity* in Figures 14.1 and 14.2. To this end, we expect that this behavior applies to other imperative languages regardless of the used compiler.

14.3.5 Possible Improvements

The basic block size is the major property of the source code that has impact on the system's total runtime and the achievable performance gains. Thus, it is natural to use methods that stimulate the grow of the basic blocks. This is achieved in two ways, either by changing location, reducing, or avoiding the control flow statements in the source code.

Control Flow Statement Location Shift

It has been reported that for some applications this task can be achieved with the superblock formation [81, 82], predicate execution [83], or hyperblock formation [84]. However, we want to point out that these sophisticated methods try to cure the disease rather than solve the problem at the origin. In consequence, when properly used, these optimizations can only help to increase basic block sizes and, as a result, they do not allow to overcome limitations for the scientific applications. Moreover, there exists a major problem with the applicability of these methods. The compiler when translating HLL into the IR code uses more than 30 different passes, which influence each other and change their order with different applications. Therefore, the functionality of the listed passes when targeting whole applications is limited

and is strongly dependent on the correct order they are executed. In our experiments, we tested the benefits of this method for selected applications and since the order had to be set up manually it was a very tedious and, in the end, not rewarding task.

Avoiding Control Flow Statements

Since the number and location of control flow statements in a source code determine the sizes of the basic blocks, the easiest way to achieve highest sizes is to reduce or completely avoid the usage of control flow statements. This behavior is used in the digital signal processing (DSP) domain [93] together with the streaming capabilities that allow to reduce the communication latencies between a memory and a computing unit. While these architectures outperform both graphic processor units and general purpose processors for more than an order of magnitude [94, 95], they target only specific algorithms developed in domain specific languages. Therefore, this approach does not apply for general purpose applications written in imperative programming languages and, in consequence, building general high performance HLS tool proves to be very challenging for them.

Other Methods

In order to achieve higher performance gains and to surpass system limitations, one could suggest to focus on a higher level of parallelism than the instruction one. To this end, we briefly investigate the loop-level and thread-level parallelisms.

The loop level parallelism can be used for transforming nested loops in benchmarks, which are used in this work to UDCIs. To this end, loop-unrolling techniques [96] that transform the computation from the temporal to the spatial domain would be used. This level of parallelism has much to offer in terms of performance gains since it allows to exploit the flexibility of reconfigurable hardware with a deep pipelining, parallel operations, and custom arithmetic methods. Unfortunately, in order to utilize it application must show streaming behavior to compensate communication latencies and delays. To this end, a memory partitioning task has to be performed, which is very challenging [91] for sequential imperative languages that are based on mutable states, mutual data, and allow for side effects. While the virtual machine and runtime capabilities allow to investigate the memory dependencies and help to perform this partitioning task, it still stays very difficult. This is mostly due to a detailed analysis of deadlocks and race conditions that have to be performed and, in consequence, no tools or algorithms capable to perform this task automatically exist. Therefore, while switching to the loop-level parallelism is a promising idea it is not applicable for the just-in-time processor customization systems.

The next raised level of parallelism corresponds to threads and tasks, which have familiar programming and execution models and do not depend on streaming methods. Unfortunately, they do not expose with a real software concurrency and contain many control flow statements that are best utilized on sequential processors. Due to significant differences between FPGA and ASIC technologes described in Section 2.2.3 the processors implemented in FPGA are easily outperformed by the ASIC ones and, in consequence, no performance gains are provided. Moreover, in practice, there exist no automated tools that would generate required hardware applicable for adaptive systems.

Summary

The best improvements are achieved when the parts intended for the hardware implementation are large and there are no control flow statements in them. These avoid sequential computing and allow for a truly parallel execution in reconfigurable architectures. While this scenario is achieved for imperative programming languages with the instruction level parallelism, better results are achieved with domain specific languages that are capable of streaming and that target specific algorithms from the digital signal processing.

14.4 Final Remarks

This thesis provides with study of an adaptive system capable of the just-in-time processor customization. The research showed that while our system is feasible and applicable in the near future for the embedded applications, it is unfeasible for the scientific applications and out-of-reach for ones utilized by personal computers.

The journey towards our adaptive system started in the end of year 2007 where Virtex-4 was state-of-the-art FPGA that contained the PowerPC 405 processor able to deliver up to 700 DMIPS. These performances were not enough to support the system back then and even the performance of the state-of-the-art embedded processors, which over these years increased performances by a few factors are still not sufficient.

What had changed in these almost four years is not only the advancement in technology but more importantly the fact that the world and companies like Intel see the demand for the just-in-time processor customization in their products [18], which outcomes in a joint cooperation with FPGA vendors like Altera or Achronix. Moreover, during these years companies like Convey Computers started to provide with products that allow to customize the HPC processors with instruction set extensions. While these systems are customized manually, they allow for certain algorithms to outperform general purpose computers by three orders of magnitude or more. These are exciting times for the reconfigurable computing and processor customization processes.

It is hard to indicate whether our research and our adaptive systems sees the daylight of a product in the near future. There are many obstacles that were precisely described in this work that limit the applicability of this task. While the technological limitations can be quite fast surpassed the major ones deal with origins of the computer architecture and as we know are extremely challenging to overcome.

Part IV

Appendices

Appendix A

Software Translation Process

The purpose of this appendix is to provide basic information about the process of software translation that has the core responsibility in our developed system. Thus, in this section, general concepts of translators are explained together with comparison of suitable frameworks for our system.

A.1 Introduction

The translation refers to the process which converts the source code written in one programming language into another programming language. In that case, the term source-to-source translator is used. The term *compiler* is used in relation when the translation occurs from the high level language (HLL) into the lower level language like assembly or the machine code. Therefore, each compiler is a special case of the translator tool. The compiler refers to the process which occurs *ahead-of time* from the code execution and thus, it is *offline* or *static* process where the code *compilation* is separated from the code execution. The compilation process is orthogonal to the *interpretation* which occurs during the runtime of the application. This process also is referred to as *runtime* or *dynamic translation*. In this process, the task of code generation and execution occur simultaneously.

A.2 Low Level Virtual Machine (LLVM)

This work is developed around open-sourced *Low Level Virtual Machine* (LLVM) compiler infrastructure [97–99]. This framework allows to build from reusable components a static compiler or a dynamic translator that include the just-in-time as well as the trace-based optimizations and transformations. This significantly reduces both development time and the cost.

LLVM consists of *virtual machine* and *static code generator* and thus, is able to translate the code either dynamically or statically. The virtual machine is able to execute the code dynamically whereas the static code generator provides backends that allow to generate an assembly code for a variety of hardware architectures, e. g., x86, IA64, PowerPC, MIPS, SPARC, etc.

The key idea behind LLVM is to use the *intermediate representation* (IR) during all phases of the compiler. This includes static or dynamic translation as well as the linking process and various optimization phases.

A.3 Intermediate Representation

LLVM is not targeted at a specific programming language but it is based on the concept of intermediate representation. The IR is independent language and consists of *virtual instruction set architecture* resembling to the RISC CPU with three-address code which is very well suited for execution by abstract machines.

The IR can exist in three forms: *textual*, *binary*, and *in-memory*. It is possible to convert between these formats without any information losses, which makes it a self contained language. The textual representation has a human assembly readable form, it can be edited by hand, and compiled into the binary form. The binary representation is kept on a disk and is meant to serve as a placeholder for bitcode representation that can be quickly loaded for execution by the virtual machine. The in-memory representation is an internal representation of a bitcode used by the compiler to perform optimizations and transformations.

In contrast to the traditional IRs like ANDF and UNCOL the LLVM IR encodes high level informations which include:

- explicit data flows through *single static assignment* (SSA),
- explicit control flow through simple constructs,
- explicit language independent type informations,
- explicit typed pointer arithmetics, able to preserve array indexing.

These informations are able to support *aggressive inter-procedural optimizations*, *loop optimization*, and *scalar optimizations* as well as other high and low level optimizations and analysis algorithms. The functionality of these algorithms is invariant in regard to the used HLL language which means that a single instance of the algorithm is functional for all HLLs that are supported by LLVM. This often caused a problem for other IRs like ANDF and UNCOL. These IRs had separated control and data flow informations that were tied to a specific HLL.

Single Static Assignment

Since LLVM has SSA IR the program *data flow graphs* (DFGs) are easily accessible. In SSA form, every result of instruction or operand is assigned to a *virtual register* exactly once, which leads to an unlimited number of virtual registers. SSA uses *use-def* and *def-use* chains with every object including instructions, operands, and functions. Therefore, SSA avoids data anti-dependencies and output-dependencies issues and allows for easy DFG construction and manipulation. Altogether, this makes from SSA a well suited form to represent data dependencies and the program structure. The use of SSA enables elegant and fast optimization algorithms such as *sparse data flow analysis*, *constant propagation*, or *dead code elimination*.

A.3.1 Type System

The type system of LLVM is presented in Table A.1. Instruction results or function arguments are able to use only the first class types. Derived types are intended for use of data types implemented in a high-level language such as arrays, functions, or classes.

Classification	Types
integer	i1, i2, ..., i16, ..., i64, ...
floating point	float, double, x86_fp80, fp128, ppc_fp128
first class	integer, floating point, pointer, vector, structure, array, label, metadata
primitive	label, void, floating point, metadata
derived	integer, array, function, pointer, structure, packed structure, vector, opaque

Table A.1: LLVM type system

A.3.2 Instruction Set Architecture

In contrast to other CISC architectures instead of hundreds of instructions, IR contains only a few dozens of them. The small number of instructions is caused by the extensive type system. A single IR instruction can be used with different data type operands and therefore there is no need to provide separate instructions for every first class type.

IR instruction can be grouped into the following categories:

- Binary instructions. Both operands as well as the result of the instruction are of the same type and separate instructions exist for integers, floating point, and bitwise operations.
- Memory access instructions. LLVM is a *load/store* system where all accesses to the memory are performed with the *load* and *store* IR instructions. Accesses are performed on *pointers* to the first class values. Moreover, separate instructions exist for *allocating* and *freeing* the memory.
- Terminator instructions that perform the control flow operations and represent edges in *control flow graphs* (CFGs). These are the last instructions found in the basic blocks and determine the next basic block for execution.
- Other instructions, which responsible for the type casting, conversion, comparisons, for aggregate values and vectors.

A.4 Program Structure

The structure of the whole program in LLVM is represented with a *module* which consists of *functions*, *global variables*, and *symbol tables* where the *function* is implemented with a set of basic blocks and arguments.

Basic Blocks

Basic Block (BB) is a sequence of IR instructions that is ended with a terminator instruction. Instructions in the BB are executed sequentially and thus, the structure of the BB can be modeled as a *direct acyclic graphs* (DAGs). Moreover, from the control flow view the BB is an atomic unit ended with the terminator instruction, which is responsible for the control flow and forwards the execution from the current BB to another BB. Therefore, the terminator instruction is responsible for the edges in the *control flow graph* (CFG) and the basic blocks represent vertices in that graph.

A.5 Modularity

LLVM has a modular design which is provided with a set of well integrated and designed libraries. These libraries were developed with runtime compilation feature in the mind and provide a framework for analyses, optimizations, code generations, dynamic compilation, support for garbage collection, or profiling. Around these libraries a collection of tools is built, such as assemblers, automatic debuggers, linkers, modular optimizers, etc.

A.6 Separation from High Level Languages

Since the *control data flow graphs* (CDFGs) are encoded directly into the IR and BB, the *abstract syntax tree* (AST) can be reconstructed from the IR and BB without the knowledge about any specific HLL constructs. This means that the IR is not tied to any specific HLL and can be used to represent an arbitrary number of them. From the LLVM point of view, IR provides a separation layer between the HLL and the middlend IR optimizations. This also means that a single IR optimization is capable to perform its task for any given HLL that is supported by the LLVM front-end. This significantly simplifies the task of designing and implementing IR optimizations.

A.7 Design of the LLVM Compiler Framework

In this section, the design of LLVM is presented with more details. The provided informations lay down fundaments for the Woolcano compiler presented in Chapter 4. The simplified construction of LLVM is demonstrated in Figure A.1 whereas the details can be found in Muchnick [101].

In LLVM there are four main abstraction layers: the *front end*, the *middle end*, and two succeeding *back ends*. These layers are connected with abstraction interfaces: previously described intermediate representation, the target instructions, and the assembly language. The purpose of every interface is to reduce the granularity of the HLL to the level which can be understood and executed by the hardware architecture.

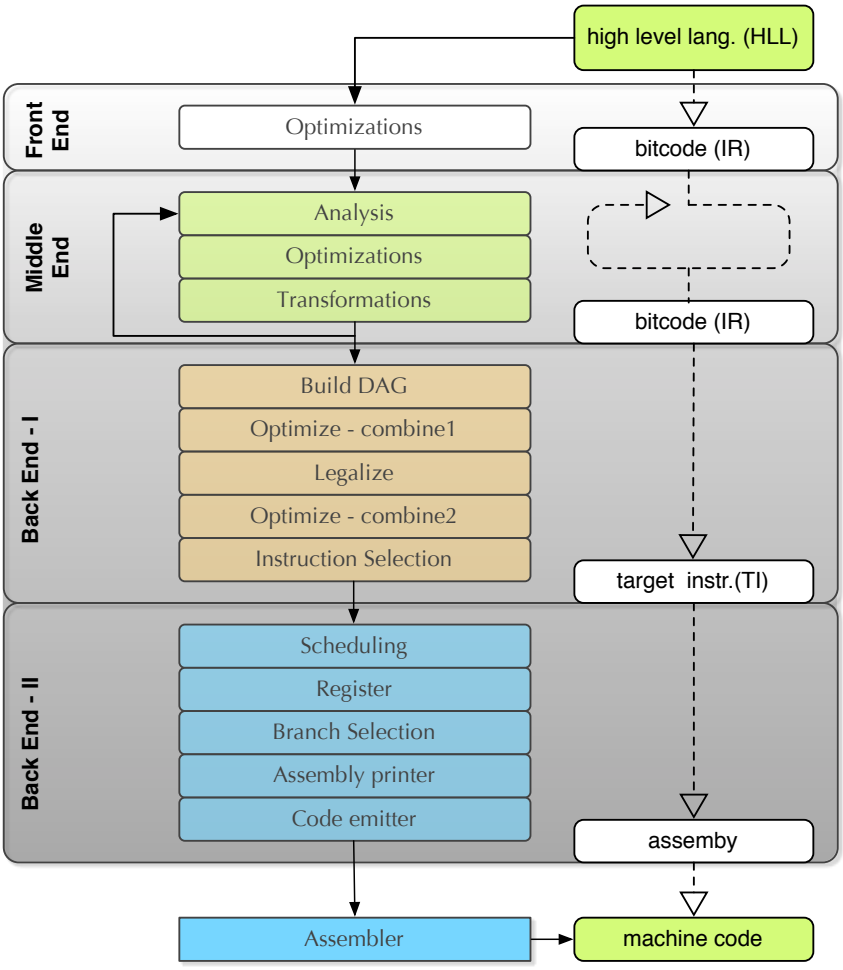

Figure A.1: Abstracted design of the LLVM

A.7.1 Compilation Process

LLVM's compilation process starts in frontend with translating the source language to the IR. This task is performed either with *llvm-gcc* or *clang* that among most widely used programming HLLs support C and C++. Once IR is generated in the middle end, the language and target independent analysis and transformations are repeatedly applied to optimize the final IR. After these optimizations, in the first backend, IR is translated to a target independent CDFGs where various graph optimizations are applied.

Only in the final part of the compiler the target independent code is translated to a target specific assembly. This transformation is provided by a code generator with the help of instruction pattern matching algorithms found in the second back end. To this end, this algorithm uses target specific informations such as opcodes, register layout, calling convention, and instruction patterns. These informations are specified in a separate target description file with declarative *domain specific language* (DSL).

A.7.2 Front End

The main task of the frontend is to translate the HLL source code into the low level code where for LLVM this translation corresponds to IR. To this end, frontends are equipped with several phases that include the *preprocessing*, *lexical* and *syntax analysis* as well as the *semantical analysis*. These define not only the meaning of various language elements like the identifiers, keywords, and symbol names found in the source code but also the structure of the HLL code, the type system, or the object bindings.

The purpose of the frontend is to generate the best quality low level code that is the most accurate and efficient one. This task is influenced by the type and by the number of used optimizations. Moreover, this task is strongly correlated with HLL expressiveness that allows to achieve high productivity. Expressive HLL have syntax and semantics that allow to describe given algorithm with ease. Unfortunately, these constructs are often cumbersome for the low level translation. Thus, the expressiveness is orthogonal to the code quality and in the frontends the balance between them is required.

In order to increase the programmers productivity, the frontend of the compiler is often equipped with debugging features which allow to accurately locate the error in the source code. This includes not only the lines of the source code where the error appeared but also the cause of the error. In addition, the static analysis of the source code is able to spot the regions which violate the common accepted programming patterns or which expose the security thread.

A.7.3 Middle End

The middlend of the compiler is responsible for the IR manipulation. To this end, it uses the analysis passes, the optimization passes as well as the transformation passes. The number of available passes is high and in the LLVM version 2.5 reached almost one hundred.

The middlend of LLVM has a modular design which helps to reduce the labor time when developing new passes. The newly developed pass can be included in the chain of passes and can reuse the results from other passes, where this task is performed automatically under the available framework. In addition, the development of new passes is simplified by the fact that LLVM has a language independent type system and uses the SSA form.

As previously stated, the IR is invariant to HLL and since the middlend passes work with the IR, in the end, a single pass is functional for many HLLs. Moreover, passes can be used many times for the same code, which provides a loop where the IR is constantly analyzed and transformed between the loop iterations.

A.7.4 Back End - I

This part of the compiler is also known under the *code generator* term. In this back end, the generated code is target independent, which means that it is not bound to any specific hardware architecture. The target independent code generator transform the IR into the target instructions (TI). While IRs instructions target the abstract machine the TIs target universal processor with universal ISA. The code generated with TIs does not contain any specific machine encodings.

The code generation task is performed with the usage of DAGs that are built from the IR in the first phase and are followed by the optimization phase. In the *legalization* phase, the parts of the DAG which cannot be expressed with the TIs are converted into other code. Therefore, the legalization phase ensures that the DAG leaving this phase can be covered with the TI. Otherwise, the legalization will inform about inconsistence problems and will precisely inform where the error occurred and which part of the DAG cannot be covered with TIs. After the legalization and the optimization phase the final *instruction selection* process happens, which is a pattern matching algorithm which disassembles the DAG into the TI instructions. Thus, it changes the representation form from the graph based to the textual one. The pattern matching algorithm consists of prioritizer and thus, it is important to adapt it and find relevant priority when supplementing the backend with additional nodes; see Appendix C.

A.7.5 Back End - II

The second backend is a target dependent process which aims at translating the independent code into a dependent one. The depended code known as assembly is suited for a given hardware architecture. To this end, the assembly code uses target specific informations that mostly correspond to the application binary interface (ABI). This not only includes the data types supported by the architecture but also the alignment and the calling convention. Most of all the target specific informations describe the ISA of the given architecture.

In this backend, several transformations are performed which for instance change the virtual registers into the physical ones (register allocation), schedule the code, and map independent instructions into the dependent ones. The target specific informations are specified in a target description file that uses declarative DSL, which simplifies the task of describing hardware architectures.

A.8 Dynamic vs. Static Translation

The difference between the dynamic and static translation relies on the following fact. The dynamic system and also interpreter, transform the application's high level source code concurrently to the code execution whereas the static system is executed *ahead-of-time*. Thus, in the case of the dynamic translator, the processor is not only used to execute the code but also to produce it. While this results in overheads that are not found in the static translation, often it outcomes in a much better processing performances. This is due to availability of informations such as application data types, values of the variables or the pointers that are not accessible by the static translation. These informations enable the set of new online adaptive optimizations that allow to translate and then execute the code more effectively than in the case of the static translation. Moreover, in contrast to the *eager evaluation* computation paradigm found in the static translation and imperative languages, this allows also for *lazy evaluation* modes and for accurate dead code elimination.

These informations also play a significant role in the hardware-software partitioning process where the execution of the application is split between the processor and dedicated hardware accelerators. This behavior is achieved due to availability since the dependencies between the variables and their layouts in the memory are known. These informations allow the

hardware accelerator to fetch the data directly from the memory without disturbing the processor and start the computation in parallel to the processor without the trouble interlocking and data sharing mechanisms.

A.9 Interpretation vs. Dynamic Translation

The difference between the interpreter and the dynamic translator relies on the fact that the interpreter produces the *bytecode* whereas the dynamic translator the *machine code*. The machine code is fed directly into the processor for the execution while the bytecode is executed by the virtual machine or by the interpreter. Another differences results from the runtimes. The bytecode generation is a simpler process than the machine code and thus, the interpretation is a faster process than the dynamic translation.

The similarity between them rely on two facts. First, both have the same input; that is a high level language source code. Second, they perform their job (interpretation and translation or compilation) concurrently into the executed code.

For both systems, the interpretation or translation of the source code and execution of that code are performed under a single step whereas for the static translation under two separated steps. This separation closes access to many important informations that are available in these online systems. On the other hand, the runtime of these online systems is accumulated together with the code execution and can outcome in significantly longer overall execution runtime. To this end, two methods were developed that try to prevent and minimize these long runtimes.

A.9.1 First Method: Caching

In general case, the online systems interpret or translate the source code every time it is invoked. Thus, this operation is very ineffective if the source code is invoked many times. In such cases, the overheads involved in these processes are accumulated and significantly increased the overall runtime of the application. In order to avoid this to happen, the caching mechanisms are used as tabulation mechanisms.

The caching mechanism is used when the code is invoked for the first time. Thus, the overheads involved in further invocations of the code are avoided. Instead of the interpretation or translation the cache reference is used to the final code; bytecode or to the machine code.

While this scenario has the advantage that every invoked code will be found in the cache, the caching itself is often not as efficient as it sounds. The problem occurs in the beginnings of the application execution where most of the application's code is needed and the caches are still empty. In this case, the overheads burden take most of the computation time and result in inefficient application's performances. In addition, the caching practice does not pay off for the code that is invoked sporadically. In particular, this is a case for the dynamic translator that requires much longer runtimes than the interpreter. For such a code, the process of interpretation brings better performances then the caching mechanisms and dynamic translation.

A.9.2 Second Method: Just-in-Time Compilation

The second method is a heterogenous one where the compilation and interpretation techniques are asymmetrically merged together. This method is known under the term just-in-time compilation and it was developed in order to solve the drawbacks of the first method. The JIT directs the compilation process only to the parts of the source code which are most frequently invoked whereas the remaining code is interpreted. This sacrifices the additional time needed for the dynamic compilation process only to the parts of the application which are most frequently executed. This leads to overall higher performances and shorter runtimes. From this perspective, the JIT compilation approach can be seen as the enhanced profile-driven extension to the dynamic compilation process.

The JIT approach has been strongly used in very effective second generation Transmeta's Efficieon processor [102]. In this design, the JIT consists of four different gears that are shifted when the code reaches certain invocation threshold (frequency). Each gear increases the time for online optimizations and results in a more effective machine code being generated.

A.10 Related Work

In general, the research compilers are usually used for prototyping new language features or language optimizations and are not robust enough or complete to handle real, large applications. The development of industrial strength compiler requires many years effort which usually is beyond the available resources of the researchers and thus, not many open sourced mature compilers are available for them. LLVM can be seen as the exception in this field since it is actively developed by the Apple team, it is used in their products, and has a licensing model similar to the BSD which is open for the researchers as well as for proprietary projects. In addition, LLVM is a very lively project with a strong technical support, broad documentation, and active developer community. This environment significantly increases the researcher's productivity.

There are many research compilers around targeting at different tasks and it is not the aim and possibility to evaluate all of them in this work. LLVM puts emphasis at the middlend inter-procedural optimizations and therefore compilers which excel in the same field will be evaluated. This includes the SUIF compiler infrastructure and the SGI's Open64 compiler.

SUIF Compiler

The SUIF compiler infrastructure [103] is probably one of the most used compilers in the research community. SUIF is a source-to-source translator and it consists of a high level abstract syntax tree (AST) representation. It has powerful inter-procedural transformations which can be profile driven. Besides that it is inactive since 2001 and it had many drawbacks. First, it was very slow because of very large and general AST representation. Next, it had limited capabilities to attach new frontends since they required adding new node types to the AST. Finally, any AST extensions broke down the compatibility of the existing optimizations and therefore required constant updates, process which was very cumbersome.

Open64 Compiler

The Open64 compiler is descendant of the SGI's commercial MIPSPro compiler developed between 2001-2003 and it has been opened to the community under the GPL license model. In proceeding years it was up-taken by various commercial and academical institutions. Thus, many forks of the original Open64 exists i.e. Path64 each resulting in different features and limitations. While every fork has it's own goals to fulfill and directions in which it advances, this outcomes in the lack of consistent documentation and consistent developer community. Besides these environmental drawbacks, Open64 is a high quality industrial strength compiler focusing on the ahead-of time compilation and generating a robust code. While it has been developed for more than a decade, it consists of many optimizations built around the inter-procedural and profile driven transformations. Open64 uses WHIRL IR which has five different levels starting from the language specific representation, ending at the machine specific representation, and it uses lowering system to transform between these levels. The development of optimizations is more difficult than in LLVM since WHIRL is not the SSA IR.

Runtime Support

Both mentioned compilers and LLVM excel at the middle level optimizations. The SUIF and Open64 do not natively support the dynamic translation in contrast to LLVM. Therefore, they were discarded as the fundamental compiler infrastructural for our system.

While in SUIF this functionality could be enabled by external extensions it is not supported in Open64 at all. Both compilers are focused on the static compilation techniques. Thus, development of our tool flow with these compilers would be significantly harder than with LLVM.

Appendix B

Strange Loop Perpsective

B.1 Introduction

Computer systems are constructed from a pile of abstractions. The two main characteristics that describe the process of computation on a computer system are: the performances and the labor time. The labor time corresponds to the amount of time the developer spends in order to design and develop algorithm that describes the process of computation. These two characteristics strongly depend between on other.

- The high labor time usually results in high performances. For instance, this is a case when a developer uses a lower computer abstraction that is not as expressive as the higher one in order to develop an application specific hardware accelerator. While this results in additional time being spent for this purpose - the developed accelerator allows to increase the processing performances.

- The lower labor time usually results in lower performances. This is a case when a programmer saves labor time by using higher abstraction that is more expressive than the ones bellow. For instance, this happens when the developer decides to use a higher level interpreted programming languages like Ruby in contrast to the compiled lower level programming languages like assemblers. While this saves the labor time, the generated code often does not achieve as high performances as the one generated with lower level language.[1]

In real world, there exists an equilibrium between these two characteristics that is regulated and driven by the economic trends; see Figure B.1. Depending on the given task to solve, this equilibrium is adjusted by moving and selecting proper level of abstraction.

End of Processing Performance Increase Era

Unfortunately, the sequential computer system as we used to know and utilize had reached his processing performance limits. This situation occurred due to limitations that appeared in semiconductor field. These limitations have strong influence on the hardware abstraction

[1] Please note that these languages are Turing complete (computationally universal).

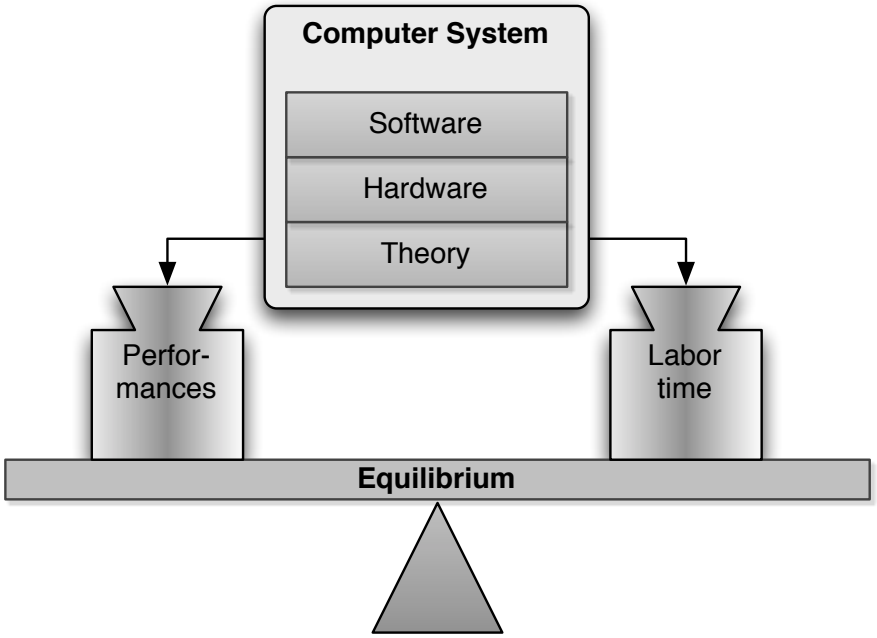

Figure B.1: Equilibrium between performances and labor time in a computer system.

presented in Figure B.1 . Therefore, the equilibrium is imbalanced and leads to an instability problem.

Contributions

In this work, we study the economics of new equilibrium (that is the performances and labor time) for a computer system that contains a *strange loop*, which is a paradox studied by Hofstadter [21]. In a convenient computer system a relation between software and hardware exists where software is executed on the hardware. In our developed system, we added another relation where the hardware is adapted by the outcomes of the software process. Both relations together lead to a paradox – to the strange loop. The very unique feature of our developed strange loop system is that it is autonomous and fully dynamic, and in consequence, the loop iteration occurs without any manual efforts.

Outline

In the first section, we describe the process of abstraction and strange loop construction from a computer design perspective. Next, we show how the process of creating abstractions reduces the labor time. To this end, we describe in details the characteristics of the abstraction process and how they interfere with each other. Afterwards, we study the characteristics of systems that are built from the abstraction structures like stacks. Since the prime example of such structures are computer systems we present it in the next section. Finally, we describe the instability problem that is caused by the equilibrium imbalance in computer systems and possible solutions.

B.2 Computer Design: Process of Abstraction and Strange Loop

The design of the modern computer is a tremendous task and many have tried before but without a success [104]. There is a golden rule which needs to be known to perform this task and the ones which did not recognized it failed early [105]. This rule is the process of abstraction.

Process of Abstraction

The idea behind the abstraction is very easy. It is to encapsulate a part of the complex system into the component. The communication with the component is performed by a strict and predefined interface which reduces the complexity and hides the unimportant details from the rest of the world. While, on one hand, the interface allows to merge the component with other components and build systems with ease, at the same time, it isolates and provides the freedom to expand independently of the others.

Advancing from Abstractions

The usage of the abstraction, components, and their interfaces can be seen as a methodology to develop better systems and tools. Every component hides the complexity and the stack of them are hidden even more. This allows to model the complex systems painlessly. Since the interface of the top component has syntax and semantics which is rich, expressive, adjusted to the modeled system, and allows to describe it in an effortless manner. The design of the compiler or the operating system are good examples of this idea.

Paradox and Strange Loop

The idea of building the complex systems from the stack of components is the first and simple method of advancing from the abstractions. There is also the other more powerful method or rather *paradox* known under the name strange loop which is presented in Figure B.2. The strange loop occurs in the heap of the components when the upper component adjusts and modifies the one bellow it. This again has the influence on the upper component and brings the performance gains which are not possible to achieve only by it. In the philosophy, this behavior is known as the *consciousness*.

Strange Loop in Computer Design

In computer science in order to perform the strange loop the upper component *emulates* the behavior of component bellow him. The results of the emulation are then used to refine the bottom components. In the design of the computer systems, this is a case between the hardware and computational theory where dedicated hardware allows to reason about the theory. Also this is a case between software and hardware were the software allows to design better hardware. The transition between abstractions has to occur neither immediately nor automatically as it is the case in the examples presented above.

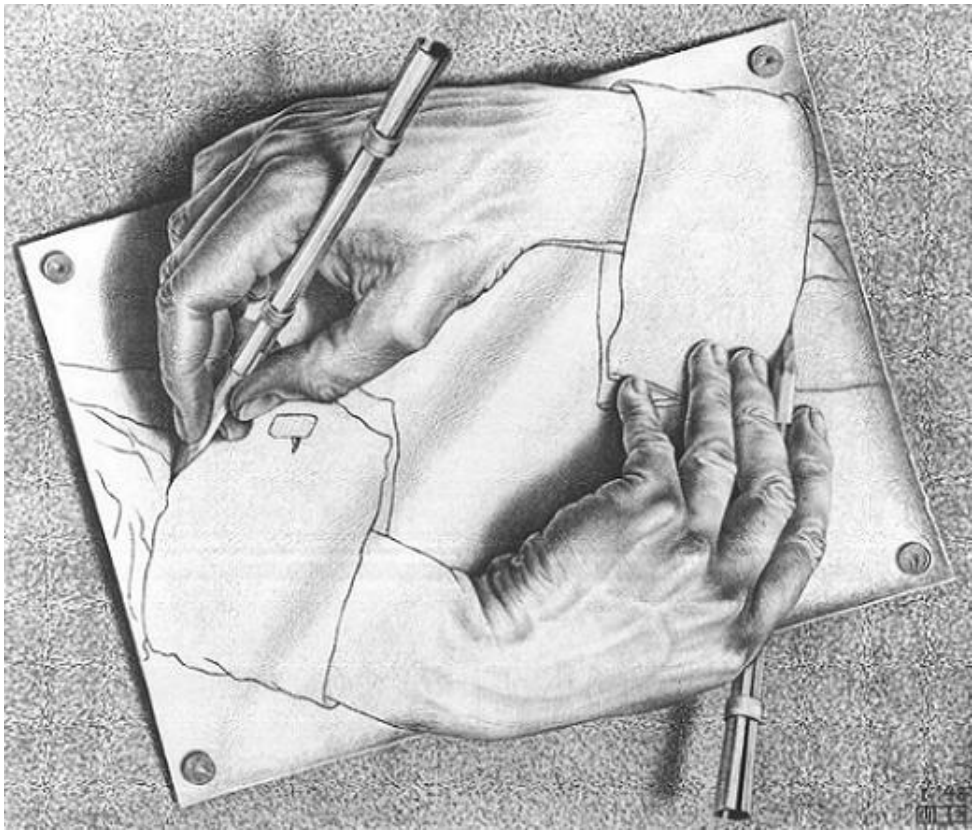

Figure B.2: *Drawing Hands* by M.C. Escher.

The *runtime* of the loop is a parameter of the system which corresponds to the time of a full iteration of the loop. This parameter strongly depends on the *design of the loop* and how the transition is performed i.e. automatically, manually, or hybrid. While the runtime can be seen as the *price of the loop iteration*, the adaptation and the *performance* impact can be seen as the *rewards of the loop iteration*, and the ratio between these two as the *effectiveness* of strange loop. It is not always the case that the effectiveness will increase with every or any loop iteration. Therefore, there exists an economical balance between factor, which is strongly influenced by the design of the loop.

Strange Loop in Arts

In computer science, the strange loop often connects two separate and unrelated science disciplines like theory and electronics or hardware and software. The phenomena of this process relies in taking parts and assembling them with the help of the *isomorphism* in a way which leads to the creation of the loop. The right patterns of this process are hard to find but they appear not only in the computer designs but also in the most profound musical and art compositions [21].

B.3 Abstraction Characteristics

The design of the computer as well as the strange loop studied in this work rely on the process of abstraction. Therefore, it is valuable to understand the properties of this process, its benefits, drawbacks, and implications. These are presented in Figure B.3.

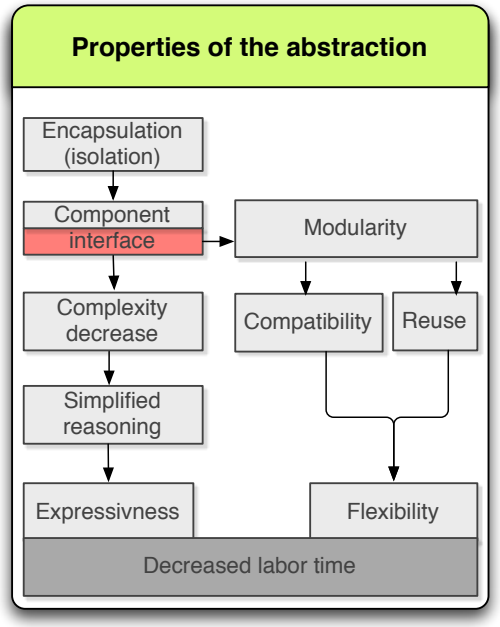

Figure B.3: Abstraction characteristics and their implications.

Properties

The abstraction is in charge of the encapsulation of the part of the system into a separate component. The details of the component are hidden underneath the interface (marked with the red color), which acts as a communication filter. It not only exposes the core functionality of the component and hides all unimportant details from the user but it also provides a boarder between the implementation and the definition. Therefore, from the user point of view the component appears as a *black box*, since all the irrelevant details are hidden. This has two fundamental consequences. First, it *decreases the complexity* of the component which leads to *simplified reasoning*. The second consequence is that while the component is black boxed it can be *replaced* by the others if the interface stays in touch and functionality intact. This, on one hand, brings the *modularity* to the design and, on the other hand, the *compatibility* and the *reuse* methodology.

Decreased Labor Time

The abstraction saves the *labor time* by letting in the *expressiveness* and the *flexibility*. The fact that the component is easy to understand allows to utilize it in an eloquent and efficient

manner. The flexibility, on the other hand, allows to easily modify or replace the component without breaking any system dependencies. The decreased labor time stands as a major property which justifies and authorizes the usage of the abstraction.

B.4 Abstraction Stacks

A single component encapsulates a part of the system and decreases the complexity of that part. However, it is hard to sufficiently decrease the complexity of the full system only within a single component. It is because the modeled systems are too large, have too many parts, and are too complex. Therefore, in order to decrease both the complexity of the system and the labor time at the same time the abstraction stack is used.

Case Study

In the left part of Figure B.4, a complex system is presented and next to it the same system is expressed with the help of abstractions. Every abstraction is laid down on the other, which forms an *abstraction stack* structure growing upwards. This has two major consequences for complex systems like compilers or operating systems.

- The first consequence is the *automation* which significantly decreases the labor time and allows to develop tools that automatically transform a given description expressed in the higher level abstraction to the lower level ones. In other words, the description found at the *(i+6)th* interface can be translated automatically to the *(i+5)th* and so on until the *bottom* line. The stack of the abstraction has the complexity denoted with the *CA* and since the automation allows to carry it automatically it significantly reduces the complexity of the full system (CFS) and the necessary labor time.

- The second outcome deals with the *granularity* of the top interface found in the stack. This granularity describes the level of the syntax and the semantics of the interfaces found in the abstractions. The larger the granularity (the higher stack), the more eloquent, expressive, and meaningful syntax of the interface. The interface which has these properties is more adapted to the modeled system and it allows to model large parts of that system with ease. It reduces the complexity of the system (CS) and with the help of the automation it shrinks the labor time.

Large Abstraction Stacks and Reduced Labor Time

The height of the stack is proportional to the number of abstractions. The large stack will significantly decrease the labor time, which is due to the automation and the large granularity of the interface. The automation releases a large portion of the system complexity from the manual efforts, whereas the large granularity allows to describe the system effectively with an expressive syntax.

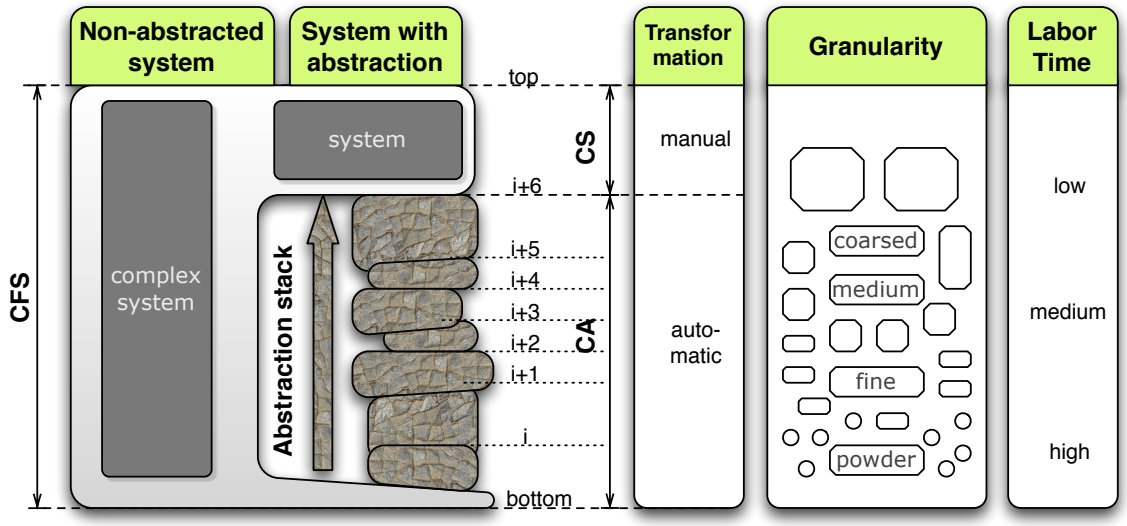

Figure B.4: The structure of the abstractions and the implications of the interface and the labor time on the *granularity*. Here, i enumerates the interface, *CFS* means the complexity of the full modeled system, *CA* complexity of the abstractions, and *CS* complexity of the system. Therefore, $CFS = CA + CS$, whereas *CA* depends on the number of i, and labor time is proportional to the *CA*.

B.5 Abstraction Price

At first sight, it might seem that it is always profitable to build higher stacks. Unfortunately, this it is not the case. Every abstraction comes at a certain price and decreases the complexity, which leads to the reduced labor time, but at the same time decreased complexity has a negative impact on the performances. It is due to the fact, that the abstraction gains the reduced complexity by hiding details and reflecting only the general use case of the component. Therefore, a given description can be implemented with the lower abstraction more precisely, reaching higher performances, but at the higher costs of the labor time. In other words the description implemented with the $i+6$th interface can be implemented with the $i+5$th interface or even with the ith interface, which will result in increased performances but at the same time it will significantly increase the labor time.

B.6 Equilibrium: Performances and Labor Time

The performances and the labor time are characteristics of the same abstraction stack. They are in inverse proportions to each other which leads to a trade-off situation. In order to achieve the modest system, these two factors are driven by economic trends and result in an equilibrium.

This situation is presented in Figure B.5. On the vertical left axis, the height of the stack is represented, which is proportional to the stack presented in Figure B.4. The horizontal

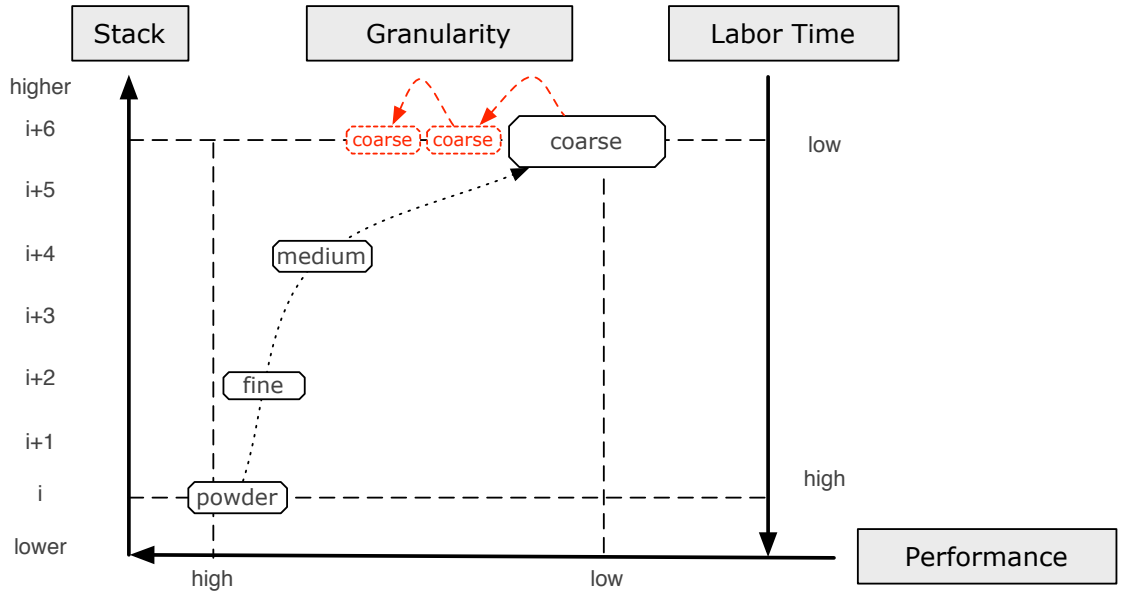

Figure B.5: Balance between the advantages and the penalties of the abstraction. The red color represents the perfect balance situation where the increase of the performances is achieved without any losses in the labor time.

axis corresponds to the performance and the right vertical axis to the labor time. The highest performance *max(performance)* is achieved when the minimal number of abstractions is used *min(i)* and when the size of the stack is minimal. Unfortunately, for this scenario the labor time reaches the maximum. It is due to the fact that $CFS = CA + CS$ and in the case when i has a minimal value so does the CA. In that case, the equation takes the form $CFS = min(CA) + CS \Rightarrow CFS = CS$ and this is the reason why the labor time has to deal with the CS which is equivalent to the CFS. This is also why no automatic transformation is achieved and the whole implementation of the CFS is shifted to manual efforts, which provides the maximum labor time.

B.6.1 Perfect Equilibrium

A perfect balance appears when the low labor time and the high performances are both sustained. This behavior is represented with the red color in Figure B.5, where the *coarse granularity* is moved to the left. This means that the same description expressed with the top interface is able to achieve higher performances. This scenario is possible to reach only when the lower abstractions are replaced with the ones of higher performances. This is not a hypothetical concept, such scenario has been indeed achieved for many years in the computer industry and will be explained bellow.

B.7 Abstracted Model of a Computer and Computation Process

In Figure B.6, the abstraction model of the simplified computer and computations is presented. The computer is based on several fields abstracted from each other. We can bucket them into three domains: theory, implementation, and model. The theory tells how to build an universal machine which allows to emulate other machines. In addition, it specifies what are the limitations of this approach i.e. what it can actually emulate. In the theoretical world, the machine is called a *formal system* and the emulation means that the universal machine is able to read the description of other machines, simulate their behavior, and provide with results. The implementation moves the theory from the paper descriptions to the real world hardware devices. These devices alone do not perform any tasks and, as in the case of the universal machine, they need the description of the other machine (formal system) to be emulated. The process of describing a machine which behavior will be emulated in the modern world is called a *software programming*. The software delivers a *program* which describes the behavior of the machine as well as the input data. The program consists of an algorithm that models the real world behavior. The knowledge, experience, and smartness of the programmer will influence the efficiency and the accuracy of the algorithm.

The computer architecture is a prime, impressive, and unique example of the stack abstractions. It is impressive that it spans over several different fields like mathematics, hardware, and software. Each of these is enlarging the granularity at a tremendous level. It is unique since it always had a perfect balance between the performances and the programmer productivity (labor time), which is represented with the red color in Figure B.5. It allows to sustain the low labor time and to increase the performances at the same time. This lets to shift the balance only on the horizontal axis without any losses in the abstraction stack, increases in the system complexity *CS* found in Figure B.4, or any refinements in the interfaces.

B.8 Instability Problem in Computer System Equilibrium

The perfect balance in years from 1978 to 1986 noticed increase of the performances by a rate of 25% per year and until 2002 by the 52% rate [4]. This means that every 5 years the performance increased more than 8 times. This raised performance was an outcome of a constant advancements in the field of the semiconductor technology [1], which constantly increased the operating frequency. On one hand, for more than three decades, the progress in this field allowed to produce faster digital switching circuits whereas the properties of the abstraction allowed to replace them without any implications in the stack. During these years the technology and frequency scaling was almost solely a factor of the performances increase and allowed to obtain the perfect balance situation between the performances and the labor time, see red color in Figure B.5.

Unfortunately, after year 2002 the grow of the technology stopped due to several technology limitations including the brick wall presented in Section 1.5. The increase of the performances as we know, was no longer possible only within the semiconductor technology - a single abstraction. That was the case almost a decade ago and it is the case even now. This

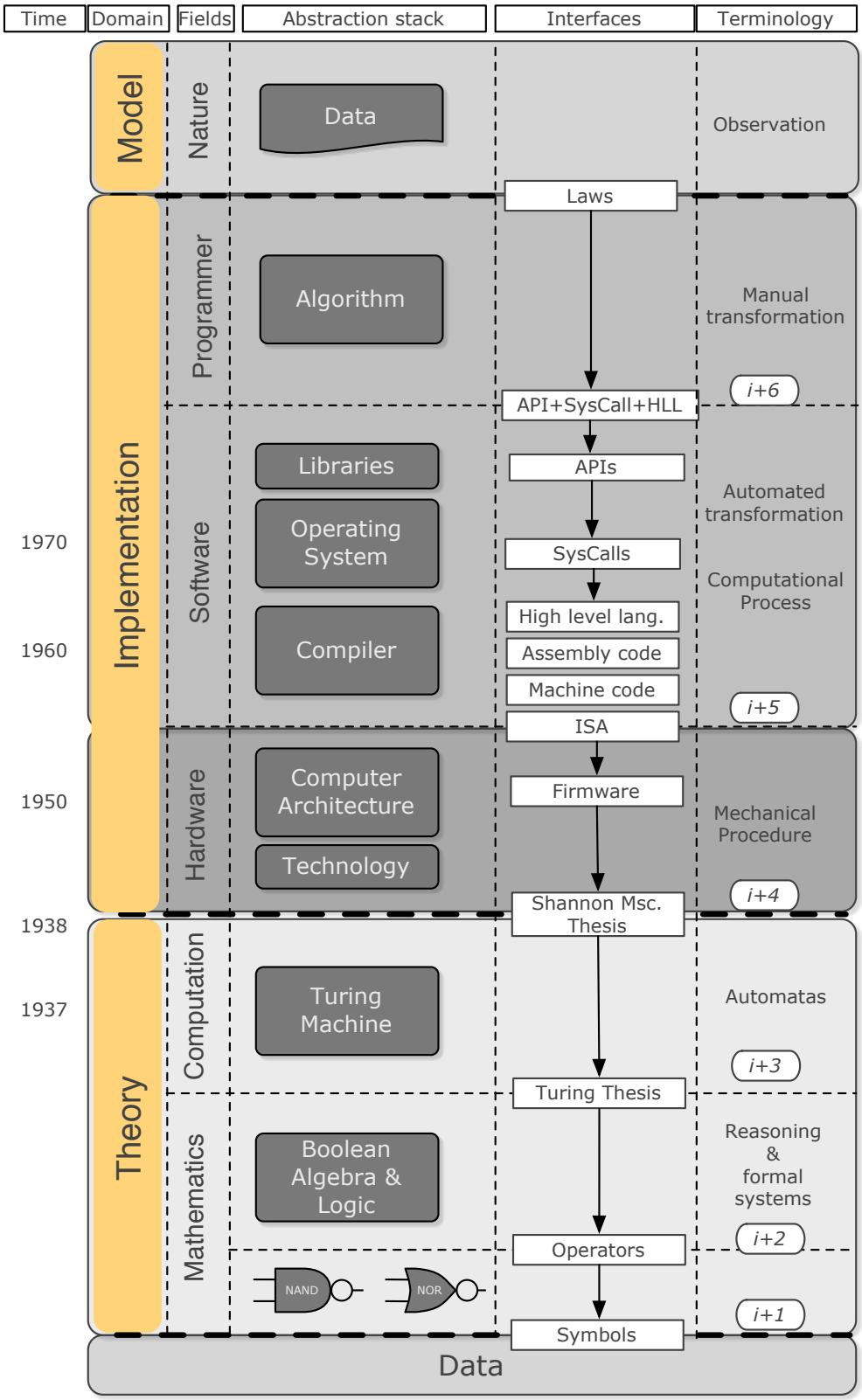

Figure B.6: Model of computation with defined abstractions.

results in the balance instability problem for which new equilibrium needs to be established and the increase rate of the performances restored.

B.8.1 Limitations of Sequential Computing

The instability problem has many more drawbacks than it might seem at first sight because it shows the boundaries of the sequential computing. The origins of the sequential computing are derived from the *Turing Machine* which is a theoretical model of the computations. The computer architecture is based on that model and the advancements of the technology allowed for frequency scaling and processing performance increase. In other words, the instability problem tells that the limits of the sequential computing have been reached and since the roots of the sequential computing are derived from the bottom abstraction, the instability problem cannot be fixed only with small refinements but rather by significant changes in the stack. These observations make the solution to the instability problem even more challenging.

B.9 Solutions and Their Limitations

The practical limits of the Turing Machine and the sequential computing have been reached. Therefore, the only solution to the instability problem and computing performances increase leads towards exploiting the parallel computing paradigm. This paradigm shift started to occur in year 2005 when the first multi-core[2] architecture has been released and in academia in year 1997 [106].

B.9.1 Top Abstraction Stays in Place

From the abstraction point of view, there are only two approaches to this scenario. The one which modifies the top abstraction and the interface, and the second one without modifications. They both have to focus on the top abstraction since skipping it in current times would require different software programming concepts and, in consequence, would damage all the software legacy models.

The top abstraction and the interface *(i+6)* provides a crossing between the manual and the automatic transformations. Models which would have to fit into lower interface would have to be develop from scratch or significantly adjusted. In both cases, that would cause a notable shift between the manual versus automatic transformations and between the complexity of the abstraction *(CA)* versus the complexity of the system *(CS)*, see Figure B.4. This shift would sacrifice too much labor time. Therefore, the top abstraction and the interface have to stay at the same place, they cannot be removed, in contrast to the ones bellow them, since this would definitively increase the labor time to the point where the instability problem could not be solved.

[2]Intel Pentium Processor Extreme Edition 840

B.9.2 Adaptions to the Top Abstraction

The top abstraction and the interface can be adapted to the parallel paradigm conditions, which differentiates the two mentioned solutions. The first one adapts the top interface and allows the programmer to control and manage the high level parallelism explicitly. The second one does not adapt the interface and expresses the lower level parallelism implicitly. The difference between these levels corresponds to the type of the parallelism that they support. The explicit parallelism targets the task and thread level parallelism, whereas the implicit parallelism at the instruction and bit level parallelism. The explicit parallelism has to be clearly and precisely specified during the code whereas the implicit one is unstated and inherent.

B.9.3 Adaptation with Explicit Parallelism

While the first solution can lead to better performance gains it brings the benefits only for models whose origins are truly parallel. In order to express that parallelism, the programmer needs to manually perform the division and synchronization as well as agglomeration and mapping of the concurrent parts of the model. These are very difficult tasks that cannot be automated and therefore, they are occupied by a significant labor time and correspond to the notable vertical downward shift in Figure B.5. In addition, the computing performances are burden with significant synchronization costs involved with managing of the concurrent tasks. Moreover, it is worth to notice that the current *von Neumann* computer architecture which is based on the concept of the Turing Machine and the sequential computing paradigm has many drawbacks when used to express the explicit parallelism, and has been criticized many times [88–90]. This can be clearly seen within the memory subsystem which is designed to support the sequential computing paradigm and not the concurrent one [91]. While it brings benefits for the sequential computing with the streaming mode, where large sequential chunks of data are accessed with only a few instructions, it is not the case for the concurrent and random accesses. This behavior leads to the memory partitioning problem but more importantly to the memory bottleneck known as the *von Neumann bottleneck* and to decreased computing performances, in particular for the parallel paradigm. The other major problems involved in the concurrent computing on the sequential computer involve a proper partitioning of the memory and ensuring the data coherence between the concurrent tasks.

B.9.4 Adaptation with Implicit Parallelism

While it may seem that the high level explicit parallelism model described above has been chosen by the manufactures to deal with the instability problem, it is not necessary the best one. It is because it is based on an old sequential computing paradigm which uses instruction-streams and faces many drawbacks when used as a concurrent paradigm that is based on data-streams. Probably the most challenging and demanding economically solution to the instability problem would be to redesign the abstraction stack from scratch and to begin with replacing the von Neumann model with the concurrent theoretical computing model. The lambda calculus with Church-Rosser property is a good example, since it has no side effects and avoids mutable states and mutable data [24]. While this would eliminate all the architectural constraints it would completely discard the von Neumann architecture at the same time. It would break all

the software legacy models, and be occupied by incredibly high labor time. This is the reason why the bridge solution between the old von Neumann and the new concurrent computing paradigm needs to be accomplished. It is also the reason why this work proposes a different approach to the instability problem. This thesis focuses on the low level implicit parallelism with the intact top abstraction and interface stack. This allows to use all programming legacy models but at the same time it provides the dynamic strange loop that modifies the hardware accordingly to the software needs and, in consequence, significantly increases the processing performances.

B.10 Contributions

This work provides a solution to the instability problem by supplying a new and unique dynamic strange loop. The developed loop is universal since it supports current legacy models, does not break them, nor puts any emphasis on the labor time since it keeps the top abstraction and the interface intact. It is based on a new field named *Reconfigurable Computing* (RC), which in contrast to the traditional hardwired fixed solutions, allows to change the hardware. In this solution, the loop adapts the hardware to the low level implicit parallelism found in the computing model. Put simply, it shifts the computation from higher level abstraction to the lower one, which is responsible for the rewards of the loop iteration and the performance gains, which are accounted from two factors. First, the shift eliminates the price of the abstractions between high and low levels, which was described in Section B.5. Secondly, it exploits the benefits of the parallelism found in the computing model. The exploited parallelism changes the computation from the temporal software to the hardware spacial domain, which increases the *instruction per cycle* rate and brings the performance gains.

The strange loop was designed as autonomous and automatic one, which means that the iteration happens without any manual efforts. The transition between the abstraction levels is automated and transparent to the user, and occurs during the execution of the computation model. The time needed for this transition includes the process of the hardware generation and the adaptation, which is known to be very time consuming. Therefore, in order to increase the effectiveness of the loop, special optimization mechanisms were designed and developed for this system and the loop.

This work provides with following contributions:

1. First, it presents a solution to the *instability problem* by designing and developing a novel dynamic strange loop. The presented loop finds and extracts the implicit parallelism in the computational model, generates corresponding hardware circuits, and adapts the software to it. It is easy to use, since it works transparently and automatically and it does not require any manual efforts. It supports all legacy models and most of all it does not increase the labor time for the new ones.

2. Next, it empirically evaluates the feasibility and applicability of the developed loop and, thus, it studies the equilibrium of a new system. For a large set of applications, it investigates the economical balance points between the rewards and the prices of the loop iteration. In addition, it studies unique optimization mechanisms that were developed in order to reduce the runtime and to provide better effectiveness ratios.

3. Finally, this work performs a detailed analysis of the proposed approach. It reveals in details where does the performance gains come from in a mixed sequential and a concurrent computing paradigm environment. To this end, detailed explanations allowing for a better understanding of the instability problem and its implications in the abstraction stack are presented. A broad range of topics is studied starting from the software and ending at the hardware architecture, i.e. automatic methods for parallelism extraction, efficient methods to prune the design and search spaces, or data path synthesis (generating hardware circuits from a software description).

Appendix C

Other Approaches to the Pathway

During the initial work on the Woolcano compiler the generation of the UDCIs in the software runtime adaptation were handled without the intrinsic support. The UDCI generation task requires DAGs representation of the code. Since DAGs are already available in both backends it seemed natural to generate the UDCI code from there and not from the middlend where external libraries are required to construct DAGs. This approach has the advantage of resulting the code being kept under a single LLVM framework without any requirements to the other libraries that duplicate the already existed functionality.

Backends Extensions

To test the feasibility of this approach the first backend of the Woolcano compiler was extended with user predefined patterns and with additional artificial DAG nodes. In the later phase, the user predefined patterns were replaced by the patterns generated by the UDCI candidate identification algorithms; see Section 5.4. The user predefined patterns were equivalent to the functionality of the UDCIs and were applied to the DAGs; see Figure C.1. If found, the DAG nodes were folded into an artificial node which later in the second backend was mapped to the UDCI; see Figure C.2.

Prioritizer and Integrity Checker

In order to perform this operations successfully, the compiler extensions had to ensure that the user predefined patterns were applied to the DAG before the other patterns. Otherwise, the DAG was split on parts and the user defined pattern was not matched in a graph. To this end, the prioritizing mechanisms were developed where the user defined patterns had the highest priorities.

After folding the DAG to the artificial node, the compiler had to ensure that the DAG could be still covered by processor instructions. If that was not possible it meant that the proper code could not be generated and, in consequence, not all user defined patterns were accepted. Therefore, a special integrity checker had to be developed that would test and validate if the DAG can be covered by all patterns.

Cumbersome Approach

While this approach is fully functional in practice and it allows to keep the development of the Woolcano compiler under a single framework, however, it occurred to be very tedious. The definitions of the user defined patterns were cumbersome to specify and additional costs involved with the integrity checker made this approach not attractive. These are the main reasons why this approach was later changed to the intrinsic one even though it required external libraries for DAG construction.

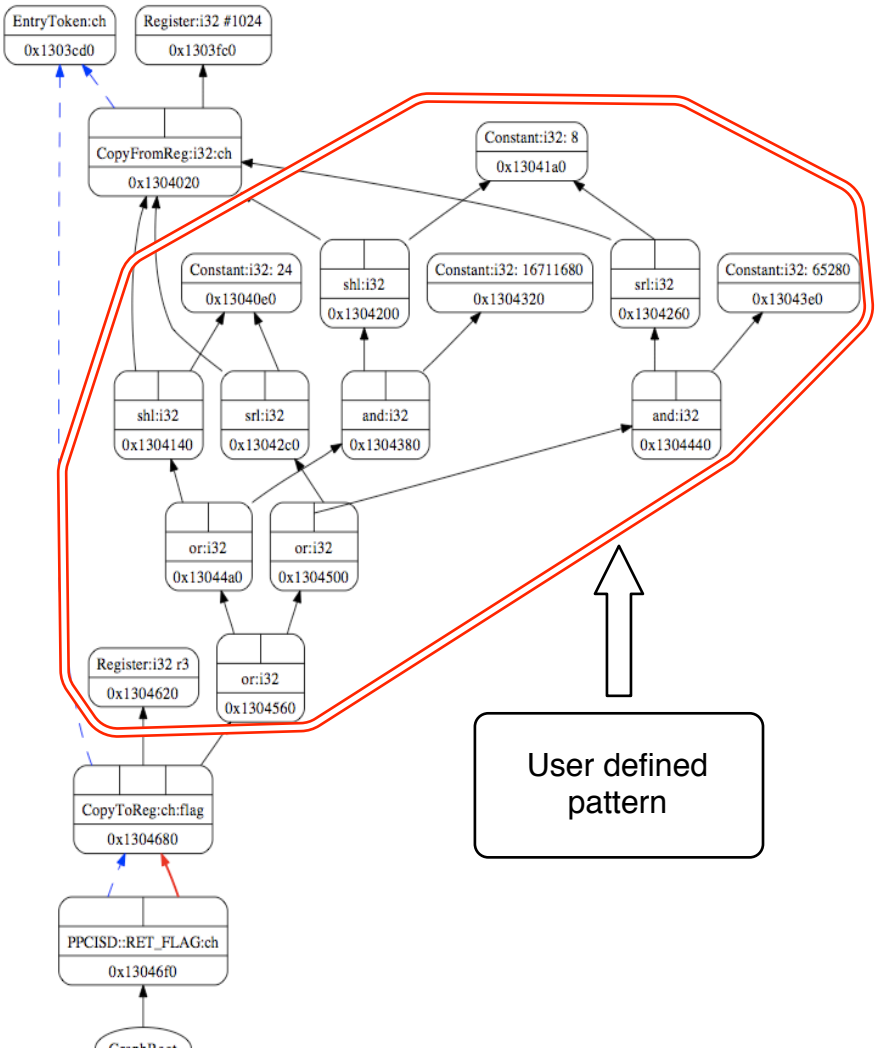

Figure C.1: Initial DAG presented from the first backend, see Figure A.1.

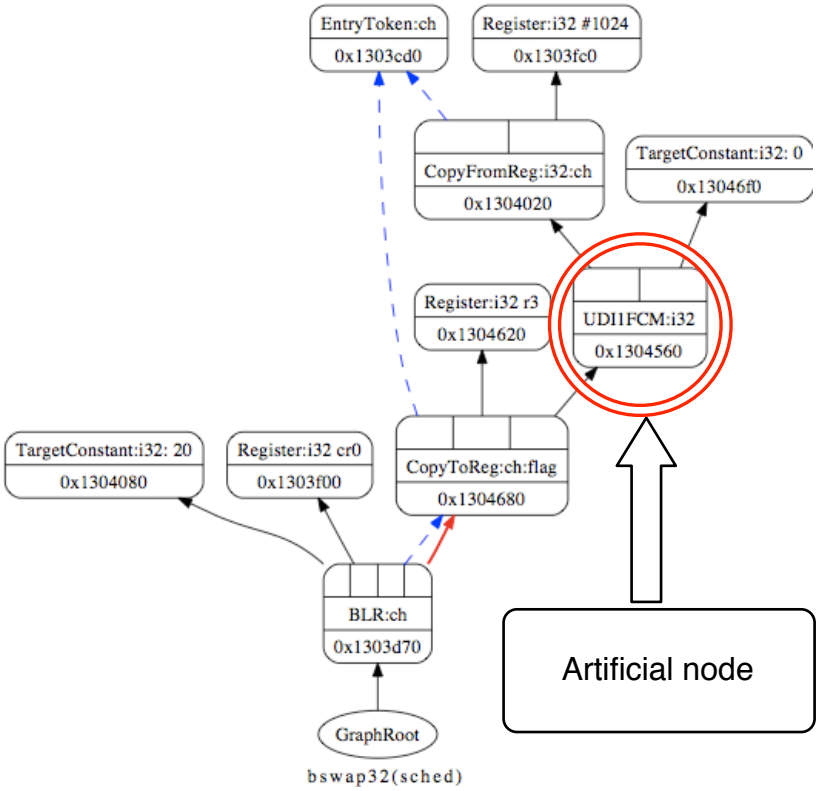

Figure C.2: DAG with an artificial node leaving the *instruction selection* process and entering the *scheduling* process in the second backend.

Appendix D

Pruning Design Space

App	# Instr	# BB	ISE algorithm runtime [ms]			Saved Clock Cycles [10^6]			Speedup		
ISE alg.			MM	SC	UN	MM	SC	UN	MM	SC	UN
adpcm	305	43	1.7	15.0	3869.4	20945.6	26718.0	25753.3	1.24	1.31	1.29
aes	8972	200	40.6	31624.9	62776.6	0.2	0.1	0.1	1.64	1.32	1.32
blowfish	1746	59	8.0	2495.7	1147.3	0.7	0.7	0.7	1.43	1.43	1.43
cjpeg	19468	2483	115.3	1522.8	6191.6	9.7	16.1	16.1	2.09	7.51	7.51
fft	304	47	1.6	9.7	33.1	6.1	8.4	8.4	3.10	14.41	14.41
md5	918	29	3.8	4400.8	20353.4	0.3	0.3	0.3	2.09	1.76	1.76
sha	471	40	2.6	37.8	1314.3	4.4	4.0	3.8	1.41	1.36	1.33
sor	129	19	0.7	4.3	14.6	5.3	5.3	5.3	14.42	14.42	14.42
whetstone	284	44	1.6	9.5	64.0	231.9	231.9	231.9	**18.01**	18.01	**18.01**
401.bzip2	14102	1604	72.3	3839.3	1979.3	0.4	28912.9	28912.9	1.18	1.13	1.13
429.mcf	1917	284	11.1	68.7	200.5	826.5	826.8	826.8	1.11	1.11	1.11
433.milc	14260	1538	78.1	5065.6	-	232062.4	957390.9	-	1.30	21.55	-
444.namd	47534	5147	227.5	35853.6	-	13426391.2	34036379.8	-	1.61	24.85	-
445.gobmk	135500	27469	906.3	5797.5	93029.4	2.1	2.1	2.1	1.11	1.11	1.11
456.hmmer	29337	5518	178.3	76436.2	105699875.0	54745.8	64889.6	64889.5	2.46	3.38	3.38
458.sjeng	20531	3373	123.7	6244.1	235195.7	2651.0	3023.4	3023.4	1.12	1.14	1.14
462.libquantum	5327	1047	32.5	140.2	1167.1	5619.8	5864.6	5864.6	1.27	1.28	1.28
470.lbm	1988	104	8.6	2777.1	0.0	214657.9	344885.1	0.0	2.55	**44.62**	1.00
473.astar	6010	757	33.4	914.8	303796653.0	5792.6	6767.1	6767.0	1.16	1.19	1.19

Table D.1: Specialization process executed for whole applications when targeting the Woolcano architecture without capacity constraints. The performance of the custom instructions has been determined with the PivPav tool. ISE algorithms: MM=MaxMiso, SC=SingleCut, UN=Union. SC & UN search is constrained to 4 inputs and 1 input.

List of Figures

List of Tables

List of Acronyms

ABI	Application Binary Interface
API	Application Programming INterface
APU	Auxiliary Processing Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instrction-Set Processor
ASIP-SP	ASIP Specialization Process
AST	Abstract Syntax Tree
BB	Basic Block
BGL	Boost Graph Library
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CMOS	Complementary Metal Oxide Semiconductor
DAG	Direct Acyclic Graph
DFG	Data Flow Graph
DMIPS	Dhrystone Milion Instruction Per Second
DPS	Data Path Synthesis
DSL	Domain Specific Language
DSP	Digital Signal Processing
EAPR	Early Access PRogramm
EDK	Embedded Design Kit
FPGA	Field Programmable Gate Array
GPR	General Purpose Register
HLL	High Level Language
HRA	Hardware Runtime Adaptaion
IC	Integrated Circuit
ICAP	Internal Configuration Access Point
ILP	Instrction Level Parallelism
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just-In-Time
LLVM	Low Level Virtual Machine
LOC	Lines Of Code
MIMO	Maximum Input Maximum Output
MM	MaxMiso (name of instruction set extension algorithm)
NG	Netlist Generator

ODBC	Open Database Connectivity
PRM	Partial Reconfiguration Module
PRR	Partal Reconfiguration Region
RC	Reconfigurable Computing
SC	SingleCut (name of instruction set extension algorithm)
SRA	Software Runtime Adaptation
SSA	Single Static Assigment
STL	Standard Library
SoC	System on Chip
TI	Target Instruction
UCF	User Constrain File
UDCI	User Defined Custom Instruction
VM	Virtual Machine
vN	von Neumann

List of Publications

[1] Mariusz Grad and Christian Plessl. On the Feasibility and Limitations of Just-in-Time Instruction Set Extension for FPGA-based Reconfigurable Processors. *Int. Journal of Reconfigurable Computing (IJRC)*, September 2011. Accepted for publication.

[2] Mariusz Grad and Christian Plessl. Just-in-Time Instruction Set Extension - Feasibility and Limitations for an FPGA-based Reconfigurable ASIP Architecture. In *Proc. 18th Reconfigurable Architectures Workshop (RAW)*, pages 278–285. IEEE Computer Society, May 2011.

[3] Mariusz Grad and Christian Plessl. Pruning the Design Space for Just-in-Time Processor Customization. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 67–72. IEEE Computer Society, December 2010.

[4] Mariusz Grad and Christian Plessl. An Open Source Circuit Library with Benchmarking Facilities. In *Proc. 10th Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 144–150. CSREA Press, July 2010.

[5] Mariusz Grad and Christian Plessl. Woolcano: An Architecture and Tool Flow for Dynamic Instruction Set Extension on Xilinx Virtex-4 FX. In *Proc. 9th Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 319–322. CSREA Press, July 2009.

[6] Mariusz Grad and Christian Plessl. Poster Abstract: Woolcano – An Architecture and Tool Flow for Dynamic Instruction Set Extension on Xilinx Virtex-4 FX. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, April 2009.

Bibliography

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), October 1974.

[2] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):144, April 1965.

[3] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19:23–29, July 1999.

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

[5] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3:54–62, September 2005.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and et al. The landscape of parallel computing research: A view from Berkeley. Technical Report 2006-183, EECS Department University of California Berkeley, 2006.

[7] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23:20–24, March 1995.

[8] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, ISCA '00, pages 248–259, New York, NY, USA, 2000. ACM.

[9] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the Spring Joint Computer Conference (SJCC)*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[10] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, May 1988.

[11] Andrew A. Chien, Allan Snavely, and Mark Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science*, 4:1987–1996, January 2011.

[12] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, 2008. AAI3353349.

[13] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41:33–38, July 2008.

[14] *Intel Advanced Vector Extensions Programming Reference*, June 2011.

[15] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, March 2000.

[16] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: architecture and implementations. *IEEE Micro*, 19(2):37–48, March 1999.

[17] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro*, 20(4):47–57, July 2000.

[18] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54:67–77, May 2011.

[19] G. Martin and H. Chang. *Winning the SoC revolution: Experiences in real design*. Springer/Kluwer Academic Publishers, 2003.

[20] Ronald Sass and Andrew G. Schmidt. *Embedded Systems Design with Platform FPGAs: Principles and Practices*. Morgan Kaufmann, 1st edition, August 2010.

[21] Douglas Hofstadter. *Gödel, Escher, Bach*. Basic Books, 1979.

[22] K. Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. on Computers*, 39:300–318, March 1990.

[23] Paolo Ienne and Rainer Leupers. *Customizable Embedded Processors: Design Technologies and Applications*, chapter 16, pages 381–423. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[24] Matthew Naylor and Colin Runciman. The reduceron reconfigured. In Paul Hudak and Stephanie Weirich, editors, *Int. Conf. on Functional Programming (ICFP)*, pages 75–86. ACM, 2010.

[25] Ian Kuon and Jonathan Rose. *Quantifying and Exploring the Gap Between FPGAs and ASICs*. Springer/Kluwer Academic Publishers, 2011.

[26] Elaine Rhodes. *ASIC Basics: Black and White Edition*. Lulu.com, 2008.

[27] Hubert Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, 2008.

[28] Clive Maxfield. *FPGAs: Instant Access*. Newnes, 2008.

[29] Mariusz Grad and Christian Plessl. Woolcano: An Architecture and Tool Flow for Dynamic Instruction Set Extension on Xilinx Virtex-4 FX. In *Proc. 9th Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 319–322. CSREA Press, July 2009.

[30] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, pages 225–235. ACM, 2000.

[31] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. 27th Int. Symp. on Microarchitecture (MICRO-27)*, pages 172–180, New York, NY, USA, 1994. ACM.

[32] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, Peter M. Athanas, Harvey F. Silverman, and S. Ghosh. PRISM-II compiler and architecture. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16. IEEE Computer Society, April 1993.

[33] Francisco Barat, Murali Jayapala, Pieter Op De Beeck, and Geert Deconinck. Reconfigurable instruction set processors: A survey. In *IEEE Trans. on Software Engineering*, pages 168–173. IEEE, 2000.

[34] Carlo Galuzzi and Koen Bertels. The instruction-set extension problem: A survey. In *Proc. Int. Conf. on Architecture of Computing Systems (ARCS)*, number 4943 in LNCS, pages 209–220. Springer/Kluwer Academic Publishers, 2008.

[35] Mariusz Grad and Christian Plessl. Pruning the Design Space for Just-in-Time Processor Customization. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 67–72. IEEE Computer Society, December 2010.

[36] Raymond J. Hookway and Mark A. Herdeg. DIGITAL FX!32: combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[37] Vasanath Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization. Technical Report HPL-1999-78, HP Laboratories Cambridge, June 1999.

[38] K. Ebcioğlu and E. Altman. DAISY: Dynamic compilation for 100 percent architectural compatibility. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, pages 26–37, New York, 1997. ACM.

[39] Jim Smith and Ravi Nair, editors. *Virtual Machines: Versatile Platforms for Systems and Processes*. Computer Architecture and Design. Morgan Kaufmann Publishers Inc., June 2005.

[40] Antonio Carlos S. Beck and Luigi Carro. Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility. In *Proc. Design Automation Conference (DAC)*, pages 732–737, New York, NY, USA, 2005. ACM.

[41] Frank Vahid, Greg Stitt, and Roman Lysecky. Warp processing: Dynamic translation of binaries to FPGA circuits. *IEEE Computer*, 41:40–46, July 2008.

[42] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, März/April 2000.

[43] Mimosys, Lugano, Switzerland. *Mimosys Clarity product datasheet*, July 2006.

[44] Xilinx. *Early Access Partial Reconfiguration User Guide*, March 2006.

[45] Xilinx, Inc. *Constraints Guide*, 9.1i edition, 2007.

[46] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[47] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In *Proc. 3rd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, pages 99–107. IEEE Computer Society, April 1995.

[48] J. E. Carrillo Esparza and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proc. 9th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 141–150. ACM, 2001.

[49] Jeffrey M. Arnold. S5: the architecture and development flow of a software configurable processor. In *Int. Conf. on Field Programmable Technology (ICFPT)*, pages 121–128. IEEE Computer Society, December 2005.

[50] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, Apr 2000.

[51] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[52] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. on Computers*, 49(5):465–481, May 2000.

[53] Paula J. Pingree, Jean-Francois L. Blavier, Geoffrey C. Toon, and Dmitriy Bekker. An FPGA/SoC approach to on-board data processing enabling new mars science with smart payloads. In *Proc. IEEE Aerospace Conference (IEEEAC)*, pages 1–12. IEEE Computer Society, March 2007.

[54] Wade S. Fife and James K. Archibald. Reconfigurable on-board vision processing for small autonomous vehicles. *EURASIP Journal on Embedded Systems*, pages 33–46, 2007.

[55] Muhammad Omer Cheema and Omar Hammami. Customized SIMD unit synthesis for system on programmable chip: a foundation for HW/SW partitioning with vectorization. In *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 54–60, Piscataway, NJ, USA, 2006. IEEE Computer Society.

[56] Juergen Wassner, Klaus Zahn, and Dersch Ulrich. Hardware-software codesign of a tightly-coupled coprocessor for video content analysis. In *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, pages 87–92. IEEE, 2010.

[57] Glenn Steiner. Code acceleration with an APU coprocessor: a case study of an LPM algorithm. Technical report, Xilinx, 2008.

[58] Mariusz Grad and Christian Plessl. An Open Source Circuit Library with Benchmarking Facilities. In *Proc. 10th Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 144–150. CSREA Press, July 2010.

[59] Xilinx. *CORE Generator Guide*.

[60] Cristian Klein Florent de Dinechin and Bogdan Pasca. Generating high-performance custom floating-point pipelines. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, August 2009.

[61] Altera. *Megafunction Overview User Guide*.

[62] Aeroflex Gaisler. *GRLIB User's Manual*.

[63] Grégory Massal. A raytracer in C++.

[64] Huynh Phung Huynh, Joon Edward Sim, and Tulika Mitra. An efficient framework for dynamic reconfiguration of instruction-set customization. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 135–144, 2007.

[65] P Yu and Tulika Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 69–78. ACM, 2004.

[66] Laura Pozzi, Kubilay Atasu, and Paolo Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1209–1029, July 2006.

[67] Cesare Alippi William, William Fornaciari, Laura Pozzi, and Mariagiovanna Sami. A DAG-based design approach for reconfigurable VLIW processors. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 778–779. ACM, January 1999.

[68] Jie Gong, Daniel D. Gajski, and Sanjiv Narayan. Software estimation from executable specifications. *Journal of Computer Software Engineering*, 2:239–258, March 1994.

[69] IBM. *The PowerPC 405TM Core*, Nov 1998.

[70] Abhijit Ray, Thambipillai Srikanthan, and Wu Jigang. Practical techniques for performance estimation of processors. In *Proc. Int. Workshop on System-on-Chip for Real-Time Applications (IWSOC)*, pages 308–311, Washington, DC, USA, 2005. IEEE Computer Society.

[71] Byoungro So, Pedro C. Diniz, and Mary W. Hall. Using estimates from behavioral synthesis tools in compiler-directed design space exploration. In *Proc. 40th Design Automation Conf. (DAC)*, pages 514–519, New York, NY, USA, 2003. ACM.

[72] *Floating-Point Operator v5.0*.

[73] Naresh Maheshwari and Sachin S. Sapatnekar. *Timing analysis and optimization of sequential circuits*. Springer/Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[74] Yan Lin Aung, Siew Kei Lam, and Thambipillai Srikanthan. Performance estimation framework for FPGA-based processors. In Jinian Bian, Qiang Zhou, Peter Athanas, Yajun Ha, and Kang Zhao, editors, *Int. Conf. on Field Programmable Technology (ICFPT)*, pages 413–416. IEEE Computer Society, 2010.

[75] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006.

[76] RJ Meeuws, Yana Yankova, Koen Bertels, Georgi Gaydadjiev, and Stamatis Vassiliadis. A quantitative prediction model for hardware/software partitioning. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 735–739, 2007.

[77] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[78] Jürgen Teich and Christian Haubelt. *Digitale Hardware/Software-Systeme. Synthese und Optimierung*, chapter 4. Springer/Kluwer Academic Publishers, Berlin Heidelberg New York, 2nd edition, 2007.

[79] Hohenauer Manuel and Leupers Rainer. *C Compilers for ASIPs: Automatic Compiler Generation with LISA*. Springer/Kluwer Academic Publishers, October 2009.

[80] João M.P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer/Kluwer Academic Publishers, 2008.

[81] Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and Wen mei W. Hwu. Superblock formation using static program analysis. In *Proc. Int. Symp. on Microarchitecture (MICRO)*, pages 247–255. IEEE Computer Society, 1993.

[82] Sweta Verma, Ranjit Biswas, and J. B. Singh. Extension of superblock technique to hyperblock using predicate hierarchy graph. In Sanjay Ranka, Arunava Banerjee, Kanad Kishore Biswas, Sumeet Dua, Prabhat Mishra, Rajat Moona, Sheung-Hung Poon, and Cho-Li Wang, editors, *Contemporary Computing - Third International Conference, IC3 2010, Noida, India, August 9-11, 2010, Proceedings, Part II*, volume 95

of *Communications in Computer and Information Science*, pages 217–229. Springer, 2010.

[83] Weihaw Chuang, Brad Calder, and Jeanne Ferrante. Phi-Predication for light-weight if-conversion. In *Proc. 2003 Int. Symp. on Code Generation and Optimization (CGO)*, CGO '03, pages 179–190, Washington, DC, USA, 2003. IEEE Computer Society.

[84] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. Int. Symp. on Microarchitecture (MICRO)*, pages 45–54. ACM, 1992.

[85] C. Claus, F.H. Muller, J. Zeppenfeld, and W. Stechele. A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration. In *Proc. Int. Symp. on Parallel and Distributed Processing (IPDPS)*, pages 1–7. IEEE, 2007.

[86] Paolo Bonzini and Laura Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 1331–1336, San Jose, CA, USA, 2007. EDA Consortium.

[87] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 178–192, Berlin, Heidelberg, 2008. Springer-Verlag.

[88] Arvind and Robert A. Iannucci. A critique of multiprocessing von Neumann style. *ACM SIGARCH Computer Architecture News*, 11:426–436, June 1983.

[89] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, August 1978.

[90] Reiner Hartenstein. The Neumann Syndrome calls for a revolution. In *Proc. Int. Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)*, HPRCTA '07, New York, NY, USA, 2007. ACM.

[91] Sarita V. Adve and Hans-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53:90–101, August 2010.

[92] David W. Wall. Limits of instruction-level parallelism. *ACM SIGARCH Computer Architecture News*, 19:176–188, April 1991.

[93] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, 3rd edition, December 2007.

[94] Jason Cong and Yi Zou. Fpga-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. on Reconfigurable Technology and Systems*, 2:17:1–17:29, September 2009.

[95] Jian Li, Marinko V. Sarunic, and Lesley Shannon. Scalable, high performance fourier domain optical coherence tomography: Why fpgas and not gpgpus. *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 0:49–56, 2011.

[96] Jack W. Davidson and Sanjay Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 125–132, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[97] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005.

[98] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. 2004 Int. Symp. on Code Generation and Optimization (CGO)*, pages 75–86. IEEE Computer Society, 2004.

[99] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[100] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[101] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[102] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12), Dec 1994.

[103] Anthony Hyman. *Charles Babbage: Pioneer of the Computer*. Princeton University Press, 1985.

[104] Bernard Cohen. *Howard Aiken: Portrait of a Computer Pioneer*. MIT Press, 2000.

[105] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30:86–93, September 1997.